Paola Festa (Ed.)

LNCS 6049

Experimental Algorithms

9th International Symposium, SEA 2010
Ischia Island, Naples, Italy, May 2010
Proceedings

Springer

Paola Festa (Ed.)

Experimental Algorithms

9th International Symposium, SEA 2010
Ischia Island, Naples, Italy, May 20-22, 2010
Proceedings

Springer

Volume Editor

Paola Festa
Department of Mathematics and Applications
University of Naples "Federico II"
Compl. MSA
Via Cintia
80126 Naples, Italy
E-mail: paola.festa@unina.it

Library of Congress Control Number: 2010926863

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-13192-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-13192-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This proceedings volume contains the invited papers and the contributed papers accepted for presentation at the 9th International Symposium on Experimental Algorithms (SEA 2010), that was held at the Continental Terme Hotel, Ischia (Naples), Italy, during May 20–22, 2010.

Previous symposia of the series were held in Riga (2001), Monte Verita (2003), Rio de Janeiro (2004), Santorini (2005), Menorca (2006), Rome (2007), Cape Cod (2008), and Dortmund (2009).

Seventy-three papers were submitted by researchers from 19 countries. Each paper was reviewed by three experts among the Program Committee members and some trusted external referees. At least two reviewers were from the same or closely related discipline as the authors. The reviewers generally provided a high-quality assessment of the papers and often gave extensive comments to the authors for the possible improvement of the presentation. The submission and review process was supported by the ConfTool conference management software and we are thankful to Harald Weinreich for letting us use it.

The Program Committee selected 40 regular papers for presentation at the conference. In addition to the 40 contributed papers, this volume includes two invited papers related to corresponding keynote talks: Giuseppe F. Italiano (University of Rome "Tor Vergata,"Italy) spoke on "Experimental Study of Resilient Algorithms and Data Structures" and Panos M. Pardalos (University of Florida, USA) spoke on "Computational Challenges with Cliques, Quasi-Cliques and Clique Partitions in Graphs."

Many people and organizations contributed to SEA 2010. We are particularly grateful for the patronage and financial support of the University of Naples "Federico II" and the Department of Mathematics and Applications "R. Caccioppoli,"and for the financial support of GNCS (Gruppo Nazionale per il Calcolo Scientifico) – INdAM (Istituto Nazionale di Alta Matematica).

We would like to thank all of the authors who responded to the call for papers submitting their scientific work. We express our sincere thanks to the invited speakers for their contributions to the program.

Our most sincere thanks go to the Program Committee members and the additional reviewers whose cooperation in carrying out quality reviews was critical for establishing a strong conference program.

We also sincerely thank Daniele Ferone for maintaining the symposium website (`http://www.sea2010.unina.it/`) and for his help in the organizing process.

I thank the SEA Steering Committee for giving me the opportunity to serve as the Program Chair and the responsibility to select the conference program. I am personally grateful to Jose Rolim, Giuseppe F. Italiano, Andrea Lodi, and

Andrew V. Goldberg for their support from the beginning to the final stage of the organization and for promptly answering my questions each time I needed their advice.

Finally, we thank Springer for publishing these proceedings in their prestigious *Lecture Notes in Computer Science* series, and in particular we would like to mention the fruitful and friendly cooperation with Alfred Hofmann and Anna Kramer during the preparation of this volume.

May 2010 Paola Festa

Organization

SEA 2010 was organized by the Department of Mathematics and Applications "R. Caccioppoli,"University of Naples "Federico II."

Program Committee

Steering Committee

Edoardo Amaldi	Politecnico di Milano, Italy
David A. Bader	Georgia Institure of Technology, USA
Josep Diaz	Universitat Politecnica de Catalunya, Spain
Giuseppe F. Italiano	University of Rome “Tor Vergata,”Italy
David Johnson	AT&T Labs Research, USA
Klaus Jansen	University of Kiel, Germany
Kurt Mehlhorn	Max-Plack-Institut für Informatik, Germany
Ian Munro	University of Waterloo, Canada
Sotiris Nikoletseas	University of Patras and CTI, Greece
Jose Rolim (Chair)	University of Geneva, Switzerland
Pavlos Spirakis	University of Patras and CTI, Greece

Organizing Committee

Daniele Ferone	University of Naples “Federico II,”Italy
Paola Festa (Chair)	University of Naples “Federico II,”Italy

Referees

Virat Agarwal
Orestis Akribopoulos
Edoardo Amaldi
Enrico Angelelli
Andreas Beckmann
Djamal Belazzougui
Silvia Canale
Claude Chaudet
Guojing Cong
Anna Corazza
Alexandre S. da Cunha
Claudia D'Ambrosio
Sanjeeb Dash
Daniel Delling
Camil Demetrescu
David Ediger
Matthias Ehrgott
Marco Faella
Juan Farré
Leonor Frias
Joaquim Gabarró
Clemente Galdi
Loukas Georgiadis
Karl Jiang
Seunghwa Kang
Christos Koninis
Evangelos Kranakis
Xu Li
Xing Liu
Michele Lombardi
Zvi Lotker
Matthias Müller -Hannemann
Kamesh Madduri
Enrico Malaguti
Othon Michail
Andrew Miller
Thomas Molhave
Carlos Oliveira
Vitaly Osipov
Apostolis Pyrgelis
Mathieu Raffinot
Jason Riedy
Frederic Roupin
Salvador Roura
Vipin Sachdeva
Hanan Samet
Peter Sanders
Ricardo Silva
Thomas Stidsen
Andrea Tramontani
Renato Werneck
Christos Zaroliagis

Sponsoring Institutions

- Department of Mathematics and Applications “R. Caccioppoli”, University of Naples “Federico II,”Italy.
- GNCS (Gruppo Nazionale per il Calcolo Scientifico) - INdAM (Istituto Nazionale di Alta Matematica), Italy.

Table of Contents

Invited Papers

Experimental Study of Resilient Algorithms and Data Structures 1
Umberto Ferraro-Petrillo, Irene Finocchi, and Giuseppe F. Italiano

Computational Challenges with Cliques, Quasi-cliques and Clique Partitions in Graphs 13
Panos M. Pardalos and Steffen Rebennack

Contributed Regular Papers

Alternative Routes in Road Networks 23
Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck

Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks 35
Annabell Berger, Martin Grimmer, and Matthias Müller-Hannemann

Space-Efficient SHARC-Routing 47
Edith Brunel, Daniel Delling, Andreas Gemsa, and Dorothea Wagner

A New Fully Dynamic Algorithm for Distributed Shortest Paths and Its Experimental Evaluation 59
Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, Daniele Frigioni, and Vinicio Maurizio

Contraction of Timetable Networks with Realistic Transfers 71
Robert Geisberger

Distributed Time-Dependent Contraction Hierarchies 83
Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter

Practical Compressed Suffix Trees 94
Rodrigo Cánovas and Gonzalo Navarro

Maximum Cliques in Protein Structure Comparison 106
Noël Malod-Dognin, Rumen Andonov, and Nicola Yanev

Exact Bipartite Crossing Minimization under Tree Constraints 118
Frank Baumann, Christoph Buchheim, and Frauke Liers

Bit-Parallel Search Algorithms for Long Patterns 129
Branislav Ďurian, Hannu Peltola, Leena Salmela, and Jorma Tarhio

Fast FPT Algorithms for Computing Rooted Agreement Forests: Theory and Experiments (Extended Abstract) 141
Chris Whidden, Robert G. Beiko, and Norbert Zeh

Experimental Evaluation of Approximation and Heuristic Algorithms for Sorting Railway Cars 154
Alain Hauser and Jens Maue

Time-Dependent Contraction Hierarchies and Approximation 166
Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders

A New Combinational Logic Minimization Technique with Applications to Cryptology 178
Joan Boyar and René Peralta

Randomized Rounding for Routing and Covering Problems: Experiments and Improvements 190
Benjamin Doerr, Marvin Künnemann, and Magnus Wahlström

The Time Dependent Traveling Salesman Problem: Polyhedra and Branch-Cut-and-Price Algorithm 202
Hernán Abeledo, Ricardo Fukasawa, Artur Pessoa, and Eduardo Uchoa

An Approximate ϵ-Constraint Method for the Multi-objective Undirected Capacitated Arc Routing Problem 214
Lucio Grandinetti, Francesca Guerriero, Demetrio Laganà, and Ornella Pisacane

A Branch-and-Price Algorithm for Multi-mode Resource Leveling 226
Eamonn T. Coughlan, Marco E. Lübbecke, and Jens Schulz

Experiments with a Generic Dantzig-Wolfe Decomposition for Integer Programs 239
Gerald Gamrath and Marco E. Lübbecke

Using Bound Sets in Multiobjective Optimization: Application to the Biobjective Binary Knapsack Problem 253
Charles Delort and Olivier Spanjaard

Improving Cutting Plane Generation with 0-1 Inequalities by Bi-criteria Separation 266
Edoardo Amaldi, Stefano Coniglio, and Stefano Gualandi

New Lower Bounds for the Vehicle Routing Problem with Simultaneous Pickup and Delivery 276
Anand Subramanian, Eduardo Uchoa, and Luiz Satoru Ochi

A Metaheuristic for a Two Echelon Location-routing Problem 288
Maurizio Boccia, Teodor G. Crainic, Antonio Sforza, and Claudio Sterle

New Fast Heuristics for the 2D Strip Packing Problem with Guillotine Constraint 302
Minh Hoang Ha, François Clautiaux, Saïd Hanafi, and Christophe Wilbaut

An Experimental Comparison of Different Heuristics for the Master Bay Plan Problem 314
Daniela Ambrosino, Davide Anghinolfi, Massimo Paolucci, and Anna Sciomachen

An Analysis of Heuristics for Vertex Colouring 326
Marco Chiarandini and Thomas Stützle

Automatic Tuning of GRASP with Path-Relinking Heuristics with a Biased Random-key Genetic Algorithm 338
Paola Festa, José F. Gonçalves, Mauricio G.C. Resende, and Ricardo M.A. Silva

Experiments with a Feasibility Pump Approach for Nonconvex MINLPs 350
Claudia D'Ambrosio, Antonio Frangioni, Leo Liberti, and Andrea Lodi

Paging Multiple Users in Cellular Network: Yellow Page and Conference Call Problems 361
Amotz Bar-Noy, Panagiotis Cheilaris, and Yi Feng

Realtime Classification for Encrypted Traffic 373
Roni Bar-Yanai, Michael Langberg, David Peleg, and Liam Roditty

Data Propagation with Guaranteed Delivery for Mobile Networks 386
Hakob Aslanyan, Pierre Leone, and Jose Rolim

Data Structures Resilient to Memory Faults: An Experimental Study of Dictionaries 398
Umberto Ferraro-Petrillo, Fabrizio Grandoni, and Giuseppe F. Italiano

Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure 411
Md. Mostofa Ali Patwary, Jean Blair, and Fredrik Manne

Policy-Based Benchmarking of Weak Heaps and Their Relatives 424
Asger Bruun, Stefan Edelkamp, Jyrki Katajainen, and Jens Rasmussen

Modularity-Driven Clustering of Dynamic Graphs 436
Robert Görke, Pascal Maillard, Christian Staudt, and Dorothea Wagner

Gateway Decompositions for Constrained Reachability Problems 449
Bastian Katz, Marcus Krug, Andreas Lochbihler, Ignaz Rutter, Gregor Snelting, and Dorothea Wagner

Robust and Efficient Delaunay Triangulations of Points on Or Close to a Sphere 462
Manuel Caroli, Pedro M.M. de Castro, Sébastien Loriot, Olivier Rouiller, Monique Teillaud, and Camille Wormser

Fault Recovery in Wireless Networks: The Geometric Recolouring Approach 474
Henk Meijer, Yurai Núñez-Rodríguez, and David Rappaport

Geometric Minimum Spanning Trees with GeoFilterKruskal 486
Samidh Chatterjee, Michael Connor, and Piyush Kumar

Practical Nearest Neighbor Search in the Plane 501
Michael Connor and Piyush Kumar

Author Index 513

Experimental Study of Resilient Algorithms and Data Structures*

Umberto Ferraro-Petrillo[1], Irene Finocchi[2], and Giuseppe F. Italiano[3]

[1] Dipartimento di Statistica, Probabilità e Statistiche Applicate, Università di Roma "La Sapienza", P.le Aldo Moro 5, 00185 Rome, Italy
umberto.ferraro@uniroma1.it

[2] Dipartimento di Informatica, Università di Roma "La Sapienza", via Salaria 113, 00198, Roma, Italy
finocchi@di.uniroma1.it

[3] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", via del Politecnico 1, 00133 Roma, Italy
italiano@disp.uniroma2.it

Abstract. Large and inexpensive memory devices may suffer from faults, where some bits may arbitrarily flip and corrupt the values of the affected memory cells. The appearance of such faults may seriously compromise the correctness and performance of computations. In recent years, several algorithms for computing in the presence of memory faults have been introduced in the literature: in particular, we say that an algorithm or a data structure is *resilient* if it is able to work correctly on the set of uncorrupted values. In this invited talk, we contribute carefully engineered implementations of recent resilient algorithms and data structures and report the main results of a preliminary experimental evaluation of our implementations.

1 Introduction

Many large-scale applications require to process massive data sets, which can easily reach the order of Terabytes. For instance, NASA's Earth Observing System generates Petabytes of data per year, while Google currently reports to be indexing and searching several billions of Web pages. In all applications processing massive data sets, there is an increasing demand for large, fast, and inexpensive memories. Unfortunately, such memories are not fully safe, and sometimes the contents of their cells may be corrupted. This may depend on manufacturing defects, power failures, or environmental conditions such as cosmic radiation and alpha particles [25,37,45,46]. As the memory size becomes larger, the mean time between failures decreases considerably: assuming standard soft error rates for the internal memories currently on the market [45], a system with Terabytes

* This work has been partially supported by the 7th Framework Programme of the EU (Network of Excellence "EuroNF: Anticipating the Network of the Future - From Theory to Design") and by MIUR, the Italian Ministry of Education, University and Research, under Project AlgoDEEP.

of memory (e.g., a cluster of computers with a few Gigabytes per node) could experience one bit error every few minutes.

A faulty memory may cause a variety of problems in most software applications, which in some cases can also pose very serious threats, such as breaking cryptographic protocols [6,7,48], taking control over a Java Virtual Machine [24] or breaking smart cards and other security processors [1,2,44]. Most of these fault-based attacks work by manipulating the non-volatile memories of cryptographic devices, so as to induce very timing-precise controlled faults on given individual bits: this forces a cryptographic algorithm to output wrong ciphertexts that may allow the attacker to recover the secret key used during the encryption. Differently from the almost random errors affecting the behavior of large size memories, in this context the errors can be considered as introduced by a malicious adversary, which can have some knowledge of the algorithm's behavior.

We stress that even very few memory faults may jeopardize the correctness of the underlying algorithms. Consider for example a simplified scenario, where we are given a set of n keys, to be sorted with a classical sorting algorithm, say mergesort. It is easy to see that the corruption of a single key during the merge step is sufficient to make the algorithm output a sequence with $\Theta(n)$ keys in a wrong position. Similar phenomena have been observed in practice [26,27].

The classical way to deal with memory faults is via error detection and correction mechanisms, such as redundancy, Hamming codes, etc. These traditional approaches imply non-negligible costs in terms of performance and money, and thus they do not provide a feasible solution when speed and cost are both at prime concern. In the design of reliable systems, when specific hardware for fault detection and correction is not available or it is too expensive, it makes sense to look for a solution to these problems at the application level, i.e., to design algorithms and data structures that are able to perform the tasks they were designed for, even in the presence of unreliable or corrupted information.

Dealing with unreliable information has been addressed in the algorithmic community in a variety of different settings, including the liar model [3,8,17,31,39], fault-tolerant sorting networks [4,34,49], resiliency of pointer-based data structures [5], and parallel models of computation with faulty memories [14,28].

In this paper we focus on the *faulty-RAM* model introduced in [20,23]. In this model, an adaptive adversary can corrupt any memory word, at any time, by overwriting its value. Corrupted values cannot be (directly) distinguished from correct ones. An upper bound δ is given on the total number of memory faults that can occur throughout the execution of an algorithm or during the lifetime of a data structure. We remark that δ is not a constant, but a parameter of the model. The adaptive adversary captures situations like cosmic-rays bursts, memories with non-uniform fault-probability, and hackers' attacks which would be difficult to be modelled otherwise. The model has been extended to external memories [12].

The faulty-RAM model assumes that there are $O(1)$ *safe* memory words which cannot be accessed by the adversary, and thus can never get corrupted. The safe memory stores the code of the algorithms / data structures (which otherwise could be corrupted by the adversary), together with a small number of running variables. This assumption is not very restrictive, since typically the space occupied by the code is orders of magnitude smaller than the space required by data, especially in case of large-scale applications: hence, one can usually afford the cost of storing the code in a smaller, more expensive, and more reliable memory. We remark that a constant-size reliable memory may even not be sufficient for a recursive algorithm to work properly: parameters, local variables, return addresses in the recursion stack may indeed get corrupted.

A natural approach to the design of algorithms and data structures in the faulty-RAM model is data replication. Informally, a *resilient variable* consists of $(2\delta + 1)$ copies x_1, x_2,..., $x_{2\delta+1}$ of a standard variable. The value of a resilient variable is given by the majority of its copies, which can be computed in linear time and constant space [9]. Observe that the value of x is reliable, since the adversary cannot corrupt the majority of its copies. The approach above, which we call *trivially resilient*, induces a $\Theta(\delta)$ multiplicative overhead in terms of both space and running time. For example, a trivially-resilient implementation of a standard dictionary based on balanced binary search trees would require $O(\delta n)$ space and $O(\delta \log n)$ time for each `search`, `insert`, and `delete` operation. Thus, it can tolerate only $O(1)$ memory faults while maintaining asymptotically optimal time and space bounds.

This type of overhead seems unavoidable for certain problems, especially if we insist on computing a totally correct output (even on corrupted data). For example, with less than $2\delta + 1$ copies of a key, we cannot avoid that its correct value gets lost. Since a $\Theta(\delta)$ multiplicative overhead could be unacceptable in several applications even for small values of δ, it seems natural to relax the definition of correctness and to require that algorithms/data structures be consistent only with respect to the uncorrupted data. For example, we might accept that the keys which get corrupted possibly occupy a wrong position in a sorted output sequence, provided that at least all the remaining keys are sorted correctly. In this framework, we say that an algorithm or data structure is *resilient* to memory faults if, despite the corruption of some memory location during its lifetime, it is nevertheless able to operate correctly (at least) on the set of uncorrupted values. Of course, every trivially-resilient algorithm is also resilient. However, there are resilient algorithms for some natural problems which perform much better than their trivially-resilient counterparts. It is worth mentioning that the algorithms and data structures that we are going to present might not always be able to detect faulty behaviors: nonetheless, they are able to operate correctly on the uncorrupted data.

In this paper, we contribute carefully engineered implementations of recent resilient algorithms and data structures and report the main results of a preliminary experimental study based on those implementations.

2 Resilient Algorithms

Sorting and searching have been the first problems studied in the faulty RAM model. More recently, the problem of counting in the presence of memory faults has been addressed in [11], and the design of resilient dynamic programming algorithms has been investigated in [13].

2.1 Resilient Sorting

The *resilient sorting* problem can be defined as follows. We are given a set K of n keys, with each key being a real value. We call a key *faithful* if it is never corrupted, and *faulty* otherwise. The problem is to compute a *faithfully sorted* permutation of K, that is a permutation of K such that the subsequence induced by the faithful keys is sorted. This is the best one can hope for, since an adversary can corrupt a key at the very end of the algorithm execution, thus making faulty keys occupy wrong positions. This problem can be trivially solved in $O(\delta\, n \log n)$ time. A resilient version of mergesort, referred to as `ResMergeSort`, with $O(n \log n + \delta^2)$ running time is presented in [21]. In [23] it is proved that `ResMergeSort` is essentially optimal, as any $O(n \log n)$ comparison-based sorting algorithm can tolerate the corruption of at most $O((n \log n)^{1/2})$ keys. In the special case of polynomially-bounded integer keys, an improved running time $O(n+\delta^2)$ can be achieved [21]. The space usage of all resilient sorting algorithms in [21] is $O(n)$. The interested reader is referred to [18] for an experimental study of resilient sorting algorithms: in particular, the experiments carried out in [18] show that a careful algorithmic design can have a great impact on the performance and reliability achievable in practice for resilient sorting.

2.2 Resilient Searching

The *resilient searching* problem is investigated in [21,23]. Here, we are given a search key κ and a faithfully sorted sequence K of n keys, i.e., a sequence in which all faithful keys are correctly ordered. The problem is to return a key (faulty or faithful) of value κ, if K contains a faithful key of that value. If there is no faithful key equal to κ, one can either return *no* or return a (faulty) key equal to κ. Note that, also in this case, this is the best possible: the adversary may indeed introduce a corrupted key equal to κ at the very beginning of the algorithm, such that this corrupted key cannot be distinguished from a faithful one. Hence, the algorithm might return that corrupted key both when there is a faithful key of value κ (rather than returning the faithful key), and when such faithful key does not exist (rather than answering *no*). There is a trivial algorithm which solves this problem in $O(\delta \log n)$ time. A lower bound of $\Omega(\log n + \delta)$ is described in [23] for deterministic algorithms, and later extended to randomized algorithms in [21]. An optimal $O(\log n + \delta)$ randomized algorithm `ResSearch` is provided in [21], while an optimal $O(\log n + \delta)$ deterministic algorithm is given in [10]. The space usage of all resilient searching algorithms in [10,21] is $O(n)$.

3 Resilient Data Structures

Besides the algorithms mentioned in Section 2, a variety of resilient data structures has been proposed in the literature, including resilient priority queues and resilient dictionaries.

3.1 Resilient Priority Queues

A *resilient priority queue* maintains a set of elements under the operations `insert` and `deletemin`: `insert` adds an element and `deletemin` deletes and returns either the minimum uncorrupted value or a corrupted value. Observe that this is consistent with the resilient sorting problem discussed in Section 2.1: given a sequence of n elements, inserting all of them into a resilient priority queue and then performing n `deletemin` operations yields a sequence in which uncorrupted elements are correctly sorted. In [30] Jørgensen *et al.* present a resilient priority queue that supports both `insert` and `deletemin` operations in $O(\log n + \delta)$ amortized time per operation. Thus, their priority queue matches the performance of classical optimal priority queues in the RAM model when the number of corruptions tolerated is $O(\log n)$. An essentially matching lower bound is also proved in [30]: a resilient priority queue containing n elements, with $n > \delta$, must ask $\Omega(\log n + \delta)$ comparisons to answer an `insert` followed by a `deletemin`. We are not aware of any implementation of the resilient priority queue of Jørgensen *et al.* [30]. One of the contribution of this paper is an implementation of this data structure, called `ResPQ`, which will be discussed in Section 4.

3.2 Resilient Dictionaries

A *resilient dictionary* is a dictionary where the `insert` and `delete` operations are defined as usual, while the `search` operation must be resilient as described in Section 2.2. In [10], Brodal *et al.* present a simple randomized algorithm `Rand` achieving optimal $O(\log n + \delta)$ time per operation. Using an alternative, more sophisticated approach, they also obtain a deterministic resilient dictionary `Det` with the same asymptotic performances. For all the mentioned implementations, the space usage is $O(n)$, which is optimal. The interested reader is referred to [19] for an experimental study of the resilient dictionaries of Brodal *et al.* [10]. One of the main findings from the experimental evalutaion of resilient dictionaries is that they tend to have very large space overheads. This might not be acceptable in practice since resilient data structures are meant for applications running on very large data sets. The work in [19] contributes an alternative implementation of resilient dictionaries, which we call `ResDict`. This implementation still guarantees optimal asymptotic time and space bounds, but performs much better in terms of memory usage without compromising the overall time efficiency.

4 Experiments

In this section we summarize our main experimental findings on resilient dictionaries, priority queues, and sorting. We restrict our experimental evaluation to the following implementations, which appeared to be the most efficient in practice according to previous experimental investigations and to our own experiments:

– `ResMergeSort`: resilient mergesort as described in [21] and implemented in [18].
– `ResPQ`: resilient priority queues supporting `insert` and `extractMin` operations as described in [30] and implemented in this paper.
– `ResDict`: resilient randomized dictionaries implemented on top of AVL trees as inspired from [10] and implemented in [19]. The implementation supports `insert`, `delete`, `search`, and `visit` operations.

On the non-resilient side, we considered standard binary heaps and AVL trees. We implemented all the algorithms in `C++`, within the same algorithmic and implementation framework, and performed experiments in a software testbed that simulates different fault injection strategies, as described in [18]. We performed experiments both on random and non-random inputs, such as the English dictionary and a few English books. For lack of space, in this paper we describe only our experiments with random inputs and on faults that are evenly spread over the entire algorithm execution and hit randomly chosen memory locations: our experiments with non-random inputs and different fault-injection strategies provide similar results. For pointer-based data structures, we distinguish between faults corrupting pointers and faults corrupting keys.

The experiments reported in this paper have been carried out on a workstation equipped with two Quad-Core Xeon E5520 processors with 2,26 Ghz clock rate and 48 GB RAM. The workstation runs Scientific Linux 4, kernel 2.6.18. All programs have been compiled through the GNU gcc compiler version 4.1.2 with optimization level `O3`.

4.1 Priority Queues

In order to operate correctly, resilient data structures must cope with memory faults and be prepared to pay some overhead in their running times. In this section we attempt to evaluate this overhead by comparing resilient priority queues against non-resilient implementations of binary heaps. The outcome of one such experiment is illustrated in Figure 1.

According to the theoretical analysis in [30], both `insert` and `extractMin` operations in a resilient priority queue have a running time of $O(\log n + \delta)$. Figure 1 shows that in practice the performances of both operations degrade substantially as δ increases. This is especially true for `extractMin` operations, that in our experiments are consistently one order of magnitude slower than insertions. This difference of performances may be explained with the following argument. The `ResPQ` data structure of [30] makes use of $O(\log n)$ up and down buffers of appropriate sizes. The keys in each buffer are faithfully ordered, i.e., all the uncorrupted elements in each buffer are correctly ordered with respect to each other. Both `insert` and `extractMin` operations need to merge adjacent buffers whenever the buffer size constraints are violated, but this expensive step can be amortized over sequences of operations. In the case of `extractMin`, however, whenever the minimum key is extracted from the appropriate buffer, all the elements from the beginning of the buffer up to the position of the minimum key

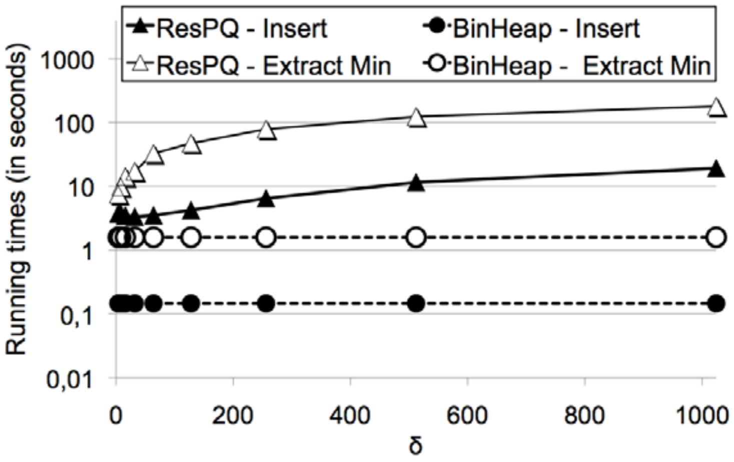


Fig. 1. Time required by $5 \cdot 10^6$ `insert` and $5 \cdot 10^6$ `extractMin` operations in non-resilient binary heaps and resilient priority queues for $\delta \in [2^2, 2^{10}]$. Times are reported using a logarithmic scale.

are right-shifted, so that elements in each buffer are always stored consecutively. This requires time proportional to δ for each operation (and cannot be amortized), resulting in larger running times in practice as δ increases. The resilient priority queues of Jørgensen *et al.* [30] appear thus to be very sensitive to δ (the upper bound on number of faults): in particular, whenever the actual number of faults is not known in advance, a bad estimate of δ may result in a substantial degradation in the performance of the data structure.

4.2 Looking at Resilient Data Structures from a Sorting Perspective

In order to get a feeling on the behavior of different resilient algorithms and data structures in a possibly coherent framework, we implemented sorting algorithms based on mergesort, on priority queues and on dictionaries. In the case of priority queues, we clearly implemented heapsort: we start from an initially empty (possibly resilient) priority queue, and perform n key insertions followed by $(n-1)$ `extractMin` operations. In the case of dictionaries, we perform n key insertions starting from an empty dictionary, and then visit the resulting tree in order to produce the sorted output sequence: we refer to this algorithm as AVL-sort.

Figure 2 shows the results of an experiment where we attempt to measure the impact of memory faults on the correctness of non-resilient mergesort, heapsort, and AVL-sort. In this experiment we assumed that faults do not affect pointers of the AVL tree (otherwise, AVL-sort is very likely to crash often). After running the algorithms, we measure the disorder in the output sequence produced by the algorithm using the k-unordered metric: namely, we estimate how many (faithful) keys need to be removed in order to obtain a (faithfully) sorted subsequence.

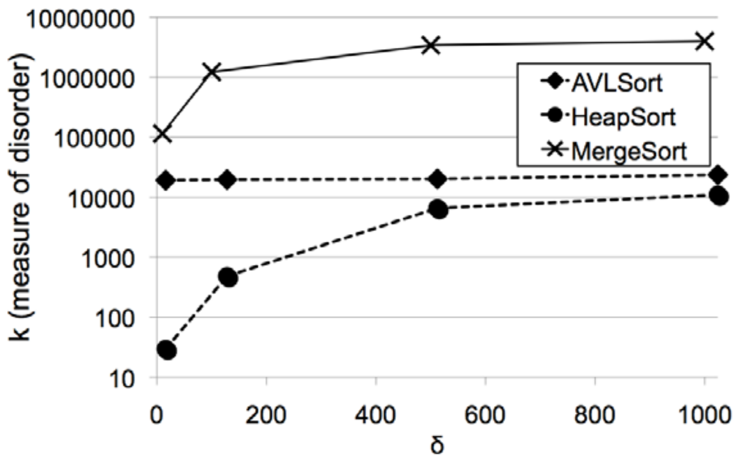


Fig. 2. Disorder produced by random memory faults in the sequence output by heapsort, AVL-based sort, and mergesort. The measure of disorder (k) is reported using a logarithmic scale.

The three algorithms considered exhibit rather different behaviors under this metric. Indeed our experiments show a deep vulnerability of mergesort even in the presence of very few random faults: when sorting 5 million elements, it is enough to have only 1000 random faults (i.e., roughly only 0.02% of the input size) to get a k-unordered output sequence for $k \approx 4 \cdot 10^6$: in other words, only 0.02% faults in the input are able to produce errors in approximately 80% of the output. The other two algorithms appear to be more resilient. Consider first the case of AVL trees. Since the tree at the beginning is empty, the first few faults that are injected are responsible of a rather large disorder in the output: when $\delta = 16$, about $20 \cdot 10^3$ elements out of $5 \cdot 10^6$ are disordered. However, this number remains almost constant as δ increases: random faults injected late during the execution are likely to hit the bottom part of the tree, and thus do not introduce much additional disorder. Heapsort is substantially more resilient than both mergesort and AVL-sort. The increasing trend of the heapsort curve in Figure 2 appears to depend on the way heaps are restructured when the minimum is deleted: faulty keys may be moved to the root of the tree, introducing errors that may easily propagate. Hence, larger numbers of faults result in larger disorder in the output.

We now analyze the performance of the sorting algorithms under examination. Figure 3 shows the outcome of experiments where each sorting algorithm considered is required to sort inputs of size $n = 5 \cdot 10^6$, for increasing values of δ. We first note that resilient mergesort (`ResMergeSort`) is always the fastest algorithm among the resilient (and even non-resilient) sorting implementations. This comes at no surprise, since `ResMergeSort` is an *ad hoc*, carefully optimized sorting algorithm. As it can be seen from Figure 3, the implementation based on priority queues is preferable to dictionary-based sorting for small values of

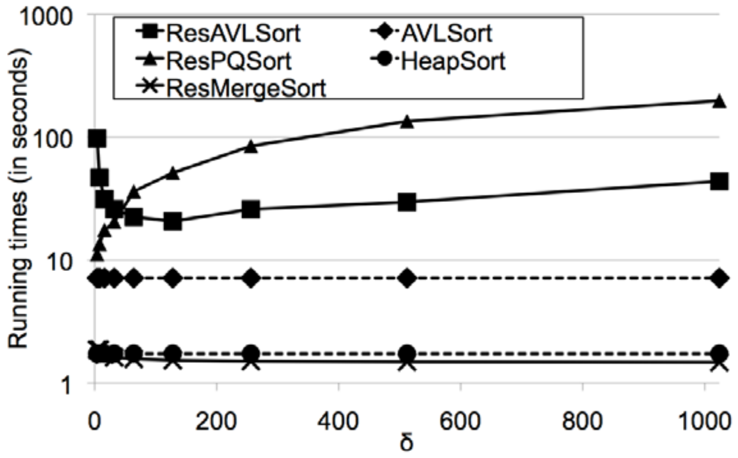


Fig. 3. Running times of non-resilient and resilient heapsort, non-resilient and resilient AVL-sort, and resilient mergesort. In this experiment $n = 5 \cdot 10^6$ and δ increases up to 1000. Times are reported using a logarithmic scale.

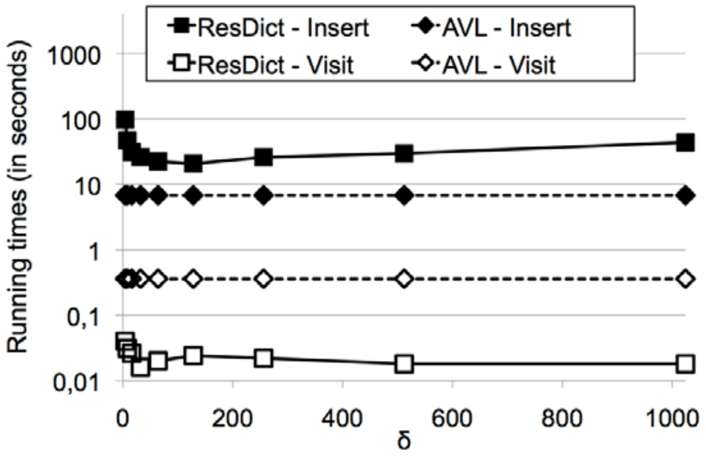


Fig. 4. Time required to carry out $5 \cdot 10^6$ `insert` operations followed by a `visit` of non-resilient and resilient AVL trees for $\delta \in [2^2, 2^{10}]$. Times are reported using a logarithmic scale.

δ, but its performance degrades quickly as δ increases, in spite of the fact that heapsort is more naturally resilient to memory faults than AVL-sort, as shown in Figure 2. As already observed in Section 4.1, this phenomenon is mainly due to the `extractMin` operations, which appear to be very inefficient in practice for large values of δ.

In order to better understand the poor performance of AVL-sort for very small values of δ, we experimented separately with `insert` and `visit` operations in dictionaries: the outcome of one of those experiments is reported in Figure 4. Although operations in dictionaries and priority queues have the same $O(\log n + \delta)$ asymptotic running time, a comparative analysis of Figure 1 and Figure 4 shows that for small values of δ insertions in priority queues are about twice as fast as insertions in dictionaries: this is not surprising, since priority queues do not need to maintain a total order between their keys, and thus they are likely to implement insertions more efficiently than dictionaries. In resilient dictionaries, nodes of the AVL tree maintain $\Theta(\delta)$ keys: when δ is small, they may be often restructured in order to keep the appropriate number of keys in each node. This operation dominates the running time, making insertions slower than in the case of priority queues.

On the other side, visiting a resilient AVL is even faster than visiting a non-resilient AVL tree of the same size n, even if in the former case pointers are stored reliably and their majority value needs to be computed during the visit. This behavior can be explained by analyzing constants used in our `ResDict` implementation: we store $2\delta + 1$ copies of each pointer, but each tree node contains about $64\,\delta$ faithfully ordered keys. Hence, the entire visit on a resilient AVL requires to follow about $(2\delta n)/(64\delta) = n/32$ pointers, which is less than the n pointer jumps taken during a standard visit of a non-resilient AVL tree.

References

1. Anderson, R., Kuhn, M.: Tamper resistance – a cautionary note. In: Proc. 2nd Usenix Workshop on Electronic Commerce, pp. 1–11 (1996)
2. Anderson, R., Kuhn, M.: Low cost attacks on tamper resistant devices. In: Proc. International Workshop on Security Protocols, pp. 125–136 (1997)
3. Aslam, J.A., Dhagat, A.: Searching in the presence of linearly bounded errors. In: Proc. 23rd STOC, pp. 486–493 (1991)
4. Assaf, S., Upfal, E.: Fault-tolerant sorting networks. SIAM J. Discrete Math. 4(4), 472–480 (1991)
5. Aumann, Y., Bender, M.A.: Fault-tolerant data structures. In: Proc. 37th IEEE Symp. on Foundations of Computer Science (FOCS 1996), pp. 580–589 (1996)
6. Blömer, J., Seifert, J.-P.: Fault based cryptanalysis of the Advanced Encryption Standard (AES). In: Wright, R.N. (ed.) FC 2003. LNCS, vol. 2742, pp. 162–181. Springer, Heidelberg (2003)
7. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
8. Borgstrom, R.S., Rao Kosaraju, S.: Comparison based search in the presence of errors. In: Proc. 25th STOC, pp. 130–136 (1993)
9. Boyer, R., Moore, S.: MJRTY — A fast majority vote algorithm. In: Automated Reasoning: Essays in Honor of Woody Bledsoe, pp. 105–118 (1991)
10. Brodal, G.S., Fagerberg, R., Finocchi, I., Grandoni, F., Italiano, G.F., Jørgensen, A.G., Moruz, G., Mølhave, T.: Optimal Resilient Dynamic Dictionaries. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 347–358. Springer, Heidelberg (2007)

11. Brodal, G.S., Jørgensen, A.G., Moruz, G., Mølhave, T.: Counting in the presence of memory faults. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 842–851. Springer, Heidelberg (2009)
12. Brodal, G.S., Jørgensen, A.G., Mølhave, T.: Fault tolerant external memory algorithms. In: Proc. 11th Int. Symposium on Algorithms and Data Structures (WADS 2009), pp. 411–422 (2009)
13. Caminiti, S., Finocchi, I., Fusco, E.G.: Resilient dynamic programming (2010) (manuscript)
14. Chlebus, B.S., Gambin, A., Indyk, P.: Shared-memory simulations on a faulty-memory DMM. In: Meyer auf der Heide, F., Monien, B. (eds.) ICALP 1996. LNCS, vol. 1099, pp. 586–597. Springer, Heidelberg (1996)
15. Chlebus, B.S., Gasieniec, L., Pelc, A.: Deterministic computations on a PRAM with static processor and memory faults. Fundamenta Informaticae 55(3-4), 285–306 (2003)
16. Dhagat, A., Gacs, P., Winkler, P.: On playing "twenty questions" with a liar. In: Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms (SODA 1992), pp. 16–22 (1992)
17. Feige, U., Raghavan, P., Peleg, D., Upfal, E.: Computing with noisy information. SIAM J. on Comput. 23, 1001–1018 (1994)
18. Ferraro-Petrillo, U., Finocchi, I., Italiano, G.F.: The Price of Resiliency: a Case Study on Sorting with Memory Faults. Algorithmica 53(4), 597–620 (2009)
19. Ferraro-Petrillo, U., Grandoni, F., Italiano, G.F.: Data Structures Resilient to Memory Faults: An Experimental Study of Dictionaries. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 398–410. Springer, Heidelberg (2010)
20. Finocchi, I., Grandoni, F., Italiano, G.F.: Designing reliable algorithms in unreliable memories. Computer Science Review 1(2), 77–87 (2007)
21. Finocchi, I., Grandoni, F., Italiano, G.F.: Optimal sorting and searching in the presence of memory faults. Theor. Comput. Sci. 410(44), 4457–4470 (2009)
22. Finocchi, I., Grandoni, F., Italiano, G.F.: Resilient Search Trees. ACM Transactions on Algorithms 6(1), 1–19 (2009)
23. Finocchi, I., Italiano, G.F.: Sorting and searching in faulty memories. Algorithmica 52(3), 309–332 (2008)
24. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: Proc. IEEE Symposium on Security and Privacy, pp. 154–165 (2003)
25. Hamdioui, S., Al-Ars, Z., van de Goor, J., Rodgers, M.: Dynamic faults in Random-Access-Memories: Concept, faults models and tests. Journal of Electronic Testing: Theory and Applications 19, 195–205 (2003)
26. Henzinger, M.R.: The Past, Present, and Future of Web Search Engines. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 3–3. Springer, Heidelberg (2004)
27. Henzinger, M.R.: Combinatorial algorithms for web search engines - three success stories. In: ACM-SIAM Symposium on Discrete Algorithms, SODA (2007) (invited talk)
28. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers 33, 518–528 (1984)
29. Indyk, P.: On word-level parallelism in fault-tolerant computing. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 193–204. Springer, Heidelberg (1996)
30. Jørgensen, A.G., Moruz, G., Mølhave, T.: Priority queues resilient to memory faults. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 127–138. Springer, Heidelberg (2007)

31. Kleitman, D.J., Meyer, A.R., Rivest, R.L., Spencer, J., Winklmann, K.: Coping with errors in binary search procedures. J. Comp. Syst. Sci. 20, 396–404 (1980)
32. Kopetz, H.: Mitigation of Transient Faults at the System Level – the TTA Approach. In: Proc. 2nd Workshop on System Effects of Logic Soft Errors (2006)
33. Lakshmanan, K.B., Ravikumar, B., Ganesan, K.: Coping with erroneous information while sorting. IEEE Trans. on Computers 40(9), 1081–1084 (1991)
34. Leighton, T., Ma, Y.: Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. SIAM J. on Comput. 29(1), 258–273 (1999)
35. Leighton, T., Ma, Y., Plaxton, C.G.: Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. J. Comp. Syst. Sci. 54, 265–304 (1997)
36. Lu, C., Reed, D.A.: Assessing fault sensitivity in MPI applications. In: Proc. 2004 ACM/IEEE Conf. on Supercomputing (SC 2004), vol. 37 (2004)
37. May, T.C., Woods, M.H.: Alpha-Particle-Induced Soft Errors In Dynamic Memories. IEEE Trans. Elect. Dev. 26(2) (1979)
38. Pelc, A.: Searching with known error probability. Theoretical Computer Science 63, 185–202 (1989)
39. Pelc, A.: Searching games with errors: Fifty years of coping with liars. Theoret. Comp. Sci. 270, 71–109 (2002)
40. Pinheiro, E., Weber, W., Barroso, L.A.: Failure trends in large disk drive populations. In: Proc. 5th USENIX Conference on File and Storage Technologies (2007)
41. Ravikumar, B.: A fault-tolerant merge sorting algorithm. In: Ibarra, O.H., Zhang, L. (eds.) COCOON 2002. LNCS, vol. 2387, pp. 440–447. Springer, Heidelberg (2002)
42. Reed, D.A., Lu, C., Mendes, C.L.: Reliability challenges in large systems. Future Gener. Comput. Syst. 22(3), 293–302 (2006)
43. Saha, G.K.: Software based fault tolerance: a survey. Ubiquity 7(25), 1 (2006)
44. Skorobogatov, S., Anderson, R.: Optical fault induction attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 2–12. Springer, Heidelberg (2003)
45. Tezzaron Semiconductor. Soft Errors in Electronic Memory – A White Paper (January 2004) (manuscript), http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf
46. van de Goor, A.J.: Testing semiconductor memories: Theory and practice. ComTex Publishing, Gouda (1998)
47. von Neumann, J.: Probabilistic logics and the synthesis of reliable organisms from unreliable components. In: Shannon, C., McCarty, J. (eds.) Automata Studies, pp. 43–98. Princeton University Press, Princeton (1956)
48. Xu, J., Chen, S., Kalbarczyk, Z., Iyer, R.K.: An experimental study of security vulnerabilities caused by errors. In: Proc. International Conference on Dependable Systems and Networks, pp. 421–430 (2001)
49. Yao, A.C., Yao, F.F.: On fault-tolerant networks for sorting. SIAM J. on Comput. 14, 120–128 (1985)

Computational Challenges with Cliques, Quasi-cliques and Clique Partitions in Graphs[⋆]

Panos M. Pardalos and Steffen Rebennack

Center for Applied Optimization
Department of Industrial & Systems Engineering
University of Florida, USA
{pardalos,steffen}@ufl.edu
http://www.ise.ufl.edu/pardalos/

Abstract. During the last decade, many problems in social, biological, and financial networks require finding cliques, or quasi-cliques. Cliques or clique partitions have also been used as clustering or classification tools in data sets represented by networks. These networks can be very large and often massive and therefore external (or semi-external) memory algorithms are needed. We discuss four applications where we identify computational challenges which are both of practical and theoretical interest.

1 Introduction

An undirected graph G is a pair (V, E) consisting of a nonempty, finite node set V, $|V| < \infty$, and a finite (possibly empty) edge set $E \subseteq V \times V$ of unordered pairs of distinct elements of V. Graphs without loops and multiple edges are so-called simple graphs. In a multigraph, we allow a graph to have multiple edges but loops are not allowed. As we mainly consider undirected simple graphs in this article we shall call them from now on just graphs and mention it otherwise explicitly. Two nodes $u, v \in V$ of graph $\mathrm{G} = (V, E)$ are adjacent if (u, v) is an edge of G; *i.e.*, $(u, v) \in E$. A graph is said to be complete if there is an edge between any to nodes. The complement $\overline{\mathrm{G}} := (V, \overline{E})$ of a graph $\mathrm{G} = (V, E)$ is the graph with the same node set as G and the complement edge set $\overline{E}$, containing only the edges which are not in E; *i.e.*, $\overline{E} := \{(i, j) \,|\, i, j \in V,\, i \neq j \wedge (i, j) \notin E\}$. A graph is connected if there is a path between any two nodes of the graph, otherwise it is disconnected. The connected components of a graph G are the connected nonempty inclusion-maximal subgraphs of G. The length of the longest path among all shortest paths between any two nodes in the graph is called the diameter. The density of a graph is defined as the ration $\frac{2|E|}{|V|^2 - |V|}$. Two graphs $\mathrm{G}_1 = (V_1, E_1)$ and $\mathrm{G}_2 = (V_2, E_2)$ are called isomorphic, if there is a bijection $\phi\colon V_1 \to V_2$ such that $(u, v) \in E_1 \Leftrightarrow (\phi(u), \phi(v)) \in E_2$. A common subgraph of two graphs G_1 and G_2 consists of subgraphs $\overline{\mathrm{G}}_1$ and $\overline{\mathrm{G}}_2$ of G_1 and G_2, respectively, such that $\overline{\mathrm{G}}_1$ is isomorphic to $\overline{\mathrm{G}}_2$. For a given node set $S \subseteq V$, $\mathrm{G}(S) := (V, E \cap S \times S)$ is

[⋆] This research is partially supported by DTRA and Air Force grants.

the subgraph induced by S. This shall be enough on general graph theoretical definitions for this article; more details can be found, for instance, in [1,2].

A set of nodes S is called a clique if the subgraph $\mathrm{G}(S)$ is complete. We distinguish between a maxim*al* clique which is not a proper subset of any other clique in G and a maxim*um* clique which is a clique of maximum cardinality; *i.e.*, the largest clique in graph G. A set of nodes S in a graph G is a stable set if any two nodes in S are not adjacent. A stable set is sometimes also called independent set, vertex packing, co-clique or anticlique. The definition of cliques can be generalized by the concept of quasi-cliques. A quasi-clique, or γ-clique, C_γ of graph $\mathrm{G} = (V, E)$ is a subset of V such that the induced subgraph $\mathrm{G}(C_\gamma)$ is connected and has at least

$$\left\lfloor \gamma \frac{q(q-1)}{2} \right\rfloor \tag{1}$$

edges; where $q := |C_\gamma|$ and $\gamma \in [0, 1]$. In the extreme case of $\gamma = 0$, $\mathrm{G}(C_\gamma)$ may have no edges and if $\gamma = 1$, then C_γ is a clique in G. A coloring of G is a partition of V into disjoint stable sets, while a clique covering is a partition into disjoint cliques. In the following, we call a clique covering a clique partition.

The maximum clique problem is to find a maximum clique in a given graph G. We denote the cardinality of a maximum clique in graph G by $\omega(\mathrm{G})$ which is also called the clique number. Analogously, the maximum stable set problem asks to find a stable set of maximum cardinality. The cardinality of such a stable set is denoted by $\alpha(\mathrm{G})$ and is called the stability number or stable set number. The coloring number or chromatic number, which is denoted by $\chi(\mathrm{G})$, is the smallest number of stable sets needed for a coloring of G. Similarly, the smallest number of cliques for a clique partition of G is called clique covering number and is abbreviated with $\overline{\chi}(\mathrm{G})$. In this article, we are interested in finding a γ-clique of maximum size for fixed density γ; there are several other optimization problems for quasi-cliques, such as, maximize γq, or fix q and maximize γ.

As a clique in G corresponds to a stable set in the complement graph $\overline{\mathrm{G}}$, we obtain the relation

$$\alpha(\mathrm{G}) = \omega(\overline{\mathrm{G}}). \tag{2}$$

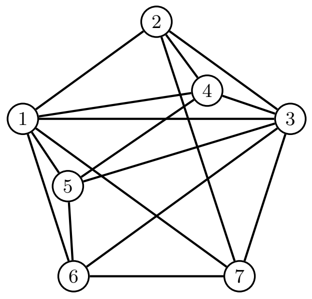
Furthermore, the following relations hold true

$$\chi(\mathrm{G}) = \overline{\chi}(\overline{\mathrm{G}}), \tag{3}$$

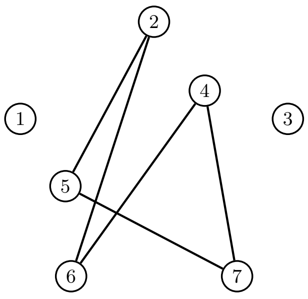
$$\omega(\mathrm{G}) \leq \chi(\mathrm{G}), \tag{4}$$

$$\alpha(\mathrm{G}) \leq \overline{\chi}(\mathrm{G}). \tag{5}$$

Since the number of stable sets needed to cover a graph is equal to the number needed to cover the complement with cliques, equality (3) is true. Hence, to find the coloring number or the clique covering number are algorithmically equivalent problems and we may discuss either of them depending on the application. To partition the node set of a graph G into disjoint stable sets, one needs at least



(a) Graph G



(b) Complement graph $\overline{\mathrm{G}}$

Fig. 1. A graph G and its complement $\overline{\mathrm{G}}$ with $\omega(\mathrm{G}) = \alpha(\overline{\mathrm{G}}) = 4$ and $\chi(\mathrm{G}) = \overline{\chi}(\overline{\mathrm{G}}) = 5$

Table 1. Maximum quasi-cliques for graph G corresponding to Figure 1 for different values of γ

γ	Maximum Cardinality	quasi-clique
1	4	{1,2,3,4}
$[13/15, 1)$	5	{1,2,3,4,5}
$[17/21, 13/15)$	6	{1,2,3,4,5,6}
$[0, 17/21)$	7	{1,2,3,4,5,6,7}

the size of a maximum clique in G. This is stated by inequality (4). Inequality (5) is the consequence of (2), (3), (4) and the observation that the complement of $\overline{\mathrm{G}}$ is again G.

Let us now have a closer look at the graph G and its complement $\overline{\mathrm{G}}$ in Figure 1. Node sets $\{1, 2, 3, 4\}$, $\{1, 2, 3, 7\}$, $\{1, 3, 4, 5\}$, $\{1, 3, 5, 6\}$ and $\{1, 3, 6, 7\}$ define all the maximum cliques in G (or maximum stable sets in $\overline{\mathrm{G}}$) leading to a clique number of 4 for G. The clique covering number for $\overline{\mathrm{G}}$ is 5 due to the nodes 2, 5, 7, 4 and 6 which form a so-called odd hole. This yields to the coloring number of 5 for the original graph G, showing that the coloring number and the clique number are not in general equal, confirming relation (4). A smaller example is given when the graph itself is an odd hole with 5 nodes. In this case, a maximum clique has size 2, but the coloring number is 3. Table 1 provides maximum quasi-cliques for different values of γ. For instance, if $\gamma < 17/21$, then the whole node set V defines a maximum quasi-clique C_γ. Increasing the γ value steadily reduces the maximum cardinality of a quasi-clique.

The maximum clique problem and the clique covering problem are one of Karp's original 21 problems shown to be NP-complete [3,4]; *i.e.*, unless P = NP, exact algorithms are guaranteed to return a solution only in a time which increases exponentially with the number of nodes in the graph. Furthermore, Arora and Safra [5] proved that for some $\varepsilon > 0$ the approximation of the clique number within a factor of $|V|^{\varepsilon}$ is NP-hard. A similar result was proven by Lund and Yanakakis [6] for the chromatic number. Computing a maximum quasi-clique for an arbitrary γ is also NP-complete, as for $\gamma = 1$ the problem is equivalent to the maximum clique problem. There is a large variety of exact and heuristic algorithms available for the maximum clique problem [7,8,9] and the clique partition problem [10]. Some recent work on quasi-clique algorithms can be found in [11].

In this article, we focus on computational challenges related to cliques, quasi-cliques as well as clique partitions arising from four applications: Call graphs (Section 2), coding theory (Section 3), matching molecular structures (Section 4), and Keller's conjecture (Section 5).

2 Call Graphs

Phone companies are faced with enormous data sets; *e.g.*, resulting from long distance phone-calls. In 1999, AT&T had approximately 300 million phone calls per day leading to a yearly storage space of 20 terabytes [12]. However, the analysis of such data is of great importance for the companies to study customer patterns and to be able to optimize their operations.

Given the data for phone calls over a specific time period (*e.g.*, ranging from days to months), one can construct a so-called call graph as follows. Each mobile user represents one node of the graph and there is a directed edge for each phone call. Hence, the resulting graph is a directed multigraph as one user may call the same user multiple times. Of interest in these graphs are especially undirected quasi-cliques as they provide information about highly interconnected users [13,14].

Graphs having millions of nodes are often referred to as massive graphs. Even the visualization of such graphs on a screen or basic statistical analysis are challenging tasks [15]. As the graphs are very large, they typically do not fit into the RAM of the computer or even into the main memory – hence, so-called external memory algorithms have been developed.

The call graphs tend to have specific properties. The most important ones are [16,17]

- the graphs are very large; *i.e.*, they have millions of nodes;
- the graphs have a very low density; *i.e.*, in the order of 0.0000001;
- the graphs are often disconnected, though connected components can be very large; *i.e.*, they may have millions of nodes;
- the undirected diameter of the graphs is low; *i.e.*, in the order of $\log(n)$;
- the node indegree d_{in} and the node outdegree d_{out} distributions follow a power-law; *i.e.*, $P(d_{in}) \sim d_{in}^{-\gamma_{in}}$ with $\gamma_{in} \in [2,3]$ and $P(d_{out}) \sim d_{out}^{-\gamma_{out}}$ with $\gamma_{out} < 2$ where $P(d)$ equals the number of nodes having degree d divided by the total number of nodes in the graph.

Abello et al. [16] studied a call graph corresponding to 1-day landline phone calls at AT&T and derived a call graph with 53 million nodes and 170 million edges.

To exploit the special structure mentioned above, the authors made extensive use of preprocessing. The algorithmic analysis of such graphs is practically very important. However, the known algorithms do not scale well on such graphs. This leads us to our first challenge.

Challenge 1 (Algorithm for Massive Graphs with Very Low Density) *Design an efficient algorithm together with a data base for the maximum γ-clique problem tailored to massive graphs characterized by very low density and by the node degree distribution following a power-law. Real world call graphs serve as an excellent test bed.*

Graphs with similar properties as the call graphs are the internet graphs, mobile graphs, graphs from social networks, SMS graphs, or www graphs [17,18,19,20]. Sometimes scale-free properties can be observed in these graphs due to self-organizing processes making them so-called small-world networks [21]. Biological networks lead to similar challenging problems with graphs [22].

3 Graphs in Coding Theory

A fundamental problem of interest is to send a message across a noisy channel with a maximum possible reliability. In coding theory, one wishes to find a binary code as large as possible that can correct a certain number of errors for a given size of the binary words.

For a binary vector $u \in \{0,1\}^n$, representing the words, denote by $F_e(u)$ the set of all vectors which can be obtained from u resulting from a certain error e, such as deletion or transposition of bits. Note that the elements in $F_e(u)$ do not necessarily have to have length n; *e.g.*, due to the deletion of digits. A subset $C \subseteq \{0,1\}^n$ is said to be an e-correcting code if $F_e(u) \cap F_e(v) = \emptyset$ for all $u, v \in C$ with $u \neq v$. The problem of interest is to find the largest correcting codes; *i.e.*, a set C of maximum size. For this to be meaningful, one has to have an idea about the nature of the error e. One may distinguish single deletion errors, two-deletion errors, single transposition including or excluding the end-around transposition errors, or one error on the Z-channel [23].

Consider a graph G having a node for every vector $u \in \{0,1\}^n$ and having an edge (u, v), if $F_e(u) \cap F_e(v) \neq \emptyset$. This way, an edge represents a conflict for an e-correcting code. Such graphs are called conflict graphs. Due to the construction of the graph, a correcting code corresponds to a stable set in G. Therefore, a largest e-correcting code can be found by solving the maximum independent set problem in the considered graph G.

Good lower bounds on the code size are especially interesting for asymmetric codes, such as codes correcting one error on the Z-channel (non zero components of any vector may change from 1 to 0). For that, several partition methods have been proposed in the literature, using minimum stable set partitions on conflict graphs. For the details of these methods, we refer the interested reader to [24,23]. The challenge for finding good lower bounds for code sizes is ongoing and tailored stable set partition algorithms are needed.

Challenge 2 (Algorithm for Conflict Graphs in Coding Theory). *Design an efficient algorithm for the minimum stable set partition problem tailored to conflict graphs resulting from applications in coding theory.*

Another example where minimum clique partitions are used as lower bounds are mandatory coverage problems, where a set of demand points has to be covered by a set of potential sites. Examples are ambulance location problems [25] or tiling problems [26], which have both to be solved in real-time, *i.e.*, within 2 minutes. For the latter problem, real-world instances resulting from cytology applications can have the size of the magnitude of tens of thousands of nodes.

4 Matching Molecular Structures

In the pharmaceutical and agrochemical industry, the problem of establishing structural relationships between pairs of three-dimensional molecular structures is an important problem. These three-dimensional molecular structures can be represented using graphs. For a protein, for instance, the nodes of the graph are given by the α-helix and β-strand secondary structure elements, whereas the edges are defined through inter-secondary structure element angles and distances [27]. In addition, both the nodes and the edges have labels, corresponding to the atomic types and the interatomic distances, respectively [28].

Consider a pair of node and edge labeled graphs $\mathrm{G}_1 = (V_1, E_1)$ and $\mathrm{G}_2 = (V_2, E_2)$. Then, one can construct the correspondence graph C as follows. Whenever two nodes $v_1 \in V_1$ and $v_2 \in V_2$ have the same label, there is a node in graph C. Hence, the nodes in C are pairs v_1v_2 and two nodes v_1v_2 and $v_1'v_2'$ are connected in C if the labels of the edges $(v_1, v_1') \in E_1$ and $(v_2, v_2') \in E_2$ are the same.

For a pair of three-dimensional chemical molecules, a maximum common subgraph in their corresponding graphs relates to the largest set of atoms that have matching distances between atoms. Hence, a maximum common subgraph is an obvious measure of structural similarity and gives important information about the two molecules.

Due to the construction of the correspondence graph C for two graphs corresponding to a pair of such molecules, it is almost obvious that the maximum common subgraphs in G_1 and G_2 correspond to cliques in their correspondence graph C. Therefore, one can find the maximum common subgraph of two arbitrary graphs by finding a maximum clique in their correspondence graph.

These correspondence graphs are characterized by their low edge densities – typically between 0.005 and 0.2 [29]. As suggested in the review by Raymond and Willett [30], the maximum clique algorithms currently available are computationally too expensive for these applications. This leads us to the following challenge.

Challenge 3 (Algorithm for Correspondence Graphs with Low Density). *Design an efficient algorithm for the maximum clique problem tailored to correspondence graphs resulting from matching of three-dimensional chemical molecules.*

Similar challenges occur when computing maximum cliques for the protein docking problem [31], where one wants to find out whether two proteins form a stable complex or not, or for comparing protein structures [32].

5 Keller's Conjecture

Keller's conjecture [33] goes back to Minkowski's conjecture [34] which stated that in a lattice tiling of $\mathbb{R}^n$ by translates of a unit hypercube, there exists two cubes that share $(n-1)$ dimensional face [35]. Minkowski's theorem was proven by Hajós [36] in 1950. Keller suggested that Minkowski's theorem can be generalized as the lattice assumption might not be necessary. Indeed, Perron [37] showed that this is true for $n \leq 6$. However, for $n \geq 8$ the lattice assumption is necessary, which was shown by Lagarias and Shor [38] and Mackey [39]. The Keller conjecture remains open for $n = 7$.

After 80 years, the rally towards the Keller's conjecture has not been ended. Almost as a by-product, a very interesting class of graphs for the maximum clique problem has been derived. For any given $n \in \mathbb{N}^+$, Corrádi and Szabó [40] constructed the so-called Keller graph Γ_n. The nodes of Γ_n are vectors of length n with values of $0, 1, 2$ or 3. Any two vectors are adjacent, if and only if in some of the n coordinates, they differ by precisely two (in absolute value). The Keller graph Γ_n is a dense graphs where the clique size is bounded by 2^n, [41]. Corrádi and Szabó [40] proved that there is an counterexample to Keller's conjecture, if and only if Γ_n has a clique of size 2^n.

With the help of these results, Lagarias and Shor [38] used a block substitution method to construct an appropriate clique, providing a counterexample for 10 dimensions. In a similar way, Mackey [39] constructed such a clique for dimension 8, proving that the Keller's conjecture does not hold true for $n \geq 8$. However, as the case of $n = 7$ remains open, we get the following challenge.

Challenge 4 (Open problem [40]). *For the Keller graph Γ_7, either find a maximum (cardinality) clique of size* 128 *or prove that none such clique exists.*

Hasselberg et al. [41] contributed several test case generators to the DIMACS challenge on cliques, among them are the Keller graphs. It turned out that the Keller graphs lead to challenging maximum clique problems and, to our best knowledge, no computational algorithms could solve the problem for $n \geq 6$. Especially as the maximum clique size for the Keller graphs are known (except for the cases of $n = 6, 7$), the Keller graphs are very useful graphs when benchmarking maximum clique algorithms with large graphs having high density. This leads us to the next challenge.

Challenge 5 (Algorithm for Dense Graphs). *Design an efficient algorithm for the maximum clique problem which is tailored to dense graphs. The Keller graphs should be used to benchmark its performance while the goal should be that the algorithm computes optimal solutions for the graphs with $2 \leq n \leq 8$.*

Table 2 summarizes the Keller graphs and their clique numbers. For the case of $n \geq 6$, we also know a lower bound on the clique number, given by $\omega(\Gamma_n) \geq \frac{57}{64}2^n$.

Table 2. Keller graphs and their clique numbers

n	# nodes	# edges	$\omega(\Gamma_n)$	reference
2	16	40	2	[40]
3	64	1,088	5	[40]
4	256	21,888	12	[40]
5	1,024	397,312	28	[40]
6	4,096	6,883,328	–	
7	16,384	116,244,480	–	
≥ 8	4^n	$\frac{1}{2}4^n(4^n - 3^n - n)$	2^n	[39,38,41]

– **open problem**

6 Conclusions

We have seen four applications leading to large graphs for clique, quasi-clique and clique partition problems. However, these graphs have different structural properties. Most significantly, the size of the graphs and their density vary greatly. Tailored algorithms to each of these problems are required to be able to solve these problems efficiently.

References

1. Diestel, R.: Graph Theory. Electronic Edition 2000. Springer, New York (2000)
2. West, D.B.: Introduction to Graph Theory, 2nd edn. Prentice-Hall, Englewood Cliffs (2000)
3. Karp, R.: Reducibility Among Combinatorial Problems. In: Miller, R.E., Thatcher, J. (eds.) Proceedings of a Symposium on the Complexity of Computer Computations, pp. 85–103. Plenum Press, New York (1972)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability, A guide to the Theory of NP-Completeness. In: Klee, V. (ed.) A series of books in the mathematical sciences. W. H. Freeman and Company, New York (1979)
5. Arora, S., Safra, S.: Probabilistic Checking of Proofs; a new Characterization of NP. In: Proceedings 33rd IEEE Symposium on Foundations of Computer Science, pp. 2–13. IEEE Computer Society, Los Angeles (1992)
6. Lund, C., Yannakakis, M.: On the hardness of approximating minimization problmes. JACM 41, 960–981 (1994)
7. Bomze, I., Budinich, M., Pardalos, P., Pelillo, M.: The maximum clique problem. In: Du, D.Z., Pardalos, P. (eds.) Handbook of Combinatorial Optimization, pp. 1–74. Kluwer Academic Publishers, Dordrecht (1999)
8. Rebennack, S.: Stable Set Problem: Branch & Cut Algorithms. In: Floudas, C.A., Pardalos, P.M. (eds.) Encyclopedia of Optimization, 2nd edn., pp. 3676–3688. Springer, Heidelberg (2008)
9. Rebennack, S., Oswald, M., Theis, D., Seitz, H., Reinelt, G., Pardalos, P.: A Branch and Cut solver for the maximum stable set problem. Journal of Combinatorial Optimization, doi:10.1007/s10878-009-9264-3

10. Pardalos, P., Mavridou, T., Xue, J.: The graph coloring problem: a bibliographic survey. In: Du, D.Z., Pardalos, P. (eds.) Handbook of Combinatorial Optimization, vol. 2, pp. 331–395. Kluwer Academic Publishers, Dordrecht (1990)
11. Brunato, M., Hoos, H., Battiti, R.: On Effectively Finding Maximal Quasi-cliques in Graphs. In: Maniezzo, V., Battiti, R., Watson, J.-P. (eds.) LION 2007 II. LNCS, vol. 5313, pp. 41–55. Springer, Heidelberg (2008)
12. Hayes, B.: Graph Theory in Practice: Part I. American Scientist 88(1), 9 (2000)
13. Cipra, B.: Massive graphs pose big problems. Technical report, SIAM NEWS, April 22 (1999)
14. Abello, J., Resende, M., Sudarsky, S.: Massive Quasi-Clique Detection. In: Rajsbaum, S. (ed.) LATIN 2002. LNCS, vol. 2286, p. 598. Springer, Heidelberg (2002)
15. Ye, Q., Wu, B., Suo, L., Zhu, T., Han, C., Wang, B.: TeleComVis: Exploring Temporal Communities in Telecom Networks. In: Buntine, W., Grobelnik, M., Mladenić, D., Shawe-Taylor, J. (eds.) ECML PKDD 2009. LNCS, vol. 5782, pp. 755–758. Springer, Heidelberg (2009)
16. Abello, J., Pardalos, P., Resende, M.: On Maximum Clique Problems in Very Lagre Graphs. In: External Memory Algorithms. DIMACS Series, pp. 119–130. American Mathematical Society, Providence (1999)
17. Nanavati, A., Singh, R., Chakraborty, D., Dasgupta, K., Mukherjea, S., Das, G., Gurumurthy, S., Joshi, A.: Analyzing the Structure and Evolution of Massive Telecom Graphs. IEEE Transactions on Knowledge and Data Engineering 20(5), 703–718 (2008)
18. Narasimhamurthy, A., Greene, D., Hurley, N., Cunningham, P.: Community Finding in Large Social Networks Through Problem Decomposition. Technical report, UCD School of Computer Science and Informatics (2008)
19. Faloutsos, M., Faloutsos, P., Faloutsos, C.: On Power-law Relationships of the Internet Topology. In: Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 251–262 (1999)
20. Hayes, B.: Connecting the Dots: Can the tools of graph theory and social-network studies unravel the next big plot? American Scientist 94(5), 400 (2006)
21. Schintler, L., Gorman, S., Reggiani, A., Patuelli, R., Nijkamp, P.: Small-World Phenomena in Communications Networks: A Cross-Atlantic Comparison. Advances in Spatial Science. In: Methods and Models in Transport and Telecommunications, pp. 201–220. Springer, Heidelberg (2005)
22. Butenko, S., Chaovalitwongse, W., Pardalos, P. (eds.): Clustering Challenges in Biological Networks. World Scientific, Singapore (2009)
23. Butenko, S., Pardalos, P., Sergieko, I., Shylo, V., Stetsyuk, P.: Estimating the size of correcting codes using extremal graph problems. In: Optimization: Structure and Applications. Springer Optimization and Its Applications, vol. 32, pp. 227–243. Springer, Heidelberg (2009)
24. van Pul, C., Etzion, T.: New lower bounds for constatn weight codes. IEEE Trans. Inform. Theory 35, 1324–1329 (1989)
25. Gendreau, M., Laporte, G., Semet, F.: Solving an ambulance location model by tabu search. Location Science 5, 75–88 (1997)
26. Brotcorne, L., Laporte, G., Semet, F.: Fast heuristics for large scale covering ocation problems. Computers and Operations Research 29, 651–665 (2002)
27. Mitchell, E., Artymiuk, P., Rice, D., Willett, P.: Use of techniques derived from graph theory to compare secondary structure motifs in proteins. J. Mol. Biol. 212, 151 (1990)

28. Brint, A., Willett, P.: Algorithms for the Identification of Three-Dimensional Maximal Common Substructures. J. Chem. ZnJ Comput. Sci. 27, 152–158 (1987)
29. Gardiner, E., Artymiuk, P., Willett, P.: Clique-detection algorithms for matching three-dimensional molecular structures. Journal of Molecular Graphics and Modelling 15, 245–253 (1997)
30. Raymond, J., Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. Journal of Computer-Aided Molecular Design 16, 521–533 (2002)
31. Gardiner, E., Willett, P., Artymiuk, P.: Graph-theoretic techniques for macromolecular docking. J. Chem. Inf. Comput. 40, 273–279 (2000)
32. Butenko, S., Wilhelm, W.: Clique-detection models in computational biochemistry and genomics. Euorpean Journal of Operational Research 173, 1–17 (2006)
33. Keller, O.: Über die lückenlose Einfüllung des Raumes mit Würfeln. J. Reine Angew. Math. 163, 231–248 (1930)
34. Minkowski, H.: Diophantische Approximationen. Teubner, Leipzig
35. Stein, S., Szabó, S.: Algebra and Tiling: Homomorphisms in the Service of Geometry. The Carus Mathematical Monographs, vol. 25. The Mathematical Associtaion of America (1994)
36. Hajós, G.: Sur la factorisation des abeliens. Casopis 50, 189–196
37. Perron, O.: Über lückenlose Ausfüllung des n-dimensioanlen Raumes durch kongruente Würfel. Math. Z. 46, 161–180 (1940)
38. Lagarias, J., Shor, P.: Keller's Cube-Tiling Conjecture is False in High Dimensions. Bulletin AMS 27, 279–283 (1992)
39. Mackey, J.: A Cube Tiling of Dimension Eight with No Facesharing. Discrete Comput. Geom. 28, 275–279 (2002)
40. Corrádi, K., Szabó, S.: A Combinatorial Approach for Keller's Conjecture. Periodica Math. Hung. 21(2), 95–100 (1990)
41. Hasselberg, J., Pardalos, P., Vairaktarakis, G.: Test Case Generators and Computational Results for the Maximum Clique Problem. Journal of Global Optimization 3, 463–482 (1993)

Alternative Routes 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. We study the problem of finding good alternative routes in road networks. We look for routes that are substantially different from the shortest path, have small stretch, and are locally optimal. We formally define the problem of finding alternative routes with a single via vertex, develop efficient algorithms for it, and evaluate them experimentally. Our algorithms are efficient enough for practical use and compare favorably with previous methods in both speed and solution quality.

1 Introduction

We use web-based and autonomous navigation systems in everyday life. These systems produce driving directions by computing a shortest path (or an approximation) with respect to a length function based on various measures, such as distance and travel time. However, optimal paths do not necessarily match the personal preferences of individual users. These preferences may be based on better local knowledge, a bias for or against a certain route segment, or other factors. One can deal with this issue by presenting a small number of alternative paths and hoping one of them will satisfy the user. The goal of an algorithm is to offer alternative paths often, and for these alternatives to look reasonable.

Recent research on route planning focused on computing only a single (shortest) path between two given vertices (see [4] for an overview). Much less work has been done on finding multiple routes. Some commercial products (by companies such as Google and TomTom) suggest alternative routes using proprietary algorithms. Among published results, a natural approach is to use k-shortest path algorithms [8], but this is impractical because a reasonable alternative in a road network is probably not among the first few thousand paths. Another approach is to use multi-criteria optimization [13,14], in which two or more length functions are optimized at once, and several combinations are returned. Efficient algorithms are presented in [5,9]. Our focus is on computing reasonable alternatives with a single cost function. In this context, the best published results we are aware of are produced by the *choice routing* algorithm [2], which we discuss in Section 4. Although it produces reasonable paths, it is not fully documented and is not fast enough for continental-sized networks.

In this work, we study the problem of finding "reasonable" alternative routes in road networks. We start by defining in Section 3 a natural class of *admissible* alternative paths. Obviously, an alternative route must be substantially different from the optimal path, and must not be much longer. But this is not enough: alternatives must feel natural to the user, with no unnecessary detours (formally, they must be *locally optimal*: every subpath up to a certain length must be a shortest path). Even with these restrictions, the number of admissible paths may be exponential, and computing the best one is hard. For efficiency, we focus on a more limited (yet useful) subset. Given an origin s and a target t, we restrict ourselves to *single via paths*, which are alternative routes obtained by concatenating the shortest paths from s to v and from v to t, for some vertex v. Section 4 discusses how the best such path can be computed in polynomial time using the bidirectional version of Dijkstra's algorithm (BD).

In practice, however, just polynomial time is not enough—we need sublinear algorithms. Modern algorithms for computing (optimal) shortest paths in road networks, such as those based on contraction hierarchies [10] and reach [11], are often based on pruning Dijkstra's search. After practical preprocessing steps, they need to visit just a few hundred vertices to answer queries on continental-sized graphs with tens of millions of vertices—orders of magnitude faster than BD. In fact, as shown in [1], their performance is sublinear on graphs with small highway dimension, such as road networks. Section 5 shows how to apply these speedup techniques to the problem of finding alternative routes, and Section 6 proposes additional measures to make the resulting algorithms truly practical.

Finally, Section 7 evaluates various algorithms experimentally according to several metrics, including path quality and running times. We show that finding a good alternative path takes only five times as much as computing the shortest path (with a pruning algorithm). Moreover, our pruning methods have similar success rates to a variant of choice routing, but are orders of magnitude faster.

Summarizing, our contributions are twofold. First, we establish a rigorous theoretical foundation for alternative paths, laying the ground for a systematic study of the problem. Second, we present efficient algorithms (in theory and in practice) for finding such routes.

2 Definitions and Background

Let $G = (V, E)$ be a directed graph with nonnegative, integral weights on edges, with $|V| = n$ and $|E| = m$. Given any path P in G, let $|P|$ be its number of edges and $\ell(P)$ be the sum of the lengths of its edges. By extension $\ell(P \cap Q)$ is the sum of the lengths of the edges shared by paths P and Q, and $\ell(P \setminus Q)$ is $\ell(P) - \ell(P \cap Q)$. Given two vertices, s and t, the *point-to-point shortest path problem* (P2P) is that of finding the shortest path (denoted by *Opt*) from s to t. Dijkstra's algorithm [7] computes $\text{dist}(s, t)$ (the distance from s to t in G) by scanning vertices in increasing order from s. Bidirectional Dijkstra (BD) runs a second search from t as well, and stops when both search spaces meet [3].

The *reach* of v, denoted by $r(v)$, is defined as the maximum, over all shortest u–w paths containing v, of $\min\{\text{dist}(u, v), dist(v, w)\}$. BD can be pruned at all vertices v for which both $\text{dist}(s, v) > r(v)$ and $\text{dist}(v, t) > r(v)$ hold [12]. The insertion of *shortcuts* (edges representing shortest paths in the original graph) may decrease the reach of some original vertices, thus significantly improving the efficiency of this approach [11]. The resulting algorithm (called RE) is three orders of magnitude faster than plain BD on continental-sized road networks.

An even more efficient algorithm (by another order of magnitude) is *contraction hierarchies* (CH) [10]. During preprocessing, it sorts all vertices by importance (heuristically), then shortcuts them in this order. (To *shortcut* a vertex, we remove it from the graph and add as few new edges as necessary to preserve distances.) A query only follows an edge (u, v) if v is more important than u.

3 Admissible Alternative Paths

In this paper, we are interested in finding an alternative path P between s and t. By definition, such a path must be significantly different from *Opt*: the total length of the edges they share must be a small fraction of $\ell(Opt)$.

This is not enough, however. The path must also be *reasonable*, with no unnecessary detours. While driving along it, every local decision must make sense. To formalize this notion, we require paths to be *locally optimal.* A first condition for a path P to be T *locally optimal* (T-LO) is that every subpath P' of P with $\ell(P') \le T$ must be a shortest path. This would be enough if P were continuous, but for actual (discrete) paths in graphs we must "round up" with a second condition. If P' is a subpath of P with $\ell(P') > T$ and $\ell(P'') < T$ (P'' is the path obtained by removing the endpoints of P'), then P' must be a shortest path. Note that a path that is not locally optimal includes a local detour, which in general is not desirable. (Users who need a detour could specify it separately.)

Although local optimality is necessary for a path to be reasonable, it is arguably not sufficient (see Figure 1). We also require alternative paths to have limited stretch. We say that a path P has $(1+\epsilon)$ *uniformly bounded stretch* ($(1+\epsilon)$-UBS) if every subpath (including P itself) has stretch at most $(1+\epsilon)$.

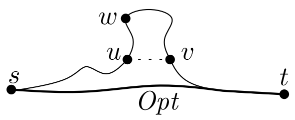


Fig. 1. Rationale for UBS. The alternative through w is a concatenation of two shortest paths, s–w and w–t. Although it has high local optimality, it looks unnatural because there is a much shorter path between u and v.

Given these definitions, we are now ready to define formally the class of paths we are looking for. We need three parameters: $0 < \alpha < 1$, $\epsilon \ge 0$, and $0 \le \gamma \le 1$. Given a shortest path *Opt* between s and t, we say that an s–t path P is an *admissible alternative* if it satisfies the following conditions:

1. $\ell(Opt \cap P) \le \gamma \cdot \ell(Opt)$ (limited sharing);
2. P is T-locally optimal for $T = \alpha \cdot \ell(Opt)$ (local optimality);
3. P is $(1+\epsilon)$-UBS (uniformly bounded stretch).

There may be zero, one, or multiple admissible alternatives, depending on the input and the choice of parameters. If there are multiple alternatives, we can sort them according to some objective function $f(\cdot)$, which may depend on any number of parameters (possibly including α, ϵ, and γ). In our experiments, we prefer admissible paths with low stretch, low sharing and high local optimality, as explained in Section 6. Other objective functions could be used as well.

Note that our definitions can be easily extended to report multiple alternative paths. We just have to ensure that the ith alternative is sufficiently different from the union of *Opt* and all $i-1$ previous alternatives. The stretch and local optimality conditions do not change, as they do not depend on other paths.

4 Single via Paths

Even with the restrictions we impose on admissible paths, they may still be too numerous, making it hard to find the best one efficiently. This section defines a subclass of admissible paths (called *single via paths*) that is more amenable to theoretical analysis and practical implementation. Given any vertex v, the *via path through* v, P_v, is the concatenation of two shortest paths, s–v and v–t (recall that we are looking for s–t paths). We look for via paths that are admissible. As we will see, these can be found efficiently and work well in practice.

Note that single via paths have interesting properties. Among all s–t paths through v (for any v), P_v is the shortest, i.e., it has the lowest stretch. Moreover, being a concatenation of two shortest paths, the local optimality of P_v can only be violated around v. In this sense, via paths are close to being admissible.

Although all $n-2$ via paths can be implicitly generated with a single run of BD (in $O(m + n \log n)$ time), not all of them must be admissible. For each via path P_v, we must check whether the three admissibility conditions are obeyed.

The easiest condition to check is sharing. Let $\sigma_f(v)$ be the sharing amount in the forward direction (i.e., how much s–v shares with *Opt*, which is known). Set $\sigma_f(s) \leftarrow 0$ and for each vertex v (in forward scanning order), set $\sigma_f(v)$ to $\sigma_f(p_f(v)) + \ell(p_f(v), v)$ if $v \in Opt$ or to $\sigma_f(p_f(v))$ otherwise (here p_f denotes the parent in the forward search). Computing $\sigma_r(v)$, the sharing in the reverse direction, is similar. The total sharing amount $\sigma(v) = \ell(Opt \cap P_v)$ is $\sigma_f(v) + \sigma_r(v)$. Note that this entire procedure takes $O(n)$ time.

In contrast, stretch and local optimality are much harder to evaluate, requiring quadratically many shortest path queries (on various pairs of vertices). Ideally, we would like to verify whether a path P is locally optimal (or is $(1+\epsilon)$-UBS) in time proportional to $|P|$ and a few shortest-path queries. We do not know how to do this. Instead, we present alternative tests that are efficient, have good approximation guarantees, and work well in practice.

For local optimality, there is a quick 2-approximation. Take a via path P_v and a parameter T. Let P_1 and P_2 be the s–v and v–t subpaths of P_v, respectively. Among all vertices in P_1 that are at least T away from v, let x be the closest to v (and let $x = s$ if $\ell(P_1) < T$). Let y be the analogous vertex in P_2 (and let $y = t$ if $\ell(P_2) < T$). We say that P_v *passes the* T*-test* if the portion of P_v between x and y is a shortest path. See Figure 2 for an example.

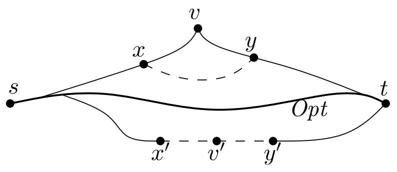


Fig. 2. Example for two T-tests. The T-test for v fails because the shortest path from u to w, indicated as a dashed spline, does not contain v. The test for v' succeeds because the shortest path from u' to w' contains v'.

Lemma 1. *If P_v passes the T-test, then P_v is T-LO.*

Proof. Suppose P_v passes the test and consider a subpath P' of P_v as in the definition of T-LO. If P' is a subpath of P_1 or P_2, then it is a shortest path. Otherwise P' contains v and is a subpath of the portion of P_v between x and y (as defined in the T-test), and therefore also a shortest path. □

This test is very efficient: it traverses P_v at most once and runs a single point-to-point shortest-path query. Although it may miss some admissible paths (a T-LO path may fail the T-test), it can be off by at most a factor of two:

Lemma 2. *If P_v fails the T-test, then P_v is not $2T$-LO.*

Proof. If P_v fails the test, then the x–y subpath in the definition of the test is not a shortest path. Delete x and y from the subpath, creating a new path P''. We know $\ell(P'') < 2T$ (v divides it into two subpaths of length less than T), which means P_v is not $2T$-LO. □

We now consider how to find the smallest ϵ for which a given path is $(1+\epsilon)$-UBS. We cannot find the exact value efficiently, but we can approximate it:

Lemma 3. *If a via path P_v has stretch $(1+\epsilon)$ and passes the T-test for $T = \beta \cdot \mathrm{dist}(s,t)$ (with $0 < \epsilon < \beta < 1$), then P_v is a $\frac{\beta}{\beta-\epsilon}$-UBS path.*

Proof. Consider a subpath P' of P_v between vertices u and w. If $v \notin P'$, or both u and w are within distance T of v, then P' is a shortest path (as a subpath of a shortest path). Assume v is between u and w and at least one of these vertices is at distance more than T from v. This implies $\ell(P') \geq T = \beta \cdot \mathrm{dist}(s,t)$. Furthermore, we know that $\ell(P') \leq \mathrm{dist}(u,w) + \epsilon \cdot \mathrm{dist}(s,t)$ (the absolute stretch of the subpath P' cannot be higher than in P_v). Combining these two inequalities, we get that $\ell(P') \leq \mathrm{dist}(u,w) + \epsilon \cdot \ell(P')/\beta$. Rearranging the terms, we get that $\ell(P') \leq \beta \cdot \mathrm{dist}(u,w)/(\beta - \epsilon)$, which completes the proof. □

BDV algorithm. We now consider a relatively fast BD-based algorithm, which we call BDV. It grows shortest path trees from s and into t; each search stops when it advances more that $(1+\epsilon)\ell(Opt)$ from its origin. (This is the longest an admissible path can be.) For each vertex v scanned in both directions, we check whether the corresponding path P_v is *approximately admissible*: it shares at most $\gamma \cdot \ell(Opt)$ with Opt, has limited stretch ($\ell(P_v) \leq (1+\epsilon)\ell(Opt)$), and passes the T-test for $T = \alpha \cdot \ell(Opt)$. Finally, we output the best approximately admissible via path according to the objective function.

The choice routing algorithm. A related method is the *choice routing algorithm* (CR) [2]. It starts by building shortest path trees from s and to t. It then looks at *plateaus*, i.e., maximal paths that appear in both trees simultaneously. In general, a plateau u–w gives a natural s–t path: follow the out-tree from s to u, then the plateau, then the in-tree from w to t. The algorithm selects paths corresponding to long plateaus, orders them according to some "goodness" criteria (not explained in [2]), and outputs the best one (or more, if desired). Because the paths found by CR have large plateaus, they have good local optimality:

Lemma 4. *If P corresponds to a plateau v–w, P is* $\text{dist}(v, w)$*-LO.*

Proof. If P is not a shortest path, then there are vertices x, y on P such that the length of P between these vertices exceeds $\text{dist}(x, y)$. Then x must strictly precede v on P, and y must strictly follow w. This implies the lemma. $\square$

Note that both CR and BDV are based on BD and only examine single-via paths. While BDV must run one point-to-point query to evaluate each candidate path, all plateaus can be detected in linear time. This means CR has the same complexity as BD (ignoring the time for goodness evaluation), which is much faster than BDV. It should be noted, however, that local optimality can be achieved even in the absence of long plateaus. One can easily construct examples where BDV succeeds and CR fails. Still, neither method is fast enough for continental-sized road networks.

5 Pruning

A natural approach to accelerate BD is to prune it at unimportant vertices (as done by RE or CH, for example). In this section, we show how known pruning algorithms can be extended to find admissible single-via paths. The results of [1] suggest that pruning is unlikely to discard promising via vertices. Because of local optimality, an admissible alternative path P contains a long shortest subpath P' that shares little with *Opt*. Being a shortest path, P' must contain an "important" (unpruned) vertex v.

For concreteness, we focus on an algorithm based on RE; we call it REV. Like BDV, REV builds two (now pruned) shortest paths trees, out of s and into t. We then evaluate each vertex v scanned by both searches as follows. First, we perform two P2P queries (s–v and v–t) to find P_v. (They are necessary because some original tree paths may be suboptimal due to pruning.) We then perform an approximate admissibility test on P_v, as in BDV. Among all candidate paths that pass, we return the one minimizing the objective function $f(\cdot)$.

The main advantage of replacing BD by RE is a significant reduction in the number of via vertices we consider. Moreover, auxiliary P2P queries (including T-tests) can also use RE, making REV potentially much faster than BDV.

An issue we must still deal with is computing the sharing amount $\sigma(v)$. RE adds shortcuts to the graph, each representing a path in the original graph. To calculate $\sigma(v)$ correctly, we must solve a *partial unpacking* subproblem: given a shortcut (a, c), with $a \in Opt$ and $c \notin Opt$, find the vertex $b \in Opt$ that belongs to the a–c path and is farthest from a. Assuming each shortcut bypasses exactly one vertex and the shortcut hierarchy is balanced (as is usually the case in practice), this can be done in $O(\log n)$ time with binary search.

Running time. We can use the results of [1] to analyze REV. Our algorithms and their implementations work for directed graphs, but to apply the results of [1] we assume the input network is undirected in the analysis.

Suppose we have a constant-degree network of diameter D and highway dimension h. The reach-based query algorithm scans $O(k \log D)$ vertices and runs in time $O((k \log D)^2)$, where k is either h or $h \log n$, depending on whether preprocessing must be polynomial or not. For each vertex v scanned in both directions, REV needs a constant number of P2P queries and partial unpackings. The total running time is therefore $O((k \log D)^3 + k \log D \log n)$, which is sublinear (unlike BDV or CR). The same algorithm (and analysis) can be applied if we use contraction hierarchies; we call the resulting algorithm CHV.

Relaxed reaches. Pruning may cause REV and CHV to miss candidate via vertices, leading to suboptimal solutions. In rare cases, they may not find any admissible path even when BDV would. For REV, we can fix this by trading off some efficiency. If we multiply the reach values by an appropriate constant, the algorithm is guaranteed to find all admissible single-via paths. The resulting algorithm is as effective as BDV, but much more efficient. It exploits the fact that vertices in the middle of locally optimal paths have high reach:

Lemma 5. *If P is T-LO and $v \in P$, then $r(v) \geq \min\{T/2, \text{dist}(s,v), \text{dist}(v,t)\}$.*

Proof. Let v be at least $T/2$ away from the endpoints of P. Let x and y be the closest vertices to v that are at least $T/2$ away from v towards s and t, respectively. Since P is T-LO, the subpath of P between x and y is a shortest path, and v has reach at least $T/2$. □

Corollary 1. *If P passes the T-test and $v \in P$, then $r(v) \geq \min\{T/4, \text{dist}(s,v), \text{dist}(v,t)\}$.*

Let δ-REV be a version of REV that uses original reach values multiplied by $\delta \geq 1$ to prune the original trees from s and to t (i.e., it uses $\delta \cdot r(v)$ instead of $r(v)$). Auxiliary P2P computations to build and test via paths can still use the original $r(v)$ values. The algorithm clearly remains correct, but may prune fewer vertices. The parameter δ gives a trade-off between efficiency and success rate:

Theorem 1. *If $\delta \geq 4(1+\epsilon)/\alpha$, δ-REV finds the same admissible via paths as* BDV.

Proof. Consider a via path P_v that passes the T-test for $T = \alpha \cdot \text{dist}(s,t)$. Then by Corollary 1 for every vertex $u \in P_v$, $r(u) \geq \min\{\text{dist}(s,v), \text{dist}(v,t), \alpha \cdot \text{dist}(s,t)/4\}$. When $\delta \geq 4(1+\epsilon)/\alpha$, then $\delta \cdot r(u) \geq \min\{\text{dist}(s,v), \text{dist}(v,t), (1+\epsilon)\text{dist}(s,t)\}$. Therefore no vertex on P_v is pruned, implying that $\text{dist}(s,v)$ and $\text{dist}(v,t)$ are computed correctly. As a result P_v is considered as an admissible via path. □

The analysis of [1] implies that multiplying reach values by a constant increases the query complexity by a constant multiplicative factor. Hence, the asymptotic time bounds for REV also apply to δ-REV, for any constant $\delta \geq 1$. Note that Theorem 1 assumes that δ-REV and BDV are applied on the same graph, with no shortcuts.

6 Practical Algorithms

The algorithms proposed so far produce a set of candidate via paths and explicitly check whether each is admissible. We introduced techniques to reduce the number of candidates and to check (approximate) admissibility faster, with a few point-to-point queries. Unfortunately, this is not enough in practice. Truly practical algorithms can afford at most a (very small) constant number of point-to-point queries *in total* to find an alternative path. Therefore, instead of actually evaluating all candidate paths, in our experiments we settle for finding one that is good enough. As we will see, we sort the candidate paths according to some objective function, test the vertices in this order, and return the first admissible path as our answer. We consider two versions of this algorithm, one using BD and the other (the truly practical one) a pruned shortest path algorithm (RE or CH). Although based on the methods we introduced in previous sections, the solutions they find do not have the same theoretical guarantees. In particular, they may not return the best (or any) via path. As Section 7 will show, however, the heuristics still have very high success rate in practice.

The practical algorithms sort the candidate paths P_v in nondecreasing order according to the function $f(v) = 2\ell(v) + \sigma(v) - pl(v)$, where $\ell(v)$ is the length of the via path P_v, $\sigma(v)$ is how much P_v shares with *Opt*, and $pl(v)$ is the length of the longest plateau containing v. Note that $pl(v)$ is a lower bound on the local optimality of P_v (by Lemma 4); by preferring vertices with high $pl(v)$, we tend to find an admissible path sooner.

We are now ready to describe X-BDV, an experimental version of BDV that incorporates elements of CR for efficiency. Although much slower than our pruning algorithms, this version is fast enough to run experiments on. It runs BD, with each search stopping when its radius is greater than $(1+\epsilon)\ell(Opt)$; we also prune any vertex u with $\mathrm{dist}(s,u) + \mathrm{dist}(u,t) > (1+\epsilon)\ell(Opt)$. In linear time, we compute $\ell(v)$, $\sigma(v)$, and $pl(v)$ for each vertex v visited by both searches. We use these values to (implicitly) sort the alternative paths P_v in nondecreasing order according to $f(v)$. We return the first path P_v in this order that satisfies three hard constraints: $\ell(P_v \setminus Opt) < (1+\epsilon)\ell(Opt \setminus P_v)$ (the detour is not much longer than the subpath it skips), $\sigma(v) < \gamma \cdot \ell(Opt)$ (sharing is limited), and $pl(v) > \alpha \cdot \ell(P_v \setminus Opt)$ (there is enough local optimality). Note that we specify local optimality relative to the detour only (and not the entire path, as in Section 3). Our rationale for doing so is as follows. In practice, alternatives sharing up to 80% with *Opt* may still make sense. In such cases, the T-test will always fail unless the (path-based) local optimality is set to 10% or less. This is too low for alternatives that share nothing with *Opt*. Using detour-based local optimality is a reasonable compromise. For consistency, ϵ is also used to bound the stretch of the detour (as opposed to the entire path).

The second algorithm we tested, X-REV, is similar to X-BDV but grows reach-pruned trees out of s and into t. The stopping criteria and the evaluation of each via vertex v are the same as in X-BDV. As explained in Section 4, RE trees may give only upper bounds on $\mathrm{dist}(s,v)$ and $\mathrm{dist}(v,t)$ for any vertex $v \notin Opt$. But we can still use the approximate values (given by the RE trees) of $\ell(v)$, $\sigma(v)$, and

$pl(v)$ to sort the candidate paths in nondecreasing order according to $f(v)$. We then evaluate each vertex v in this order as follows. We first compute the actual via path P_v with two RE queries, s–v and v–t (as an optimization, we reuse the original forward tree from s and the backward tree from t). Then we compute the exact values of $\ell(v)$ and $\sigma(v)$ and check whether $\ell(P_v \setminus Opt) < (1+\epsilon)\ell(Opt \setminus P_v)$, whether $\sigma(v) < \gamma \cdot \ell(Opt)$, and run a T-test with $T = \alpha \cdot \ell(P_v \setminus Opt)$. If P_v passes all three tests, we pick it as our alternative. Otherwise, we discard v as a candidate, penalize its descendants (in both search trees) and try the next vertex in the list. We penalize a descendant u of v in the forward (backward) search tree by increasing $f(u)$ by $\text{dist}(s, v)$ ($\text{dist}(v, t)$). This gives less priority to vertices that are likely to fail, keeping the number of check queries small.

A third implementation we tested was X-CHV, which is similar to X-REV but uses contraction hierarchies (rather than reaches) for pruning.

7 Experiments

We implemented the algorithms from Section 6 in C++ and compiled them with Microsoft Visual C++ 2008. Queries use a binary heap as priority queue. The evaluation was conducted on a dual AMD Opteron 250 running Windows 2003 Server. It is clocked at 2.4 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. Our code is single-threaded and runs on a single processor at a time.

We use the European road network, with 18 million vertices and 42 million edges, made available for the 9th DIMACS Implementation Challenge [6]. It uses travel times as the length function. (We also experimented with TIGER/USA data, but closer examination revealed that the data has errors, with missing arcs on several major highways and bridges. This makes the results less meaningful, so we do not include them.) We allow the detour to have maximum stretch $\epsilon = 25\%$, set the maximum sharing value to $\gamma = 80\%$, and set the minimum detour-based local optimality to $\alpha = 25\%$ (see Section 6).

X-REV and X-CHV extend the point-to-point query algorithms RE [11] and CH [10]. Since preprocessing does not change, we use the preprocessed data given in [10,11]. In particular, preprocessing takes 45 minutes for RE and 25 for CH.

We compare the algorithms in terms of both query performance and path quality. Performance is measured by the number of vertices scanned and by query times (given in absolute terms and as a multiple of the corresponding P2P method). Quality is given first by the *success rate*: how often the algorithm finds as many alternatives as desired. Among the successful runs, we also compute the average and worst uniformly-bounded stretch, sharing, and detour-based local optimality (reporting these values requires $O(|P|^2)$ point-to-point queries for each path P; this evaluation is not included in the query times). Unless otherwise stated, figures are based on 1 000 queries, with source s and target t chosen uniformly at random.

In our first experiment, reported in Table 1, we vary p, the desired number of alternatives to be computed by the algorithms. There is a clear trade-off between success rates and query times. As expected, X-BDV is successful more often, while

Table 1. Performance of various algorithms on the European road network as the number of desired alternatives (p) changes. Column *success rate* reports how often the algorithm achieves this goal. For the successful cases, we report the (average and worst-case) quality of the p-th alternative in terms of UBS, sharing, and detour-based local optimality. Finally, we report the average number of scanned vertices and query times (both in milliseconds and as a multiple of the corresponding point-to-point variant).

		PATH QUALITY							PERFORMANCE		
		success	UBS[%]		sharing[%]		locality[%]		#scanned	time	slow-
p	algo	rate[%]	avg	max	avg	max	avg	min	vertices	[ms]	down
1	X-BDV	94.5	9.4	35.8	47.2	79.9	73.1	30.3	16 963 507	26 352.0	6.0
	X-REV	91.3	9.9	41.8	46.9	79.9	71.8	30.7	16 111	20.4	5.6
	X-CHV	58.2	10.8	42.4	42.9	79.9	72.3	29.8	1 510	3.1	4.6
2	X-BDV	81.1	11.8	38.5	62.4	80.0	71.8	29.6	16 963 507	29 795.0	6.8
	X-REV	70.3	12.2	38.1	60.3	80.0	71.3	29.6	25 322	33.6	9.2
	X-CHV	28.6	10.8	45.4	55.3	79.6	77.6	30.3	1 685	3.6	5.3
3	X-BDV	61.6	13.2	41.2	68.9	80.0	68.7	30.6	16 963 507	33 443.0	7.7
	X-REV	43.0	12.8	41.2	66.6	80.0	74.9	33.3	30 736	42.6	11.7
	X-CHV	10.9	12.0	41.4	59.3	80.0	79.0	36.1	1 748	3.9	5.8

X-CHV is fastest. Unfortunately, X-BDV takes more than half a minute for each query and X-CHV finds an alternative in only 58.2% of the cases (the numbers are even worse with two or three alternatives). The reach-based algorithm, X-REV, seems to be a good compromise between these extremes. Queries are still fast enough to be practical, and it is almost as successful as X-BDV. In only 20.4 ms, it finds a good alternative in 91.3% of the cases.

Alternative paths, when found, tend to have similar quality, regardless of the algorithm. This may be because the number of admissible alternatives is small: the algorithms are much more successful at finding a single alternative than at finding three (see Table 1). On average, the first alternative has 10% stretch, is 72% locally optimal, and shares around 47% with the optimum. Depending on the success rate and p, the alternative query algorithm is 4 to 12 times slower than a simple P2P query with the same algorithm. This is acceptable, considering how much work is required to identify good alternatives.

As observed in Section 5, we can increase the success rate of REV by multiplying reach values by a constant $\delta > 1$. Our second experiment evaluates how this multiplier affects X-REV, the practical variant of REV. Table 2 reports the success rate and query times of δ-X-REV for several values of δ when $p = 1$. As predicted, higher reach bounds do improve the success rate, eventually matching that of X-BDV on average. The worst-case UBS is also reduced from almost 42% to around 30% when δ increases. Furthermore, most of the quality gains are already obtained with $\delta = 2$, when queries are still fast enough (less than 10 times slower than a comparable P2P query).

The original reach values used by X-REV are not exact: they are upper bounds computed by a particular preprocessing algorithm [11]. On a smaller graph (of the Netherlands, with $n \approx 0.9$M and $m \approx 2.2$M), we could actually afford to

Table 2. Performance of X-REV when varying the multiplier (δ) for reach values

		PATH QUALITY							PERFORMANCE		
		success	UBS[%]		sharing[%]		locality[%]		#scanned	time	slow-
algo	δ	rate[%]	avg	max	avg	max	avg	min	vertices	[ms]	down
X-REV	1	91.3	9.9	41.8	46.9	79.9	71.8	30.7	16 111	20.4	5.6
	2	94.2	9.7	31.6	46.6	79.9	71.3	27.6	31 263	34.3	9.4
	3	94.2	9.5	29.2	46.7	79.9	71.9	31.2	53 464	55.3	15.2
	4	94.3	9.5	29.3	46.7	79.9	71.8	31.2	80 593	83.2	22.8
	5	94.4	9.5	29.3	46.7	79.9	71.8	31.4	111 444	116.6	31.9
	10	94.6	9.5	30.2	46.8	79.9	71.7	31.4	289 965	344.3	94.3
X-BDV	–	94.5	9.4	35.8	47.2	79.9	73.1	30.3	16 963 507	26 352.0	6.0

compute exact reaches on the shortcut-enriched graph output by the standard RE preprocessing. On this graph, the success rate drops from 83.4% with the original upper bounds to 81.7% with exact reaches. Multiplying the exact reaches by $\delta = 2$ increases the success rate again to 83.6%. With $\delta = 5$, we get 84.7%, very close to the 84.9% obtained by X-BDV. Note that the Dutch subgraph tends to have fewer alternatives than Europe as a whole.

To examine this issue in detail, Figure 3 reports the success rate of our algorithms for $p = 1$ and various Dijkstra ranks on Europe. The *Dijkstra rank* of v with respect to s is i if v is the ith vertex taken from the priority queue when running a Dijkstra query from s. The results are based on 1 000 queries for each rank. We observe that the success rate is lower for local queries, as expected. Still, for mid-range queries we find an alternative in 60% to 80% of the cases, which seems reasonable. (Recall that an admissible alternative may not exist.) Multiplying reach values helps all types of queries: 2-X-REV has almost the same success rate as X-BDV for all ranks and number of alternatives examined.

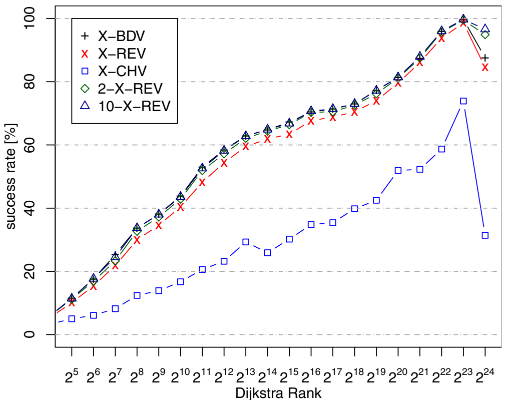


Fig. 3. Success rate for X-BDV, X-REV, 2-X-REV, 10-X-REV, and X-CHV when varying the Dijkstra rank of queries for $p = 1$

8 Conclusion

By introducing the notion of admissibility, we have given the first formal treatment to the problem of finding alternative paths. The natural concept of local optimality allows us to prove properties of such paths. Moreover, we have given theoretically efficient algorithms for an important subclass, that of single via paths. We concentrated on making the approach efficient for real-time applications by designing approximate admissibility tests and an optimization function biased towards admissible paths. Our experiments have shown that these simplified versions are practical for real, continental-sized road networks.

More generally, however, our techniques allow us to do optimization constrained to the (polynomially computable) set of admissible single via paths. We could optimize other functions over this set, such as fuel consumption, time in traffic, or tolls. This gives a (heuristic) alternative to multi-criteria optimization.

Our work leads to natural open questions. In particular, are there efficient exact tests for local optimality and uniformly bounded stretch? Furthermore, can one find admissible paths with multiple via vertices efficiently? This is especially interesting because it helps computing arbitrary admissible paths, since any admissible alternative with stretch $1+\epsilon$ and local optimality $\alpha \cdot \ell(Opt)$ is defined by at most $\lceil (1+\epsilon)/\alpha \rceil - 1$ via points.

References

1. Abraham, I., Fiat, A., Goldberg, A.V., Werneck, R.F.: Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In: SODA, pp. 782–793 (2010)
2. Cambridge Vehicle Information Technology Ltd. Choice Routing (2005), http://www.camvit.com
3. Dantzig, G.B.: Linear Programming and Extensions. Princeton Univ. Press, Princeton (1962)
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.: Pareto Paths with SHARC. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 125–136. Springer, Heidelberg (2009)
6. Demetrescu, C., Goldberg, A.V., Johnson, D.S. (eds.): 9th DIMACS Implementation Challenge - Shortest Paths (2006), http://www.dis.uniroma1.it/~challenge9/
7. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, 269–271 (1959)
8. Eppstein, D.: Finding the k shortest paths. In: Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1994), pp. 154–165 (1994)
9. Geisberger, R., Kobitzsch, M., Sanders, P.: Route Planning with Flexible Objective Functions. In: ALENEX, pp. 124–137. SIAM, Philadelphia (2010)
10. 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)
11. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Reach for A*: Shortest Path Algorithms with Preprocessing. In: Demetrescu, C., Goldberg, A.V., Johnson, D.S. (eds.) The Shortest Path Problem: Ninth DIMACS Implementation Challenge. DIMACS Book, vol. 74, pp. 93–139. American Mathematical Society, Providence (2009)
12. Gutman, R.J.: Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In: ALENEX, pp. 100–111. SIAM, Philadelphia (2004)
13. Hansen, P.: Bricriteria Path Problems. In: Fandel, G., Gal, T. (eds.) Multiple Criteria Decision Making: Theory and Application, pp. 109–127. Springer, Heidelberg (1979)
14. Martins, E.Q.: On a Multicriteria Shortest Path Problem. European Journal of Operational Research 26(3), 236–245 (1984)

Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks*

Annabell Berger, Martin Grimmer, and Matthias Müller-Hannemann

Department of Computer Science, Martin-Luther-University Halle-Wittenberg,
Von-Seckendorff-Platz 1, 06120 Halle, Germany
{berger,muellerh}@informatik.uni-halle.de,
martin.grimmer@particle-dev.com

Abstract. We introduce two new speed-up techniques for time-dependent point-to-point shortest path problems with fully-dynamic updates in a multi-criteria setting. Our first technique, called SUBITO, is based on a specific substructure property of time-dependent paths which can be lower bounded by their minimal possible travel time. It requires no preprocessing, and the bounds can be computed on-the-fly for each query. We also introduce k-flags, an extension of arc flags, which assigns to each arc one of k levels for each region of a vertex partition. Intuitively, the higher the level of an arc for a certain destination, the larger the detour with respect to travel time. k-flags allow us to handle dynamic changes without additional time-consuming preprocessing.

In an extensive computational study using the train network of Germany we analyze these and other speed-up techniques with respect to their robustness under high and realistic update rates. We show that speed-up factors are conserved under different scenarios, namely a typical day of operation, distributed delays after "heavy snowfall", and a major disruption at a single station. In our experiments, k-flags combined with SUBITO have led to the largest speed-up factors, but only marginally better than SUBITO alone. These observations can be explained by studying the distribution of k-flags. It turns out that only a small fraction of arcs can be excluded if one wants to guarantee exact Pareto-optimal point-to-point queries.

Keywords: Shortest paths; dynamic graphs; speed-up technique.

1 Introduction

Up to now, route planning is usually based on a static road map or a published schedule in public transport. Unfortunately, car traffic is affected by time-dependent speed profiles (like every day rush hours) and unforeseeable events,

* This work was partially supported by the DFG Focus Program Algorithm Engineering, grant Mu 1482/4-1. We wish to thank Deutsche Bahn AG for providing us timetable data for scientific use.

like traffic jams, and blocked roads because of construction work. Likewise, public transportation systems suffer from delays for various reasons. While speed profiles can be modelled quite well by time-dependent models, unforeseeable events like traffic jams and train delays require in addition a truly dynamic model to support on-trip routing.

The demands for an online tool are quite high: To give a concrete example, on a typical day of operation of German Railways, an online system has to handle about 6 million forecast messages about (mostly small) changes with respect to the planned schedule and the latest prediction of the current situation. Initial primary train delays induce a whole cascade of secondary delays of other trains which have to wait according to certain waiting policies between connecting trains. Thus, the graph model has to be updated at a rate of about 70 update operations per second [1]. From the traveler's point of view, a navigation or timetable information system should offer a choice of reasonable alternatives subject to optimization criteria like earliest arrival time, travel costs, sustainability, or traveler comfort. Hence, the task is to solve a multi-criteria optimization problem, i.e., to find all Pareto-optimal itineraries. Since central timetable information servers have to answer millions of queries per day, it is obvious that low query times are crucial. The algorithmic challenge is to design speed-up techniques which are provably correct and work in a multi-criteria fully-dynamic scenario.

In recent years, a multitude of powerful speed-up techniques have been developed for point-to-point shortest path queries in static transportation networks. These techniques work empirically extremely well for road networks, yielding query times of a few microseconds in continental-sized networks, for a recent survey see [2]. Quite recently, Abraham et al. [3] complemented these observations with the first rigorous proofs of efficiency for many of the speed-up techniques suggested over the past decade. They introduced the notion of highway dimension and showed that networks with small highway dimension have good query performance. In contrast, several empirical studies have shown that these speed-up techniques work less well for public transportation networks (with scheduled traffic for trains, buses, ferries, and airplanes) [4,5]. To understand this difference, it is helpful to consider time-expanded graph models where each arrival or departure event corresponds to a vertex. These graphs are highly non-planar and have relatively high highway dimension. A detailed explanation of further reasons why timetable information in public transport is more challenging than routing in road networks has been given by Bast [6].

Speed-up techniques in a dynamic context. The general challenge is to design a speed-up technique based on preprocessing with the following features:

1. Even after many dynamic graph changes, we do not require a further time-consuming preprocessing. Moreover, the speed-ups in a time-dependent scenario with and without dynamic updates should be comparable.
2. The method should allow us to find *all* Pareto-optimal paths within a reasonable range, not only one representative for each Pareto-optimal solution vector.

Related Work. The literature on point-to-point shortest path problems uses the term dynamic in several different ways, leading to problem variants which can be categorized as follows:

Problem version 1: ("static shortest path problem with cost updates") This model considers a static graph with cost updates. Static here refers to the arc costs which are constant between two consecutive updates. This version can be seen as a first step towards modelling traffic jams or blocked roads in road networks. However, the shortcoming of this model is that it does not capture the time dimension correctly. For example, if some bridge or road is blocked for half an hour, it may or may not have an influence for a routing request, depending on the point of time when one could reach this place. So at query time, the static graph includes only one of these two possibilities.

Problem version 2: ("time-dependent earliest arrival problem with dynamic updates") This model is built on a time-dependent graph, where each arc has a discrete travel-time function. The travel functions can be used to model speed profiles. These functions are dynamically updated at discrete time points. Updates are unplanned changes of the speed profile, for example due to a blocked bridge as above. This version is usually treated as a single-criterion optimization problem with the objective to find a route with earliest arrival. A standard approach is to look only for *greedy paths*. In road networks, this means that the traveler immediately decides at a junction which road to take next (he does not wait for better alternatives). In public transportation, greedy means that the passenger takes the very first possibility on each line. Unfortunately, optimal greedy paths can be sub-optimal in the set of all time-dependent paths in a dynamic scenario. In road networks, it can now be reasonable to wait before a blocked bridge for some minutes to get the faster path. Thus the problem is closer to public transport where waiting times have to be considered already in an undelayed scenario. These complications can be handled, but such queries require a larger search effort for non-greedy paths [7].

Problem version 3: ("fully-dynamic multi-criteria on-trip problem") This version comprises discrete event-dependent multi-criteria search for all shortest (s, t)-paths where one optimization criterion corresponds to the earliest arrival time. If an additional optimization criterion depends on other attributes than time, the problem becomes event-dependent (for example, on the specific train). Fully dynamic here means that the travel time functions can be varied freely, including arc deletions or temporarily blocked arcs.

Work on problem version 1 includes, for example, dynamic shortest path containers [8], dynamic highway node routing [9], landmark-based routing in dynamic graphs [10], dynamic arc flags [11], and contraction hierarchies [12].

Problem version 2 has found much less attention. Nannicini et al. [13] have chosen a slightly different approach to handle a realistic dynamic situation in road networks where traffic information data are updated each minute. They relax the problem of finding shortest paths to determining paths which are very close to an optimal solution. Their underlying graph model can be seen as time-dependent, the travel time functions are determined heuristically based on real-time traffic data. Unfortunately, they cannot give a guarantee for the optimality of a solution. Surprisingly, the proposed solution works well in practice, with 0.55% average deviation from the optimal solution and a recorded maximum deviation of 17.59%. In another paper, Nannicini et al. [14] proposed a two-phase polynomial time approximation scheme for the point to point shortest path problem. In summary, there is no exact method available which has been tested for problem version 2, and to the best of our knowledge, problem version 3 is first studied in this paper.

Our contribution. Our approach makes use of the following assumption which is valid in a railroad scenario: the travel time on each time-dependent arc can be lower-bounded by some positive value which we call the minimum-theoretical travel time. Above this lower bound value, the travel time function can be freely modified by dynamically changing delays. In particular, we allow the symbolic value $+\infty$ to model the unavailability of an arc. We only forbid dynamic updates which insert new arcs into our model. This is no strong restriction since new railroad tracks will not be provided all of a sudden, they will usually be known right in advance to be included into the preprocessing. Lower travel time bounds on arcs extend to lower bounds on time- and event-dependent paths. This leads us to the new notion of additive c-optimality for static and event-dependent paths. Its properties are useful to restrict the search space with respect to a concrete (s, t)-query. We introduce two new speed-up techniques which we call *SUBITO* and *k-flags*. The speed-up technique SUBITO reduces the number of events in a timetable and can be used on-the-fly without time-consuming preprocessing. k-flags allow us to restrict the search to a subgraph after a preprocessing phase. Both techniques can also be combined. For the first time, these techniques meet all mentioned goals for a dynamic context.

The proposed techniques have been experimentally evaluated on the German train networks with several realistic delay scenarios (a typical day of operation, distributed delays due to bad weather like a “snow chaos”, and a major disruption at a single station). We observe a strong positive correlation of the observed running time and the travel time. The best speed-ups are obtained for the most difficult queries, i.e., those with large travel times. We also show that speed-up factors are conserved under high update rates. In our experiments k-flags combined with SUBITO have led to the largest speed-up factors, but only marginally better than SUBITO alone. These observations can be explained by studying the distribution of k-flags. It turns out that only a small fraction of arcs can be excluded if one wants to guarantee exact Pareto-optimal point-to-point queries.

Overview. The remainder of the paper is structured as follows. In Section 2, we discuss the on-trip problem and briefly review our concept of time- and event-dependent paths and provide basic definitions. Our main contribution — the speed-up techniques SUBITO and k-flags — are introduced in Section 3. In Section 4, we give the results of our experiments testing our speed up techniques with real-time traffic data of German Railways.

2 The On-Trip Problem

2.1 Graph Models

A timetable $TT := (Z, S, C)$ consists of a tuple of sets. Let Z be the set of trains, S the set of stations and C the set of *elementary connections*, that is

$$C := \left\{ c = (z, s, s', t_d, t_a) \,\middle|\, \begin{array}{l} \text{Train } z \in Z \text{ leaves station } s \text{ at time } t_d. \\ \text{The next stop of } z \text{ is at station } s' \text{ at time } t_a. \end{array} \right\}.$$

A timetable TT is valid for a number of N traffic days. A timetable-valid function $v : Z \to \{0,1\}^N$ determines on which traffic days the train operates. We denote a time value t in TT by $t := a \cdot 1440 + b$ with $a \in [0, N], b \in [0, 1439]$. The actual time within a day is then $t_{day} = t \ mod\ 1440$ and the actual day $d = \lfloor \frac{t}{1440} \rfloor$. We define the *station graph* $G = (V, A)$ with $V := S$ and $A := \{(v, u) | \ v, u \in V, \ \exists c \in C(TT) \text{ from } v \text{ to } u\}$. For several reasons it is useful to construct an extended version of the station graph by splitting each arc of the station graph G into parallel arcs. We denote a sequence $(v_1, \ldots, v_l)$ of pairwise disjoint vertices $v_i \in V$ as a *route* r if there exists at least one train using exactly these stations v_i in the given order. This setting allows us to construct the so-called *station route graph* $G_r = (V, A_r)$ with $A_r := \{(v, u)_r \mid (v, u) \in A, r \text{ is route in TT using arc } (v, u)\}$.

Let $T \subset \mathbb{N}$ be a set of time points. With respect to the set of elementary connections C we define a set of departure events Dep_v and arrival events Arr_v for all $v \in V$. Each event $dep_v := (time, train, route) \in Dep_v$ and $arr_v := (time, train, route)$ represents exactly one departure or arrival event which consists of the three attributes *time*, *train* and *route*. We assign to each arc $a = (v, u) \in A_r$ and departure event dep_v at v an arrival event arr_u which defines the event that we reach vertex u if we depart in v with departure event dep_v and traverse arc a on route r. This setting represents a corresponding elementary connection $c \in C$. Hence, the event dep_v departs at time $dep_v(time) \in \mathbb{N}$ in v, travels on the arc $a = (v, u) \in A$ and arrives at the corresponding event arr_u at $arr_u(time) \in T$. Staying in any vertex can be limited to minimum and maximum staying times $minstay(arr_v, dep_v), maxstay(arr_v, dep_v) \in \mathbb{R}_+$ which have to be respected between different events in v. Staying times ensure the possibility to transfer from one train to the next. The following definitions are given with respect to an (s, t)-query for an earliest start time $start_s(time)$ at s. For an (s, t)-query we ignore all arrival events at s, but introduce an artificial "arrival event" $start_s$ with an earliest start time $start_s(time) \in \mathbb{N}$ which can be interpreted as an arrival time. Furthermore, we define one artificial "departure event" end_t which can be interpreted as a departure time in t.

Event-dependent vertices v_e are 2-tuples which consist of an arrival event $arr_v \in Arr_v$ and a departure event $dep_v \in Dep_v$ for a vertex $v \in V$, respecting minimum and maximum staying times $minstay, maxstay$ at v. *Event-dependent edges* a_e are tuples which consist of a departure event at $v \in V$ and the corresponding arrival event at $u \in V$ by traversing the edge $(v, u) \in A_r$. These notions have been introduced in [7]. We define a weighted discrete event-dependent graph

$\mathcal{G}(G_r, s, t, start_s) := (V_E, A_E, w)$ with respect to an (s,t)-query with an earliest start time $start_s(time)$ and underlying station route graph G_r. It is a tuple consisting of the set of event-dependent vertices V_E, the set of event-dependent edges A_E and a weight-function $w \in (\mathbb{R}^k)^{V_E \cup A_E}$. The weight-function w assigns to each event-dependent vertex v_E a k-dimensional vector $w(v_e) \in \mathbb{R}^k$ and to each event-dependent edge a_e a k-dimensional vector $w(a_e) \in \mathbb{R}^k$.

2.2 On-Trip Paths

Definition 1. *An* event-dependent path P_E *is an alternating sequence of event-dependent vertices and edges with event-dependent vertices on its ends. The stations of the event-dependent vertices in* P_E *are pairwise different. We call two event-dependent paths* comparable *if their first event-dependent vertices possess an identical arrival event and their last event-dependent vertices possess an identical departure event.*

We distinguish between two notations for event-dependent paths. If we focus on the origin v and the destination u of a path ("stations"), we call it *event-dependent* (v,u)*-path*. On the other hand if we focus on starting and ending events of a path we denote an event-dependent path which starts with the event-dependent vertex $v_T = (arr_v, dep_v)$ and ends with the event-dependent vertex $u_T = (arr_u, dep_u)$ as P_{arr_v, dep_u}. Note that a single event-dependent vertex $v_T = (arr_v, dep_v)$ is already an event-dependent path P_{arr_v, dep_v}. All event-dependent (s,t)-paths in our definition are comparable because they possess identical arrival events $arr_s := start_s$ and identical departure-events $dep_t := end_t$. Following Orda and Rom [15] we define a function *top* which assigns to each event-dependent path P_E its underlying sequence p of vertices $v \in V$. We call p *topological path of* P_E and denote it by $top(P_E) =: p$. Obviously, p is a simple path in the station-route-graph G_r, because P_E is vertex-disjoint due to Definition 1. For railway networks, we use the following weight-function $ontrip := (eatt, transfers) : V_E \cup A_E \to \mathbb{R}_+$ for all event-dependent vertices $v_e := (arr_v, dep_v) \in V_E$ and for all time-dependent arcs $a_e := (dep_v, arr_u) \in A_E$.

$$eatt(v_e) := \begin{cases} dep_v(time) - arr_v(time), & \text{for all } v \in V \setminus \{s,t\} \\ dep_s(time) - start_s(time), & \text{for } v = s \\ 0, & \text{for } v = t. \end{cases}$$

$$eatt(a_e) := \Big\{ arr_u(time) - dep_v(time), \quad \text{for all } v \in V.$$

$$transfers(v_e) := \begin{cases} 1 & \text{for all } v \in V \setminus \{s,t\} \text{ with } dep_v(train) \neq arr_v(train) \\ 0 & \text{otherwise.} \end{cases}$$

$$transfers(a_e) := 0$$

The first function *eatt* computes the *earliest arrival travel time*, it includes travel times on arcs and staying times at stations. The second function *transfers* counts the number of transfers between trains. For an event-dependent path

P_E we define $ontrip(P_E) := \sum_{v_e \in V_E} ontrip(v_e) + \sum_{a_e \in A_E} ontrip(a_e)$. Pareto-optimal on-trip paths can be computed by a Dijkstra-like labeling algorithm, see [7] for details. We would like to point out that Pareto-optimal on-trip greedy paths are not necessarily Pareto-optimal on-trip paths in the station graph. Fortunately, one can show that the set of Pareto-optimal on-trip greedy paths is a subset of all Pareto-optimal on-trip paths in the station route graph G_r.

3 Speed-Up Techniques SUBITO and *k*-Flags

3.1 *c*-Optimality and the SUBITO Technique

In this section we consider a dynamic timetable, i.e., one with dynamic changes for departure and arrival events. In particular, departure times and arrival times can change. Let $G_r = (V, A_r)$ be the station route graph with a static weight g. Let $V^* := \{V^1, \ldots, V^\nu\}$ be a partition of vertex set V, each element $V^j \in V^*$ is called a *region*. Furthermore, we denote all vertices $b \in V^j$ as *boundary vertices of region* V^j if and only if there exists at least one arc $(b, u), (u, b) \in A_r$ with $u \notin V^j$. We define the arc-weight-function $g : A_r \to \mathbb{R}^+$, denoting the *minimal theoretical travel time* on arc $a \in A_r$ on the station route graph G_r. Furthermore, we denote the minimal theoretical travel time on a topological path p by $g(p) := \sum_{a \in p} g(a)$. We denote by $d_g(s,t) := \min_{p \in p_{st}} g(p)$ the *smallest* minimal theoretical travel time between two stations s and t. We call a topological (s,t)-path p with $d_g(s,t) \leq g(p) \leq d_g(s,t) + c$ a *c-optimal topological path.* These definitions give us the desired substructure properties of c-optimality.

Theorem 1 (substructure property of c-optimal paths). *Let p' be a topological subpath of an (s,t)-topological path p in the station route graph $G_r = (V, A_r, g)$. Then we have:* p is c-optimal $\Rightarrow$ p' is c-optimal.

Next we define a subset of event-dependent paths. We call an event-dependent path P_{arr_u, dep_v} with $eatt(P_{arr_u, dep_v}) \leq d_g(v,v) + c$ and $c \in \mathbb{R}_+$ an *event-dependent c-optimal* path. This definition relates static and event-dependent paths. Similar to static c-optimal paths we also observe a substructure property for c-optimal event-dependent paths.

Theorem 2 (event-dependent c-optimality and substructure property) *Let P_{start_s, end_t} be an event-dependent c-optimal path in $\mathcal{G}(G_r, s, t, start_s)$ with $eatt(P_{start_s, end_t}) \leq d_g(s,t) + c$. Then each subpath P_{start_s, dep_u} is an event-dependent c-optimal path with $eatt(P_{start_s, dep_u}) \leq d_g(s,u) + c$.*

Our goal is to find all on-trip paths arriving at most max_{time} time after the earliest possibility. In the first step of our algorithm we determine the smallest minimum theoretical travel times $d_g(s,v)$ for all $v \in V$. In a next step we determine the earliest arrival time at t with respect to the earliest start time $start_s(time)$ in s. Then, we determine the maximum required value c_{max} by computing $c_{max} := eatt + max_{time} - d_g(s,t)$. Finally, we apply a Dijkstra-like labeling algorithm where we now can ignore departure events dep_u which do not fulfill $dep_u(time) \leq d_g(s,u) + c_{max}$. We denote these speed-up technique as *SUBITO-technique* for *sub*structure *i*n *t*ime-dependent *o*ptimization.

3.2 The Idea and Correctness of *k*-Flags

In this section we introduce the notion of *k-flags* which generalize classical arc-flags.

Definition 2 (*k*-flags). *Let $G_r = (V, A_r, g)$ be the station route graph, V^* a vertex partition of V, $c_i \in \mathbb{N}_+ \cup \{0\}$ with $c_1 < \cdots < c_k$, and $i \in \{1, \cdots, k\}$. We define for each $a = (v, u) \in A_r$ the k-flag function $f_a \in \{1, \cdots, k\}^{V^*}$* with

$$f_a(V^j) = \min\{i \mid \exists\ t \in V^j \text{ and a } c_i\text{-optimal path } p = (v, u, \cdots, t)\}.$$

Note, that there may exist vertices v with $d_g(v, t) = \infty$ for all $t \in V^j$. In this case all arcs (v, u) do not get a k-flag for region V^j. We construct a sequence of *query graphs* $G^i_{V^j} := (V^i_{V^j}, A^i_{V^j})$, $i \in \{1, \cdots, k\}$ with respect to an (s, t)-query. We define $A^i_{V^j} := \{a \in A_r \mid f_a(V^j) \le i, t \in V^j\}$ and $V^i_{V^j} := \{v \in V \mid (v, u) \in A^i_{V^j} \vee (u, v) \in A^i_{V^j}\}$. It follows $A^i_{V^j} \subset A^{i+1}_{V^j} \subset A_r$ for all $i \in \{1, \cdots, k-1\}$. The next theorem proves that we find all c_i-optimal topological (s, t)-paths p in query graph $G^i_{V^j}$. This reduces the number of arcs in our station route graph for the search of c_i-paths with $i < k$.

Theorem 3 (consistence of c-optimality). *Let $G_r = (V, A_r, g)$ be the station route graph and $G^i_{V^j} := (V^i_{V^j}, A^i_{V^j})$ the query-graph from level i. Then it follows: p is c_i-optimal topological (s, t)-path in $G_r = (V, A_r)$ $\Leftrightarrow$ p is c_i-optimal topological (s, t)-path in $G^i_{V^j} := (V^i_{V^j}, A^i_{V^j})$.*

The next theorem shows that we are able to find all optimal on-trip paths (with an arrival time at most c_i time units after the fastest path) in the event-dependent graph possessing as underlying station graph our reduced query graph.

Theorem 4 (arc reduced event-dependent query graph). *Let $G_r = (V, A_r)$ be the station route graph, $\mathcal{G}(G_r, s, t, start_s)$ the corresponding event-dependent graph. Let $G^i_{V^j} = (V^i_{V^j}, A^i_{V^j})$ the query graph, $\mathcal{G}(G^i_{V^j}, s, t, start_s)$ the corresponding event-dependent graph. Then it follows: P_{arr_v, dep_u} is an event-dependent c_i-optimal path in $\mathcal{G}(G_r, s, t, start_s)$*
$\Rightarrow P_{arr_v, dep_u}$ is an event-dependent c_i-optimal path in $\mathcal{G}(G^i_{V^j}, s, t, start_s)$.

SUBITO and k-flags can be combined, that means, we search for event-dependent c_i-optimal paths in $\mathcal{G}(G^i_{V^j}, s, t, start_s)$ where we ignore all event-dependent (s, u)-subpaths P_{start_s, dep_u} which are not c_i-optimal. The correctness of this approach follows by Theorems 2 and 4. See a pseudocode in Algorithm 1.

4 Experimental Study

Test instances and environment. Our computational study is based on the German train schedule of 2008. This schedule consists of 8817 stations, 40034 trains on 15428 routes, 392 foot paths, and 1,135,479 elementary connections. In our station route graph model we obtain a graph with 189,214 arcs. For our tests, we used queries with randomly chosen start stations and destinations, and different earliest start times (namely 0:00, 4:00, 8:00, 12:00, 16:00, 20:00).

Algorithm 1. On-Trip-Algorithm with SUBITO k-flags

```
Input: Origin $s$, destination $t \in V^j$, earliest start event $start_s$, preprocessed schedule,
    parameter $max_{time} \geq 0$.
Output: Set of $c_i$-optimal on-trip paths for $i = 1, \ldots, i_{max} \leq k$. //Let $P_{start_s,end_t}$
    be an event-dependent Pareto-optimal path with minimum eatt. We deter-
    mine all Pareto-optimal on-trip paths $P'_{start_s,end_t}$ with $eatt(P'_{start_s,end_t}) \leq$
    $eatt(P_{start_s,end_t}) + max_{time} \leq d_g(s,t) + c_{i_{max}}$.
1: Compute the minimum theoretical travel time $d_g(s,v)$ for all $v \in V$ in $G_r$.
2: Compute the minimal earliest arrival travel time (eatt) at $t \in V$. //Do a single-
    criteria event-dependent Dijkstra search with respect to eatt level by level until t is
    settled.
3: Compute the maximum required level $i_{max}$ with respect to $max_{time}$.
4: Compute Pareto-optimal on-trip-paths with a label setting algorithm (see [7])
    on the query graph $G^{i_{max}}_{V^j}$. Ignore departure events which do not possess $c_{i_{max}}$-
    optimality (SUBITO).
```

We considered three types of delay scenarios:

1. We used the complete delay information of a typical day of operation in Germany available at 8:00 a.m.
2. We simulated a number of considerable delays distributed over all Germany. Our experiments are parameterized with the number $x \in \{10, 50, 100, 150\}$ of primary delays and the delay size $s \in \{5, 10, \ldots, 60\}$.
3. We simulated a major disruption at a single station, more precisely, we stopped all trains passing through this station by two hours. This scenario models disruptions due to a power failure, the breakdown of a signal tower, or the evacuation of a station after a bomb threat.

For the simulation of delays we used the prototype MOTIS (multi-objective traffic information system) which provides a fully realistic model using the standard waiting policies of German Railways to calculate secondary delays. MOTIS can handle massive data streams [1]. All experiments were run on a standard PC (Intel(R) Xeon(R), 2.93GHz, 4MB cache, 47GB main memory under ubuntu linux version 8.10). Only one core has been used by our program. Our code is written in C++ and has been compiled with g++ 4.4.1 and compile option -O3.

Algorithmic variants and combinations. We used the following variants:

– **base:** This variant is a reasonably tuned multi-criteria Dijkstra-like algorithm which uses the speed-up technique dominance by results in combination with lower travel time bounds towards the destination (cf. [4]).
– **goal:** This version adds to base a version of goal-directed search.
– **SUBITO** refers to our new speed-up technique.
– **kFlags** refers to a version which combines all other techniques with k-flags.
– If we search only for greedy paths, we denote this by the attribute **greedy**.

Each variant is parameterized by max_{time}, i.e., we search for paths which arrive at most max_{time} minutes after the earliest possibility at the destination.

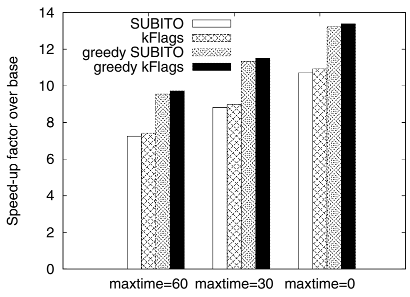


Fig. 1. Comparison of different speed-up techniques

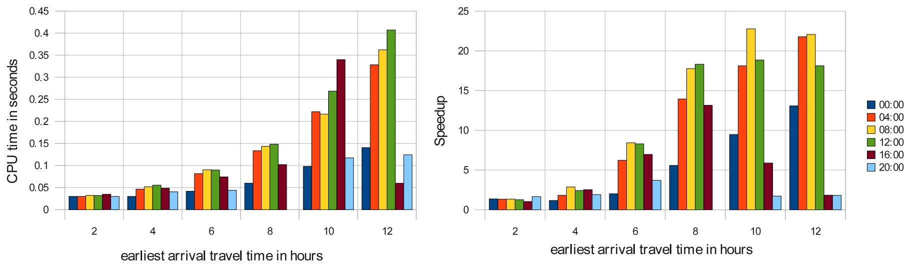


Fig. 2. Experiment 1: Results of greedy k-flags with $max_{time} = 0$. Each column corresponds to a class of queries with the same earliest start time.

Preprocessing. SUBITO either requires no preprocessing at all (the variant used in our experiments) or a single all-pairs shortest path computation on the station route graph which takes less than two minutes (and requires no update after dynamic changes). The computation of k-flags is computationally expensive, it requires about 3h 40 minutes for 128 regions.

Experiment 1: No delays. In our first experiment, we are interested in the comparison of variants in a scenario without delays. In Figure 1, we display the results of the base variant in comparison with goal-direction, SUBITO and kFlags (with and without greedy heuristic), and $max_{time} \in \{0, 30, 60\}$. The baseline variant requires an average CPU time of $1.63s$. Turning on goal-direction, already reduces the average CPU time to $0.25s$. For fixed parameter max_{time}, the kFlags variant is consistently the fastest method, but only marginally faster than SUBITO (an explanation follows below). Searching for greedy paths allows a significant speed-up over non-greedy paths. It is interesting to observe that CPU times and speed-ups depend on the earliest start time and the earliest arrival travel time (eatt). Figure 2 shows this dependency. The longer the travel time, the more CPU time is needed, but speed-ups are also higher for such queries.

Experiment 2 (delays on an ordinary day of operation). We have run an experiment with original delay data for a typical day of operation. Using several thousands of passenger queries, we observed almost no measurable effect on the running time. A corresponding table is therefore omitted.

Experiment 3 (distributed delays "snow chaos"). The purpose of our next experiment is to analyze whether the speed-up techniques are robust against an increasing number of update operations. We varied the number of primary delays from 10 to 150, leading to 188 and 4978 graph update operations, respectively. While several previous computational studies observed a major drop down of speed-ups already after a few update operations [8,11], there is almost no measurable effect on the CPU time and speed-up factors when we increase the number of primary delays.

Experiment 4 (closing down a central station for two hours). Next we analyzed the effect of stopping all trains passing through some central station for two hours (from 11:55 to 13:55). For that purpose, we selected three major German railway stations (Frankfurt am Main Hbf, Hannover Hbf, and Leipzig Hbf). For this experiment, we use a special set of 1000 queries each, designed such that it is likely that the fastest route usually would pass the closed station. Indeed, we observed a posteriori that a high percentage of passengers were affected and had to suffer from an increased earliest arrival time. For example, in the case of Frankfurt, even 98% of the passengers arrived later than without delays. As a consequence, queries require slightly more CPU time, but the speed-up factors over the baseline variant increase even slightly, see Table 1.

Table 1. Closing down a major station (Frankfurt a.M. Hbf, Hannover Hbf, Leipzig Hbf) for two hours. Average query time in seconds and speed-up factors over base.

Variant	Frankfurt a.M. Hbf		Hannover Hbf		Leipzig Hbf	
	CPU	speed-up	CPU	speed-up	CPU	speed-up
base	2.21	1.00	2.69	1.00	2.49	1.00
goal	0.79	2.80	0.34	7.91	0.27	9.27
SUBITO, $max_{time} = 60$	0.24	9.11	0.28	9.78	0.23	10.76
kFlags, $max_{time} = 60$	0.24	9.32	0.27	9.96	0.23	10.92
greedy, kFlags, $max_{time} = 60$	0.19	11.93	0.20	13.54	0.18	14.20
greedy, kFlags, $max_{time} = 0$	0.13	17.39	0.15	17.93	0.13	19.18

Analysis of *k*-flag distribution. The distribution of k-flags sheds some light on the potential speed-up which can be gained by techniques which try to thin out the station route graph.

For train networks we observe that a multi-criteria search requires a large fraction of arcs in order to guarantee exact solutions for point-to-point queries. The average maximal required c-value c_{max} for our queries is $c_{max} = 143$, which means that the underlying search subgraph for the k-flag technique consists of about 72% of all arcs. About 25% of all arcs can be excluded since the destination cannot be reached on simple paths.

References

1. Müller-Hannemann, M., Schnee, M.: Efficient timetable information in the presence of delays. In: Zaroliagis, C. (ed.) Robust and Online Large-Scale Optimization. LNCS, vol. 5868, pp. 249–272. Springer, Heidelberg (2009)
2. 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)
3. Abraham, I., Fiat, A., Goldberg, A.V., Werneck, R.F.: Highway dimension, shortest paths, and provably efficient algorithms. In: SODA 2010: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 782–793. SIAM, Philadelphia (2010)
4. Berger, A., Delling, D., Gebhardt, A., Müller-Hannemann, M.: Accelerating time-dependent multi-criteria timetable information is harder than expected. In: Clausen, J., Stefano, G.D. (eds.) ATMOS 2009 - 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany (2009)
5. Bauer, R., Delling, D., Wagner, D.: Experimental study on speed-up techniques for timetable information systems. Networks (to appear, 2010)
6. Bast, H.: Car or public transport—two worlds. In: Albers, S., Alt, H., Näher, S. (eds.) Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday. LNCS, vol. 5760, pp. 355–367. Springer, Heidelberg (2009)
7. Berger, A., Müller-Hannemann, M.: Subpath-optimality of multi-criteria shortest paths in time-dependent and event-dependent networks. Technical report, Martin-Luther-Universität Halle-Wittenberg, Department of Computer Science (2009)
8. Wagner, D., Willhalm, T., Zaroliagis, C.D.: Dynamic shortest paths containers. Electr. Notes Theor. Comput. Sci. 92, 65–84 (2004)
9. Schultes, D., Sanders, P.: Dynamic highway-node routing. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 66–79. Springer, Heidelberg (2007)
10. 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)
11. Berrettini, E., D'Angelo, G., Delling, D.: Arc-flags in dynamic graphs. In: Clausen, J., Stefano, G.D. (eds.) ATMOS 2009 - 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany (2009)
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. Nannicini, G., Baptiste, P., Krob, D., Liberti, L.: Fast computation of point-to-point paths on time-dependent road networks. In: Yang, B., Du, D.-Z., Wang, C.A. (eds.) COCOA 2008. LNCS, vol. 5165, pp. 225–234. Springer, Heidelberg (2008)
14. Nannicini, G., Baptiste, P., Barbier, G., Krob, D., Liberti, L.: Fast paths in large-scale dynamic road networks. Computational Optimization and Applications (published online, 2008)
15. Orda, A., Rom, R.: Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. J. ACM 37, 607–625 (1990)

Space-Efficient SHARC-Routing[⋆]

Edith Brunel[1], Daniel Delling[1,2], Andreas Gemsa[1], and Dorothea Wagner[1]

[1] Institute of Theoretical Informatics,
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
edith@brunel-online.de, {gemsa,dorothea.wagner}@kit.edu
[2] Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043
dadellin@microsoft.com

Abstract. Accelerating the computation of quickest paths in road networks has been undergoing a rapid development during the last years. The breakthrough idea for handling road networks with tens of millions of nodes was the concept of *shortcuts*, i.e., additional arcs that represent long paths in the input. Very recently, this concept has been transferred to *time-dependent* road networks where travel times on arcs are given by functions. Unfortunately, the concept of shortcuts is very space-consuming in time-dependent road networks since the travel time functions assigned to the shortcuts may become quite complex.

In this work, we present how the space overhead induced by time-dependent SHARC, a technique relying on shortcuts as well, can be reduced significantly. We are able to reduce the overhead stemming from SHARC by a factor of up to 11.5 for the price of a loss in query performance of a factor of 4. The methods we present allow a trade-off between space consumption and query performance.

1 Introduction

Route Planning is a prime example of algorithm engineering. Modeling the network as graph G with arc weights depicting travel times, the shortest path in G equals the quickest connection in the transportation network. In general, DIJKSTRA's algorithm [12] solves this task, but unfortunately, the algorithm is way to slow to be used in transportation networks with tens of millions of nodes. Therefore, so called *speed-up techniques* split the work into two parts. During an *offline* phase, called preprocessing, additional data is computed that accelerates queries during the *online* phase. The main concept for route planning in road networks was the introduction of so called *shortcuts*, i.e., arcs representing long paths, to the graph. A speed-up technique then relaxes the shortcut instead of the whole path if the target is "sufficiently far away".

However, adapting the concept of shortcuts to *time-dependent* road networks yields several problems. A travel time function assigned to the shortcut is as complex as all arc functions the shortcut represents. The reason for this is that we need to *link* piece-wise linear functions (cf. [8] for details). For example, a straightforward adaption of Contraction Hierarchies [13], a technique relying solely on shortcuts yields an overhead of $\approx 1\,000$ bytes per node [1] in a time-dependent scenario whereas the overhead in a time-independent scenario is almost negligible.

[⋆] Partially supported by the DFG (project WA 654/16-1).

[9] gives an overview over time-independent speed-up techniques, while [11] summarizes the recent work on time-dependent speed-up techniques. As already mentioned, all efficient speed-up techniques for road networks rely on adding shortcuts to the graph making the usage of them in a limited time-dependent environment complicated. Memory efficient variants of *time-independent* speed-up techniques however exist. For example, Contraction Hierarchies [13] have been implemented on a mobile device [21]. The straightfoward time-dependent variant [1] is very space-consuming. The ALT [14] algorithm, which works in a time-dependent scenario as well [19], has been implemented on an external device [15] as well. However, space consumption of ALT is rather high and performance is clearly inferior to SHARC. Work on the compression of time-independent graph data structures can also be found in [4,5]. To the best of our knowledge, we were the first who studied the problem of compressing a high-performance time-dependent speed-up technique [7]. However, since the publication of [7], the memory consumption of time-dependent Contraction Hierarchies has been reduced [2]. Still, our approach yields a factor of 1.6 less overhead.

In this work, we present how to compress the preprocessing of SHARC, introduced in [3] and augmented to the time-dependent scenario in [8], without too high of a loss in query performance. The key idea is to identify unimportant parts of the preprocessing and remove them in such a way that correctness of SHARC can still be guaranteed. After settling preliminaries and recalling SHARC in Section 2, we present our main contribution in Section 3. There, we show how to reduce the overhead stemming from arc-flags stored to the graph, by mapping unimportant arc-flag vectors to important ones. The advantage of this approach over other compression schemes such as bloom filters [6] is that we do not need to change the query algorithm of SHARC. Due to this fact, we keep the additional computational effort limited. Moreover, we show that we can remove shortcuts from SHARC, again without changing the query algorithm. Finally, we may even remove the complex travel time functions from the shortcuts by reproducing the length function on-the-fly. In Section 4 we run extensive tests in order to show the feasibility of our compression schemes. It turns out that we can safely remove 40% of the arc-flag information *without* any loss in query performance. Moreover, about 30% of the shortcuts added by SHARC are of limited use as well. So, we may also remove them. Finally, it turns out that by removing the travel time functions from the remaining shortcuts, we can reduce the overall overhead of SHARC significantely. As a result, we are able to reduce the overhead induced by SHARC by a factor of up to 11.5. The resulting memory efficient variant of SHARC yields an overhead of 13.5 (instead of 156) bytes per node combined with average query times of about 3 *ms* (on the German road network with realistic time-dependent traffic), around 500 times faster than DIJKSTRA's algorithm.

2 Preliminaries

A (directed) graph $G = (V,A)$ consists of a finite set V of nodes and a finite set E of arcs. An arc is an ordered pair (u,v) of nodes $u,v \in V$, the node u is called the *tail* of the arc, v the *head*. The number of nodes $|V|$ is denoted by n, the number of arcs by m. Throughout the whole work we restrict ourselves to directed graphs which are weighted by a piece-wise linear periodic travel time function *len*. A travel time function $len(e)$ is defined by several *interpolation points*, each consisting of a timestamp t and a travel time $w > 0$, depicting the travel time on e at time t. The travel time between two interpolation points is done by *linear interpolation*. The composition of two travel time functions f,g is defined by $f \oplus g := f + (g \circ (f + id))$.

A *partition* of V is a family $\mathscr{C} = \{C_0, C_1, \ldots, C_k\}$ of sets $C_i \subseteq V$ such that each node $v \in V$ is contained in exactly one set C_i. An element of a partition is called a *cell*. A *multilevel partition* of V is a family of partitions $\{\mathscr{C}^0, \mathscr{C}^1, \ldots, \mathscr{C}^l\}$ such that for each $i < l$ and each $C_n^i \in \mathscr{C}^i$ a cell $C_m^{i+1} \in \mathscr{C}^{i+1}$ exists with $C_n^i \subseteq C_m^{i+1}$. In that case the cell C_m^{i+1} is called the *supercell* of C_n^i. The supercell of a level-l cell is V.

The original arc-flag approach [17,16] first computes a partition $\mathscr{C}$ of the graph and then attaches a *label* AF to each arc e. A label contains, for each cell $C_i \in \mathscr{C}$, a flag $AF_{C_i}(e)$ which is `true` if a shortest path to a node in C_i starts with e. A modified DIJKSTRA then only considers those arcs for which the flag of the target node's cell is `true`. Given two arc-flag vectors AF_1, AF_2. The OR arc-flags vector $AF_1 \vee AF_2$ has all arc-flags set to `true` that are `true` in AF_1 or AF_1. The AND of two arc-flags vectors is defined analogously and is denoted by $AF_1 \wedge AF_2$.

Note that more and more arcs have a flag set for the target's cell when approaching the target cell (called the *coning effect*) and finally, all arcs are considered as soon as the search enters the target cell. Hence, [18] introduces a second layer of arc-flags for each cell. Therefore, each cell is again partitioned into several subcells and arc-flags are computed for each. This approach can be extended to a multi-level arc-flags scenario easily. A multi-level arc-flags query then first uses the flags on the topmost level and as soon as the query enters the target's cell on the topmost level, the lower-level arc-flags are used for pruning. In the following we denote by the level of an arc-flag the level of layer it is responsible for.

SHARC [3]. The main disadvantage of a multi-level arc-flags approach is the time-consuming preprocessing [16]. SHARC improves on this by the integrating of contraction, i.e., a routine iteratively removing unimportant nodes and adding shortcuts in order to preserve distances between non-removed nodes. Preprocessing of SHARC is an iterative process: during each iteration step i, we contract the graph and then compute the level i arc-flags. One key observation of SHARC is that we are able to assign arc-flags to all bypassed arcs during contraction. More precisely, any arc (u,v) outgoing from a non-removed node and heading to a removed one gets only one flag set to `true`, namely, for the region v is assigned to. Any other bypassed arc gets all flags set to `true`. By this procedure, unimportant arcs are only relaxed at the beginning and end of a query. Although these suboptimal arc-flags already yield a good query performance, SHARC improves on this by a (very local) arc-flag refinement routine. The key observation here is that bypassed arcs may inherit flags from arcs not bypassed during contraction (cf. [3] for details). It should be noted SHARC integrates contraction in such a natural way that the multi-level arc-flags query can be applied to SHARC without modification.

Due to its unidirectional query algorithm, SHARC was a natural choice for augmenting it to a time-dependent [8] and a multi-criteria scenario [10]. The idea is the same for both augmentations: adapt the basic ingredients of the preprocessing, i.e., arc-flags, contraction, and arc-flags refinement, such that correctness of them can still be guaranteed and leave the basic concept untouched. It turns out that SHARC performs pretty well in both augmented scenarios. However, a crucial problem for time-dependent route planning are shortcuts representing paths in the original graph. While this is "cheap" in time-independent networks, the travel time functions assigned to time-dependent shortcuts may become quite complex. In fact, the number of interpolation points defining the shortcut is approximately the sum of all interpolation points assigned to the arcs the shortcut represents. See [8] for details. In fact, the overhead of SHARC increases by a factor of up to 10 when switching from a time-independent to a time-dependent scenario exactly because of these complex travel time functions.

SHARC adds auxiliary data to the graph. More precisely, the overhead stems from several ingredients: Region information, arc-flags, topological information of shortcuts, the travel time functions of shortcuts and shortcut unpacking information. We call the graph enriched by shortcuts and any other auxiliary data the *output graph*.

The first overhead, i.e., the region information, is used for determining which arc-flag to evaluate during query times. This information is encoded by an integer and cannot be compressed without a significant performance penalty. We have a arc-flags vector for each arc. However, the number of unique arc-flags vectors is much smaller than the number of arcs. So, instead of storing the arc-flags directly at each arc, we use a separate table containing all possible unique arc-flags sets. In order to access the flags efficiently, we assign an additional pointer to each arc indexing the correct arc-flags set in the table. The main overhead, however, stems from the shortcuts we add to the graph. For each added shortcut, we need to store the topological information, i.e., the head and tail of the arc, and the travel time function depicting the travel time on the path the shortcut represents. Moreover, we need to store the arcs the shortcut represents in order to retrieve the complete description of a computed path.

3 Preprocessing Compression

In this section, we show how to reduce the space consumption of SHARC by removing unimportant arc-flags, shortcuts, and functions without violating correctness.

Arc-Flags. The first source of overhead for SHARC is storing the arc-flags for each arc. As already mentioned, our original SHARC implementation already compresses the arc-flag information by storing each unique arc-flag set separately in a table (called the *arc-flags table*) and each arc stores an index to the arc-flag table. Figure 3 gives a small example.

Lemma 1. *Given a correct SHARC preprocessing. Flipping (arbitrary) arc-flags from* `false` *to* `true` *does not violate the correctness of SHARC.*

Proof. Let $P = (u_1, \dots, u_k)$ be an arbitrary shortest path in G. Since SHARC-Routing is correct, we know that each arc (u_i, u_{i+1}) has the arc-flag being responsible for t set to `true`. Since we only flip bits from `false` to `true`, this also holds after bit-flipping. □

This lemma allows us to flip bits in the arc-flag table from `false` to `true`. Hence, we compress the arc-flag table by *bit-flipping*. Let AF_r, AF_m be two unique arc-flag vectors

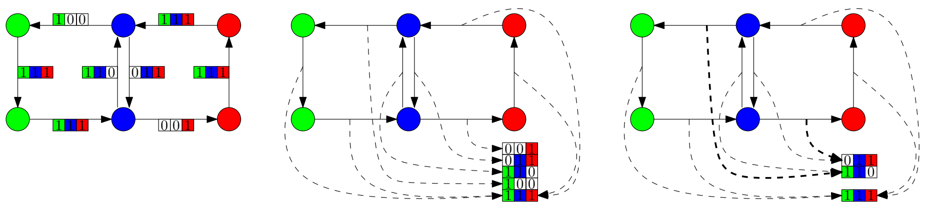


Fig. 1. Compression of Arc-Flags. The input is partitioned into three cells, indicated by coloring. A set arc-flag is indicated by color and a one. Instead of storing the flags directly at the arcs (left figure), we store each unique arc-flags vector in a separate table with arcs indexing the right arc-flags vector (middle figure). We additionally compress the table by flipping bits from `false` to `true` (right figure) and thus, we reduce the number of entries in the table. The remapped indices are drawn thicker.

such that $AF_r \subseteq AF_m$, i.e., all arc-flags set in AF_r are also set in AF_m. Then, we may remove AF_r from our arc-flag table and all arcs indexing AF_r are remapped to AF_m. Note that this compression scheme has no impact on the query algorithm.

The compression rate achieved by bit-flipping highly depends on which arc-flag vectors to remove and to which arc-flag vectors they are mapped. We introduce an *arc-flag costs* function cost assigning an importance value to each arc-flags vector. The idea is as follows: for each layer i of the multi-level partition, we introduce a value cost_i. Let $|AF_i|$ be the number of flags set to `true` on level i. Then we define $\text{cost}(AF) = \sum_{i=0}^{l} \text{cost}_i \cdot |AF_i|$. The higher the costs of an arc-flags vector scores, the more important it is. So, a good candidate for removing it from the table should have low costs. The remaining question is what a good candidate for mapping is. Therefore, we define the *flipping costs* between two arc-flag vectors AF_r,AF_m with $AF_r \subseteq AF_m$ as $\text{cost}(AF_r \wedge AF_m)$. A good mapping candidate AF_m for a vector AF_r to be removed is the arc-flags vector with minimal flipping costs. It is easy to see that $\text{cost}(AF_r \wedge AF_m) = \text{cost}(AF_m) - \text{cost}(AF_r)$ holds. We reduce the overhead induced by arc-flags by iteratively removing arc-flags vectors from the table. Therefore, we order the unique arc-flags vectors non-decreasing by their costs. Then, we remove the arc-flags vector AF_r with lowest costs from the table and remap all arcs indexing AF_r to the arc-flags vector AF_m with minimal costs and for which $AF_r \subseteq AF_m$ holds.

Shortcuts. In Section 2, we discussed that, at least in the time-dependent scenario, the main source of overhead derives from the (time-dependent) shortcuts added to the graph. In [3], we already presented a subroutine to remove *all* shortcuts from SHARC. However, this yielded a high penalty in query performance. The following lemma recaps the main idea.

Lemma 2. *Given a correct SHARC preprocessing. Let* (u,v) *be an added shortcut during preprocessing and let* $P_{uv} = (u,u_0,\dots,u_k,v)$ *be the path it represents. By removing* (u,v) *from the graph and setting* $AF(u,u_0) = AF(u,u_0) \vee AF(u,v)$*, correctness of SHARC is not violated.*

Proof. Let $P = (s,\dots,u,v,\dots,t)$ be an arbitrary shortest path using shortcut (u,v) in the output graph of SHARC. Let also (u,u_0) be the first edge of the path (u,v) represents in the original graph. We need to show that after removing (u,v), the path $P' = (s,\dots,u,u_0,\dots,t)$ has all flags for t set to `true`. Since SHARC is correct, we know that the subpath $(s,\dots,u)$ has correct flags set. Moreover, (u,u_0) has proper flag set as well, since we propagate all flags from (u,v) to (u,u_0). We also know that the shortest path from u_0 to t must not contain (u,v) since we restrict ourselves to positive length functions (cf. Section 2). Due to correctness of SHARC, the shortest path from u_0 to t must have proper flags set. Hence, P' has proper flags set as well. □

In other words, we reroute any shortest path query using a removed shortcut (u,v) to its path it represents, Figure 2 gives an example. The lemma allows us to remove arbitrary shortcuts from the output graph. Note that we may again leave the SHARC query untouched. Some shortcuts are more important than others and the ordering in which we remove shortcuts has a high impact on the resulting query performance. Generally, a removed shortcut should only have low arc-flags costs (cf. Section 4). Furthermore, let $l(u)$ be the level of an arbitrary node u, given by iteration u was removed during the original preprocessing of SHARC (cf. Section 2). We define the *tail-level* of a shortcut (u,v) by $l(u)$, while $l(v)$ is the *head-level*. Presumably, shortcuts with low head and tail levels are less important than those with high ones. A fourth indicator for the importance of a shortcut is the so-called *search space coning coefficient*. Let (u,v) be a shortcut and let $P_{uv} = (u,u_0,\dots,u_k,v)$ be the path it represents. Then the search space coning coefficient of (u,v) is given by $\mathrm{sscc}(u,v) = \sum_{u_i \in P_{uv}} \sum_{(u_i,w)\in E, w\neq u_{i+1}} \mathrm{cost}\left(AF(u,v) \wedge AF(u_i,w)\right)$. In other words, the search space coning coefficient depicts how many arcs may be relaxed additionally if (u,v) was removed, i.e., the search cones. Therefore, the arc-flags of any outgoing arc from $u_i \in P_{uv}$ is examined and whenever a flag is set that is also set for the shortcut, the search space coning coefficient increases.

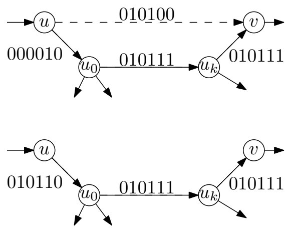


Fig. 2. Example for removing the shortcut (u,v), representing (u,u_0,u_k,v), by propagating flags from (u,v) to (u,u_0)

Our *shortcut-removal* compression scheme iteratively removes shortcuts from the graph and sets arc-flags according to Lemma 2. We use a priority queue to determine which shortcut to remove next. The priority of a shortcut is given by a linear combination of its head level, its tail level, its arc-flag costs, and the search space coning coefficient. We normalize these values by their maximal values during initialization.

Removing Travel Time Functions. Removing shortcuts from the output graph increases the search space since unnecessary arcs may be relaxed during traversing the path the shortcut represents. However, the main problem of time-dependent shortcuts are their complex travel time functions. Another possibility to remedy this space consumption is to remove the travel time functions but to keep the shortcut itself. Now, when a shortcut is relaxed, we compute the weight of it by unpacking the shortcut on-the-fly. The advantage of this over complete removal of the shortcut is that the search space does not increase. However, due to on-the-fly unpacking query times may increase.

Again, like for removing shortcuts and arc-flags compression, we are able to remove the travel time functions only from some of the shortcuts. On the one hand, we want to remove a shortcut with a complex function, and on the other hand, it should not be relaxed too frequentely. Hence, we use a priority queue in order to determine which function to delete next. As key for (u,v) we use a linear combination of the number of interpolation points of $len(u,v)$ and the arc-flag costs of $AF(u,v)$.

4 Experiments

Our experimental evaluation has been done on one core of an AMD Operon 2218 running SUSE Linux 11.1. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.3, using optimization level 3. Our implementation is written in C++. As priority queue we use a binary heap.

We use the German road network as input, it has approximately 4.7 million nodes and 10.8 million arcs. We have access to five different traffic scenarios, generated from realistic traffic simulations: Monday, midweek (Tuesday till Thursday), Friday, Saturday, and Sunday. All data has been provided by PTV AG [20] for scientific use.

For testing our compression schemes, we run a complete time-dependent SHARC preprocessing (heuristic variant [8]) on our Monday instance, we use our default parameters from [8]. The input has a space consumption of 44.2 bytes per node. SHARC adds about 2.7 million shortcuts and increases the number of interpolation points (for time-dependent arcs) from 12.7 million to about 92.1 million, yielding a total overhead of 156.94 additional bytes per node. Form those 156.94 bytes per node, 11.55 are stemming from shortcuts (topology and unpacking information), 8.2 from the arc-flag information, 2.0 from the region information, while 135.31 bytes per node stem from the additional interpolation points added to the graph. Hence, the main overhead stems from the latter. Note that the total overhead is slightly higher than reported in [8]. The reason for this is a change (we now store no interpolation point for a time-independent edge) to a more space-efficient graph data structure. Note that the heuristic variant of SHARC may compute a path that is slightly longer than the shortest in very few occasions [8]. However, all insights gained her also hold for any other variant.

In order to evaluate how well our schemes work, we evaluate the query performance of this SHARC preprocessing after compression. Therefore, we run 100 000 *s*-*t* queries for which we pick *s*, *t*, and the departure time uniformly at random. Then, we provide the average query time. Note that we do not report the time for unpacking the whole path. However, this can be done in less than 0.1 ms.

Arc-Flags. Figure 4 depicts the performance of time-dependent SHARC in our Monday scenario after removing a varying amount of arc-flags vectors for different cost functions. Note that we do not provide running times for compression since it takes less than one minute to compress the arc-flags. This is only a small fraction of the time the SHARC preprocessing takes (3-4 hours). We observe that the choice of the cost function has a high impact on the success of our flag compression scheme. As expected, a cost function that prefers flipping of low-level flags (cost function 1,3,9,27,243) performs better than one that prefers high-level flags (cost function 16,8,4,2,1). Interestingly, we

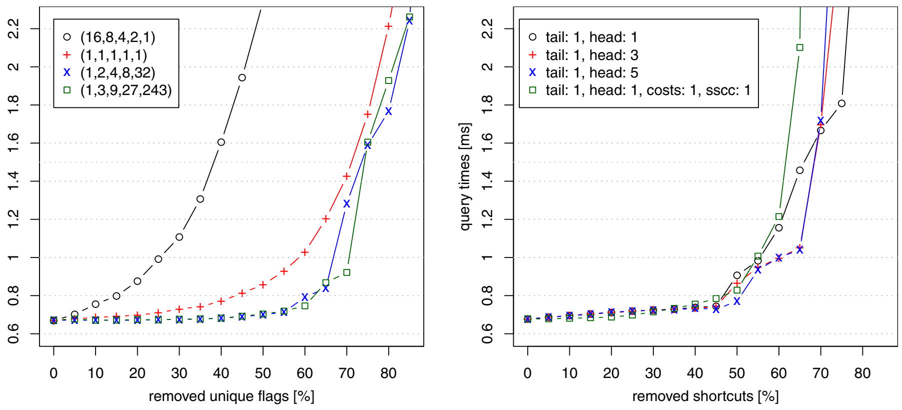


Fig. 3. Removing arc-flags (left) and shortcuts (right) from the output graph with cost functions. For arc-flags, the entry on the left indicates the costs for flipping a low level flag, while the right most entry shows the costs for flipping the highest level.

may remove up to 40% of the arc-flags vectors without any loss in performance. This reduces the overhead induced by arc-flags from 8.2 bytes per node to 7.2. By removing 60% of the vectors (resulting overhead: 6.9), the query performance decreases only by 10%. However, removing more than 70% of the flags yields a significant penalty in performance, although the overhead (for arc-flags) is only reduced to 6.3 bytes per node. So, since arc-flags contribute only a small fraction to the overhead, it seems reasonable to settle for a arc-flag compression rate of 40%.

Shortcuts. Figure 4 depicts the performance of SHARC after removing a varying amount of shortcuts for different linear coefficients introduced in Section 3. Note again that we do not report running times for compression since it takes less than one minute to obtain the reduced preprocessed data. We observe that we can remove up to 45% of the shortcuts yielding a mild increase in query performance ($\approx$ 10%). This reduces the overhead of the shortcuts from 11.55 bytes per node to 7.51. Moreover, the overhead induced by travel time functions is reduced from 135.31 to 118.72 bytes per node since some of the removed shortcuts are time-dependent. Up to 65%, the loss in query performance is still acceptable (a factor of 2), especially when keeping the gain in mind: the overhead for shortcuts reduces to 5.3 bytes per node and 97.7 bytes per node for additional interpolation points. Analyzing the impact of ordering, we observe that the level of head and tail seem to be the most important parameters. Interestingly, the head level seems to be more important than the tail level. The reason for this is that shortcuts from lower to higher levels are relaxed at the beginning of a query, while shortcuts from higher to lower levels are relaxed at the end. Since removing shortcuts cones the query (cf. Section 3), the latter shortcuts are less important. The influence of the search space coning coefficient is minor and only observable for very low compression rates: at 20%, the loss in performance is almost negligible. In the following, we will use values of 20%, 45%, and 65% as default parameters.

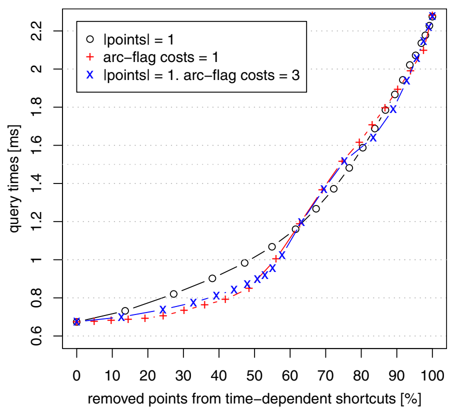


Fig. 4. Removing travel time functions. The x-axis indicates how many of the additional interpolation points are removed by the compression scheme. Coefficients not indicated are set to zero.

Travel Time Functions. Figure 4 indicates the query performance of SHARC after removing travel time functions from time-dependent shortcuts. We here evaluate different orderings given by different weights for the coefficients arc-flag costs and number of interpolation points, as explained in Section 3. We observe that for low compression rates, the arc-flag costs are more important than the number of points on the shortcut. However, the situation is vice versa between compression rates between 60% and 85%: here an ordering based on the number of points performs better than the order based on arc-flag costs. However, the differences are marginal, hence we use arc-flag costs as default for determing the ordering. In general, we may remove up to 40% of the additional points for a loss of query performance of about 25%. This already reduces the overhead induced by additional interpolation points to 81.2 bytes per node. The corresponding figures for a compression rate of 60% are a query performance penalty of factor 2 and a resulting overhead of 63.1. Most remarkably, we may even remove *all* additional interpolation points from the output graph with paying "only" a loss of performance of a factor of 3.2. This yields a *total* overhead of 21.6 bytes per node, a reduction of factor of 7.5 over the uncompressed preprocessing. Still the average query performance of 2.3 ms is still a speedup of a factor of 678 over DIJKSTRA's algorithm.

Combinations. Up to now, we evaluated each compression scheme separately. Table 1 gives an overview if we combine all three schemes among each other. We here report the overhead of the preprocessed data in terms of *additional* bytes per node. For evaluating the query performance, we not only provide query times but also the average number of settled nodes and relaxed arcs for 100 000 random *s*-*t* queries. Moreover, we report the speed-up over our (efficient) implementation of time-dependent Dijkstra. On this input, the latter settles about 2.2 million arcs in about 1.5 seconds on average. We observe that we may vary the compression rate yielding different total overhead and query performance. A good trade-off seems to be achieved for compressing shortcuts by 20%, interpolation points by 60%, and flags by 40%. This reduces space overhead in total by a factor of 2 and yields a loss in query performance by a factor of 1.85. For this is reason, we call these values a *medium* compression setup. Our *high* compression setup removes 65% of all shortcuts, removes all interpolation points from remaining time-dependent shortcuts and reduces the flag-table by 40%. This reduces the overhead induced by SHARC by a factor of almost 11.5 while query performance is ≈ 4 times slower than without compression. Any compression beyond this point yields a big performance loss without a significant reduction in preprocessed data.

Table 1. Query performance for different combinations of our compression schemes. As input, we use the German road network with traffic scenario Monday.

SHORTCUTS		POINTS		FLAGS		TOTAL		QUERIES			
rem.	overhead	rem.	overhead	rem.	overhead	overhead	comp.	#sett.	#rel.	time	speed
	[bytes/n]	[%]	[bytes/n]	[%]	[bytes/n]	[bytes/n]	[%]	nodes	arcs	[ms]	up
0	11.55	0	135.31	0	8.20	156.94	0.0	775	984	0.68	2 288
20	9.72	40	75.67	40	7.03	94.29	39.9	876	1 095	0.87	1 786
20	**9.72**	**60**	**50.22**	**40**	**7.03**	**68.84**	**56.1**	**876**	**1 095**	**1.26**	**1 238**
20	9.72	100	0.00	40	7.03	18.62	88.1	876	1 095	2.44	636
45	7.51	40	67.19	40	6.67	83.24	47.0	949	1 167	0.94	1 654
45	7.51	60	44.78	40	6.67	60.83	61.2	949	1 167	1.43	1 087
45	7.51	100	0.00	40	6.67	16.05	89.8	949	1 167	2.56	606
65	5.30	40	54.66	40	6.34	68.17	56.6	1 717	1 971	1.39	1 118
65	5.30	60	37.97	40	6.34	51.48	67.2	1 717	1 971	1.88	827
65	**5.30**	**100**	**0.00**	**40**	**6.34**	**13.52**	**91.4**	**1 717**	**1 971**	**2.98**	**521**
65	5.30	100	0.00	60	6.10	13.27	91.5	1 811	2 085	3.07	506

Traffic Scenarios. Our final testset evaluates the impact of different traffic scenarios on our compression schemes. Besides our Monday scenario which we evaluated up to this point, we now also apply a midweek (Tuesday to Thursday), Friday, Saturday, Sunday, and “no traffic” scenario. Note that the latter is a time-independent network. Our graph data structures occupy 44.2, 44.1, 41.0, 31.4, 27.8, and 22.4 bytes per node, respectively. We here also report the *additional* overhead induced by SHARC, as well as the total time of preprocessing (including compression). The resulting figures can be found in Tab. 2.

We observe that for all time-dependent inputs, our medium compression setup reduces the space consumption by a factor of 2 combined with a performance penalty between 1.5 and 1.9, depending on the degree of time-dependency in the network. Our high compression rate yields an overhead of $\approx$ 13.5 bytes per node, independently of the applied traffic scenario. This even holds for our “no traffic scenario”. The query performance however, varies between 0.37 ms (no traffic) and 3.06 ms (midweek). The reason for this is that in a high traffic scenario, more are arcs are time-dependent and hence, more arcs need to be evaluated when unpacking a (time-dependent) shortcut on-the-fly. Since the no traffic input contains no time-dependent arcs, no shortcut has a travel time function assigned. Hence, the costly on-the-fly unpacking needs not to be done during query times.

Comparison. Finally, we compare our memory-efficient version of SHARC (high compression) with the most recent variant of Contraction Hierachies (CH) [2]. The input is Germany midweek. CH achieves a speed-up of 714 over Dijkstra’s algorithm, assembling 23 additional bytes per node in 37 minutes. SHARC yields a slightly lower speed-up (491) with less overhead (13.5 bytes per node) for the price of a longer preprocessing time (around 4 hours). However, our heuristic variant of SHARC (which we use in this paper) drops correctness while CH is provably correct. Still, in (very) space-limited time-dependent environment, SHARC seems to be the first choice.

Table 2. Query performance of heuristic time-dependent SHARC applying different traffic scenarios for our German road network. Column *compression rate* indicates our default rates from Section 4. Columns *increase edges, points* indicate the increase in number of edges and points compared to the input.

	PREPROCESSING						QUERIES			
scenario	comp. rate	time [h:m]	inc. edges	inc. points	space [bytes/n]	comp [%]	#sett. nodes	#rel. arcs	time [ms]	speed up
Monday	none	3:52	25.2%	621.1%	156.94	0.0	775	984	0.68	2 288
	med	3:54	19.9%	230.5%	68.84	56.1	876	1 095	1.26	1 238
	high	3:54	8.2%	0.0%	13.52	91.4	1 717	1 971	2.98	521
midweek	none	3:46	25.2%	621.8%	156.45	0.0	777	990	0.68	2 203
	med	3:48	20.0%	229.3%	68.35	56.3	880	1 102	1.28	1 177
	high	3:48	8.2%	0.0%	13.54	91.3	1 715	1 971	3.06	491
Friday	none	3:22	25.1%	654.4%	142.90	0.0	733	930	0.63	2 400
	med	3:24	19.9%	240.4%	63.22	55.8	837	1 043	1.17	1 302
	high	3:24	8.2%	0.0%	13.59	90.5	1 691	1 932	2.79	543
Saturday	none	2:24	24.7%	784.2%	92.37	0.0	624	782	0.49	3 001
	med	2:26	19.6%	286.8%	44.58	51.7	726	892	0.81	1 825
	high	2:26	8.0%	0.0%	13.69	85.2	1 615	1 817	1.95	752
Sunday	none	1:53	24.3%	859.4%	67.43	0.0	593	735	0.44	3 299
	med	1:55	19.2%	317.8%	35.58	47.2	693	844	0.68	2 159
	high	1:55	7.9%	0.0%	13.64	79.8	1 576	1 762	1.64	893
no	none	0:10	23.3%	0.0%	20.90	0.0	277	336	0.18	6 784
	med	0:12	18.4%	0.0%	17.80	14.8	327	390	0.20	5 990
	high	0:12	7.5%	0.0%	13.12	37.2	758	838	0.37	3 332

5 Conclusion

In this work, we showed how to reduce the space consumption of SHARC without too high of a loss in query performance. The key idea is to identify unimportant parts of the preprocessing and remove them in such a way that correctness of SHARC can still be guaranteed. More precisely, we showed how to reduce the overhead stemming from arc-flags stored to the graph, how to remove shortcuts and how to remove complex travel time functions assigned to shortcuts. As a result, we were able to reduce the overhead induced by SHARC by a factor of up to 11. We thereby solved the problem of high space consumption of time-dependent route planning: SHARC does not yield a space-consumption penalty for switching from time-independent to time-dependent route planning, making it an interesting candidate for mobile devices.

Regarding future work, it would be interesting to compress the time-dependent input graphs. Techniques from [4,5,21] show how to compress the topology information of a graph. The main challenge, however, seems to be the reduction of the space consumption needed for storing the travel time functions. A possible approach could be the following: Real-world networks often assign a so called delay-profile to each edge. So, instead of storing the travel functions at the edges, one could use an index pointing to the (small number of) delay profiles. Note that this approach is similar to the arc-flags compression scheme.

References

1. Batz, G.V., Delling, D., Sanders, P., Vetter, C.: Time-Dependent Contraction Hierarchies. In: ALENEX, pp. 97–105. SIAM, Philadelphia (2009)
2. 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. Bauer, R., Delling, D.: SHARC: Fast and Robust Unidirectional Routing. ACM Journal of Experimental Algorithmics 14, 2.4 (2009)
4. Blandford, D.K., Blelloch, G.E., Kash, I.A.: Compact Representation of Separable Graphs. In: SODA, pp. 679–688. SIAM, Philadelphia (2003)
5. Blandford, D.K., Blelloch, G.E., Kash, I.A.: An Experimental Analysis of a Compact Graph Representation. In: ALENEX, pp. 49–61. SIAM, Philadelphia (2004)
6. Bloom, B.H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM 13(7), 422–426 (1970)
7. Brunel, E., Delling, D., Gemsa, A., Wagner, D.: Space-Efficient SHARC-Routing. Technical Report 13, ITI Wagner, Faculty of Informatics, Universität Karlsruhe, TH (2009)
8. Delling, D.: Time-Dependent SHARC-Routing. Algorithmica (July 2009)
9. 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)
10. Delling, D., Wagner, D.: Pareto Paths with SHARC. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 125–136. Springer, Heidelberg (2009)
11. Delling, D., Wagner, D.: Time-Dependent Route Planning. In: Zaroliagis, C. (ed.) Robust and Online Large-Scale Optimization. LNCS, vol. 5868, pp. 207–230. Springer, Heidelberg (2009)
12. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, 269–271 (1959)
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. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A* Search Meets Graph Theory. In: SODA, pp. 156–165 (2005)
15. Goldberg, A.V., Werneck, R.F.: Computing Point-to-Point Shortest Paths from External Memory. In: ALENEX, pp. 26–40. SIAM, Philadelphia (2005)
16. Hilger, M., Köhler, E., Möhring, R.H., Schilling, H.: Fast Point-to-Point Shortest Path Computations with Arc-Flags. In: Demetrescu, C., Goldberg, A.V., Johnson, D.S. (eds.) The Shortest Path Problem: Ninth DIMACS Implementation Challenge. DIMACS Book, vol. 74, pp. 41–72. American Mathematical Society, Providence (2009)
17. Lauther, U.: An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In: Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung, vol. 22, pp. 219–230. IfGI prints (2004)
18. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning Graphs to Speedup Dijkstra's Algorithm. ACM Journal of Experimental Algorithmics 11, 2.8 (2006)
19. 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)
20. PTV AG - Planung Transport Verkehr (2008), http://www.ptv.de
21. Sanders, P., Schultes, D., Vetter, C.: Mobile Route Planning. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 732–743. Springer, Heidelberg (2008)

A New Fully Dynamic Algorithm for Distributed Shortest Paths and Its Experimental Evaluation

Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, Daniele Frigioni, and Vinicio Maurizio

Dipartimento di Ingegneria Elettrica e dell'Informazione, Università dell'Aquila, I-67040 Monteluco di Roio, L'Aquila - Italy
{serafino.cicerone,gianlorenzo.dangelo,gabriele.distefano, daniele.frigioni}@univaq.it, vinicio.maurizio@cc.univaq.it

Abstract. In this paper we study the problem of dynamically update *all-pairs shortest paths* in a distributed network while edge update operations occur to the network. Most of the previous solutions for this problem suffer of two main limitations: they work under the assumption that before dealing with an edge update operation, the algorithm for each previous operation has to be terminated, that is, they are not able to update shortest paths *concurrently*; they concurrently update shortest paths, but their convergence can be very slow (possibly infinite) due to the well-known *looping* and *count-to-infinity* phenomena; they are not suitable to work in the realistic *fully dynamic* case, where an arbitrary sequence of edge change operations can occur to the network in an unpredictable way.

In this paper, we make a step forward in the area of shortest paths routing, by providing a new fully dynamic solution that overcomes some of the above limitations. In fact, our algorithm is able to concurrently update shortest paths, it heuristically reduces the cases where the looping and count-to-infinity phenomena occur and it is experimentally better than the Bellman-Ford algorithm.

Keywords: Dynamic algorithms, shortest paths, distributed algorithms.

1 Introduction

The problem of updating efficiently *all-pairs shortest paths* in a distributed network whose topology dynamically changes over the time, in the sense that links can change status during the lifetime of the network, is considered crucial in today's practical applications.

The algorithms for computing shortest-paths used in computer networks are classified as *distance-vector* and *link-state* algorithms. In a distance vector algorithm, as for example the classical Bellman-Ford method (originally introduced in the Arpanet [13]), a node knows the distance from each of its neighbors to every destination and uses this information to compute the distance and the next node in the shortest path to each destination. In a link-state algorithm, as for example the *OSPF* protocol widely used in the Internet (e.g., see [14]), a node must know the entire network topology to compute its distance to any network destination (usually running the centralized Dijkstra's algorithm for shortest paths).

The main drawbacks of distance-vector algorithms, when used in a dynamic environment, are the well-known *looping* and *count-to-infinity* phenomena (see, e.g., [4]) that lead in many cases to a very slow convergence. A loop is a path induced by the routing table entries, such that the path visits the same node more than once before reaching the intended destination. A node "counts to infinity" when it increments its distance to a destination until it reaches a predefined maximum distance value. Link-state algorithms are free of the looping and count-to-infinity problems. However, each node needs to receive up-to-date information on the entire network topology after a network change. This is achieved by broadcasting each change of the network topology to all nodes [14,17]. In any case, it is very important to find efficient dynamic distributed algorithms for shortest paths, since real networks are inherently dynamic and the recomputation from scratch can result very expensive in practice.

If the topology of a dynamic network is represented as a weighted undirected graph, then the typical update operations on that network can be modelled as insertions and deletions of edges and edge weight changes (weight decrease and weight increase). When arbitrary sequences of the above operations are allowed, we refer to the *fully dynamic problem*; if only *insert* and *weight decrease* (*delete* and *weight increase*) operations are allowed, then we refer to the *incremental* (*decremental*) problem.

A number of dynamic solutions for the distributed shortest paths problem have been proposed in the literature (e.g., see [8,9,10,12,15,16]). Most of these solutions are distance-vector algorithms that rely on the classical Bellman-Ford method which has been shown to converge to the correct distances if the edge weights stabilize and all cycles have positive lengths [4]. However, the convergence can be very slow (possibly infinite) in the case of *delete* and *weight increase* operations, due to the looping and count-to-infinity phenomena. Furthermore, if the nodes of the network are not synchronized, even though no change occurs in the network, the message complexity of the Bellman-Ford method is exponential in the size of the network (e.g., see [3]).

In this paper, we are interested in the practical case of a dynamic network in which an edge change can occur while another change is under processing. A processor v could be affected by both these changes. As a consequence, v could be involved in the *concurrent* executions related to both the changes. In general, it is possible to classify the algorithms known in the literature for the dynamic distributed shortest paths problem in the following two categories:

1. Algorithms which *are not* able to *concurrently* update shortest paths as those in [8,9,12,16]. In particular, algorithms that work under the assumption that before dealing with an edge operation, the algorithm for the previous operation has to be terminated. This is a limitation in real networks, where changes can occur in an unpredictable way.

2. Algorithms which *are* able to *concurrently* update shortest paths as those in [6,10], but they present one or more of the following drawbacks: they suffer of the looping and count-to-infinity phenomena; their convergence is very slow in the case of weight increase and delete operations; they are not able to work in the realistic fully dynamic case, where an arbitrary sequence of edge change operations can occur to the network in an unpredictable way.

In this paper, we make a step forward in the area of shortest paths routing by proposing a new algorithm which falls in the second category, and overcomes some of the limitations of the known solutions in this category. In fact, despite our algorithm still suffer of the looping and count-to-infinity phenomena, it heuristically reduces the cases where these phenomena occur, it is able to work in the realistic *fully dynamic* case, and it is experimentally better than the Bellman-Ford algorithm.

In order to experimentally evaluate our algorithm, we implemented it and the classical Bellman-Ford method in the OMNeT++ simulation environment [1]. Then, we performed several tests both on real-world data [11] and randomly generated graphs and update sequences. These experiments show that our algorithm performs better than the Bellman-Ford algorithm in terms of either number of messages sent or space occupancy per node.

Structure of the paper. The paper is organized as follows. In Section 2 we introduce some useful notation and the distributed asynchronous model used in the paper. In Section 3 we describe our new fully dynamic algorithm. In Section 4 we experimentally analize the algorithm and compare its performances against the well-known Bellman-Ford algorithm. Finally, in Section 5 we give some concluding remarks and outline possible future research directions.

2 Preliminaries

We consider a network made of processors linked through communication channels. Each processor can send messages only to its neighbors. We assume that messages are delivered to their destination within a finite delay but they might be delivered out of order. We consider an asynchronous system, that is, a sender of a message does not wait for the receiver to be ready to receive the message. Finally, there is no shared memory among the nodes of the network.

We represent the network by an undirected weighted graph $G = (V, E, w)$, where: V is a finite set of n nodes, one for each processor; E is a finite set of m edges, one for each communication channel; and w is a weight function $w : E \to \mathbb{R}^+ \cup \{\infty\}$. An edge in E that links nodes $u, v \in V$ is denoted as $u \to v$. Given $v \in V$, $N(v)$ denotes the set of neighbors of v and $deg(v)$ the degree of v. The maximum degree of the nodes in G is denoted by *maxdeg*. A path P in G between nodes u and v is denoted as $P = u \rightsquigarrow v$. We define the *length* of P as the number of edges of P and denote it by $\ell(P)$, and define the *weight* of P as the sum of the weights of the edges in P and denote it by $weight(P)$. A *shortest path* between nodes u and v is a path from u to v with the minimum weight. The *distance* from u to v is the weight of a shortest path from u to v, and is denoted as $d(u, v)$. Given two nodes $u, v \in V$, the *via* from u to v is the set of neighbors of u that belong to a shortest path from u to v. Formally: $via(u, v) \equiv \{z \in N(u) \mid d(u, v) = w(u, z) + d(z, v)\}$.

Given a graph $G = (V, E, w)$, we suppose that a sequence $\mathcal{C} = \{c_1, c_2, ..., c_k\}$ of k operations is performed on edges $(x_i, y_i) \in E$, $i \in \{1, 2, ..., k\}$. The operation c_i either inserts a new edge in G, or deletes an edge of G, or modifies (either increases or decreases) the weight of an existing edge in G. We consider the case in which $\mathcal{C} = \{c_1, c_2, ..., c_k\}$ is a sequence of *weight increase* and *weight decrease* operations, that is operation c_i either increases or decreases the weight of edge (x_i, y_i) by a quantity $\epsilon_i > 0$, $i \in \{1, 2, ..., k\}$. The extension to *delete* and *insert* operations, respectively, is straightforward: deleting an edge (x, y) is equivalent to increase $w(x, y)$ to $+\infty$, and inserting an edge (x, y) with weight α is equivalent to decrease $w(x, y)$ from $+\infty$ to α.

Assuming $G^0 \equiv G$, we denote as G^i the graph obtained by applying the operation c_i to G^{i-1}. We denote as $d^i()$ and $via^i()$ the distance and the via over G^i, $0 \leq i \leq k$, respectively. Given a path P in G, we denote as $weight^i(P)$ the weight of P in G^i, $0 \leq i \leq k$.

Asynchronous model. Given an asynchronous system, the model summarized below is based on that proposed in [2]. The *state* of a processor v is the content of the data structure at node v. The *network state* is the set of states of all the processors in the network plus the network topology and the edge weights. An *event* is the reception of a message by a processor or a change to the network state. When a processor p sends a message m to a processor q, m is stored in a buffer in q. When q reads m from its buffer and processes it, the event "reception of m" occurs. An *execution* is an alternate sequence (possibly infinite) of network states and events. A non negative real number is associated to each event, the *time* at which that event occurs. The time is a *global* parameter and is not accessible to the processors of the network. The times must be non decreasing and must increase without bound if the execution is infinite. Events are ordered according to the times at which they occur. Several events can happen at the same time as long as they do not occur on the same processor. This implies that the times related to a single processor are strictly increasing.

Concurrent executions. We consider a dynamic network in which a weight change can occur while another weight change is under processing. A processor v could be affected by both these changes. As a consequence, v could be involved in the executions related to both the changes. Hence, according to the asynchronous model described above we need to define the notion of *concurrent* executions as follows. Let us consider an algorithm A that maintains shortest paths on G after a weight change operation in $\mathcal{C}$. Given c_i and c_j in $\mathcal{C}$, we denote as: t_i and t_j the times at which c_i and c_j occur, respectively; $\mathcal{A}_i$ and $\mathcal{A}_j$ the executions of A related to c_i and c_j, respectively; and $t_{\mathcal{A}_i}$ the time when $\mathcal{A}_i$ terminates. If $t_i \leq t_j$ and $t_{\mathcal{A}_i} \geq t_j$, then $\mathcal{A}_i$ and $\mathcal{A}_j$ are *concurrent*, otherwise they are *sequential*.

3 The Fully Dynamic Algorithm

In this Section we describe our new fully dynamic solution for the concurrent update of distributed all-pairs shortest paths. Without loss of generality, we assume that operations in $\mathcal{C} = \{c_1, c_2, ..., c_k\}$ occur at times $t_1 \leq t_2 \leq ... \leq t_k$, respectively.

Data structures. Each node of G knows the identity of every other node of G, the identity of all its neighbors and the weights of the edges incident to it. The information on the shortest paths in G are stored in a data structure called *routing table* RT distributed over all nodes. Each node v maintains its own routing table $\mathtt{RT}_v[\cdot]$, that has one entry $\mathtt{RT}_v[s]$, for each $s \in V$. The entry $\mathtt{RT}_v[s]$ consists of two fields:

– $\mathtt{RT}_v[s].\mathtt{D}$, the estimated distance between nodes v and s in G;
– $\mathtt{RT}_v[s].\mathtt{VIA} \equiv \{v_i \in N(v) \mid \mathtt{RT}_v[s].\mathtt{D} = w(v, v_i) + \mathtt{RT}_{v_i}[s].\mathtt{D}\}$, the estimated via from v to s.

For sake of simplicity, we write $\mathtt{D}[v, s]$ and $\mathtt{VIA}[v, s]$ instead of $\mathtt{RT}_v[s].\mathtt{D}$ and $\mathtt{RT}_v[s].\mathtt{VIA}$, respectively. Furthermore, in what follows we denote as $\mathtt{D}_t[v, s]$ and $\mathtt{VIA}_t[v, s]$ the value of the data structures at time t; we simply write $\mathtt{D}[v, s]$ and $\mathtt{VIA}[v, s]$ when the time is clear by the context.

Given a destination s the set $\mathtt{VIA}[v, s]$ contains at most $deg(v)$ elements. Hence, each node v requires $O(n \cdot deg(v))$ space and the space complexity is hence $O(maxdeg \cdot n)$ per node.

Algorithm. The proposed fully dynamic algorithm is reported in Fig. 1, 2 and 3, and is described in what follows with respect to a source $s \in V$. Before the algorithm starts, we assume that, for each $v, s \in V$ and for each $t < t_1$, $\mathtt{D}_t[v, s]$ and $\mathtt{VIA}_t[v, s]$ are correct, that is $\mathtt{D}_t[v, s] = d^0(v, s)$ and $\mathtt{VIA}_t[v, s] = via^0(v, s)$. The algorithm starts at each t_i, $i \in \{1, 2, ..., k\}$. The event related to operation c_i on edge $x_i \to y_i$ is detected only by nodes x_i and y_i. As a consequence, if c_i is a *weight increase* (*weight decrease*) operation, x_i sends the message $increase(x_i, s)$ ($decrease(x_i, s, \mathtt{D}_{t_i}[x_i, s])$) to y_i and y_i sends the message $increase(y_i, s)$ ($decrease(y_i, s, \mathtt{D}_{t_i}[y_i, s])$) to x_i, for each $s \in V$.

If an arbitrary node v receives the message $decrease(u, s, \mathtt{D}[u, s])$, then it performs Procedure Decrease in Fig. 1. Basically, Decrease performs a relaxation of edge (u, v). In particular, if $w(v, u) + \mathtt{D}[u, s] < \mathtt{D}[v, s]$ (Line 1), then v needs to update its estimated distance to s. To this aim, v performs phase improve-table, that updates the data structures $\mathtt{D}[v, s]$ and $\mathtt{VIA}[v, s]$ (Lines 3–4), and propagates the updated values to the nodes in $N(v)$ (Line 6). Otherwise, if $w(v, u) + \mathtt{D}[u, s] = \mathtt{D}[v, s]$ (Line 9), then u is a new estimated via for v wrt destination s, and hence v performs phase extend-via, that simply adds u to $\mathtt{VIA}[v, s]$ (Line 10).

If a node v receives the message $increase(u, s)$, then it performs Procedure Increase in Fig. 2. While performing Increase, v simply checks whether the message comes from a node in $\mathtt{VIA}[v, s]$ (Line 1). In the affirmative case, v needs

```
Event: node $v$ receives the message $decrease(u, s, \mathtt{D}[u, s])$ from $u$
Procedure DECREASE
1.  if $w(v, u) + \mathtt{D}[u, s] < \mathtt{D}[v, s]$ then
2.    begin   $\boxed{\text{Lines 2-7: phase IMPROVE-TABLE}}$
3.      $\mathtt{D}[v, s] := w(v, u) + \mathtt{D}[u, s]$
4.      $\mathtt{VIA}[v, s] := \{u\}$
5.      for each $v_i \in N(v)$ do
6.        send $decrease(v, s, \mathtt{D}[v, s])$ to $v_i$
7.    end
8.  else
9.    if $\mathtt{D}[v, s] = w(v, u) + \mathtt{D}[u, s]$ then
10.     $\mathtt{VIA}[v, s] := \mathtt{VIA}[v, s] \cup \{u\}$   $\boxed{\text{Line 10: phase EXTEND-VIA}}$
```

Fig. 1.

```
Event: node $v$ receives the message $increase(u, s)$ from $u$
Procedure INCREASE
1.  if $u \in \mathtt{VIA}[v, s]$ then
2.    begin
3.      $\mathtt{VIA}[v, s] := \mathtt{VIA}[v, s] \setminus \{u\}$   $\boxed{\text{Line 3: phase REDUCE-VIA}}$
4.      if $\mathtt{VIA}[v, s] \equiv \emptyset$ then
5.        begin   $\boxed{\text{Lines 5-17: phase REBUILD-TABLE}}$
6.          $\mathtt{old\_distance} := \mathtt{D}[v, s]$
7.          for each $v_i \in N(v)$ do
8.            receive $\mathtt{D}[v_i, s]$ by sending $get\text{-}dist(v, s)$ to $v_i$
9.          $\mathtt{D}[v, s] := \min_{v_i \in N(v)} \{w(v, v_i) + \mathtt{D}[v_i, s]\}$
10.         $\mathtt{VIA}[v, s] := \{v_i \in N(v) \mid w(v, v_i) + \mathtt{D}[v_i, s] = \mathtt{D}[v, s]\}$
11.         for each $v_i \in N(v)$ do
12.           begin
13.             if $\mathtt{D}[v, s] > \mathtt{old\_distance}$ then
14.               send $increase(v, s)$ to $v_i$
15.             send $decrease(v, s, \mathtt{D}[v, s])$ to $v_i$   $\boxed{\text{LRH}_2}$
16.           end
17.       end
18.   end
```

Fig. 2.

to remove u from $\mathtt{VIA}[v, s]$. To this aim, v performs phase REDUCE-VIA (Line 3). As a consequence of this deletion, $\mathtt{VIA}[v, s]$ may become empty. In this case, v performs phase REBUILD-TABLE, whose purpose is to compute the new estimated distance and via of v to s. To do this, v asks to each node $v_i \in N(v)$ for its current estimated distance, by sending v_i message *get-dist*(v, s) (Lines 7–8).

```
Event: node $v_i$ receives the message get-dist$(v, s)$ from $v$
Procedure DIST
1.   if $(\texttt{VIA}[v_i, s] \equiv \{v\})$ $\boxed{\text{LRT}_1}$ or
         ($v_i$ is performing REBUILD-TABLE or IMPROVE-TABLE wrt destination $s$) $\boxed{\text{LRT}_2}$
2.   then send $\infty$ to $v$ $\boxed{\text{LRH}_1}$
3.   else send $\texttt{D}[v_i, s]$ to $v$
```

Fig. 3.

When v_i receives message *get-dist*(v, s) by v, it performs Procedure DIST in Fig. 3. While performing DIST, v_i basically sends $\texttt{D}[v_i, s]$ to v, unless one of the following two conditions holds: 1) $\texttt{VIA}[v_i, s] \equiv \{v\}$; 2) v_i is performing REBUILD-TABLE or IMPROVE-TABLE wrt destination s. The test of these two conditions is part of our strategy to reduce the cases in which the looping and count-to-infinity phenomena appear. The test is performed at Line 1 of DIST, where the conditions are labelled as LRT_1 and LRT_2, respectively (the acronym stands for Loop Reducing Test). If LRT_1 or LRT_2 are true, then v_i sends ∞ to v. This action is performed at Line 2, and is labelled as LRH_1 (the acronym stands for Loop Reducing Heuristic).

Once node v has received the answers to the *get-dist* messages by all its neighbors, it computes the new estimated distance and via to s (Lines 9–10). Now, if the estimated distance has been increased, v sends an *increase* message to its neighbors (Line 14). In any case, v sends to its neighbors the message *decrease* (Line 15), to communicate them $\texttt{D}[v, s]$. This action, that we call LRH_2, is also part of our strategy to reduce the looping and count-to-infinity phenomena. In fact, at some point, as a consequence of LRH_1, v could have sent ∞ to a neighbor v_j. Then, v_j receives the message sent by v at Line 15, and it performs Procedure DECREASE to check whether $\texttt{D}[v, s]$ can determine an improvement to the value of $\texttt{D}[v_j, s]$.

The correctness of the algorithm is stated in the next theorem, whose proof is given in [7].

Theorem 1. *There exists t_F such that, for each pair of nodes $v, s \in V$, and for each time $t \geq t_F$, $\texttt{D}_t[v, s] = d^k(v, s)$ and $\texttt{VIA}_t[v, s] \equiv via^k(v, s)$.*

4 Experimental Analysis

Experimental environment. The experiments have been carried out on a workstation equipped with a 2,66 GHz processor (Intel Core2 Duo E6700 Box) and 8Gb RAM.

The experiments consist of simulations within the OMNeT++ environment, version 4.0p1 [1]. OMNeT++ is an object-oriented modular discrete event network simulator, useful to model protocols, telecommunication networks, multiprocessors and other distributed systems. An OMNeT++ model consists of hierarchically nested modules, that communicate through message passing.

In our model, we defined a basic module *node* to represent a node in the network. A node v has a communication *gate* with each node in $N(v)$. Each node can send messages to a destination node through a *channel* which is a module that connects gates of different nodes (both gate and channel are OMNeT++ predefined modules). In our model, a channel connects exactly two gates and represents an edge between two nodes. We associate two parameters per channel: a *weight* and a *delay.* The former represents the cost of the edge in the graph, and the latter simulates a finite but not null transmission time.

Implemented algorithms. We implemented the algorithm described in Section 3, denoted as ConFu. In order to compare its performances with respect to algorithms used in practice, we also implemented the well known Bellman-Ford algorithm [4] which is denoted as BF. In BF, a node v updates its estimated distance to a node s, by simply executing the iteration $\mathtt{D}[v,s] := \min_{u \in N(v)}\{w(v,u) + \mathtt{D}[u,s]\}$, using the last estimated distance $\mathtt{D}[u,s]$ received from a neighbor $u \in N(v)$ and the latest status of its links. Eventually, node v transmits its new estimated distance to nodes in $N(v)$. BF requires $O(n \cdot \mathit{maxdeg})$ space per node to store the last estimated distance vector $\{\mathtt{D}[u,s] \mid s \in V\}$ received from each neighbor $u \in N(v)$.

Executed tests. For our experiments we used both real-world and artificial instances of the problem. In detail, we used the CAIDA IPv4 topology dataset [11] and Erdös-Rényi random graphs [5].

CAIDA (Cooperative Association for Internet Data Analysis) is an association which provides data and tools for the analysis of the Internet infrastructure. The CAIDA dataset is collected by a globally distributed set of monitors. The monitors collect data by sending probe messages continuously to destination IP addresses. Destinations are selected randomly from each routed IPv4/24 prefix on the Internet such that a random address in each prefix is probed approximately every 48 hours. The current prefix list includes approximately 7.4 million prefixes. For each destination selected, the path from the source monitor to the destination is collected, in particular, data collected for each path probed includes the set of IP addresses of the hops which form the path and the Round Trip Times (RTT) of both intermediate hops and the destination.

We parsed the files provided by CAIDA to obtain a weighted undirected graph G_{IP} where a node represents an IP address contained in the dataset (both source/destination hosts and intermediate hops), edges represent links among hops and weights are given by RTTs.

As the graph G_{IP} consists of $n \approx 35000$ nodes, we cannot use it for the experiments, as the amount of memory required to store the routing tables of all the nodes is $O(n^2 \cdot \mathit{maxdeg})$. Hence, we performed our tests on connected subgraphs of G_{IP} induced by the settled nodes of a breadth first search starting from a node taken at random. We generated a set of different tests, each test consists of a dynamic graph characterized by: a subgraph of G_{IP} of 5000 nodes, a set of k concurrent edge updates, where k assumes values in $\{5, 10, \ldots, 100\}$. An edge update consists of multiplying the weight of a random selected edge by a percentage value randomly chosen in $[50\%, 150\%]$. For each test configuration, we performed 5 different experiments and we report average values.

The graph G_{IP} turns out to be very sparse (i.e. $m/n \approx 1.3$), so it is worth analyzing also dense graphs. To this aim we generated Erdös-Rényi random graphs. In detail, we randomly generated a set of different tests, where a test consists of a dynamic graph characterized by: an Erdös-Rényi random graphs G_{rand} of 1000 nodes; the density *dens* of the graph, computed as the ratio between m and the number of the edges of the n-complete graph; and the number k of edge update operations. We chosen different values of *dens* ranging from 0.01 to 0.41. The number k assumes values in $\{30, 100, 1000\}$. Edge weights are non-negative real numbers randomly chosen in $[1, 10000]$. Edge updates are randomly chosen as in the CAIDA tests. For each test configuration, we performed 5 different experiments and we report average values.

Analysis. The results of our experiments are shown in Fig. 4, 5, and 6. In Fig. 4 (left), we report the number of messages sent by algorithms ConFu and BF on subgraphs of G_{IP} having 5000 nodes and an everage value of 6109 edges in the cases where the number k of modifications is in $\{5, 10, 15, 20\}$. Fig. 4 (left) shows that ConFu always performs better than BF. In particular, it always sends less messages than BF. The tests for $k \in \{25, 30, \ldots, 100\}$ are not reported as in these cases we experimentally checked that BF always falls in looping and then the number of messages is unbounded, while ConFu always converges to the correct routing tables. Fig. 4 (right) shows the same results as Fig. 4 (left) from a different point of view, that is, it shows the ratio between the number of messages sent by BF and ConFu. It is worth noting that the ratio is within 21 and 230.

To conclude our analysis on G_{IP}, we experimentally analyze the space occupancy per node of ConFu and BF. The latter requires a node v to store, for each destination, the estimated distance given by each of its neighbors, while ConFu only needs the estimated distance of v and the set `VIA`, for each destination. Since in these sparse graphs it is not common to have more than one

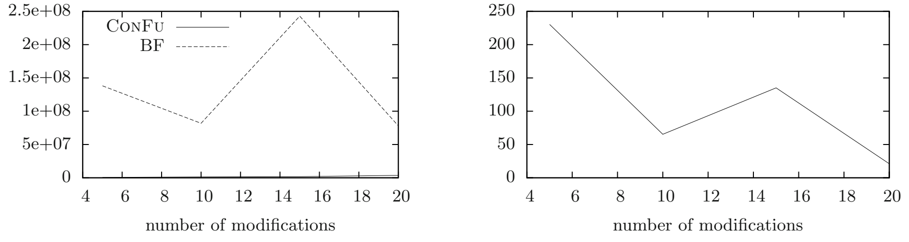
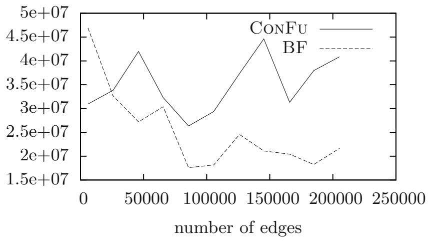


Fig. 4. Left: Number of messages sent by ConFu and BF on subgraphs of G_{IP}. Right: Ratio between the number of messages sent by BF and ConFu on subgraphs of G_{IP}.



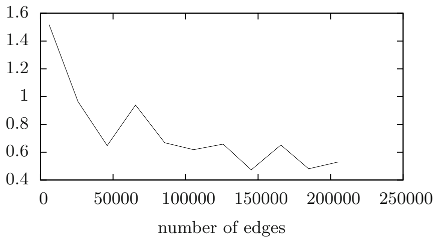
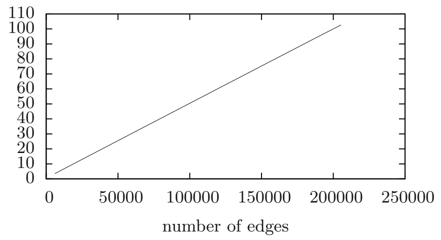


Fig. 5. Left: Number of messages sent by ConFu and BF on graphs G_{rand}. Right: Ratio between the number of messages sent by BF and ConFu on graphs G_{rand}.

via to a destination, the memory requirement of ConFu is much smaller than that of BF. In particular, ConFu requires in average 40000 bytes per node and 40084 bytes per node in the worst case. BF requires in average 44436 bytes per node and $4M$ bytes per node in the worst case. This implies that ConFu is in average 1.11 times more space efficient than BF and it is 99.79 times more space efficient than BF in the worst case.

The good performances of ConFu are mainly due to the sparsity of G_{IP}. In fact, ConFu uses three kind of messages: *decrease*, *increase* and *get-dist*. Messages *decrease* and *increase* are sent only when a node v changes its routing table and they are used to propagate this change, while *get-dist* is used by v in order to know the estimated distances of its neighbors. Hence, the number of *get-dist* messages is proportional to the average node degree of a graph. Note that, BF does not need to use *get-dist* messages as it stores, for each node, the estimated distances of its neighbors. Hence, in sparse graphs, where the average degree of a graph is small, the number of *get-dist* messages sent by ConFu is also small and this implies that, in these cases, ConFu sends less messages than BF.

By the above discussion, it is worth investigating how the two algorithms perform when the graph is dense. Fig. 5 (left) shows the number of messages sent by algorithms ConFu and BF on dynamic Erdös-Rényi random graphs with 1000 nodes, 1000 edge weight changes and *dens* ranging from 0.01 to 0.41 which leads to a number m of edges which ranges from about 5000 to about 200000. The number of messages sent by ConFu is less than the number of messages sent by BF when the number of edges is less than 20000. In most of the cases when the number of edges is more than 20000, BF is better than ConFu. This is due to the fact that ConFu does not require a node to store the estimated distances of its neighbors but it sends a *get-dist* message to each neighbor (see Line 8 of Procedure Increase). Hence, the number of *get-dist* messages is high when the average number of neighbors is high. Contrarily, BF does not need to send such messages as it stores for each node v the estimated distances of each neighbor of v. This implies an increase in the space occupancy of BF as highlighted by Fig. 6. In detail, Fig. 6 (left) shows the ratio between the average space occupancy per node required by BF and ConFu in G_{rand}, while Fig. 6 (right) shows the ratio between the worst case space occupancy per node required by BF and ConFu. The average space occupancy ratio grows linearly



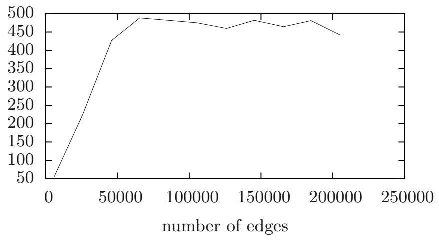


Fig. 6. Ratio between the space required by BF and ConFu on graphs G_{rand} in the average case (left) and in the worst case (right)

with the number of edges as the space occupancy of ConFu remains almost constant while the space occupancy of BF is proportional to the average node degree. The worst case space occupancy of BF grows very fast as in the executed tests where $dens > 0.10$ there exists at least a node v such that $deg(v) = n - 1$.

A different point of view is given in Fig. 5 (right) which shows the ratio between the number of messages sent by BF and ConFu. Note that, the ratio is about 1.5 in the sparse graphs and it decreases until it assumes a value of smaller than 1 for dense graphs.

Fig. 5 and 6 refer to the case where $k = 1000$, as it is the case where ConFu performs worse. In cases where $k = 30, 100$ ConFu performs better than BF and hence they are not reported.

5 Conclusion and Future Work

Most of the solutions known in the literature for the dynamic distributed *all-pairs shortest paths* problem suffer of three main drawbacks: they are not able to update shortest paths concurrently; they suffer of the looping and counting-to-infinity phenomena thus having a slow convergence; they are not suitable to work in the realistic fully dynamic case. In this paper we provide a new *fully dynamic* solution that overcomes most of the above problems and that behaves better than the well-known Bellman-Ford algorithm in practical cases.

A research line that deserve investigation is the evaluation of the new algorithm from an experimental point of view against other concurrent solution known in the literature as for example that given in [9].

Acknowledgments

Support for the IPv4 Routed/24 Topology Dataset is provided by National Science Foundation, US Department of Homeland Security, WIDE Project, Cisco Systems, and CAIDA.

References

1. Omnet++: the discrete event simulation environment, http://www.omnetpp.org/
2. Attiya, H., Welch, J.: Distributed Computing. John Wiley and Sons, Chichester (2004)
3. Awerbuch, B., Bar-Noy, A., Gopal, M.: Approximate distributed bellman-ford algorithms. IEEE Transactions on Communications 42(8), 2515–2517 (1994)
4. Bertsekas, D., Gallager, R.: Data Networks. Prentice Hall International, Englewood Cliffs (1992)
5. Bollobás, B.: Random Graphs. Cambridge University Press, Cambridge (2001)
6. Cicerone, S., D'Angelo, G., Di Stefano, G., Frigioni, D.: Partially dynamic efficient algorithms for distributed shortest paths. Theoretical Computer Science 411, 1013–1037
7. Cicerone, S., D'Angelo, G., Di Stefano, G., Frigioni, D., Maurizio, V.: A new fully dynamic algorithm for distributed shortest paths and its experimental evaluation. Technical Report R.10.111, Department of Electrical and Information Engineering, University of L'Aquila (2010)
8. Cicerone, S., Di Stefano, G., Frigioni, D., Nanni, U.: A fully dynamic algorithm for distributed shortest paths. Theoretical Computer Science 297(1-3), 83–102 (2003)
9. Garcia-Lunes-Aceves, J.J.: Loop-free routing using diffusing computations. IEEE/ACM Transactions on Networking 1(1), 130–141 (1993)
10. Humblet, P.A.: Another adaptive distributed shortest path algorithm. IEEE Transactions on Communications 39(6), 995–1002 (1991)
11. Hyun, Y., Huffaker, B., Andersen, D., Aben, E., Shannon, C., Luckie, M., Claffy, K.: The CAIDA IPv4 routed/24 topology dataset, http://www.caida.org/data/active/ipv4_routed_24_topology_dataset.xml
12. Italiano, G.F.: Distributed algorithms for updating shortest paths. In: Toueg, S., Kirousis, L.M., Spirakis, P.G. (eds.) WDAG 1991. LNCS, vol. 579, pp. 200–211. Springer, Heidelberg (1992)
13. McQuillan, J.: Adaptive routing algorithms for distributed computer networks. Technical Report BBN Report 2831, Cambridge, MA (1974)
14. Moy, J.T.: OSPF - Anatomy of an Internet routing protocol. Addison-Wesley, Reading (1998)
15. Orda, A., Rom, R.: Distributed shortest-path and minimum-delay protocols in networks with time-dependent edge-length. Distributed Computing 10, 49–62 (1996)
16. Ramarao, K.V.S., Venkatesan, S.: On finding and updating shortest paths distributively. Journal of Algorithms 13, 235–257 (1992)
17. Rosen, E.C.: The updating protocol of arpanet's new routing algorithm. Computer Networks 4, 11–19 (1980)

Contraction of Timetable Networks with Realistic Transfers[⋆]

Robert Geisberger

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
geisberger@kit.edu

Abstract. We contribute a fast routing algorithm for timetable networks with realistic transfer times. In this setting, our algorithm is the first one that successfully applies precomputation based on node contraction: gradually removing nodes from the graph and adding shortcuts to preserve shortest paths. This reduces query times to 0.5 ms with preprocessing times below 4 minutes on all tested instances, even on continental networks with 30 000 stations. We achieve this by an improved contraction algorithm and by using a station graph model. Every node in our graph has a one-to-one correspondence to a station and every edge has an assigned collection of connections. Also, our graph model does not require parallel edges.

Keywords: route planning; public transit; algorithm engineering.

1 Introduction

Route planning is one of the showpieces of algorithm engineering. Many hierarchical route planning algorithms have been developed over the past years and are very successful on static road networks (overview in [1]). Recently, Contraction Hierarchies (CH) [2] provided a particularly simple approach with fast preprocessing and query times. CH is solely based on the concept of *node contraction*: removing "unimportant" nodes and adding shortcuts to preserve shortest path distances. One year later, *time-dependent* CH (TCH) [3] was published and works well for time-dependent road networks, but *completely fails* for timetable networks of public transportation systems. In this paper, we show how to adapt CH successfully to timetable networks with realistic transfers, i. e. minimum transfer times. The positive outcome is partly due to our *station graph model*, where each station is a single node and no parallel edges between stations are necessary. Additionally, we change the contraction algorithm significantly and deal with special cases of timetable networks, e. g. loops.

[⋆] Partially supported by DFG grant SA 933/5-1, and the 'Concept for the Future' of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

1.1 Related Work

Public transportation networks have always been time-dependent, i. e. travel times depend on the availability of trains, buses or other vehicles. That makes them naturally harder than road networks, where simple models can be independent of the travel time and still achieve good results. There are two intensively studied models for modeling timetable information: the *time-expanded* [4,5,6], and the so-called *time-dependent* model[1] [7,8,9,10]. Both models answer queries by applying some shortest-path algorithm to a suitably constructed graph. In the time-expanded model, each node corresponds to a specific time event (departure or arrival), and each edge has a constant travel time. In the time-dependent model, each node corresponds to a station, and the costs on an edge are assigned depending on the time in which the particular edge will be used by the shortest-path algorithm.

To model more realistic transfers in the time-dependent model, [7] propose to model each platform as a separate station and add walking links between them. [11] propose a similar extension for constant and variable transfer and describe it in more detail. Basically, the stations are expanded to a *train-route graph* where no one-to-one correspondence between nodes and stations exists anymore. A *train route* is the maximal subset of trains that follow the exact same route, at possibly different times and do not overtake each other. Each train route has its own node at each station. Those are interconnected within a station with the given transfer times. This results in a significant blowup in the number of nodes and creates a lot of redundancy information that is collected during a query. Recently, [12,13] independently proposed a model that is similar to ours. They call it the *station graph* model and mainly use it to compute all Pareto-optimal paths in a fully realistic scenario. For unification, we will give our model the same name although there are some important differences in the details. The most significant differences are that (1) they require parallel edges, one for each train route and (2) their query algorithm computes connections per incoming edge instead per node. Their improvement over the time-dependent model was mainly that they compare all connections at a station and remove dominated ones.

Speed-up techniques are very successful when it comes to routing in time-dependent road networks, see [14] for an overview. However, timetable networks are very different from road networks [15] and there is only little work on speed-up techniques for them. Goal-directed search (A*) brings basic speed-up [16,17,11,18]. Time-dependent SHARC [19] brings better speed-up by using arc flags in the same scenario as we do and achieves query times of 2 ms but with preprocessing times of more than 5 hours[2]. Based on the station graph model, [13] also applied some speed-up techniques, namely arc flags that are valid on time periods and route contraction. They could not use node contraction because there

[1] Note that the time-dependent model is a special technique to model the time-dependent information rather than an umbrella term for all these models.

[2] We scaled timings by a factor of 0.42 compared to [14] based on plain Dijkstra timings.

were too many parallel edges between stations. Their preprocessing time is over 33 CPU hours resulting in a full day profile query time of more than 1 second (speed-up factor 5.2). These times are 2-3 orders of magnitude slower than ours but a comparison is not possible since they use a fully realistic bi-criteria scenario with footpaths, traffic days and graphs that are not available to us.

2 Preliminaries

We propose a model that is similar to the realistic time-dependent model introduced in [11], but we keep a one-to-one mapping between nodes in the graph and real stations.

A *timetable* consists of data concerning: *stations* (or bus stops, ports, etc), *trains* (or buses, ferries, etc), connecting stations, *departure* and *arrival times* of trains at stations, and *traffic days*. More formally, we are given a set of stations $\mathcal{B}$, a set of *stop events* $\mathcal{Z}_{\mathcal{S}}$ per station $S \in \mathcal{B}$, and a set of *elementary connections* $\mathcal{C}$, whose elements c are 6-tuples of the form $c = (Z_1, Z_2, S_1, S_2, t_d, t_a)$. Such a tuple (elementary connection) is interpreted as train that leaves station S_1 at time t_d after stop Z_1 and the *immediately next* stop is Z_2 at station S_2 at time t_a. If x denotes a tuple's field, then the notation of $x(c)$ specifies the value of x in the elementary connection c. A stop even is the consecutive arrival and departure of a train at a station, where no transfer is required. For the corresponding arriving elementary connection c_1 and the departing one c_2 holds $Z_2(c_1) = Z_1(c_2)$. If a transfer between some elementary connections c_1' and c_2' at station $S_2(c_1') = S_1(c_2')$ is required, $Z_2(c_1') \neq Z_1(c_2')$ must hold. We introduce additional stop events for the begin (no arrival) and the end (no departure) of a train.

The *departure* and *arrival times* $t_d(c)$ and $t_a(c)$ of an elementary connection $c \in \mathcal{C}$ within a day are integers in the interval $[0, 1439]$ representing time in minutes after midnight. Given two time values t and t', $t \leq t'$, the *cycledifference*(t, t') is the smallest nonnegative integer ℓ such that $\ell \equiv t' - t \pmod{1440}$. The *length* of an elementary connection c, denoted by *length*(c), is *cycledifference*$(t_d(c), t_a(c))$. We generally assume that trains operate daily but our model can be extended to work with traffic days. At a station $S \in \mathcal{B}$, it is possible to *transfer* from one train to another, if the time between the arrival and the departure at the station S is larger than or equal to a given, station-specific, *minimum transfer time*, denoted by *transfer*(S).

Let $P = (c_1, \ldots, c_k)$ be a sequence of elementary connections together with departure times $dep_i(P)$ and arrival times $arr_i(P)$ for each elementary connection c_i, $1 \leq i \leq k$. We assume that the times $dep_i(P)$ and $arr_i(P)$ also include day information to model trips that last longer than a day. Define $S_1(P) := S_1(c_1)$, $S_2(P) := S_2(c_k)$, $Z_1(P) := Z_1(c_1)$, $Z_2(P) := Z_2(c_k)$, $dep(P) := dep_1(P)$, and $arr(P) := arr_k(P)$. Such a sequence P is called a *consistent connection* from station $S_1(P)$ to $S_2(P)$ if it fulfills the following two consistency conditions: (1) the departure station of c_{i+1} is the arrival station of c_i; (2) the time values $dep_i(P)$ and $arr_i(P)$ correspond to the time values t_d and t_a, resp., of the elementary connections (modulo 1440) and respect the transfer times at stations.

Given a timetable, we want to solve the earliest arrival problem (EAP), i.e. to compute the earliest arriving consistent connection between given stations A and B departing not earlier than a specified time t_0. We refer to the algorithm that solves the EAP as *time query*. In contrast, a *profile query* computes an optimal set of all consistent connections independent of the departure time.

3 Station Graph Model

We introduce a model that represents a timetable as a directed graph $G = (\mathcal{B}, E)$ with exactly one node per station. For a simplified model without transfer times, this is like the time-dependent model. The novelty is that even with positive transfer times, we keep one node per station and require no parallel edges. The attribute of an edge $e = (A, B) \in E$ is a set of consistent connections $\mathit{fn}(e)$ that depart at A and arrive at B, usually all elementary connections. Here and in the following we assume that all connections are consistent. Previous models required that all connections of a single edge fulfill the FIFO-property, i. e. they do not overtake each other. In contrast, we do not require this property. So we can avoid parallel edges, as this is important for CH preprocessing. However, even for time queries, we need to consider multiple dominant arrival events per station.

We say that a connection P *dominates* a connection Q if we can replace Q by P (Lemma 1). More formally, let Q be a connection. Define $\mathit{parr}(Q)$ as the (**p**revious) **arr**ival arrival time of the train at station $S_1(Q)$ before it departs at time $\mathit{dep}(Q)$, or $\perp$ if this train begins there. If $\mathit{parr}(Q) \neq \perp$ then we call $\mathit{res}_d(Q) := \mathit{dep}(Q) - \mathit{parr}(Q)$ the *residence time at departure*. We call Q a *critical departure* when $\mathit{parr}(Q) \neq \perp$ and $\mathit{res}_d(Q) < \mathit{transfer}(S_1(Q))$. Symmetrically, we define $\mathit{ndep}(Q)$ as the (**n**ext) **dep**arture time of the train at station $S_2(Q)$, or $\perp$ if the train ends there. When $\mathit{ndep}(Q) \neq \perp$ then we call $\mathit{res}_a(Q) := \mathit{ndep}(Q) - \mathit{arr}(Q)$ the *residence time at arrival*. And Q is a *critical arrival* when $\mathit{ndep}(Q) \neq \perp$ and $\mathit{res}_a(Q) < \mathit{transfer}(S_2(Q))$.

A connection P *dominates* Q iff all of the following conditions are fulfilled:
(1) $S_1(P) = S_1(Q)$ and $S_2(P) = S_2(Q)$
(2) $\mathit{dep}(Q) \leq \mathit{dep}(P)$ and $\mathit{arr}(P) \leq \mathit{arr}(Q)$
(3) $Z_1(P) = Z_1(Q)$, or Q is not a critical departure, or $\mathit{dep}(P) - \mathit{parr}(Q) \geq \mathit{transfer}(S_1(P))$
(4) $Z_2(P) = Z_2(Q)$, or Q is not a critical arrival, or $\mathit{ndep}(Q) - \mathit{arr}(P) \geq \mathit{transfer}(S_2(P))$

Conditions (1),(2) are elementary conditions. Conditions (3),(4) are necessary to respect the minimum transfer times, when Q is a subconnection of a larger connection.

Given connection $R = (c_1, \ldots, c_k)$, we call a connection $(c_i, \ldots, c_j)$ with $1 \leq i \leq j \leq k$ a *subconnection* of R, we call it *prefix* iff $i = 1$ and *suffix* iff $j = k$.

Lemma 1. *A consistent connection P dominates a consistent connection Q iff for all consistent connections R with subconnection Q, we can* replace *Q by P to get a consistent connection R' with $\mathit{dep}(R) \leq \mathit{dep}(R') \leq \mathit{arr}(R') \leq \mathit{arr}(R)$.*

3.1 Time Query

In this section we describe our baseline algorithm to answer a time query (A, B, t_0). We use a Dijkstra-like algorithm on our station graph that stores labels with each station and incrementally corrects them. A *label* is a connection P stored as a tuple (Z_2, arr)[3], where Z_2 is the arrival stop event and arr is the arrival time including days. The source station is always A, the target station $S_2(P)$ is implicitly given by the station that stores this label. Furthermore, we only consider connections departing not earlier than t_0 at A and want to minimize the arrival time. As we do not further care about the actual departure time at A, we call such a connection *arrival connection.* We say that an arrival connection P *dominates* Q iff all of the following conditions are fulfilled:
(1) $S_2(P) = S_2(Q)$
(2) $arr(P) \leq arr(Q)$
(3) $Z_2(P) = Z_2(Q)$, or Q is not a critical arrival, or $ndep(Q) - arr(P) \geq transfer(S_2(P))$

Lemma 2 shows that dominant arrival connections are sufficient for a time query.

Lemma 2. *Let (A, B, t_0) be a time query. A consistent arrival connection P dominates a consistent arrival connection Q iff for all consistent arrival connections R with prefix Q, we can* replace *Q by P to get a consistent arrival connection R' with $arr(R') \leq arr(R)$.*

Our algorithm manages a set of dominant **a**rrival **c**onnections $ac(S)$ for each station S. The initialization of $ac(A)$ at the departure station A is a special case since we have no real connection to station A. That is why we introduce a special stop event $\perp$ and we start with the set $\{(\perp, t_0)\}$ at station A. Our query algorithm then knows that we are able to board all trains that depart not earlier than t_0. We perform a label correcting query that uses the minimum arrival time of the (new) connections as key of a priority queue. This algorithm needs two elementary operations: (1) *link*: We need to traverse an edge $e = (S, T)$ by linking a given set of arrival connections $ac(S)$ with the connections $fn(e)$ to get a new set of arrival connections to station T. (2) *minimum*: We need to combine the already existing arrival connections at T with the new ones to a dominant set. We found a solution to the EAP once we extract a label of station B from the priority queue, as Theorem 1 proves.

Theorem 1. *The time query in the station graph model solves the EAP.*

Proof. The query algorithm only creates consistent connections because link and minimum do so. Lemma 2 ensures that there is never a connection with earlier arrival time. The connections depart from station A not before t_0 by initialization. Since the length of any connection is non-negative, and by the order in the priority queue, the first label of B extracted from the priority queue represents a solution to the EAP.

[3] Such a label does not uniquely describe a connection but stores all relevant information for a time query.

The link and minimum operation dominate the runtime of the query algorithm. The most important part is a suitable order of the connections, primarily ordered by arrival time. The minimum operation is then mainly a linear merge operation, and the link operation uses precomputed intervals to look only at a small relevant subset of $fn(e)$. We gain additional speed-up by combining the link and minimum operation.

3.2 Profile Query

A profile query (A, B) is similar to a time query. However, we compute dominant connections $con(S)$ instead of dominant arrival connections. Also we cannot just stop the search when we remove a label of B from the priority queue for the first time. We are only allowed to stop the search when we know that we have a dominant set of *all* consistent connections between A and B. For daily operating trains, we can compute a maximum length for a set of connections and can use it to prune the search. The efficient implementations of the minimum and link operation are also more complex. Similar to a time query, we use a suitable order of the connections, primarily ordered by departure time. The minimum operation is an almost linear merge: we merge the connections in descending order and remove dominated ones. This is done with a sweep buffer that keeps all previous dominant connections that are relevant for the current departure time. The link operation, which links connections from station A to S with connections from station S to T, is more complex: in a nutshell, we process the sorted connections from A to S one by one, compute a relevant interval of connections from S to T as for the time query, and remove dominated connections using a sweep buffer like for the minimum operation.

4 Contraction Hierarchies (CH)

CH performs preprocessing based on node contraction to accelerate queries. Contracting a node (= station) v in the station graph removes v and all its adjacent edges from the graph and adds *shortcut edges* to preserve dominant connections between the remaining nodes. A shortcut edge bypasses node v and represents a set of whole connections. Practically, we contract one node at a time until the graph is empty. All original edges together with the shortcut edges form the result of the preprocessing, a *CH*.

4.1 Preprocessing

The most time consuming part of the contraction is the witness search: given a node v and an incoming edge (u, v) and an outgoing edge (v, w), is a shortcut between u and w necessary when we contract v? We answer this question usually by a one-to-many profile search from u omitting v (*witness search*). If we find for every connection of the path $\langle u, v, w\rangle$ a dominating connection (*witness*), we can omit a shortcut, otherwise we add a shortcut with all the connections that have not been dominated. To keep the number of profile searches small, we maintain a set of necessary shortcuts for each node v. They do not take a lot of space since timetable networks are much smaller than road networks. Then, the contraction of node v is reduced to just adding the stored shortcuts. Initially, we perform a one-to-many profile search from each node u and store with each neighbor v the necessary shortcuts (u, w) that bypass v. The search can be limited by the length of the longest potential shortcut connection from u. After the contraction, we need to update the stored shortcuts of the remaining nodes. The newly added shortcuts (u, w) may induce other shortcuts for the neighbors u and w. So we perform one forward profile search from u and add to w the necessary shortcuts (u, x) bypassing w. A backward profile search from w updates node u. To omit the case that two connections witness each other, we add a shortcut when the witness has the same length and is not faster. So at most two profile searches from each neighbor of v are necessary. When we add a new shortcut (u, w), but there is already an edge (u, w), we merge both edges using the minimum operation, so there are never parallel edges. Avoiding these parallel edges is important for the contraction, which performs worse on dense graphs. Thereby, we also ensure that we can uniquely identify an edge with its endpoints.

We also limit the number of hops and the number of transfers of a witness search. As observed in [2], this accelerates the witness search at the cost of potentially more shortcuts.

We could omit loops in static and time-dependent road networks. But for station graph timetable networks, loops are sometimes necessary when transfer times differ between stations. For example, assume there is a train $T1$: (station sequence) $A \rightarrow B \rightarrow C$ and another train $T2$: $C \rightarrow B \rightarrow D$. A large minimum transfer time at B and a small one at C can forbid the transfer from $T1$ to $T2$ at B but make it possible at C. Contracting station C requires a loop at station B to preserve the connection between A and D. These loops also make the witness computation and the update of the stored shortcuts more complex. A shortcut (u, w) for node v with loop (v, v) must not only represent the path $\langle u, v, w\rangle$, but also $\langle u, v, v, w\rangle$. So when we add a shortcut (v, v) during the contraction of another node, we need to recompute all stored shortcuts of node v.

The order in which the nodes are contracted is deduced from a node priority consisting of: (a) The edge quotient, the quotient between the amount of shortcuts added and the amount of edge removed from the remaining graph. (b) The hierarchy depth, an upper bound on the amount of hops that can be performed in the resulting hierarchy. Initially, we set depth$(u) = 0$ and when a node v is contracted, we set depth(u) = max(depth(u),depth(v)+1) for all neighbors u. We weight (a) with 10 and (b) 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 by computing independent node sets with a 2-neighborhood [20]. Also note that [2,3] perform a simulated contraction of a node to compute its edge quotient. [20] improves this by caching witnesses, but still needs to perform a simulated contraction when a shortcut is necessary. We can omit this due to our stored sets of necessary shortcuts.

Interestingly, we cannot directly use the algorithms used for time-dependent road networks [3]. We tried using the time-dependent model for the timetable networks, but too many shortcuts were added, especially a lot of shortcuts between the different train-route nodes of the same station pair occur.[4] Additionally, [3] strongly base their algorithm on min-max search that only uses the time-independent min./max. length of an edge to compute upper and lower bounds. However, in timetable networks, the max. travel time for an edge is very high, e. g. when there is no service during the night. So the computed upper bounds are too high to bring any exploitable advantages. Without min-max search, the algorithm of [3] is drastically less efficient, i. e. the preprocessing takes days instead of minutes.

4.2 Query

Our query is a bidirectional Dijkstra-like query in the CH. A directed edge (v, w), where w is contracted after v, is an *upward* edge, otherwise a *downward* edge. Our forward search only relaxes upward edges and our backward search only downward edges [2]. The node contraction ensures the correctness of the search.

For a CH time query, we do not know the arrival time at the target node. We solve this by marking all downward edges that are reachable from the target node. The standard time query algorithm, using only upward edges and the marked downward edges, solves the EAP. The CH profile query is based on the standard profile query algorithm. Note that using further optimizations that work for road networks (stall-on-demand, min-max search) [3] would even slowdown our query.

5 Experiments

Environment. The experimental evaluation was done on one core of a Intel Xeon X5550 processors (Quad-Core) clocked at 2.67 GHz with 48 GiB of RAM[5] and 2x8MiB 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.

Test Instances. We have used real-world data from the European railways. The network of the long distance connections of Europe (`eur-longdist`) is from the winter period 1996/97. The network of the local traffic in Berlin/Brandenburg (`ger-local1`) and of the Rhein/Main region in Germany (`ger-local2`) are from the winter period 2000/01. The sizes of all networks are listed in Table 1.

Results. We selected 1 000 random queries and give average performance measures. We compare the time-dependent model and our new station model using a simple unidirectional Dijkstra algorithm in Table 2. Time queries have a good query time speed-up above 4.5 and even more when compared to the number of

[4] We tried to merge train-route nodes but this brought just small improvements.

[5] We never used more than 556 MiB of RAM, reported by the kernel.

Table 1. Network sizes and number of nodes and edges in the graph for each model

network	stations	trains/ buses	elementary connections	time-dependent nodes	time-dependent edges	station based nodes	station based edges
`eur-longdist`	30 517	167 299	1 669 666	550 975	1 488 978	30 517	88 091
`ger-local1`	12 069	33 227	680 176	228 874	599 406	12 069	33 473
`ger-local2`	9 902	60 889	1 128 465	167 213	464 472	9 902	26 678

Table 2. Performance of the station graph model compared to the time-dependent model on plain Dijkstra queries. We report the total *space*, the *#delete mins* from the priority queue, query *times*, and the *speed-up* compared to the time-dependent model.

			TIME-QUERIES				PROFILE-QUERIES			
network	model	space [MiB]	#delete mins	spd up	time [ms]	spd up	#delete mins	spd up	time [ms]	spd up
`eur-longdist`	time-dep.	27.9	259 506	1.0	54.3	1.0	1 949 940	1.0	1 994	1.0
	station	48.3	14 504	17.9	9.4	5.8	48 216	40.4	242	8.2
`ger-local1`	time-dep.	11.3	112 683	1.0	20.9	1.0	1 167 630	1.0	1 263	1.0
	station	19.6	5 969	18.9	4.0	5.2	33 592	34.8	215	5.9
`ger-local2`	time-dep.	10.9	87 379	1.0	16.1	1.0	976 679	1.0	1 243	1.0
	station	29.3	5 091	17.2	3.5	4.6	27 675	35.3	258	4.8

delete mins. However, since we do more work per delete min, this difference is expected. Profile queries have very good speed-up around 5 to 8 for all tested instances. Interestingly, our speed-up of the number of delete mins is even better than for time queries. We assume that more re-visits occur since there are often parallel edges between a pair of stations represented by its train-route nodes. Our model does not have this problem since we have no parallel edges and each station is represented by just one node. It is not possible to compare the space consumption per node since the number of nodes is in the different models different. So we give the absolute memory footprint: it is so small that we did not even try to reduce it, altough there is some potential.

Before we present our results for CH, we would like to mention that we were unable to contract the same networks in the time-dependent model. The contraction took days and the average degree in the remaining graph exploded. Even when we contracted whole stations with all of its route nodes at once, it did not work. It failed since the necessary shortcuts between all the train-route nodes multiplied quickly. So we developed the station graph model to fix these problems. Table 3 shows the resulting preprocessing and query performance. We get preprocessing times between 3 to 4 minutes using a hop limit of 7. The number of transfers is limited to the maximal number of transfers of a potential shortcut + 2. These timings are exceptional low (minutes instead of hours) compared to previous publications [19,13] and reduce time queries below 550 μs for all tested instances. CH work very well for `eur-longdist` where we get speed-ups of more than 37 for time queries and 65 for profile queries. When we multiply the speed-up of the comparison with the time-dependent model, we even get a speed-up

Table 3. Performance of CH. We report the preprocessing *time*, the *space* overhead and the increase in edge count. For query performance, we report the *#delete mins* from the priority queue, query *times*, and the *speed-up* over a plain Dijkstra (Table 2).

		PREPROCESSING			TIME-QUERIES				PROFILE-QUERIES			
network	hop-limit	time [s]	space [MiB]	edge inc.	#del. mins	spd up	time [μs]	spd up	#del. mins	spd up	time [ms]	spd up
`eur-longdist`	7	210	45.7	88%	192	75.7	251	37.5	260	186	3.7	65.1
	15	619	45.3	86%	183	79.3	216	43.5	251	192	3.4	71.4
`ger-local1`	7	216	27.9	135%	207	28.8	544	7.3	441	76	27.0	8.0
	15	685	26.9	128%	186	32.1	434	9.2	426	79	24.2	8.9
`ger-local2`	7	167	36.0	123%	154	33.1	249	14.0	237	117	9.5	27.1
	15	459	35.0	117%	147	34.6	217	16.1	228	121	8.2	31.3

of 218 (time) and 534 (profile) respectively. These speed-ups are one order of magnitude larger than previous speed-ups [19]. The network `ger-local2` is also suited for CH, the ratio between elementary connections and stations is however very high, so there is more work per settled node. More difficult is `ger-local1`; in our opinion, this network is less hierarchically structured. We see that on the effect of different hop limits for precomputation. (We chose 7 as a hop limit for fast preprocessing and then selected 15 to show further tradeoff between preprocessing and query time.) The smaller hop limit increases time query times by about 25%, whereas the other two networks just suffer an increase of about 16%. So important witnesses in `ger-local1` contain more edges, indicating a lack of hierarchy.

We do not really have to worry about preprocessing space since those networks are very small. The number of edges roughly doubles for all instances. We observe similar results for static road networks [2], but there we can save space with bidirectional edges. But in timetable networks, we do not have bidirectional edges with the same weight, so we need to store them separately. CH on timetable networks are inherently space efficient as they are event-based, they increase the memory consumption by not more than a factor 2.4 (`ger-local1`: 19.6 MiB $\rightarrow$ 47.5 MiB). In contrast, CH time-dependent road networks are not event-based and get very complex travel time functions on shortcuts, leading to an increased memory consumption (Germany midweek: 0.4 GiB $\rightarrow$ 4.4 GiB). Recent work reduces the space consumption by using approximations to answer queries exactly [21].

6 Conclusions

Our work has two contributions. First of all the station graph model, which has just one node per station, is clearly superior to the time-dependent model for the given scenario. Although the link and minimum operations are more expensive, we are still faster than in the time-dependent model since we need to execute them less often. Also all known speed-up techniques that work for the time-dependent model should work for our new model. Most likely, they even work better since the hierarchy of the network is more visible because of the one-to-one mapping of stations to nodes and the absence of parallel edges. Our second contribution is the combination of the CH algorithm and the station graph model. With preprocessing times of a few minutes, we answer time queries in half a millisecond. Our algorithm is therefore suitable for web services with high load, where small query times are very important and can compensate for our restricted scenario.

In our opinion, our presented ideas build the algorithmic core to develop efficient algorithms in more realistic scenarios. Especially the successful demonstration of the contraction of timetable networks brings speed-up techniques to a new level. It allows to reduce network sizes and to apply other speed-up techniques only to a core of the hierarchy, even in case that the contraction of all nodes is infeasible.

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. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: [22], pp. 319–333
3. Batz, G.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)
4. Müller-Hannemann, M., Weihe, K.: Pareto Shortest Paths is Often Feasible in Practice. In: Brodal, G.S., Frigioni, D., Marchetti-Spaccamela, A. (eds.) WAE 2001. LNCS, vol. 2141, pp. 185–197. Springer, Heidelberg (2001)
5. Marcotte, P., Nguyen, S. (eds.): Equilibrium and Advanced Transportation Modelling. Kluwer Academic Publishers Group, Dordrecht (1998)
6. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. ACM Journal of Experimental Algorithmics 5, 12 (2000)
7. Brodal, G., Jacob, R.: Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. In: [23], pp. 3–15
8. Nachtigall, K.: Time depending shortest-path problems with applications to railway networks. European Journal of Operational Research 83(1), 154–166 (1995)
9. Orda, A., Rom, R.: Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. Journal of the ACM 37(3), 607–625 (1990)
10. Orda, A., Rom, R.: Minimum Weight Paths in Time-Dependent Networks. Networks 21, 295–319 (1991)
11. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Efficient Models for Timetable Information in Public Transportation Systems. ACM Journal of Experimental Algorithmics 12, Article 2.4 (2007)
12. Berger, A., Müller–Hannemann, M.: Subpath-Optimality of Multi-Criteria Shortest Paths in Time-and Event-Dependent Networks. Technical Report 1, University Halle-Wittenberg, Institute of Computer Science (2009)

13. Berger, A., Delling, D., Gebhardt, A., Müller–Hannemann, M.: Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In: Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS 2009), Dagstuhl Seminar Proceedings (2009)
14. Delling, D., Wagner, D.: Time-Dependent Route Planning. In: Ahuja, R.K., Möhring, R.H., Zaroliagis, C. (eds.) Robust and Online Large-Scale Optimization. LNCS, vol. 5868, pp. 207–230. Springer, Heidelberg (2009)
15. Bast, H.: Car or Public Transport – Two Worlds. In: Albers, S., Alt, H., Näher, S. (eds.) Efficient Algorithms. LNCS, vol. 5760, pp. 355–367. Springer, Heidelberg (2009)
16. Hart, P.E., Nilsson, N., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics 4, 100–107 (1968)
17. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Towards Realistic Modeling of Time-Table Information through the Time-Dependent Approach. In: [22], pp. 85–103
18. Disser, Y., Müller–Hannemann, M., Schnee, M.: Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In: [22], pp. 347–361
19. Delling, D.: Time-Dependent SHARC-Routing. Algorithmica (July 2009); In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 332–343. Springer, Heidelberg (2008)
20. Vetter, C.: Parallel Time-Dependent Contraction Hierarchies (2009), Student Research Project, http://algo2.iti.kit.edu/documents/routeplanning/vetter_sa.pdf
21. 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)
22. McGeoch, C.C. (ed.): WEA 2008. LNCS, vol. 5038. Springer, Heidelberg (2008)
23. Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS 2003). Electronic Notes in Theoretical Computer Science, vol. 92 (2004)

Distributed Time-Dependent Contraction Hierarchies*

Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter

Karlsruher Institut für Technologie, Germany
{luxen,sanders}@kit.edu,
tim@kieritz.de, vetter@ira.uka.de

Abstract. Server based route planning in road networks is now powerful enough to find quickest paths in a matter of milliseconds, even if detailed information on time-dependent travel times is taken into account. However this requires huge amounts of memory on each query server and hours of preprocessing even for a medium sized country like Germany. This is a problem since global internet companies would like to work with transcontinental networks, detailed models of intersections, and regular re-preprocessing that takes the current traffic situation into account. By giving a distributed memory parallelization of the arguably best current technique – time-dependent contraction hierarchies, we remove these bottlenecks. For example, on a medium size network 64 processes accelerate preprocessing by a factor of 28 to 160 seconds, reduce per process memory consumption by a factor of 10.5 and increase query throughput by a factor of 25.

Keywords: time-dependent shortest paths, distributed computation, message passing, algorithm engineering.

1 Introduction

For planning routes in road networks, Dijkstra's algorithm is too slow. Therefore, there has been considerable work on accelerating route planning using preprocessing. Refer to [4,7] for recent overview papers. For road networks with ordinary, static edge weights this work has been very successful and leading to methods that are several orders of magnitudes faster than Dijkstra's classical algorithm. Recent work shows that similarly fast routing is even possible when the edge weights are time-dependent travel time functions defined by piece-wise linear functions that allow no overtaking [5,1]. This is practically important since it allows to take effects such as rush-hour congestion into account. While these methods are already fast enough to be used in practice in a server based scenario on "medium-sized" networks such as the road network of Germany they leave several things to be desired. First, globally operating companies providing routing services might want to offer a seamless service for transcontinental networks as for EurAsiAfrica or for the Americas. Second, we would like to move to

* Partially supported by DFG grant SA 933/5-1.

more-and-more detailed network models with an ever increasing fraction of time-dependent edges and multi-node models for every road intersection that allows to model turn-penalties, traffic-light delays, etc. First studies indicate that such models increase memory requirements by a factor of 3–4 [13]. On top of this, we would like to recompute the preprocessing information frequently in order to take information on current traffic (e.g., traffic jams) into account. Indeed, in the future we may want to perform massive sub-real-time traffic simulations that predict congestion patterns for the near future. These simulations put even more stringent requirements on a route planner supporting them. Finally, even when the preprocessed information fits on a single large machine with a lot of expensive main memory, a large company may have to replicate this information in order to be able the handle a large flow of queries.

In this paper we address all these issues by developing an approach that distributes both preprocessing and query processing to a cluster of inexpensive machines each equipped with limited main memory. The available large cumulative memory allows large networks while the cumulative processing power allows fast preprocessing and high query throughput. Our distributed implementation (DTCH) is based on time-dependent contraction hierarchies (TCH) [1]. Among the techniques available for time-dependent route planning, TCHs have several features that make its parallelization attractive: they are currently the fastest approach available both with respect to preprocessing time and query time. Storage requirements are critical because TCHs introduce many additional edges which represent long paths with complex travel-time function. Furthermore, TCH queries have small search spaces mostly concentrated around source and target node and hence exhibit a degree of locality that make it attractive for a distributed implementation.

First, we give an overview over the relevant literature for the time-dependent shortest path problem in Section 2. Then we introduce basic concepts in Section 3. Sections 4 and 5 explain distributed approaches to preprocessing and queries respectively. Section 6 gives results on a first implementation of our approach. The distributed preprocessing time in one of our test cases falls from more than an hour to a little more than two and a half minutes while the throughput of distributed batch queries almost benefits linearly from more processes. Section 7 summarizes the results and outlines future improvements.

2 Related Work

For a survey on the rich literature on speedup techniques for static edge weights we refer to [7,4]. Static contraction hierarchies [8] are a simple and efficient hierarchical approach to fast route planning. The idea is to build an n-level hierarchy by iteratively removing the "least important" node v from the network by *contracting it* – shortest paths leading through v are bypassed using new *shortcut* edges. For example, consider a graph consisting of three nodes a, b, c and two direct edges (a, b) and (b, c). Node b is shortcutted by removing incoming edge (a, b) as well as outgoing edge (b, c) and inserting a new edge (a, c) with cumulative edge weights. Node ordering is done heuristically. The resulting contraction hierarchy (CH) consists of all original nodes and edges plus the shortcut edges thus introduced. A bidirectional query can then restrict itself to edges leading to more important nodes – resulting in very small search spaces. It was thought that bidirectional query algorithms would not work for time-dependent networks because it is not clear how to perform a backward search time-dependently without knowing the arrival time. However, in [1] it was shown how CHs can be made time-dependent. First, time-dependent forward search and non-time-dependent backward search are combined to identify a small corridor network that has to contain the shortest path. This path is then identified using time-dependent forward search in the corridor.

Delling et al. [5,6] developed several successful time-dependent route planning techniques based on a combination of node contraction and goal-directed techniques. The most promising of these methods rely on forward search. Since they do not use long-range shortcuts, they need less memory than TCHs. However they exhibit slower query time and/or preprocessing time and distributed memory parallelization looks more difficult since the query search space is less localized around source and target.

Vetter [12] has developed a shared memory parallelization of the preprocessing stage that is also the basis for our distributed memory parallelization. While sequential CH techniques perform node ordering online using a priority queue, Vetter contracts nodes in a batched fashion by identifying sets of nodes that are both sufficiently unimportant and sufficiently far away of each other so that their concurrent contraction does not lead to unfavorable hierarchies. Note that a shared memory parallelization is much easier than using distributed memory since all data is available everywhere.

3 Preliminaries

We are considering a road network in the form of a directed graph $G = (V, E)$ with n nodes and m edges. We assume that the nodes $V = 1, \ldots, n$ are ordered by some measurement of importance $\prec$ and that 1 is the least important node. This numbering defines the so-called *level* of each node and $u < v$ denotes that the level of u is lower than the level of v.

Time-dependent networks do not have static edge weights, but rather a travel time function. The weight function $f(t)$ specifies the travel time at the endpoint of an edge when the edge is entered at time t. We further assume that each edge obeys the FIFO property: $(\forall\tau < \tau') : \tau + f(\tau) \leq \tau' + f(\tau')$ and as mentioned before, each travel time function is represented by a piece-wise linear function defined by a number of supporting points. This allows us to use a time-dependent and label correction variant of Dijkstra's algorithm. For an explanation of the inner workings of the edge weight data type see the paper of Batz et al. [1] where it is explained in-depth.

A node u can be contracted by deleting it from the graph and replacing paths of the form $\langle v, u, w\rangle$ by shortcut edges (v, w). Shortcuts that at no point in time represent a shortest path can be omitted. To prove this property, so-called *witness searches* have to be performed, which are profile searches to check whether there is a shorter path $\langle u, w \rangle$ not going over v that is valid at some point in time.

We consider a system where p processes run on identical processors each with their own local memory. Processes interact by exchanging messages, e.g., using the message passing interface MPI.

4 Distributed Node Ordering and Contraction

The nodes of the input graph are partitioned into one piece $\mathcal{N}_i$ for each process $i \in \{0, \dots, p-1\}$ using graph partitioning software that produces sets of about equal size and few cut edges. See Section 6 for more details on the partitioning software.

The goal of preprocessing is to move each node into its level of the hierarchy. Initially, no node is in a level. The sequential node order is computed on the fly while contracting nodes. Which node is to be contracted next is decided using a priority function whose most important term among others is the *edge difference* between the number of introduced shortcuts and the number of adjacent edges that would be removed by a contraction step.

The distributed node ordering and contraction is an iterative algorithm. While the remaining graph (containing the nodes not yet contracted) is nonempty, we identify an independent set I of nodes to be contracted in each iteration. Each process contracts the nodes under its authority independently of the other processes. Furthermore, we define I to be the set of nodes whose contraction does not depend on the contraction of any other node in the remaining graph. Thus the nodes in I can be contracted in any order without changing the result. While any independent set could be used in principle, we have to try to approximate the ordering a sequential algorithm would use. We therefore use nodes that are locally minimal with respect to a heuristic importance function within their local 2-hop neighborhood. In [12] this turned out to be a good compromise between a low number of iterations and good approximation of the behavior of sequential contraction.

As in static CHs [8], the search for witness paths is pruned heuristically by a limit h on the number of edges in the witness (hop-limit). This feature helps us to achieve an efficient distributed implementation. We maintain the invariant that before contracting I, every process stores its local partition $\mathcal{N}_i$ plus the nodes within a ℓ-neighborhood[1] (a *halo*) where $\ell = \lfloor h/2 \rfloor + 1$. It is easy to show that all witness paths with at most h hops must lie inside this local node set L_i. When the halo information is available, each iteration of the node contraction can be performed completely independently of the other processes, possibly using shared-memory parallelism locally. At the end of each iteration, all newly generated shortcuts (u, v) are sent to the owners of nodes u and v which subsequently forward them to processes with a copy of u or v in their halo. Messages between the same pair of processes can be joined into a single message to reduce

[1] Node v is in the x-neighborhood of node set M if there is a path with $\leq x$ edges from v to a node in M or a path with $\leq x$ edges from a node in M to v.

the communication overhead. This way, two global communication phases suffice to exchange information on new shortcuts generated. Then, the halo-invariant has to be repaired since new shortcuts may result in new nodes reachable within the hop limit. This can be done in two stages. First, a local search from the border nodes in $\mathcal{N}_i$ (those nodes that have a neighbor outside $\mathcal{N}_i$) establishes the current distances of nodes in the halo. Then, nodes with hop distance $< \ell$ with neighbors outside L_i requests information on these neighbors (using a single global data exchange). This process is iterated until the full ℓ-hop halo information is available again at each process. Thus, this repair operation takes at most ℓ global communication phases.

5 Distributed Query

The query is easily distributable. The forward search is performed on the cluster node that authoritative for the starting node. The temporary results are communicated the authorative node for the backward search where the query is completed.

We explain the method in more detail now. To distribute the query, we use the same partitioning of the nodes that was previously used during the contraction of the road network. Each process i is responsible for the search spaces of start and target nodes that lie in $\mathcal{N}_i$. We denote the authoritative process for node v by $p^{(v)}$. In addition to the nodes, each process keeps track of the nodes that do not lie in $\mathcal{N}_i$ but are reachable from the border nodes in $\mathcal{N}_i$. To be more precise, process i is the authority to the nodes in $\mathcal{N}_i$ and knows those nodes that are reachable by a forward search in the graph $G_\uparrow$ and by a backward search in $G_\downarrow$. Here, $G_\uparrow$ contains all edges or shortcuts of the form (u, v) where v was contracted later than u. $G_\downarrow$ contains all edges or shortcuts of the form (u, v) where u was contracted later than v.

To perform a shortest path query from s to t, a request enters the system at possibly any process and is sent to the authoritative process for s. If that process is the authority to node t as well, then the entire query is answered locally. Otherwise a time-dependent forward search is conducted starting at node s in $G_\uparrow^s$. Note that this search can eliminate some nodes that cannot be on a shortest path using the technique of stall-on-demand [8]. The arrival time at all non-pruned nodes reached by this search is then sent to process $p^{(t)}$ which now has all the information needed to complete the query. Same as the backward search in the sequential algorithm, $p^{(t)}$ goes on to mark all edges leading to t in $G_\downarrow$ pruning some edges than cannot be part of a shortest path at any time. Finally, the forward search is resumed starting from those nodes reached by both forward search and backward search and only exploring marked edges.

6 Experiments

We now report on experiments with a prototypical implementation using C++ and MPI. We use 1, 2, 4, ... 128 nodes each equipped with 2 2 667 MHz Quad-Core Intel Xeon X5355 processors and 16 gigabytes of RAM. The nodes are connected by an InfiniBand 4xDDR switch. The resulting point-to-point peak bandwidth between two nodes is more than 1 300 MB/s. We use Intel C/C++ compiler version 10.1 with full optimization turned on and OpenMPI 1.3.3 for communication. We used only a single process per node to simplify implementation. Using shared memory parallelism within the nodes should lead to significant further reductions of execution time for a fixed number of nodes. For the partitioning of the input graph we used the same partitioner as SHARC [3] which is a locally optimized variant of SCOTCH [9] and was kindly provided by Daniel Delling. The running time for graph partitioning is not included but negligible in our setting.

We used the real-world road network of Germany with realistic traffic patterns as well as a European network with synthetic time-dependent travel times using the methodology of [6,5]. All networks were provided by PTV AG for scientific use. The German road network has about 4.7 million nodes and 10 million edges and is augmented with time-dependent edge weights of two distinct traffic patterns. The first is strongly time-dependent and reflects midweek (Tuesday till Thursday) traffic while the other reflects the more quiet Sunday traffic patterns. Both data sets were collected from historical data. The German midweek scenario has about 8% time-dependent edge weights while the sunday scenario consists of about 3% time-dependent edges. The European graph is highly time-dependent and has approximately 18 million nodes and 42.6 million edges of which 6% are time-dependent with an average number of more than 26 supporting points each.

6.1 Distributed Contraction

Figure 1 shows the speedups and execution times obtained. We use relative speedup compared to the sequential code of [1] Batz et al. . Since the European road network cannot be contracted on less than 4 nodes we use the execution time of our distributed algorithm on four cluster nodes as a baseline and measure speedups against this number in this case. For the German midweek road network our systems scales well up to 16 processes. Indeed, we obtain a slight superlinear speedup which we attribute to a larger overall capacity of cache memory. More than 64 processes make no sense at all. For the German Sunday network, the behavior is similar although both overall execution times and speedups are smaller. This is natural since the limited time-dependence incurs less overall work. As to be expected for a larger network, scalability for the European network is better, being quite good up to 32 processes and still improving at 128 processes. However, a closer look shows that there is still potential for improvement here. Our analysis indicates that the execution times per iteration are fluctuating wildly for the European network. Closer inspection indicates that this is due to partitions that are (at least in some iterations) much more difficult to contract than others. Therefore, in future versions we plan to try a more adaptive contraction strategy: Once a significant percentage of all processes have

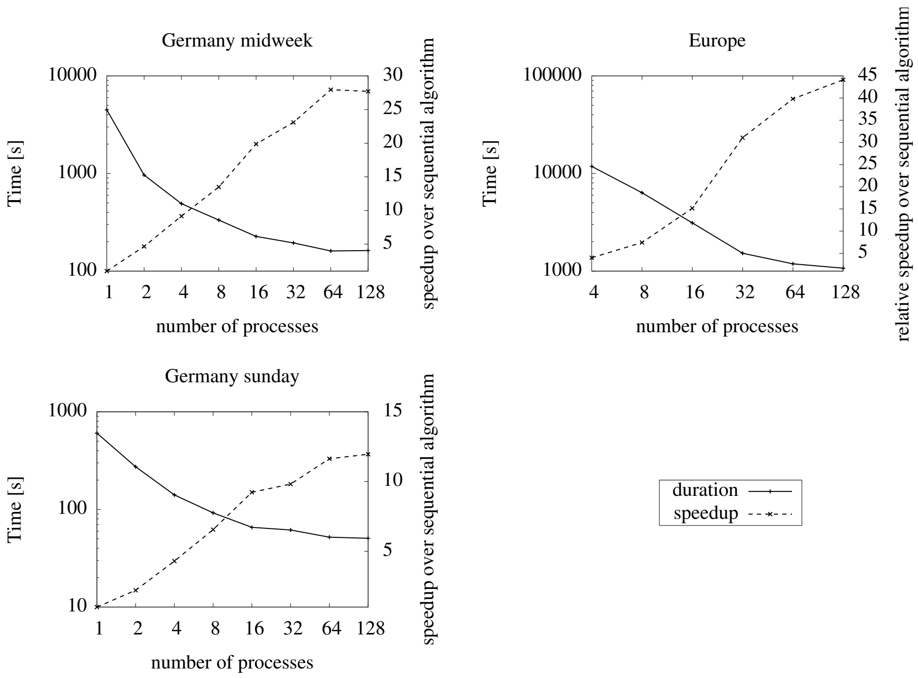


Fig. 1. Running Times and speedup for the distributed contraction with a varying number of processes

finished contracting the nodes alloted to them, we stop contraction everywhere. If contracted nodes are spread uniformly (e.g. using randomization) this should smooth out some of these fluctuations.

Next, we analyze the memory requirements for each process at query time[2] As explained in Section 4 we get an overhead because every process holds the complete search space for each node under its authority. In Figure 2 we plot the memory consumption in two ways. First, we look at the maximum memory requirements for any single process. Second, we analyze how this maximum m compares to the sequential memory requirements s by looking at the ratio of $p{\cdot}m/s$ which quantifies the blow-up of overall memory requirement. Although the maximum m decreases, the memory blowup only remains in an acceptable range for around 16–32 processes. Even then a blowup factor around 2 is common. This is in sharp contrast to (unpublished) experiments we made in connection with a mobile implementation of static CHs [11] where memory blowup was negligible. The crucial difference here is that the relatively small number of important nodes that show up on most processes have many incident nodes with highly complex travel-time functions.

[2] Memory requirements during contraction time are much lower in our implementation since we write edges incident to contracted nodes out to disk.

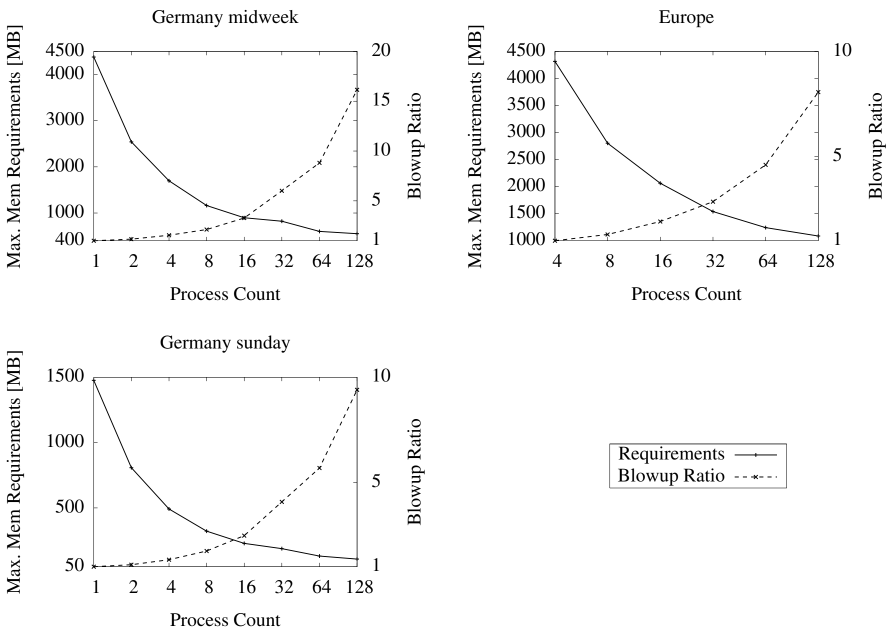


Fig. 2. Maximum Memory Requirements and Overheads of the Distributed search Data Structure

6.2 Distributed Query

The query times are averaged over a test set of 100 000 queries that were precomputed using a plain time-dependent Dijkstra. For each randomly selected start and destination node a random departure time was selected. The length of a resulting shortest path was saved. The query and its running times can be evaluated in two distinct settings.

Figure 3 shows the speedup obtained for performing all 100 000 queries. This figure is relevant for measuring system thoughput and thus the cost of the servers. Scalability for the German networks is even better than for the contraction time with efficiency near 50 % for up to 32 PEs. It should also be noted that we measured average message lengths of only around 4 000 bytes for communicating the results of the forward search space. This means that we could expect similar performance also on machines with much slower interconnection network. For the European network we even get superlinear speedup. This might again be connected to cache effects. The reason why we do *not* have superlinear speedup for the German networks might be that the smaller data set do not put that heavy requirements on cache capacity anyway. Still, the amount of superlinear speedup remains astonishing. We refrain from further attempts at an explanation because we lack intuition on the nature of the synthetic data used. Before claiming that this effect is interesting and useful we would prefer to wait for realistic data for large networks.

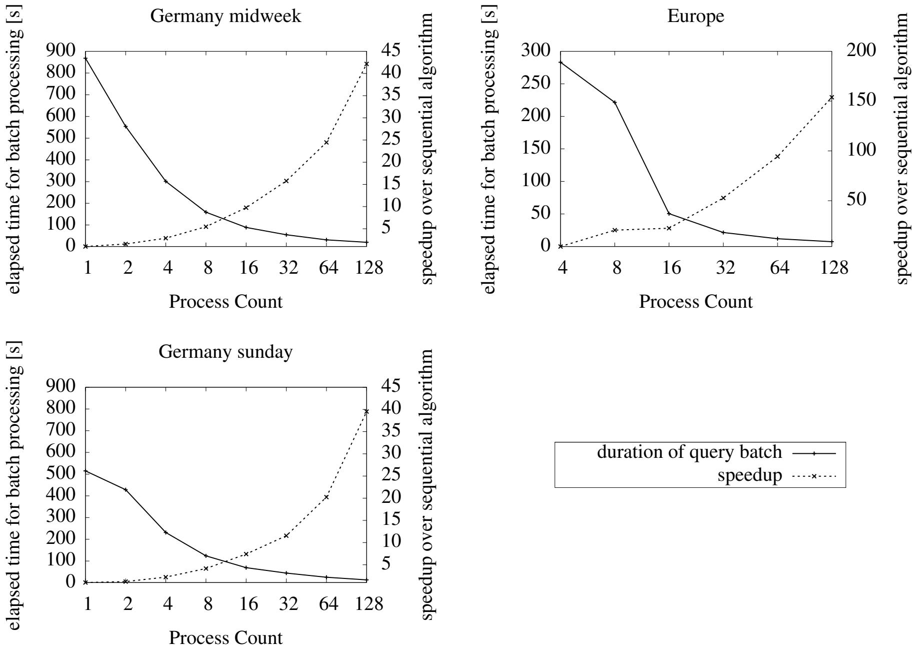


Fig. 3. Average time [ms] for a single query in a batch run of 100 000 with a given number of processes on the German road networks

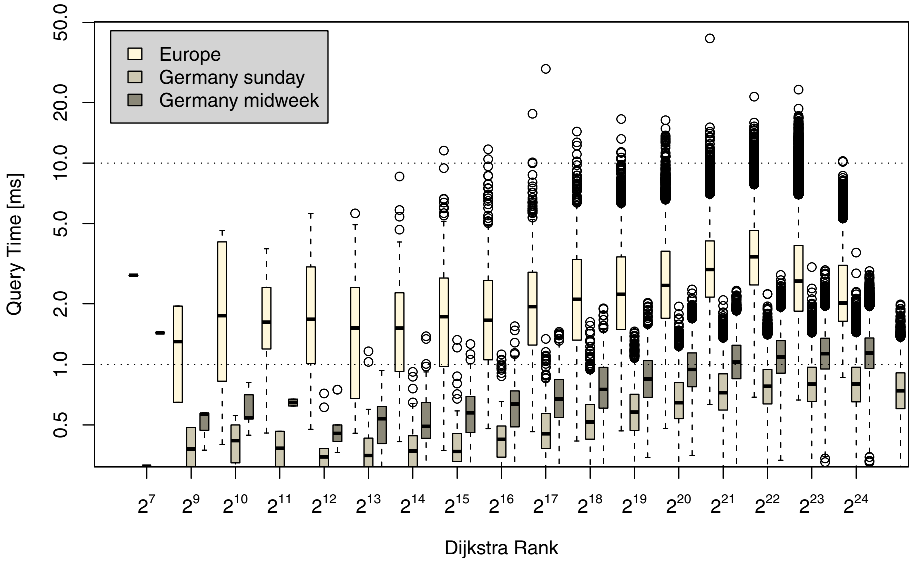


Fig. 4. Rank plots for the various road networks

Next, we analyze the single query behavior and measure how much time each individual distributed shortest path query takes. We give a detailed look into query time distribution in Figure 4 using the well-established methodology of [10]. It shows a plot of the individual query times of 100 000 random distributed queries on each of the road networks with the property that a plain time-dependent bidirectional Dijkstra settles 2^i nodes. We observed mean query times of 1.12 ms (Germany midweek), 0.44 ms (Germany Sunday) average query times for the sequential query algorithm on the same hardware. In the parallel system we observe somewhat larger times but these latencies are still negligible compared to the latencies usual in the internet which are at least an order of magnitude larger. We see considerably smaller query times for local queries which might constitute a significant part of the queries seen in practice. Again, the European network behaves differently. The overall query latencies of about 2ms are good, but we do not see improvements for local queries.

7 Conclusions and Future Work

We successfully distributed time-dependent Contraction Hierarchies including the necessary precomputation to a cluster of machines with medium sized main memory. For large networks we approach two orders of magnitude in reduction of preprocessing time and at least a considerable constant factor in required local memory size. We believe that there is considerable room for further improvement: reduce per-node memory requirements, use shared-memory parallelism on each node and more adaptive node contraction.

Batz et al. [2] developed improved variants of TCHs that reduce space consumption by storing only approximate travel-time functions for shortcuts. These features should eventually be integrated into our distributed approach but by themselves they do not solve the scalability issues stemming from larger networks, more detailed modelling, and stringent requirements on preprocessing time for incorporating real-time information and for traffic simulation.

With respect to the targeted applications we can be quite sure that reduced turnaround times for including upto-date traffic information are realistic. Regarding larger networks with more detailed modelling we are optimistic yet further experiments with such large networks would be good to avoid bad surprises.

References

1. Batz, G.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)
2. 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. Bauer, R., Delling, D.: SHARC: Fast and Robust Unidirectional Routing. In: Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX 2008), pp. 13–26. SIAM, Philadelphia (2008)

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. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 303–318. Springer, Heidelberg (2008)
5. Delling, D.: Time-dependent SHARC-routing. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 332–343. Springer, Heidelberg (2008)
6. Delling, D., Nannicini, G.: Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 812–823. Springer, Heidelberg (2008)
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, pp. 117–139. Springer, Heidelberg (2009)
8. 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)
9. Pellegrini, F.: SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package (2007)
10. 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)
11. Sanders, P., Schultes, D., Vetter, C.: Mobile route planning. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 732–743. Springer, Heidelberg (2008)
12. Vetter, C.: Parallel time-dependent contraction hierarchies. Student Research Project, Universität Karlsruhe (TH) (2009), http://algo2.iti.kit.edu/1588.php
13. Volker, L.: Route planning in road networks with turn costs. Student Research Project, Universität Karlsruhe (TH) (2008), http://algo2.iti.kit.edu/1154.php

Practical Compressed Suffix Trees*

Rodrigo Cánovas and Gonzalo Navarro

Department of Computer Science, University of Chile, Chile
{rcanovas,gnavarro}@dcc.uchile.cl

Abstract. The suffix tree is an extremely important data structure for stringology, with a wealth of applications in bioinformatics. Classical implementations require much space, which renders them useless for large problems. Recent research has yielded two implementations offering widely different space-time tradeoffs. However, each of them has practicality problems regarding either space or time requirements. In this paper we implement a recent theoretical proposal and show it yields an extremely interesting structure that lies in between, offering both practical times and affordable space. The implementation is by no means trivial and involves significant algorithm engineering.

1 Introduction

The suffix tree [18,30] is arguably the most important data structure for string analysis. It has been said to have a myriad of virtues [2] and there are books dedicated to its applications in areas like bioinformatics [12]. Many complex sequence analysis problems are solved through sophisticated traversals over the suffix tree, and thus a fully-functional implementation supports a variety of *navigation operations*. These involve not only the classical tree navigation operations (parent, child) but also specific ones such as suffix links and lowest common ancestors.

One serious problem of suffix trees is that they take much space. A naive implementation can easily require 20 bytes per character, and a very optimized one reaches 10 bytes [14]. A way to reduce this space to about 4 bytes per character is to use a simplified structure called a suffix array [17], but it does not contain sufficient information to carry out all the complex tasks suffix trees are used for. Enhanced suffix arrays [1] extend suffix arrays so as to recover the full suffix tree functionality, raising the space to about 6 bytes per character in practice. Another heuristic space-saving methods [20] achieve about the same.

For example, on DNA, each character could be encoded with 2 bits, whereas the alternatives we have considered require 32 to 160 bits per character (bpc). Using suffix trees on secondary memory makes them orders of magnitude slower as most traversals are non-local. This situation is also a heresy in terms of Information Theory: whereas the information contained in a sequence of n symbols over an alphabet of size σ is $n \log \sigma$ bits in the worst case, all the alternatives above require $\Theta(n \log n)$ bits. (Our logarithms are in base 2.)

* Partially funded by Millennium Institute of Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, and by Fondecyt Grant 1-080019, Chile.

Recent research on *compressed suffix trees (CSTs)* has made much progress in terms of reaching space requirements that approach not only the worst-case space of the sequence, but even its information content. All these can be thought of as a *compressed suffix array (CSA)* plus some extra information that encodes the tree topology and longest common prefix (LCP) information.

The first such proposal was by Sadakane [27]. It requires $6n$ bits on top of his CSA [26], which in turn requires $nH_0 + O(n \log\log\sigma)$ bits, where H_0 is the zero-order entropy of the sequence. This structure supports most of the tree navigation operations in constant time (except, notably, going down to a child, which is an important operation). A recent implementation [29] achieves a few tens of microseconds per operation, but in practice the structure requires about 25–35 bpc (close to a suffix array), and thus its applicability is limited.

The second proposal was by Russo et al. [25]. It requires only $o(n)$ bits on top of a CSA. By using an FM-index [6] as the CSA, one achieves $nH_k + o(n\log\sigma)$ bits of space, where H_k is the k-th order empirical entropy of the sequence, for sufficiently low $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$. The navigation operations are supported in polylogarithmic time (at best $\Theta(\log n \log\log n)$ in their paper). This structure was implemented by Russo and shown to achieve very little space, around 4–6 bpc, which makes it extremely attractive when the sequence is large compared to the available main memory. On the other hand, the structure is much slower than Sadakane's. Each navigation operation takes the order of milliseconds, which is comparable to disk operation times.

Both existing implementations are unsatisfactory in either time or space (though certainly excell on the other aspect), and become very far extremes of a tradeoff: Either one has sufficient main memory to spend 30 bpc, or one has to spend milliseconds per navigation operation.

In this paper we present a third implementation, which offers a relevant space/time tradeoff between these two extremes. One variant shows to be superior to the implementation of Sadakane's CST in *both* space and time: it uses 13–16 bpc (i.e., half the space) and requires a few microseconds (i.e., several times faster) per operation. A second alternative works within 8–12 bpc and requires a few hundreds of microseconds per operation, that is, smaller than our first variant and still several times faster than Russo's implementation.

Our implementation is based on a third theoretical proposal, by Fischer et al. [8], which achieves $nH_k(2\max(1,\log(1/H_k))+1/\epsilon+O(1))+o(n\log\sigma)$ bpc (for the same k as above and any constant $\epsilon > 0$) and navigation times of the form $O(\log^\epsilon n)$. Their proposal involves several theoretical solutions, whose efficient implementation was far from trivial, requiring significant algorithm engineering that completely changed the original proposal in some cases. After experimental study of several alternatives, we choose the two variants described above.

Our work opens the door to a number of practical suffix tree applications, particularly relevant to bioinformatics. Our implementation will be publicly available in the *Pizza&Chili* site (`http://pizzachili.dcc.uchile.cl`). We plan to apply it to solve concrete bioinformatic problems on large instances.

Table 1. Operations over the nodes and leaves of the suffix tree

Operation	Description
Root()	the of the suffix tree.
Locate(v)	suffix position i if v is the leaf of suffix $T_{i,n}$, otherwise NULL.
Ancestor(v, w)	true if v is an ancestor of w.
SDepth(v)/TDepth(v)	string-depth/tree-depth of v.
Count(v)	number of leaves in the subtree rooted at v.
Parent(v)	parent node of v.
FChild(v)	alphabetically first child of v.
NSibling(v)	alphabetically next sibling of v.
SLink(v)	suffix-link of v; i.e., the node w s.th. $\pi(w) = \beta$ if $\pi(v) = a\beta$ for $a \in \Sigma$.
SLink^i(v)	iterated suffix-link: the node w s.th. $\pi(w) = \beta$ if $\pi(v) = a\beta$ for $a \in \Sigma^i$.
LCA(v, w)	lowest common ancestor of v and w.
Child(v, a)	node w s.th. the first letter on edge (v, w) is $a \in \Sigma$.
Letter(v, i)	*ith* letter of v's path-label, $\pi(v)[i]$.
LAQ$_S$(v, d)/LAQ$_T$(v, d)	the hightest ancestor of v with string-depth/tree-depth $\leq d$.

2 Compressed Suffix Trees

A *suffix array* over a text $T[1, n]$ is an array $A[1, n]$ of the positions in T, lexicographically sorted by the suffix starting at the corresponding position of T. That is, $T[A[i], n] < T[A[i+1], n]$ for all $1 \leq i < n$. Note that every substring of T is the prefix of a suffix, and that all suffixes starting with a given pattern P appear consecutively in A, hence a couple of binary searches find the area $A[sp, ep]$ containing all the positions where P occurs in T.

There are several *compressed suffix arrays (CSAs)* [21,5], which offer essentially the following functionality: (1) Given a pattern $P[1, m]$, find the interval $A[sp, ep]$ of the suffixes starting with P; (2) obtain $A[i]$ given i; (3) obtain $A^{-1}[j]$ given j. An important function the CSAs implement is $\Psi(i) = A^{-1}[(A[i] \bmod n) + 1]$ and its inverse, usually much faster than computing A and A^{-1}. This function lets us move virtually in the text, from the suffix i that points to text position $j = A[i]$, to the one pointing to $j + 1 = A[\Psi(i)]$.

A *suffix tree* is a compact trie (or digital tree) storing all the suffixes of T. This is a labeled tree where each text suffix is read in a root-to-leaf path, and the children of a node are labeled by different characters. Leaves are formed when the prefix of the corresponding suffix is already unique. Here "compact" means that unary paths are converted into a single edge, labeled by the string formed by concatenating the involved character labels. If the children of each node are ordered lexicographically by their string label, then the leaves of the suffix tree form the suffix array of T. Several navigation operations over the nodes and leaves of the suffix tree are of interest. Table 1 lists the most common ones.

In order to get a suffix tree from a suffix array, one needs at most two extra pieces of information: (1) the tree topology; (2) the *longest common prefix (LCP)* information, that is, $LCP[i]$ is the length of the longest common prefix between $T[A[i-1], n]$ and $T[A[i], n]$ for $i > 1$ and $LCP[1] = 0$ (or, seen another way, the length of the string labeling the path from the root to the lowest common ancestor node of suffix tree leaves i and $i - 1$). Indeed, the suffix tree topology can be implicit if we identify each suffix tree node with the suffix array interval containing the leaves that descend from it. This range uniquely identifies the node because there are no unary nodes in a suffix tree.

Consequently, a *compressed suffix tree (CST)* is obtained by enriching the CSA with some extra data. Sadakane [27] added the topology of the tree (using $4n$ extra bits) and the LCP data. The LCP was compressed to $2n$ bits by noticing that, if sorted by text order rather than suffix array order, the LCP numbers decrease by at most 1. Let LCP' be the permuted LCP array, then $LCP'[j+1] \geq LCP'[j]-1$. Thus the numbers can be differentially encoded, $h[j+1] = LCP'[j+1] - LCP'[j] + 1 \geq 0$, and then represented in unary over a bitmap $H[1,2n] = 0^{h[1]}10^{h[2]}\dots 10^{h[n]}1$. Then, to obtain $LCP[i]$, we look for $LCP'[A[i]]$, and this is extracted from H via *rank*/*select* operations. Here $rank_b(H,i)$ counts the number of bits b in $H[1,i]$ and $select_b(H,i)$ is the position of the i-th b in H. Both can be answered in constant time using $o(n)$ extra bits of space [19]. Then $LCP'[j] = select_1(H,j) - 2j$, assuming $LCP'[0] = 0$.

Russo et al. [25] get rid of the parentheses, by instead identifying suffix tree nodes with their corresponding suffix array interval. By sampling some suffix tree nodes, most operations can be carried out by moving, using suffix links, towards a sampled node, finding the information stored in there, and transforming it as we move back to the original node. The suffix link operation, defined in Table 1, can be computed using Ψ and the lowest common ancestor operation [27].

A New Theoretical CST Proposal. Fischer et al. [8] prove that array H in Sadakane's CST is compressible as it has at most $2r \leq 2(nH_k + \sigma^k)$ *runs* of 0s or 1s, for any k. Let $z_1, z_2, \dots, z_r$ the lengths of the runs of 0s and $o_1, o_2, \dots, o_r$ the same for the 1s. They create arrays $Z = 10^{z_1-1}10^{z_2-1}\dots$ and $O = 10^{o_1-1}10^{o_2-1}\dots$, with overall $2r$ 1s out of $2n$, and thus can be compressed to $2r\log\frac{n}{r} + O(r) + o(n)$ bits and support constant-time *rank* and *select* [24].

Their other improvement over Sadakane's CST is to get rid of the tree topology and replace it with suffix array ranges. Fischer et al. show that all the navigation can be simulated by means of three operations: (1) $RMQ(i,j)$ gives the position of the minimum in $LCP[i,j]$; (2) $PSV(i)$ finds the last value smaller than $LCP[i]$ in $LCP[1,i-1]$; and (3) $NSV(i)$ finds the first value smaller than $LCP[i]$ in $LCP[i+1,n]$. All these could easily be solved in constant time using $O(n)$ extra bits of space on top of the LCP representation, but Fischer et al. give sublogarithmic-time algorithms to solve them with only $o(n)$ extra bits.

As examples, the parent of node $[i,j]$ is $[PSV(i), NSV(i)-1]$; the LCA between nodes $[i,j]$ and $[i',j']$ is $[PSV(p), NSV(p)-1]$, where $p = RMQ(\min(i,i'), \max(j,j'))$; and the suffix link of $[i,j]$ is $[PSV(\Psi(i)), NSV(\Psi(j))-1]$.

Our Contribution. The challenge faced in this paper is to implement this CST. This can be divided into (1) how to represent LCP efficiently in practice, and (2) how to compute efficiently RMQ, PSV, and NSV over this LCP representation. We study each subproblem and compare the resulting CST with previous ones.

Our experiments were performed on 100 MB of the protein, sources, XML and DNA texts from *Pizza&Chili*. The computer is an Intel Core2 Duo at 3.16 GHz, with 8 GB of RAM and 6 MB cache, running Linux version 2.6.24-24.

3 Representing Array *LCP*

The following alternatives were considered to represent LCP:

Sad-Gon: Encodes H in plain, using the *rank*/*select* implementation of González [10], which takes $0.1n$ bits over the $2n$ used by H itself and answers *select* in $O(\log n)$ time via binary search.

Sad-OS: Like the previous one, but using the *dense array* implementation of Okanohara and Sadakane [22] for H. This requires about the same space as the previous one and answers *select* in $O(\log^4 r/\log n)$ time.

FMN-RRR: Encodes H in compressed form as in Fischer et al. [8], i.e., by encoding bitmaps Z and O. We use the compressed representation by Raman et al. [24] as implemented by Claude [4]. This costs $0.54n$ extra bits on top of the entropy of the two bitmaps, $2r\log\frac{n}{r}+O(r)$. *Select* takes $O(\log n)$ time.

FMN-OS: Like the previous one, but instead of Raman et al. technique, we use the *sparse array* implementation by Okanohara and Sadakane [22]. This requires $2r\log\frac{n}{r}+O(r)$ bits and solves *select* in time $O(\log^4 r/\log m)$.

PT: Inspired on an LCP construction algorithm [23], we store a particular sampling of LCP values, and compute the others using the sampled ones. Given a parameter v, the sampling requires $n+O(n/\sqrt{v}+v)$ bytes of space and computes any $LCP[i]$ by comparing at most some $T[j,j+v]$ and $T[j',j'+v]$. As we must obtain these symbols using Ψ up to $2v$ times, the idea is slow.

PhiSpare: This is inspired in another construction [13]. For a parameter q, store in text order an array LCP'_q with the LCP values for all text positions $q\cdot k$. Now assume $SA[i]=qk+b$, with $0\le b<k$. If $b=0$, then $LCP[i]=LCP'_q[k]$. Otherwise, $LCP[i]$ is computed by comparing at most $q+LCP'_q[k+1]-LCP'_q[k]$ symbols of the suffixes $T[SA[i-1],n]$ and $T[SA[i],n]$. The space is n/q integers and the computation requires $O(q)$ applications of Ψ on average.

DAC: The *directly addressable codes* of Ladra et al. [3]. Most LCP values are small ($O(\log_\sigma n)$ on average), and thus require few bits. Yet, some can be much longer. Thus we can fix a block length b and divide each number, of ℓ bits, into $\lceil \ell/b\rceil$ blocks of b bits. Each block is stored using $b+1$ bits, the last one telling whether the number continues in the next block or finishes in the current one. Those blocks are then rearranged to allow for fast random access. There are two variants of this structure, both implemented by Ladra: one with fixed b (*DAC*), and another using different b values for the first, second, etc. blocks, so as to minimize the total space (*DAC-Var*). Note we represent LCP and not LCP', thus we do not need to compute $A[i]$.

RP: *Re-Pair* [15] is a grammar-based compression method that factors out repetitions in a sequence. It has been used [11] to compress the differentially encoded suffix array, $SA'[i]=SA[i]-SA[i-1]$, which contains repetitions because SA can be partitioned into r areas that appear elsewhere in SA with the values shifted by 1 [16]. Note that LCP must then contain the same repetitions shifted by 1, and therefore Re-Pair compression of the differential LCP should perform similarly [8]. To obtain $LCP[i]$ we store sampled absolute LCP values and decompress the nonterminals since the last sample.

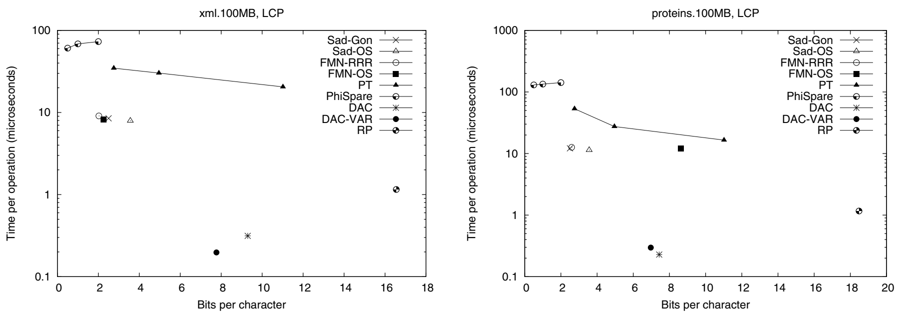


Fig. 1. Space/time for accessing *LCP* array

Experimental Comparison. We tested the different *LCP* implementations by accessing 100,000 random positions of the *LCP* array. Fig. 1 shows the space/times achieved on two texts (the others gave similar results). Only *PT* and *PhiSpare* display a space/time tradeoff; in the first we use $v = 4, 6, 8$ and for the second $q = 16, 32, 64$.

As it can be seen, *DAC*/*DAC-Var* and the representations of H dominate the space-time tradeoff map (*PhiSpare* and *PT* can use less space but they become impractically slow). For the rest of the paper we will keep only *DAC* and *DAC-Var*, which give the best time performance, and *FMN-RRR* and *Sad-Gon*, which have the most robust performance at representing H.

4 Computing *RMQ*, *PSV*, and *NSV*

Once a representation for *LCP* is chosen, one must carry out operations *RMQ*, *PSV*, and *NSV* on top of it (as they require to access *LCP*). We first implemented verbatim the theoretical proposals of Fischer et al. [8]. For *NSV*, the idea is akin to the recursive *findclose* solution for compressed trees [9]: the array is divided into blocks and some values are chosen as *pioneers* so that, if a position is not a pioneer, then its *NSV* answer is in the same block of that of its preceding pioneer (and thus it can be found by scanning that block). Pioneers are marked in a bitmap so as to map them to a reduced array of pioneers, where the problem is recursively solved. We experimentally verified that it is convenient to continue the recursion until the end instead of storing the explicit answers at some point. The block length L yields a space/time tradeoff since, at each level of the recursion, we must obtain $O(L)$ values from *LCP*. *PSV* is symmetric, needing another similar structure.

For *RMQ* we apply an existing implementation [7] on the *LCP* array, remembering that we do not have direct access to *LCP* but have to use any of the access methods we have developed for it. This accesses at most 5 cells of *LCP*, yet it requires $3.25n$ bits. In the actual theoretical proposal [8] this is reduced to $o(n)$ but many more accesses to *LCP* would be necessary; we did not implement that verbatim as it has little chances of being practical.

The final data structure, that we call *FMN-NPR*, is composed of the structure to answer *NSV* plus the one for *PSV* plus the structure to calculate *RMQ*.

4.1 A Novel Practical Solution

We propose now a different solution, inspired in Sadakane and Navarro's succinct tree representation [28]. We divide *LCP* into blocks of length L. Now we form a hierarchy of blocks, where we store the minimum *LCP* value of each block i in an array $m[i]$. The array uses $\frac{n}{L}\log n$ bits. On top of array m, we construct a perfect *L-ary* tree T_m where the leaves are the elements of m and each internal node stores the minimum of the values stored in its children. The total space for T_m is $\frac{n}{L}\log n(1+O(1/L))$ bits, so if $L=\omega(\log n)$, the space used is $o(n)$ bits.

To answer $NSV(i)$, we look for the first $j>i$ such that $LCP[j]<p=LCP[i]$, using T_m to find it in time $O(L\log(n/L))$. We first search sequentially for the answer in the same block of i. If it is not there, we go up to the leaf that represents the block and search the right siblings of this leaf. If some of these sibling leaves contain a minimum value smaller than p, then the answer to $NSV(i)$ is within their block, so we go down to their block and find sequentially the leftmost position j where $LCP[j]<p$. If, however, no sibling of the leaf contains a minimum smaller than p, we continue going up the tree and considering the right siblings of the parent of the current node. At some node we find a minimum smaller than p and start traversing down the tree as before, finding at each level the first child of the current node with a minimum smaller than p. *PSV* is symmetric. As the minima in T_m are explicitly stored, the heaviest part of the cost in practice is the $O(L)$ accesses to *LCP* cells at the lowest levels.

To calculate $RMQ(x,y)$ we use the same T_m and separate the search in three parts: (a) We calculate sequentially the minimum value in the interval $[x, L\lceil\frac{x}{L}\rceil-1]$ and its leftmost position in the interval; (b) we do the same for the interval $[L\lfloor\frac{y}{L}\rfloor, y]$; (c) we calculate $RMQ(L\lceil\frac{x}{L}\rceil, L\lfloor\frac{y}{L}\rfloor-1)$ using T_m. Finally we compare the results obtained in (a), (b) and (c) and the answer will be the one holding the minimum value, choosing the leftmost to break ties. For each node in T_m we also store the local position in the children where the minimum occurs, so we do not need to scan the child blocks when we go down the tree. The extra space incurred is just $\frac{n}{L}\log L(1+O(1/L))$ bits. The final data structure, if $L=\omega(\log n)$, requires $o(n)$ bits and can compute *NSV*, *PSV* and *RMQ* all using the same auxiliary structure. We call it *CN-NPR*.

Experimental Comparison. We tested the performance of the different *NPR* implementations by performing 100,000 *NSV* and *RMQ* queries at different random positions in the *LCP* array. Fig. 2 shows the space/time achieved for each implementation on two texts (the others gave very similar results). We used the slower *Sad-Gon* implementation for *LCP* to enhance the differences in time performance. We obtained space/time tradeoffs by using different block sizes $L=8,16,32$ (so the times for *RMQ* on *FMN-NPR* are not affected). Clearly *CN-NPR* displays the best performance for *NSV*, both in space and time. For *RMQ*, one can see that the best time obtained with *CN-NPR* dominates, in time and space, the *FMN-NPR* curve. Thus *CN-NPR* is our chosen implementation.

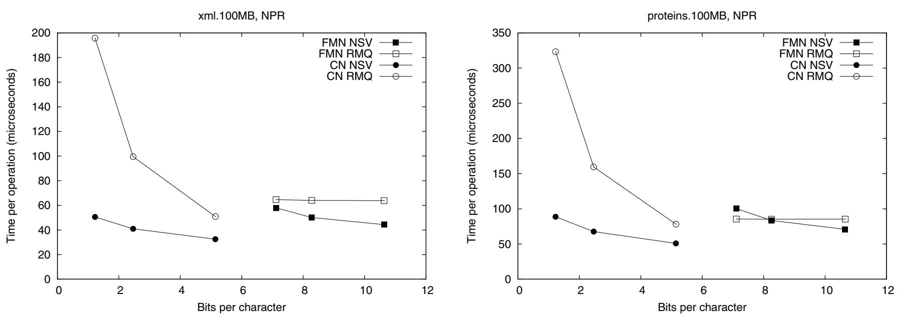


Fig. 2. Space/time for the operations NSV and RMQ

5 Our Compressed Suffix Tree

Our CST implementation applies our *CN-NPR* algorithms of Section 4 on top of some *LCP* representation from those chosen in Section 3. This solves most of the tree traversal operations by using the formulas provided by Fischer et al. [8], which we do not repeat for lack of space. In some cases, however, we have deviated from the theoretical algorithms for practical considerations.

TDepth: We proceed by brute force using *Parent*, as there is no practical solution in the proposal.

NSib: There is a bug in the original formula [8] in the case v is the next-to-last child of its parent. According to them, $NSib([v_l, v_r])$ first obtains its parent $[w_l, w_r]$, then checks whether $v_r = w_r$ (in which case there is no next sibling), then checks whether $w_r = v_r + 1$ (in which case the next sibling is leaf $[w_r, w_r]$), and finally answers $[v_r+1, z-1]$, where $z = RMQ(v_r+2, w_r)$. This RMQ is aimed at finding the end of the next sibling of the next sibling, but it fails if we are near the end. Instead, we replace it by the faster $z = NSV'(v_r+1, LCP[v_r+1])$. $NSV'(i,d)$ generalizes NSV by finding the next value smaller or equal to d, and is implemented almost like NSV using T_m.

Child: The children are ordered by letter. We need to extract the children sequentially using $FChild$ and $NSib$, to find the one descending by the correct letter, yet extracting the *Letter* of each is expensive. Thus we first find all the children sequentially and then binary search the correct letter among them, thus reducing the use of *Letter* as much as possible.

LAQ$_S(v,d)$**:** Instead of the slow complex formula given in the original paper, we use NSV' (and PSV'): $LAQ_S([v_l, v_r], d) = [PSV'(v_l+1, d), NSV'(v_r, d)-1]$. This is a complex operation we are supporting with extreme simplicity.

LAQ$_T(v,d)$**:** There is no practical solution in the original proposal. We proceed as follows to achieve the cost of d *Parent* operations, plus sume LAQ_S ones, all of which are reasonably cheap. Since $SDepth(v) \geq TDepth(v)$, we first try $v' = LAQ_S(v,d)$, which is an ancestor of our answer; let $d' = TDepth(v')$. If $d' = d$ we are done; else $d' < d$ and we try $v'' = LAQ_S(v, d + (d - d'))$.

We compute $d'' = TDepth(v'')$ (which is measured by using $d'' - d'$ *Parent* operations until reaching v') and iterate until finding the right node.

6 Comparing the CST Implementations

We compare all the CST implementations: Välimäki et al.'s [29] implementation of Sadakane's compressed suffix tree [27] *(CST-Sadakane)*; Russo's implementation of Russo et al.'s "fully-compressed" suffix tree [25] *(FCST)*; and our best variants. These are called *Our CST* in the plots. Depending on their *LCP* representation, they are suffixed with *Sad-Gon*, *FMN-RRR*, *DAC*, and *DAC-Var*. We do not compare some operations like *Root* and *Ancestor* because they are trivial in all implementations; *Locate* and *Count* because they depend only on the underlying compressed suffix array (which is mostly orthogonal, thus *Letter* is sufficient to study it); $SLink^i$ because it is usually better to do *SLink* i times; and LAQ_S and LAQ_T because they are not implemented in the alternative CSTs.

We typically show space/time tradeoffs for all the structures, where the space is measured in bpc (recall that these CSTs replace the text, so this is the overall space required). The times are averaged over a number of queries on random nodes. We use four types of node samplings, which make sense in different typical suffix tree traversal scenarios: (a) Collecting the nodes visited over 10,000 traversals from a random leaf to the root (used for *Parent*, *SDepth*, and *Child* operations); (b) same but keeping only nodes of depth at least 5 (for *Letter*); (c) collecting the nodes visited over 10,000 traversals from the parent of a random leaf towards the root via suffix links (used for *SLink* and *TDepth*); and (d) taking 10,000 random leaf pairs (for *LCA*). For space limitations, and because the outcomes are consistent across texts, we show the results of each operation over one text only, choosing in each case a different text. The standard deviation divided by the average is in the range [0.21,2.56] for *CST-Sadakane*, [0.97,2.68] for *FCST*, [0.65,1.78] for *Our CST Sad-Gon*, [0.64,2.50] for *Our CST FMN-RRR*, [0.59,0.75] for *Our CST DAC*, and [0.63,0.91] for *Our CST DAC-Var*. The standard deviation of the estimator is thus at most 1/100th of that.

Fig. 3 shows space/time tradeoffs for six operations. The general conclusion is that our CST implementation does offer a relevant tradeoff between the two rather extreme existing variants. Our CSTs can operate within 8–12 bpc (that is, at most 50% larger than the plain byte-based representation of the text, and replacing it) while requiring a few hundred microseconds for most operations (the "small and slow" variants *Sad-Gon* and *FMN-RRR*); or within 13–16 bpc and carry out most operations within a few microseconds (the "large and fast" variants *DAC*/*DAC-Var*). In contrast, the FCST requires only 4–6 bpc (which is, remarkably, as little as half the space required by the plain text representation), but takes the order of milliseconds per operation; and Sadakane's CST takes usually a few tens of microseconds per operation but requires 25–35 bpc, which is close to uncompressed suffix arrays (not uncompressed suffix trees, though).

We remark that, for many operations, our "fast and large" variant takes half the space of Sadakane's CST implementation and is many times faster. Exceptions are *Parent* and *TDepth*, where Sadakane's CST stores the explicit tree

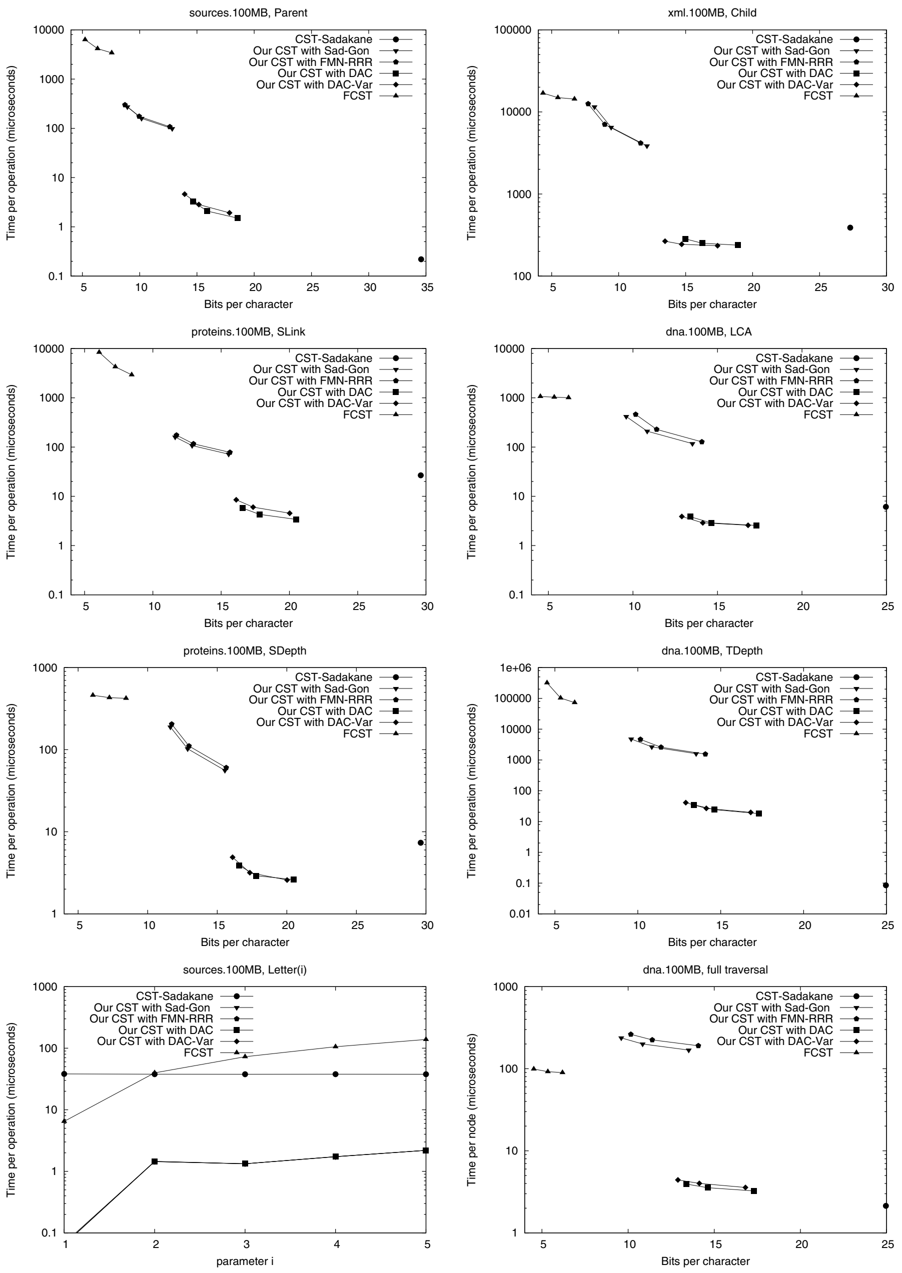


Fig. 3. Space/time tradeoff performance figures for several operations, time for *Letter*(i) as a function of i, and a full traversal computing *SDepth*. Note the logscale.

topology, and thus takes a fraction of a microsecond. On the other hand, our CST carries out LAQ_S (not shown) in the same time of *Parent*, whereas this is much more complicated for the alternatives (they do not even implement it). For *Child*, where we descend by a random letter from the current node, the times are higher than for other operations as expected, yet the same happens to all the implementations. We note that the FCST is more efficient on operations LCA and *SDepth*, which are its kernel operations, yet it is still slower than our "small and slow" variant. Finally, *TDepth* is an operation where all but Sadakane's CST are relatively slow, yet on most suffix tree algorithms the string depth is much more relevant than the tree depth. Our $LAQ_T(v, d)$ (not shown) would cost about d times the time of our *TDepth*.

At the bottom of the figure we show *Letter(i)*, as a function of i. It depends only on the CSA structure, and requires either applying $i-1$ times Ψ, or applying once SA and SA^{-1}. The former choice is preferred for the FCST and the latter in Sadakane's CST. For our CST, using Ψ iteratively was better for these i values, as the alternative requires around 70 microseconds.

The figure finishes with a basic suffix tree traversal algorithm: the classical one to detect the longest repetition in a text. This traverses all of the internal nodes using *FChild* and *NSib* and reports the maximum *SDepth*. Although Sadakane's CST takes advantage of locality, our "large and fast" variant is pretty close using half the space. Our "small and slow" variant, instead, requires a few hundred microseconds as expected, yet the FCST has a special implementation for full traversals and, this time, it beats our slow variant in space and time.

Acknowledgements. We thank Francisco Claude, Johannes Fischer, Rodrigo González, Juha Kärkkäinen, Susana Ladra, Veli Mäkinen, Simon Puglisi, Luís Russo, and Kunihiko Sadakane for code, help to use it, and discussions.

References

1. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discr. Algorithms 2(1), 53–86 (2004)
2. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words. NATO ISI Series, pp. 85–96. Springer, Heidelberg (1985)
3. Brisaboa, N., Ladra, S., Navarro, G.: Directly addressable variable-length codes. In: Hyyro, H. (ed.) SPIRE 2009. LNCS, vol. 5721, pp. 122–130. Springer, Heidelberg (2009)
4. 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)
5. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM J. Exp. Algor. 13, article 12 (2009)
6. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM TALG 3(2), article 20 (2007)
7. 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)

8. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theor. Comp. Sci. 410(51), 5354–5364 (2009)
9. Geary, R., Rahman, N., Raman, R., Raman, V.: A simple optimal representation for balanced parentheses. Theor. Comp. Sci. 368, 231–246 (2006)
10. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Proc. 4th WEA (posters), pp. 27–38 (2005)
11. 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)
12. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
13. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
14. Kurtz, S.: Reducing the space requirements of suffix trees. Soft. Pract. Exp. 29(13), 1149–1171 (1999)
15. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proc. of the IEEE 88(11), 1722–1732 (2000)
16. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic J. Comp. 12(1), 40–66 (2005)
17. Manber, U., Myers, E.: Suffix arrays: a new method for on-line string searches. SIAM J. Comp., 935–948 (1993)
18. McCreight, E.: A space-economical suffix tree construction algorithm. J. ACM 32(2), 262–272 (1976)
19. Munro, I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
20. Munro, I., Raman, V., Rao, S.: Space efficient suffix trees. J. Algor. 39(2), 205–222 (2001)
21. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
22. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. 9th ALENEX (2007)
23. Puglisi, S., Turpin, A.: Space-time tradeoffs for longest-common-prefix array computation. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 124–135. Springer, Heidelberg (2008)
24. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. 13th SODA, pp. 233–242 (2002)
25. Russo, L., Navarro, G., Oliveira, A.: Fully-Compressed Suffix Trees. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 362–373. Springer, Heidelberg (2008)
26. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Algor. 48(2), 294–313 (2003)
27. Sadakane, K.: Compressed suffix trees with full functionality. Theor. Comp. Sys. 41(4), 589–607 (2007)
28. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. 21st SODA, pp. 134–149 (2010)
29. Välimäki, N., Gerlach, W., Dixit, K., Mäkinen, V.: Engineering a compressed suffix tree implementation. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 217–228. Springer, Heidelberg (2007)
30. Weiner, P.: Linear pattern matching algorithms. In: IEEE Symp. Swit. and Aut. Theo., pp. 1–11 (1973)

Maximum Cliques in Protein Structure Comparison

Noël Malod-Dognin[1], Rumen Andonov[2,*], and Nicola Yanev[3,4]

[1] South-West University "Neofit Rilski", Blagoevgrad, Bulgaria
nmaloddg@swu.bg

[2] INRIA Rennes - Bretagne Atlantique and University of Rennes 1, France
randonov@irisa.fr

[3] Faculty of Mathematics and Informatics, University of Sofia, Bulgaria
choby@math.bas.bg

[4] Institute of Mathematics and Informatics, Bulgarian Academy of Sciences

Abstract. Computing the similarity between two protein structures is a crucial task in molecular biology, and has been extensively investigated. Many protein structure comparison methods can be modeled as maximum clique problems in specific k-partite graphs, referred here as alignment graphs. In this paper, we propose a new protein structure comparison method based on internal distances (DAST), which main characteristic is that it generates alignments having RMSD smaller than any previously given threshold. DAST is posed as a maximum clique problem in an alignment graph, and in order to compute DAST's alignments, we also design an algorithm (ACF) for solving such maximum clique problems. We compare ACF with one of the fastest clique finder, recently conceived by Östergård. On a popular benchmark (the Skolnick set) we observe that ACF is about 20 times faster in average than the Östergård's algorithm. We then successfully use DAST's alignments to obtain automatic classification in very good agreement with SCOP.

Keywords: protein structure comparison, maximum clique problem, k-partite graphs, combinatorial optimization, branch and bound.

1 Introduction

A fruitful assumption in molecular biology is that proteins of similar three-dimensional (3D) structures are likely to share a common function and in most cases derive from a same ancestor. Understanding and computing the protein structures similarities is one of the keys for developing protein based medical treatments, and thus it has been extensively investigated [1,2]. Evaluating the similarity of two protein structures can be done by finding an optimal (according to some criterions) order-preserving matching (also called alignment) between their components. In this paper, we propose a new protein structure comparison method based on internal distances (DAST). Its main characteristic is to generate alignments having RMSD smaller than any previously given threshold. We show that finding such alignments is equivalent to solving maximum clique problems in specific k-partite graphs referred here as alignment graphs. These graphs

[*] Corresponding author.

could be very large (more than 25000 vertices and 3×10^7 edges) when comparing real proteins. Even very recent general clique finders [3,4] are oriented to notably smaller instances and are not able to solve problems of such size (the available code of [4] is limited to graphs with up to 1000 vertices).

For solving the maximum clique problem in this context we conceive an algorithm, denoted by *ACF* (for Alignment Clique Finder), which profits from the particular structure of the alignment graphs. We furthermore compare ACF to an efficient general clique solver [5] and the obtained results clearly demonstrate the usefulness of our dedicated algorithm. In addition, we show that the scores obtained by DAST allow to obtain an automatic classification in agreement with SCOP [6]. The main focus here is on designing an algorithm able to generate alignments with guaranteed small RMSD. Evaluating the quality of these alignments and its comparison with other structure alignment methods is beyond the scope of this paper and is a subject of our coming research.

Strickland et al. [7] also exploit the properties of the maximum cliques in protein-based alignment graphs. However, their approach considerably differs from ours: the alignment graphs are defined in a different manner (see section 1.3) and the authors in [7] concentrate on specialized preprocessing techniques in order to accelerate the solution of another optimization problem–Contact Map Overlap Maximization. The maximum cliques instances that are solved in [7] are much smaller than ours.

1.1 The Maximum Clique Problem

We usually denote an undirected graph by $G = (V,E)$, where V is the set of vertices and E is the set of edges. Two vertices i and j are said to be adjacent if they are connected by an edge of E. A clique of a graph is a subset of its vertex set, such that any two vertices in it are adjacent.

Definition 1. *The **maximum clique problem** (also called maximum cardinality clique problem) is to find a largest, in terms of vertices, clique of an arbitrary undirected graph G, which will be denoted by $MCC(G)$.*

The maximum clique problem is one of the first problem shown to be NP-complete [8] and it has been studied extensively in literature. Interested readers can refer to [9] for a detailed state of the art about the maximum clique problem. It can be easily proven that solving this problem in the context of k-partite graphs does not reduce its complexity.

1.2 Alignment Graphs

In this paper, we focus on grid alike graphs, which we define as follows.

Definition 2. *A $m \times n$ **alignment graph** $G = (V,E)$ is a graph in which the vertex set V is depicted by a (m-rows) $\times$ (n-columns) array T, where each cell $T[i][k]$ contains at most one vertex i.k from V (note that for both arrays and vertices, the first index stands for the row number, and the second for the column number). Two vertices i.k and j.l can be connected by an edge $(i.k, j.l) \in E$ only if $i < j$ and $k < l$. An example of such alignment graph is given in Fig 2a.*

It is easily seen that the m rows form a m-partition of the alignment graph G, and that the n columns also form a n-partition. In the rest of this paper we will use the following notations. A successor of a vertex $i.k \in V$ is an element of the set $\Gamma^+(i.k) = \{j.l \in V$ s.t. $(i.k, j.l) \in E, i < j$ and $k < l\}$. $V^{i.k}$ is the subset of V restricted to vertices in rows j, $i \leq j \leq m$, and in columns l, $k \leq l \leq n$. Note that $\Gamma^+(i.k) \subset V^{i+1.k+1}$. $G^{i.k}$ is the subgraph of G induced by the vertices in $V^{i.k}$. The cardinality of a vertex set U is $|U|$.

1.3 Relations with Protein Structure Similarity

In graph-theoretic language, two proteins P_1 and P_2 can be represented by two undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ where the sets of vertices V_1 and V_2 stand for residues/SSE, while edges depict contacts/relationships between them. The similarity between P_1 and P_2 can be estimated by finding the longest alignment between the elements of V_1 and V_2. In our approach, this is modeled by an alignment graph $G = (V, E)$ of size $|V_1| \times |V_2|$, where each row corresponds to an element of V_1 and each column corresponds to an element of V_2. A vertex $i.k$ is in V (i.e. matching $i \leftrightarrow k$ is possible), only if elements $i \in V_1$ and $k \in V_2$ are compatible. An edge $(i.k, j.l)$ is in E if and only if : (i) $i < j$ and $k < l$, for order preserving, and (ii) matching $i \leftrightarrow k$ is compatible with matching $j \leftrightarrow l$. A feasible alignment of P_1 and P_2 is then a clique in G, and the longest alignment corresponds to a maximum clique in G.

At least two protein structure similarity related problems from the literature can be converted into clique problems in alignment graphs : the secondary structure alignment in VAST[10], and the Contact Map Overlap Maximization problem (CMO)[11].

VAST, or Vector Alignment Search Tool, is a software for aligning protein 3D structures largely used in the National Center for Biotechnology Information[1]. In VAST, V_1 and V_2 contain 3D vectors representing the secondary structure elements (SSE) of P_1 and P_2. Matching $i \leftrightarrow k$ is possible if vectors i and k have similar norms and correspond either both to α-helices or both to β-strands. Finally, matching $i \leftrightarrow k$ is compatible with matching $j \leftrightarrow l$ only if the couple of vectors (i, j) from P_1 can be well superimposed in 3D-space with the couple of vectors (k, l) from P_2.

CMO is one of the most reliable and robust measures of protein structure similarity. Comparisons are done by aligning the residues (amino-acids) of two proteins in a way that maximizes the number of common contacts (when two residues that are close in 3D space are matched with two residues that are also close in 3D space). We have already dealt with CMO, but not using cliques [12]. The above definition of the alignment graph is inspired by the one we used and proved to be very successful in the case of CMO. There is a multitude of other alignment methods and they differ mainly by the nature of the elements of V_1 and V_2, and by the compatibility definitions between elements and between pairs of matched elements. One essential difference between our approach and the one used in [7] resides in the definition itself of the alignment graph. Every vertex in the so-called specially defined graph from [7] corresponds to an overlap of an edge/contact from P_1 with an edge/contact from P_2 and hence the graph size is $|E_1| \times |E_2|$, versus $|V_1| \times |V_2|$ in our definition.

1.4 DAST: An Improvement of CMO Based on Internal Distances

The objective in CMO is to maximize the number of common contacts. It has been shown that this objective finds a good global similarity score which can be successfully used

[1] http://www.ncbi.nlm.nih.gov/Structure/VAST/vast.shtml

for classification of structures [13,12]. However, such a strategy also introduces some "errors" in the structure based alignment–like aligning two residues that are close in 3D space with two residues that are remote, as illustrated in Fig 1. These errors could potentially yield alignments with big root mean square deviations (RMSD) which is not desirable for structures comparison. To avoid such problems we propose DAST (Distance-based Alignment Search Tool)–an alignment method based on internal distances which is modeled in an alignment graph. In DAST, two proteins P_1 and P_2 are represented by their ordered sets of residues V_1 and V_2. Two residues $i \in V_1$ and $k \in V_2$ are compatible if they come from the same kind of secondary structure elements (i.e. i and k both come from α-helices, or from β-strands) or if both come from loops. Let us denote by d_{ij} (resp. $d_{k.l}$) the euclidean distance between the α-carbons of residues i and j (resp. k and l). Matching $i \leftrightarrow k$ is compatible with matching $j \leftrightarrow l$ only if $|d_{ij} - d_{kl}| \leq \tau$, where τ is a distance threshold. The longest alignment in terms of residues, in which each couple of residues from P_1 is matched with a couple of residues from P_2 having similar distance relations, corresponds to a maximum clique in the alignment graph G. For example the clique $(2.1), (3.2), (4.3)$ in Fig 2a is generated by aligning residues $2, 3, 4$ from P_1 (rows) with residues $1, 2, 3$ from P_2 (columns).

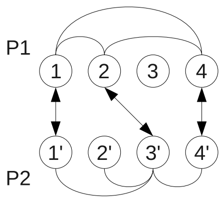


Fig. 1. An optimal CMO matching

Two proteins (P_1 and P_2) are represented by their contact map graphs where the vertices corresponds to the residues and where edges connect residues in contacts (i.e. close). The matching "$1 \leftrightarrow 1', 2 \leftrightarrow 3', 4 \leftrightarrow 4'$", represented by the arrows, yields two common contacts which is the maximum for the considered case. However, it also matches residues 1 and 4 from P_1 which are in contacts with residues $1'$ and $4'$ in P_2 which are remote.

Given a set of n deviations $S = \{s_1, s_2, \ldots, s_n\}$, its Root Mean Square Deviation ($RMSD$) is : $RMSD(S) = \sqrt{\frac{1}{n} \times \sum_{i=1}^{n} s_i^2}$. For assessing the quality of an alignment, the biologists use two different $RMSD$ measures which differ on the deviations they take into account. The first one is the $RMSD$ of superimposed coordinates ($RMSD_c$). After superimposing the two protein structures, the measured deviations are the euclidean distances between the matched amino-acid d_{ik}, for all matching pairs $i \leftrightarrow k$. The second one is the $RMSD$ of internal distances ($RMSD_d$). The measured deviations are $|d_{ij} - d_{kl}|$, for all couples of matching pairs "$i \leftrightarrow k, j \leftrightarrow l$". Let us denote by P the later set and by N_m its cardinality. We therefore have that $RMSD_d = \sqrt{\frac{1}{N_m} \times \sum_{(ij,jk) \in P} (|d_{ij} - d_{kl}|^2)}$ and

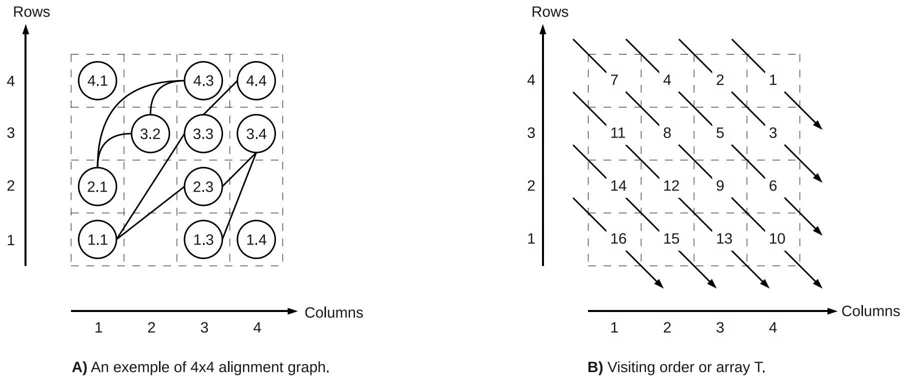


Fig. 2. A 4×4 alignment graph and the visiting order of its array T

since $|d_{ij} - d_{kl}| \leq \tau$ holds for all matching pairs "$i \leftrightarrow k, j \leftrightarrow l$", the alignments generated by DAST are characterized by the desired property $RMSD_d \leq \tau$.

2 Branch and Bound Approach

We have been inspired by [5] to propose our own algorithm which is more suitable for solving the maximum clique problem in the previously defined $m \times n$ alignment graph $G = (V, E)$. Let *Best* be the biggest clique found so far (first it is set to $\emptyset$), and $|\overline{MCC}(G)|$ be an over-estimation of $|MCC(G)|$. By definition, $V^{i+1.k+1} \subset V^{i.k+1} \subset V^{i.k}$, and similarly $V^{i+1.k+1} \subset V^{i+1.k} \subset V^{i.k}$. >From these inclusions and from definition 2, it is easily seen that for any $G^{i.k}$, $MCC(G^{i.k})$ is the biggest clique among $MCC(G^{i+1.k})$, $MCC(G^{i.k+1})$ and $MCC(G^{i+1.k+1}) \bigcup \{i.k\}$, but for the latter only if vertex $i.k$ is adjacent to all vertices in $MCC(G^{i+1.k+1})$. Let C be a $(m+1) \times (n+1)$ array where $C[i][k] = |\overline{MCC}(G^{i.k})|$ (values in row $m+1$ or column $n+1$ are equal to 0). For reasoning purpose, let assume that the upper-bounds in C are exact. If a vertex $i.k$ is adjacent to all vertices in $MCC(G^{i+1.k+1})$, then $C[i][k] = 1 + C[i+1][k+1]$, else $C[i][k] = \max(C[i][k+1], C[i+1][k])$. We can deduce that a vertex $i.k$ cannot be in a clique in $G^{i.k}$ which is bigger than *Best* if $C[i+1][k+1] < |Best|$, and this reasoning still holds if values in C are upper estimations. Another important inclusion is $\Gamma^+(i.k) \subset V^{i+1.k+1}$. Even if $C[i+1][k+1] \geq |Best|$, if $|\overline{MCC}(\Gamma^+(i.k))| < |Best|$ then $i.k$ cannot be in a clique in $G^{i.k}$ bigger than *Best*.

Our main clique cardinality estimator is constructed and used according to these properties. A function, Find_clique(G), will visit the cells of T according to north-west to south-est diagonals, from diagonal "$i+k = m+n$" to diagonal "$i+k = 2$" as illustrated in Fig 2b. For each cell $T[i][k]$ containing a vertex $i.k \in V$, it may call Extend_clique($\{i.k\}$, $\Gamma^+(i.k)$), a function which tries to extend the clique $\{i.k\}$ with vertices in $\Gamma^+(i.k)$ in order to obtain a clique bigger than *Best* (which cannot be bigger than |Best| +1). If such a clique is found, *Best* is updated. However, Find_clique() will call Extend_clique() only if two conditions are satisfied : (i) $C[i+1][k+1] = |Best|$ and (ii) $|\overline{MCC}(\Gamma^+(i.k))| \geq |Best|$. After the call to Extend_clique(), $C[i][k]$ is set to $|Best|$. For all other cells $T[i][k]$, $C[i][k]$ is set to $\max(C[i][k+1], C[i+1][k])$ if $i.k \notin V$, or to $1+C[i+1][k+1]$) if $i.k \in V$. Note that the order used for visiting the cells in T guaranties that when computing the value of $C[i][k]$, the values of $C[i+1][k]$, $C[i][k+1]$ and $C[i+1][k+1]$ are already computed.

Array C can also be used in function Extend_clique() to fasten the maximum clique search. This function is a branch a bound (B&B) search using the following branching rules. Each node of the B&B tree is characterized by a couple (Cli, $Cand$) where Cli is the clique under construction and $Cand$ is the set of candidate vertices to be added to Cli. Each call to Extend_clique($\{i.k\}$, $\Gamma^+(i.k)$) create a new B&B tree which root node is ($\{i.k\}$, $\Gamma^+(i.k)$). The successors of a B&B node (Cli,$Cand$) are the nodes ($Cli \bigcup \{i'.k'\}$, $Cand \bigcap \Gamma^+(i'.k')$), for all vertices $i'.k' \in Cand$. Branching follows lexicographic increasing order (row first). According to the branching rules, for any given B&B node (Cli, $Cand$) the following cutting rules holds : (i) if $|Cli|$ + $|Cand| \leq |Best|$ then the current branch cannot lead to a clique bigger than $|Best|$ and can be fathomed, (ii) if $|\overline{MCC}(Cand)| \leq |Best| - |Cli|$, then the current branch cannot lead to a clique bigger than $|Best|$, and (iii) if $|\overline{MCC}(Cand \bigcap \Gamma^+(i.k))| \leq |Best| - |Cli| - 1$, then branching on $i.k$ cannot lead to a clique bigger than $|Best|$. For any set $Cand$ and any vertex $i.k$, $Cand \bigcap \Gamma^+(i.k) \subset \Gamma^+(i.k)$, and $\Gamma^+(i.k) \subset G^{i+1.k+1}$. From these inclusions we can deduce two way of over-estimating $|MCC(Cand \bigcap \Gamma^+(i.k))|$. First, by using $C[i+1][k+1]$ which over-estimate $|MCC(G^{i+1.k+1})|$ and second, by over-estimating $|MCC(\Gamma^+(i.k))|$. All values $|\overline{MCC}(\Gamma^+(i.k))|$ are computed once for all in Find_clique() and thus, only $|\overline{MCC}(Cand)|$ needs to be computed in each B&B node.

3 Maximum Clique Cardinality Estimators

Even if the described functions depend on array C, they also use another upper-estimator of the cardinality of a maximum clique in an alignment graph. By using the properties of alignment graphs, we developed the following estimators.

3.1 Minimum Number of Rows and Columns

Definition 2 implies that there is no edge between vertices from the same row or the same column. This means that in a $m \times n$ alignment graph, $|MCC(G)| \leq \min(m,n)$. If the numbers of rows and columns are not computed at the creation of the alignment graph, they can be computed in $O(|V|)$.

3.2 Longest Increasing Subset of Vertices

Definition 3. *An* ***increasing subset of vertices*** *in an alignment graph* $G = \{V,E\}$ *is an ordered subset* $\{i_1.k_1, i_2.k_2, \ldots, i_t.k_t\}$ *of* V*, such that* $\forall j \in [1,t-1]$, $i_j < i_{j+1}$, $k_j < k_{j+1}$. $LIS(G)$ *is the longest, in terms of vertices, increasing subset of vertices of* G.

Since any two vertices in a clique are adjacent, definition 2 implies that a clique in G is an increasing subset of vertices. However, an increasing subset of vertices is not necessarily a clique (since vertices are not necessarily adjacent), and thus $|MCC(G)| \leq |LIS(G)|$. In a $m \times n$ alignment graph $G = (V,E)$, $LIS(G)$ can be computed in $O(n \times m)$ times by dynamic programming. However, it is possible by using the longest increasing subsequence to solve $LIS(G)$ in $O(|V| \times \ln(|V|))$ times which is more suited in the case of sparse graph like in our protein structure comparison experiments.

Definition 4. *The* ***longest increasing subsequence*** *of an arbitrary finite sequence of integers $S =$ "$i_i, i_2, \ldots, i_n$" is the longest subsequence $S' =$ "$i'_i, i'_2, \ldots, i'_t$" of S respecting the original order of S, and such that for all $j \in [1,t], i'_j < i'_{j+1}$. By example, the longest increasing subsequence of "1,5,2,3" is "1,2,3".*

For any given alignment graph $G = \{V,E\}$, we can easily reorder the vertex set V, first by increasing order of columns, and second by decreasing order of rows. Let's denote by V' this reordered vertex set. Then we can create an integer sequence S corresponding to the row indices of vertices in V'. For example, by using the alignment graph presented in Fig2a, the reordered vertex set V' is $\{4.1,\ 2.1,\ 1.1,\ 3.2, 4.3, 3.3,\ 2.3,\ 1.3,\ 4.4,\ 3.4,\ 1.4\}$, and the corresponding sequence of row indices S is "4, 2, 1, 3, 4, 3, 2, 1, 4, 3, 1". An increasing subsequence of S will pick at most one number from a column, and thus an increasing subsequence is longest if and only if it covers a maximal number of increasing rows. This proves that solving the longest increasing subsequence in S is equivalent to solving the longest increasing subset of vertices in G. Note that the longest increasing subsequence problem is solvable in time $O(l \times \ln(l))$ [14], where l denotes the length of the input sequence. In our case, this corresponds to $O(|V| \times \ln(|V|))$.

3.3 Longest Increasing Path

Definition 5. *An* ***increasing path*** *in an alignment $G = \{V,E\}$ is an increasing subset of vertex $\{i_1.k_1,\ i_2.k_2,\ \ldots,\ i_t.k_t\}$ such that $\forall j \in [1,t-1]$, $(i_j.k_j, i_{j+1}.k_{j+1}) \in E$. The longest increasing path in G is denoted by $LIP(G)$*

As the increasing path take into account edges between consecutive vertices, $|LIP(G)|$, should better estimate $MCC(G)|$. $|LIP(G)|$ can be computed in $O(|V|^2)$ by the following recurrence. Let $DP[i][k]$ be the length of the longest increasing path in $G^{i.k}$ containing vertex $i.k$. $DP[i][k] = 1 + \max_{i'.k' \in \Gamma^+ i.k}(DP[i'][k'])$. The sum over all $\Gamma^+(i.k))$ is done in $O(|E|)$ time complexity, and finding the maximum over all $DP[i][k]$ is done in $O(|V|)$. This results in a $O(|V| + |E|)$ time complexity for computing $|LIP(G)|$.

Any of the previously defined estimators can be used as bound generator in our B&B, and without them our algorithm is about 2.21 times slower than the Östergård's one. Experimentally, the longest increasing subset of vertices (solved using the longest increasing subsequence) exhibits the best performances, allowing our algorithm to be about 20 times faster than the Östergård's one, and is the bound generator that we used for obtaining the optimal alignments presented in the next section.

4 Results

All results presented in this section come from real protein structure comparison instances. Our algorithm, denoted by *ACF* (for Alignment Clique Finder), has been implemented in C and was tested in the context of DAST. *ACF* will be compared to the fast clique finder (denoted by here *Östergård*) which has been proposed in [5] and which code is publicly available.

4.1 Residues Alignment

In this section we compare *ACF* to *Östergård* in the context of residue alignments in DAST. Computations were done on a PC with an Intel Core2 processor at 3Ghz, and for both algorithms the computation time was bounded to 5 hours per instance. Secondary structures assignments were done by KAKSI [15], and the threshold distance τ was set to 3Å. The protein structures come from the well known Skolnick set, described in [16]. It contains 40 protein chains having from 90 to 256 residues, classified in SCOP[6] (v1.73) into five families. Amongst the 780 corresponding alignment instances, 164 align protein chains from the same family and will be called “similar”. The 616 other instances align protein chains from different families and thus will be called “dissimilar”. Characteristics of the corresponding alignment graphs are presented in **table 1**.

Table 1. DAST alignment graphs characteristics

		array size	$\|V\|$	$\|E\|$	density	$\|MCC\|$
similar instances	min	97×97	4018	106373	8.32%	45
	max	256×255	25706	31726150	15.44%	233
dissimilar instances	min	97×104	1581	77164	5.76%	12
	max	256×191	21244	16839653	14.13%	48

All alignment graphs from DAST have small edge density (less than 16%). Similar instances are characterized by bigger maximum cliques than the dissimilar instances.

Table 2. Number of solved instances comparison

	Östergård	ACF
Similar instances (164)	128	**155**
Dissimilar instances (616)	545	**616**
Total (780)	673	**771**

On the Skolnick set *ACF* solves 21% more similar instances and 13% more dissimilar instances than *Östergård* when the running time was upper-bounded by 5 hours per instance.

Table 2 compares the number of instances solved by each algorithm on Skolnick set. Note that when an instance is solved, the B&B algorithm finds both the optimal score (maximum clique cardinality), as well as the corresponding residues alignment. *ACF* solved 155 from 164 similar instances, while *Östergård* solved 128 instances. *ACF* was able to solve all 616 dissimilar instances, while *Östergård* solved 545 instances only. Thus, on this popular benchmark set, *ACF* clearly outperforms *Östergård* in terms of number of solved instances.

Figure 3 compares the running time of *ACF* to the one of *Östergård* on the set of 673 instances solved by both algorithms (all instances solved by *Östergård* were also solved by *ACF*). For all but one instances, *ACF* is significantly faster than *Östergård*. More precisely, *ACF* needed 12 hs. 29 min. 56 sec. to solve all these 673 instances,

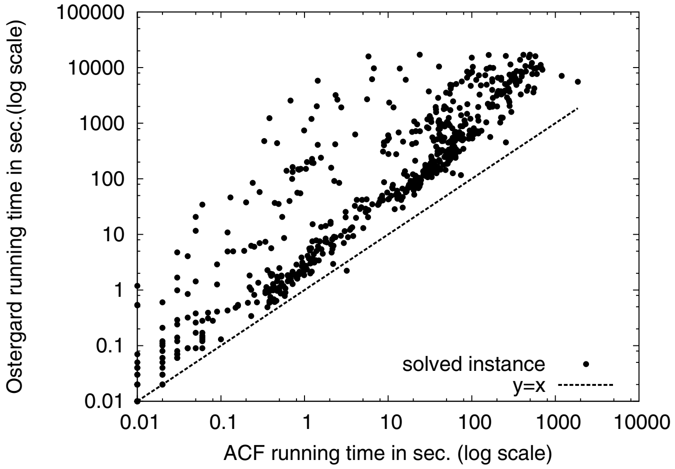


Fig. 3. Running time comparison on Skolnick set

ACF versus *Östergård* running time comparison on the set of the 673 Skolnick instances solved by both algorithms. The *ACF* time is presented on the x-axis, while the one of *Östergård* is on the y-axis. For all instances except one, *ACF* is faster than *Östergård.*

while *Östergård* needed 260 hs. 10 min. 10 sec. Thus, on the Skolnick set, *ACF* is about 20 times faster in average than *Östergård*, (up to 4029 times for some instances).

4.2 Comparison between DAST's and CMO's Alignments

In order to compare the alignments of DAST to the ones of CMO[12], we extracted from the Skolnick set 10 instances that are optimally solved by both methods (see table 3). The five "similar" instances compare protein structures coming from the same SCOP family, while the five "dissimilar" instances compare protein structures coming from different SCOP families. The distance threshold of DAST was set to 3 Å(which corresponds to the desired $RMSD_d$ of alignments), while the contact threshold of CMO was set to 7.5 Å(optimal value according to [13]).

Table 3 compares the obtained alignments, both in terms of length (percentage of aligned amino-acids) and in terms of $RMSD_d$. The alignments of CMO for similar proteins are very good : they are both long and possess small $RMSD_d$ values. However, for dissimilar proteins, the alignments of CMO possess very bad $RMSD_d$ values, which means that they do not correspond to common substructures. On the other hand, for both similar and dissimilar proteins, the alignments of DAST always possess small $RMSD_d$ values (smaller than the perviously fixed threshold). DAST's alignments are shorter than the ones of CMO, but their lengths better reflect the similarity between two proteins, since the alignments between similar proteins are always much longer than the alignments between dissimilar proteins. Note that this property does not hold for CMO's alignments.

Table 3. CMO vs DAST alignments

		Length (AA %)		$RMSD_d$ (Å)	
	Instance	CMO	DAST	CMO	DAST
similar instances	1amkA–1aw2A	**97.4 %**	78.9 %	1.39	**0.68**
	1amkA–1htiA	**99.0 %**	81.8 %	1.24	**0.74**
	1qmpA–1qmpB	**99.2 %**	90.8 %	**0.22**	**0.22**
	1ninA–1plaA	**96.0 %**	57.4 %	1.42	**0.96**
	1tmhA–1treA	**99.8 %**	91.6 %	0.90	**0.44**
dissimilar instances	1amkA–1b00A	**63.5 %**	21.7 %	5.62	**1.23**
	1amkA–1dpsA	**78.0 %**	15.3 %	13.01	**1.06**
	1b9bA–1dbwA	**68.3 %**	24.4 %	6.02	**1.11**
	1qmpA–2pltA	**83.3 %**	15.0 %	7.36	**1.18**
	1rn1A–1b71A	**70.5 %**	17.6 %	11.22	**0.82**

Similar instances compare proteins coming from the same SCOP family, while dissimilar instances compare proteins coming from different SCOP families. The distance threshold of DAST was set to 3 Å, while the contact threshold of CMO was set to 7.5 Å. Columns 3 and 4 compare the length of the alignments (in percentage of aligned amino-acids), while columns 5 and 6 compare the $RMSD_d$ of the alignments. DAST's alignments always possess good (small) $RMSD_d$ values, but are shorter than CMO's ones.

Table 4. DAST classification of the Skolnick set

DAST class	SCOP Family	Proteins
1	CheY-related	1b00A, 1dbwA, 1natA, 3chyA 1qmp(A,B,C,D), 4tmy(A,B)
2	CheY-related	1ntrA
3	Plastocyanin /azurin-like	1bawA, 1byo(A,B), 1kdiA, 1ninA 1plaA, 2b3iA, 2pcyA, 2pltA
4	Triosephosphate isomerase (TIM)	1amkA, 1aw2A, 1b9bA, 1btmA, 1htiA 1tmhA, 1treA, 1triA, 1ydvA, 3ypiA, 8timA
5	Ferritin	1b71A, 1bcfA, 1dpsA, 1fhaA, 1ierA, 1rcdA
6	Fungal ribonucleases	1rn1(A,B,C)

The classification returned by CHAVL based on similarity score found by DAST, is very similar to the SCOP classification, except for the protein chain 1ntrA (class 2) which is not recognized as a CheY-related protein.

4.3 Automatic Classification

In this section, we test the possibility to obtain good automatic classifications based on DAST's alignments. For this purpose we used the following protocol: on the Skolnick set, the runs of DAST were limited to 5 hours per instance. The similarity score between two proteins P_1 and P_2 (having respectively $|V_1|$ and $|V_2|$ amino-acids) was defined as $SIM(P_1,P_2) = \dfrac{2 \times N_m}{|V_1|+|V_2|}$, where N_m is the number of aligned amino-acids (i.e. the size of the biggest clique found by DAST). These scores were given to CHAVL [17], an unsupervised ascendant classification tool based on likelihood maximization, and the obtained classification was compared to SCOP classification [6], which is a curated classification of the protein structures.

Table 4 presents the obtained classification. It is very similar to the one of SCOP, except that the protein chain "1ntrA" is not classified with the other members of its SCOP family. We detected that this error was provoked by Kaksi's secondary structure assignment of 1ntrA, which is not in agreement with the one used in SCOP.

5 Conclusion and Future Work

In this paper we introduce a novel protein structure comparison approach DAST, for Distance-based Alignment Search Tool. For any fixed threshold τ, it finds the longest alignment in which each couple of pairs of matched residues shares the same distance relation (+/- τ), and thus the RMSD of the alignment is $\leq \tau$. This property is not guaranteed by the CMO approach, which inspired initially DAST. From computation standpoint, DAST requires solving the maximum clique problem in a specific *k*-partite graph. By exploiting the peculiar structure of this graph, we design a new maximum clique solver which significantly outperforms one of the best general maximum clique solver. Our solver was successfully integrated into DAST and will be freely available soon. We are currently studying the quality of DAST alignments from practical viewpoint and compare the obtained results with other structure comparison methods.

Acknowledgments

Rumen Andonov is supported by PROTEUS "ANR-06-CIS6-008" and by BioWIC "ANR-08-SEGI-005" projects. Noël Malod-Dognin is supported by project DVU/01/197 from the South-West University, Blagoevgrad, while Nicola Yanev is supported by projects DO 02359/2008 and DTK02/71. All computations were done on the Ouest-genopole bioinformatics platform (http://genouest.org). We would like to express our gratitude to Jean-François Gibrat for numerous helpful discussions.

References

1. Godzik, A.: The structural alignment between two proteins: Is there a unique answer? Protein Science (7), 1325–1338 (1996)
2. Sierk, M., Kleywegt, G.: Déjà vu all over again: Finding and analyzing protein structure similarities. Structure 12(12), 2103–2111 (2004)
3. Konc, J., Janezic, D.: An efficient branch-and-bound algorithm for finding a maximum clique. Discrete Mathematics and Theoretical Computer Science 58, 220 (2003)
4. Tomita, E., Seki, T.: An improved branch and bound algorithm for the maximum clique problem. Communications in Mathematical and in Computer Chemistry / MATCH 58, 569–590 (2007)
5. Östergård, P.R.J.: A fast algorithm for the maximum clique problem. Discrete Applied Mathematics 120(1-3), 197–207 (2002)

6. Andreeva, A., Howorth, D., Chandonia, J.M., Brenner, S., Hubbard, T., Chothia, C., Murzin, A.: Data growth and its impact on the SCOP database: new developments. Nucl. Acids Res. 36, 419–425 (2007)
7. Strickland, D., Barnes, E., Sokol, J.: Optimal protein structure alignment using maximum cliques. Oper. Res. 53(3), 389–402 (2005)
8. Karp, R.: Reducibility among combinatorial problems. Complexity of Computer Computations 6, 85–103 (1972)
9. Bomze, I., Budinich, M., Pardalos, P., Pelillo, M.: The maximum clique problem. Handbook of Combinatorial Optimization (1999)
10. Gibrat, J.F., Madej, T., Bryant, S.: Surprising similarities in structure comparison. Current Opinion in Structural Biology 6, 377–385 (1996)
11. Godzik, A., Skolnick, J.: Flexible algorithm for direct multiple alignment of protein structures and sequences. CABIOS 10, 587–596 (1994)
12. Andonov, R., Yanev, N., Malod-Dognin, N.: An efficient lagrangian relaxation for the contact map overlap problem. In: Crandall, K.A., Lagergren, J. (eds.) WABI 2008. LNCS (LNBI), vol. 5251, pp. 162–173. Springer, Heidelberg (2008)
13. Caprara, A., Carr, R., Israil, S., Lancia, G., Walenz, B.: 1001 optimal PDB structure alignments: integer programming methods for finding the maximum contact map overlap. J. Comput. Biol. 11(1), 27–52 (2004)
14. Fredman, M.: On computing the length of longest increasing subsequences. Discrete Mathematics 11, 29–35 (1975)
15. Martin, J., Letellier, G., Marin, A., Taly, J.F., de Brevern, A., Gibrat, J.F.: Protein secondary structure assignment revisited: a detailed analysis of different assignment methods. BMC Structural Biology 5, 17 (2005)
16. Lancia, G., Carr, R., Walenz, B., Istrail, S.: 101 optimal pdb structure alignments: a branch-and-cut algorithm for the maximum contact map overlap problem. In: RECOMB 2001: Proceedings of the fifth annual international conference on Computational biology, pp. 193–202 (2001)
17. Lerman, I.: Likelihood linkage analysis (lla) classification method (around an example treated by hand). Biochimie 75(5), 379–397 (1993)

Exact Bipartite Crossing Minimization under Tree Constraints*

Frank Baumann[1], Christoph Buchheim[1], and Frauke Liers[2]

[1] Technische Universität Dortmund, Fakultät für Mathematik, Vogelpothsweg 87, 44227 Dortmund, Germany
[2] Universität zu Köln, Institut für Informatik, Pohligstraße 1, 50969 Köln, Germany

Abstract. A *tanglegram* consists of a pair of (not necessarily binary) trees. Additional edges, called *tangles*, may connect the leaves of the first with those of the second tree. The task is to draw a tanglegram with a minimum number of tangle crossings while making sure that the trees are drawn crossing-free. This problem has relevant applications in computational biology, e.g., for the comparison of phylogenetic trees. Most existing approaches are only applicable for binary trees. In this work, we show that the problem can be formulated as a quadratic linear ordering problem (QLO) with side constraints. Buchheim et al. (INFORMS J. Computing, to appear) showed that, appropriately reformulated, the QLO polytope is a face of some cut polytope. It turns out that the additional side constraints do not destroy this property. Therefore, any polyhedral approach to max-cut can be used in our context. We present experimental results for drawing random and real-world tanglegrams defined on both binary and general trees. We evaluate linear as well as semidefinite programming techniques. By extensive experiments, we show that our approach is very efficient in practice.

Keywords: tanglegram, graph drawing, computational biology, crossing minimization, quadratic programming, maximum cut problem.

1 Introduction

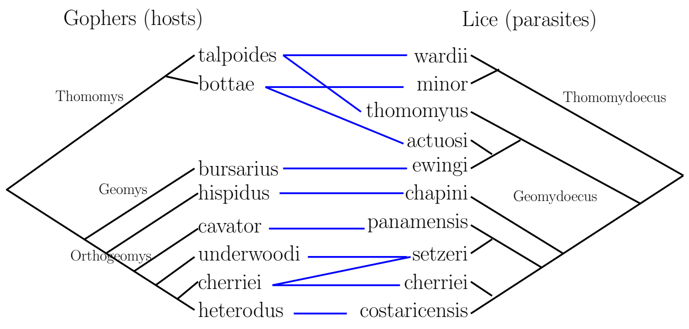
A *tanglegram* [11] consists of a pair of trees T_1, T_2 and a correspondences between the leaf sets L_1 and L_2 of T_1 and T_2, respectively. The correspondence is represented by edges between leaves in L_1 and L_2 called *tangles*. When visualizing a tanglegram, it is natural to ask for a drawing in which no edge crossings occur within either of the trees, while the number of tangle crossings is minimized. We require that the leaves in L_1 and L_2 are drawn on two parallel lines, while the trees are drawn outside the strip bounded by these lines.

The task of drawing tanglegrams arises in several relevant applications, e.g., in computational biology for the comparison of phylogenetic trees [11]. A phylogenetic tree represents a hypothesis of the evolutionary history of a set of species.

* Financial support from the German Science Foundation (DFG) is acknowledged under contracts Bu 2313/1–1 and Li 1675/1–1.

These species are drawn as the leaves of the tree, their ancestors as inner nodes. Different reconstruction methods may lead to a set of different candidate trees; a tanglegram layout then allows to compare a pair of such trees visually.

As another application, consider a phylogenetic tree of some set of species that serve as hosts for a certain set of parasites. The hypothesis that the evolution of hosts and their parasites is strongly correlated can be tested by analyzing a tanglegram layout. A tangle then specifies which host is affected by which parasite. Whereas in the first application the number of tangles incident to a leaf is always one, in the latter it can be higher, as shown below in a tanglegram from Hafner et al. [6] (here the hypothesis seems to be true).



Tanglegrams also occur in hierarchical clusterings, which can be visualized by so-called *dendrograms*. Dendrograms consist of trees where the elements to be clustered are identified with the leaves. Internal nodes determine clusters that contain the elements or sub-clusters. A tanglegram layout helps comparing the results of different clustering methods. Moreover, tanglegrams occur when analyzing software projects in which a tree represents package, class and method hierarchies. Hierarchy changes are analyzed over time, or automatically generated decompositions are compared with human-made ones. This application yields tanglegrams on trees that are not binary in general [10].

In the next section, we review related work. In Section 3, we introduce an exact model for tanglegrams that can be applied to pairs of general (not necessarily binary) trees with arbitrary tangle density. To this end, we show that the task is to optimize a quadratic function over the linear ordering polytope intersected with further hyperplanes. We show that the corresponding polytope is isomorphic to a face of a cut polytope. We compare and evaluate different solution methods based on both linear and semidefinite approaches in Section 4. We show results for random as well as real-world instances. The results prove that our approach is very efficient in practice.

2 Related Work

Most of the literature is concerned with the case of binary trees and leaves that are in one-to-one correspondence. Whereas several of the presented methods could easily be generalized to arbitrary tangle layers, an extension to non-binary trees is usually not possible. When allowing general trees, one extreme case would be a star, where all leaves are adjacent to the root node. If both T_1 and T_2 are stars, there are no constraints on the orders of leaves on either shore, so that the problem specializes to the bipartite crossing minimization problem [9, 2].

We are not aware of any implementation of an exact method for drawing tanglegrams with non-binary trees. Fernau et al. [5] showed the NP-hardness of tanglegram layout, even in the case of binary trees. They also presented a fixed-parameter algorithm for binary tanglegrams. Recently, an improved fixed-parameter algorithm was presented by Böcker et al. [1] which can solve large binary instances quickly in practice, provided that the number of crossings is not too large. Finally, while in the recent paper by Venkatachalam et al. [13] the focus is on binary instances, a fixed-parameter algorithm for general tanglegram instances is presented. According to our knowledge, this is the only algorithm that could deal with non-binary trees; however, no implementation or running times are provided making it impossible to evaluate its practical performance.

Besides analyzing the performance and quality of several heuristics in a computational study for binary tanglegrams with one-to-one tangles, Nöllenburg et al. [10] also implemented a branch-and-bound algorithm and an exact integer-programming (IP) based approach for this case.

As we will compare our approach with the exact IP-approach of Nöllenburg et al. [10], we describe it in more detail in the following. A feasible but not necessarily optimal tanglegram layout is given as an input. For each inner node, a binary variable x_i is introduced. In the case of complete binary trees with n leaves each, this gives rise to $2(n-1)$ variables. If $x_i = 1$, the subtree rooted in node i is flipped with respect to the input drawing, otherwise it remains unchanged. As by definition there are no crossings within the trees, the number of crossings can be determined by counting the number of tangle crossings. Let (a, c) and (b, d) be tangles with $a, b \in L_1$ and $c, d \in L_2$. Let i be the lowest common ancestor of a, b in T_1 and j that of c, d in T_2. If the tangles cross each other in the input drawing, then a crossing occurs in the output drawing if and only if either both subtrees below i and j are flipped or both remain unchanged. This can be expressed as $x_i x_j = 1$ or $(1-x_i)(1-x_j) = 1$. Similarly, if the edges do not cross each other in the input drawing, then there is a crossing in the output drawing if and only if either $(1-x_i)x_j = 1$ or $x_i(1-x_j) = 1$.

Thus minimizing the number of tangle crossings reduces to minimizing the sum of the given products. The latter is an instance of the unconstrained quadratic binary optimization problem, which is well-known to be equivalent to a maximum cut problem in some associated graph with an additional node [3]. In an undirected graph $G = (V, E)$, the cut $\delta(W)$ induced by a set $W \subseteq V$ is defined as the set of edges (u, v) such that $u \in W$ and $v \notin W$. If edge weights are given, the weight of a cut is the total weight of edges in the cut. Now the maximum cut problem asks for a cut of maximal weight or cardinality.

While Nöllenburg et al. used this model only for instances with one-to-one tangles, they briefly note that it could be extended to leaves of higher degree as well. However, their model cannot be generalized to instances with non-binary trees in a straightforward way. In many applications, the trees are not necessarily binary. In the next section we will present an exact model for tanglegrams that neither restricts the degree of inner nodes in the trees nor the number of tangles incident to a leaf.

3 An Exact Model for General Tanglegrams

The problem of drawing tanglegrams is closely related to bipartite crossing minimization. As argued above, the latter problem can be considered a special case of the former. Therefore, we first review approaches for drawing bipartite graphs.

3.1 Bipartite Crossing Minimization

Let $G = (V_1 \cup V_2, E)$ be a bipartite graph. The task is to draw G with straight line edges. The nodes in V_1 and V_2 have to be placed on two parallel lines H_1 and H_2 such that the number of edge crossings is minimal. Both heuristic and exact methods [9] exist for this problem.

Assume for a moment that the nodes on the first layer H_1 are fixed, and only the nodes on layer H_2 are permuted. For each pair of nodes on H_2, we introduce a variable x_{uv} such that $x_{uv} = 1$ if u is drawn to the left of v and $x_{uv} = 0$ otherwise. For edges (i, k) and (j, l) with $i, j \in H_1$ and $k, l \in H_2$, such that i is left of j, a crossing exists if and only if l is left of k. We thus have to punish x_{lk} in the objective function. The task of minimizing the number of crossings is now equivalent to determining a minimum linear ordering on the nodes of H_2. Exploiting $x_{uv} = 1 - x_{vu}$, we can eliminate half of the variables and only keep those with $u < v$. Note that bipartite crossing minimization with one fixed layer is already NP-hard [4].

If the nodes on both layers are allowed to permute, the number of crossings depends on the order of the nodes on each layer. Therefore, the problem can be modeled as a quadratic optimization problem over linear ordering variables. We write the quadratic linear ordering problem (QLO) in its general form as

$$(QLO) \quad \begin{array}{ll} \min & \sum_{(i,j,k,l)\in I} c_{ijkl} x_{ij} x_{kl} \\ \text{s.t.} & x \in P_{LO} \\ & x_{ij} \in \{0,1\} \ \text{ for all } (i,j) \in J \end{array}$$

where P_{LO} is the linear ordering polytope, i.e. the convex hull of the incidence vectors of all linear orderings. The index set I consists of all quadruples (i, j, k, l) such that $x_{ij} x_{kl}$ occurs as a product in the objective function, while J is the set of all pairs (i, j) for which a linear ordering variable x_{ij} is needed. For the bipartite crossing minimization case, I and J are given as

$$I = \{(i, j, k, l) \mid i, j \in H_1,\ i < j, \text{ and } k, l \in H_2,\ k < l\}$$
$$J = \{(i, j) \mid i, j \in H_1 \text{ or } i, j \in H_2,\ i < j\}$$

In order to linearize the objective function, we introduce a new binary variable y_{ijkl} for each $(i,j,k,l) \in I$, modeling the product $x_{ij}x_{kl}$. (Note that $y_{ijkl} = y_{klij}$.) Applying the standard linearization, the corresponding linearized quadratic linear ordering problem (LQLO) can be written as

$$\begin{array}{lll} & \min & \sum_{(i,j,k,l)\in I} c_{ijkl}y_{ijkl} & \\ (LQLO) & \text{s.t.} & \quad x \in P_{LO} & \\ & & x_{ij} \in \{0,1\} & \text{for all } (i,j) \in J \\ & & y_{ijkl} \leq x_{ij}, x_{kl} & \text{for all } (i,j,k,l) \in I \\ & & y_{ijkl} \geq x_{ij} + x_{kl} - 1 & \text{for all } (i,j,k,l) \in I \\ & & y_{ijkl} \in \{0,1\} & \text{for all } (i,j,k,l) \in I. \end{array}$$

Buchheim et al. [2] introduced the above model for bipartite crossing minimization. Additionally, a quadratic reformulation of the constraints defining P_{LO} was given: it was shown that a 0/1 vector (x,y) satisfying $y_{ijkl} = x_{ij}x_{kl}$ is contained in (LQLO) if and only if

$$x_{ik} - y_{ijik} - y_{ikjk} + y_{ijjk} = 0 \text{ for all } (i,j,k,l) \in I. \tag{1}$$

Note that (LQLO) is a quadratic binary optimization problem where the feasible solutions need to satisfy further side constraints, namely those restricting the set of feasible solutions to linear orderings. As unconstrained binary quadratic optimization is equivalent to the maximum cut problem [3], the task is to intersect a cut polytope with a set of hyperplanes.

In general, the convex hull of the corresponding feasible incidence vectors has a structure that is very different from that of a cut polytope. In the above context, however, Buchheim et al. [2] showed that the hyperplanes (1) cut out faces of the cut polytope.

3.2 Modeling Tanglegrams

Crossing minimization in tanglegrams can be seen as a generalization of bipartite crossing minimization. The set of feasible orderings is implicitly restricted by the given tree structures. Starting from the model discussed above, we formalize these restrictions as follows: let us consider a triple of leaves a, b, c in one of the trees, say T_1. In case all pairwise lowest common ancestors coincide, all relative orderings between a, b, and c are feasible. However, if the lowest common ancestor of, say, a and b is on a lower level than that of, say, a and c (in this case, the former is a descendant of the latter), then c must not be placed between a and b, as an intra-tree crossing would be induced; see Figure 1.

Therefore, we derive a betweenness restriction for every triple of leaves such that two of the leaf pairs have different lowest common ancestors. Each such betweenness restriction of the form 'c cannot be placed between a and b' can be written in linear ordering variables as $x_{ac}x_{cb} = 0$ and $x_{bc}x_{ca} = 0$. In the linearized model (LQLO), the latter amounts to requiring

$$y_{accb} = 0 \text{ and } y_{cabc} = 0\ . \tag{2}$$

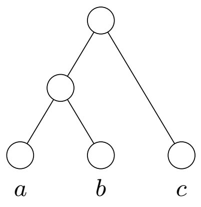


Fig. 1. Leaf c is not allowed to lie between a and b

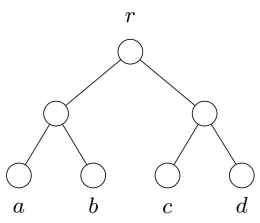


Fig. 2. Variables x_{ac} and x_{bd} can be identified

For binary trees with n leaves each, all triples of leaves have two different lowest common ancestors, so in this case the number of additional equations is $2\binom{n}{3}$.

In summary, we now obtain a quadratic linear ordering problem (QLO) on a smaller number of variables, with additional constraints of the form (2), where

$$J = \{(i,j) \mid i,j \text{ are leaves of the same tree, } i<j\}$$
$$I = \{(i,j,k,l) \mid (i,j),(k,l) \in J \text{ belong to different trees}\} \, .$$

For complete binary trees with n leaves each, the total number of linear ordering variables is $2\binom{n}{2}$. The same number of variables is necessary in the corresponding bipartite crossing minimization model [2].

As mentioned above, the polytope corresponding to the linearized problem (LQLO) is isomorphic to a face of a cut polytope [2]. Since all y-variables are binary, constraints of the form (2) are always face-inducing for (LQLO). In summary, we derive the following result:

Theorem 1. *The problem of drawing tanglegrams with a minimum number of edge crossings can be solved by optimizing over a face of a suitable cut polytope.*

3.3 Binary Case

In the binary case, the model introduced in the preceding sections is closely related to the model presented in [10]. To see this, first observe that the two equations (2) can be written as

$$x_{ac} = x_{bc} \, . \tag{3}$$

This replacement does not affect the set of feasible solutions, even the corresponding LP-relaxations of (LQLO) are equivalent. Note however that introducing the y-variables allows to strengthen the model, see Theorem 1.

When using the linear equations (3) instead of the quadratic equations (2), we end up with a set of equivalence classes of linear ordering variables such that all pairwise orderings corresponding to variables in the same class can only be flipped simultaneously. Two variables x_{ac} and x_{bd} belong to the same class if and only if there is a node r such that a, b and c, d are descendants of different children of r; see Figure 2. In the binary case, a class of linear ordering variables thus corresponds to the decision of flipping the children of node r or not, which is modeled explicitly by a single variable in the model of Nöllenburg et al. [10].

In the general case, however, where node r has k children, there are $k!$ different orderings. As these cannot be modeled by a single binary variable, the model of Nöllenburg et al. [10] cannot be applied here.

4 Computational Results

We implemented the model detailed in Section 3.2. Instead of adding equations (3) explicitly, we used one variable for each equivalence class of linear ordering variables, thereby significantly reducing the number of variables. For evaluating the IP-based methods, we used CPLEX 11.2 [8]. The naive approach is to solve the linearized model (LQLO) using a standard integer programming solver. A more advanced approach is to solve the quadratic reformulation (1), using separation of cutting planes for max-cut, both in the context of integer and semidefinite programming. For the SDP approaches, we used a branch-and-bound approach by Rendl et al. [12] employing the bundle method. For comparison, we also implemented the IP model by Nöllenburg et al. [10] that only works for binary tanglegrams. For the tested binary instances, the running times for solving the latter are very comparable to those for our model. This can be expected since our model generalizes [10].

We generated random instances on general binary, ternary and quad trees. I.e., the degree of each internal node is at most 2, 3 or 4, respectively. Each tree has n leaves, either having one-to-one tangles or a certain tangle density $d\%$. Instances are generated following the description in [10], with obvious extensions to the more general cases considered here. Finally, we solved real-world binary tanglegram instances from [10] arising in applications in biology and general real-world instances from visualizing software hierarchies [7].

Average results are always computed over 5 randomly generated instances. For each instance, we imposed an upper limit of 10h of CPU time. Instances that could not be solved within this limit count with 10h in the averages. Runs were performed on Intel Xeon machines with 2.33GHz.

In Table 1, we present the average cpu time in seconds for real-world binary instances. Table 2 shows results for random ternary and quad trees, respectively. Figure 4 visualizes the results from Table 2 for $n = 128$. Running times for real-world general tanglegram instances are presented in Table 3.

The first column *SDP* shows results obtained by semidefinite optimization, whereas the remaining columns refer to IP-based approaches for solving the model from Section 3. *IP* refers to solving the standard linearization using CPLEX default, *QP* its quadratic reformulation (1). In the options *IP+cyc* and *QP+cyc*, cycle inequalities for max-cut are additionally separated, and all CPLEX cuts are switched off.

Clearly, for real-world binary trees with one-to-one tangles, the SDP approach usually needs considerably more time than the IP-based methods. Furthermore,

Table 1. Average cpu time in seconds for real-world one-to-one binary trees having n leaves each [10]. Instances are grouped by their number of leaves. The second column shows the number of instances in each group. The nine largest instances with up to 540 leaves could not all be computed within the time and memory constraints and are omitted.

n	# inst	SDP	IP	QP	IP+cyc	QP+cyc
0–49	2988	<1	<1	<1	<1	14
50–99	717	3	<1	2	<1	428
100–149	102	31	<1	7	<1	1100
150–199	42	125	1	32	1	1282
200–249	18	437	1	66	2	2704
250–299	18	4786	2	2529	7	10602
300–349	9	6483	2	80	9	8862
400–449	3	20508	12	12067	69	14931

Table 2. Average cpu time in seconds for random general ternary (left) and quad trees (right) having n leaves each, density d%

n	d	SDP	IP	QP	IP+cyc	QP+cyc	SDP	IP	QP	IP+cyc	QP+cyc
64	1	3	1	<1	<1	1	<1	<1	<1	<1	<1
	5	15	12	4	67	6	2	3	1	1	<1
	10	18	27	19	1301	17	3	8	2	5	1
	15	78	54	23	3893	37	4	16	6	57	3
	20	41	54	48	12508	66	4	19	7	65	3
128	1	165	16	32	78	63	32	19	3	2	4
	5	362	448	280	22253	448	80	59	35	1045	51
	10	436	2179	791	36000	2746	73	1406	119	10147	495
	15	4049	3247	1551	36000	5583	77	844	352	31582	1184
	20	8293	2326	1408	36000	15426	1420	1560	1908	36000	10111

memory requirements strongly increase with system size and so the largest instances could not be solved. On average, the fastest approaches for solving the largest instances are the pure standard linearization *IP* and the quadratic reformulation *QP*.

In fact, we can optimize tanglegrams with more than 500 leaves in each tree. This is the range of sizes arising in real-world applications. The real-world instances can be solved particularly fast. Interestingly, cycle separation for max-cut usually does not pay off for binary one-to-one tanglegrams: the running time increases, although the dual bounds are usually considerably better when cycle separation is included. Often, an optimum solution can be determined in the root node. However, although the root bound is weak in the standard linearization, after few branching steps the optimum LP solution is often feasible and the program can stop. We found similar characteristics for random binary tanglegram instances.

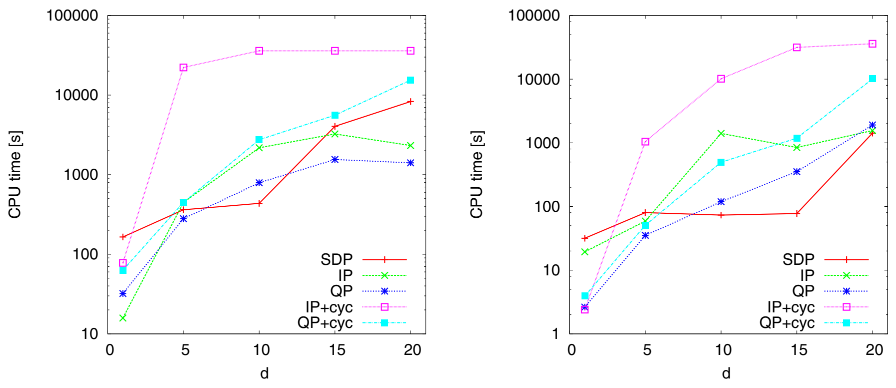


Fig. 3. Plot showing results for $n = 128$ of Table [2] for ternary (left) and quad trees (right)

Table 3. CPU time in seconds for real-world general tanglegram instances [7]. δ specifies the maximum node degree.

instance	δ	SDP	IP	QP	IP+cyc	QP+cyc
philips_orig_4a	14	27618	4725	73	4321	32692
philips_orig_4b	14	26093	4205	74	2756	23443
philips_orig_4c	14	36000	2666	122	4030	36000
philips_orig_4d	14	27891	3208	314	4178	28902
philips_orig_4e	14	36000	2789	90	5665	36000
philips_4a_4b	9	5238	1395	4	420	5
philips_4a_4c	9	4769	1858	3	637	4
philips_4a_4d	9	2924	1467	4	494	3
philips_4a_4e	9	2575	827	2	338	3
philips_4b_4c	8	6526	1965	8	510	7
philips_4b_4d	8	4872	2070	5	577	5
philips_4b_4e	8	2127	649	4	124	36
philips_4c_4d	7	4611	804	3	217	5
philips_4c_4e	7	6403	1074	3	396	5
philips_4d_4e	7	3738	891	4	811	11

The picture changes when varying the density of the tangles: for big enough tangle density the SDP approach usually outperforms the IP approaches. Memory requirements, however, usually prohibit solving instances with more than 500 leaf nodes and tangle density of 1%. On the IP side, reformulation is often preferable. Indeed, the best performance is observed when the problems are quadratically reformulated. For $n = 512$ and 1% tangle density, the average solution time is 2120.42 seconds. These instances cannot be solved within the given time limits when using only the standard linearization, with or without separation of cycle inequalities.

The instances for ternary and quad trees are computationally slightly more difficult. This is mainly due to the fact that the number of betweenness restrictions decreases in comparison to binary trees. Here again, the SDP approach performs well for denser instances however memory requirements strongly increase with system size. For larger instances, best performance is often found for the quadratic reformulation.

Comparing *IP* with *IP+cyc* and *QP* with *QP+cyc* for not necessarily binary trees, it turns out that the performance of separating cycle inequalities improves for ternary and quad trees. The special case of a star, where the degree is maximal, is equivalent to the quadratic linear ordering problem, for which we know that separation of cycle inequalities improves over *IP* [2].

The real-world general instances have 131 one-to-one tangles and between 371 and 414 nodes. The average degree of an internal node ranged between 2.71 and 3.38. We show the results in Table 3. Here, solving the reformulation usually yields best performance. Note that these non-binary instances could not be solved before by any other exact method.

Acknowledgments

We are grateful to Franz Rendl, Giovanni Rinaldi, and Angelika Wiegele for the permission to use their bundle program for the SDP-based approach, as well as to Martin Nöllenburg and Danny Holten for sending us real-world tanglegram instances. We thank Henning Fernau for drawing our attention to tanglegrams.

References

[1] Böcker, S., Hüffner, F., Truss, A., Wahlström, M.: A faster fixed-parameter approach to drawing binary tanglegrams. In: Proc. of International Workshop on Parameterized and Exact Computation, IWPEC 2009 (2009) (to appear)
[2] Buchheim, C., Wiegele, A., Zheng, L.: Exact algorithms for the quadratic linear ordering problem. INFORMS J. on Computing (to appear)
[3] De Simone, C.: The cut polytope and the boolean quadric polytope. Discrete Mathematics 79, 71–75 (1989)
[4] Eades, P., Wormald, N.C.: Edge crossings in drawing bipartite graphs. Algorithmica 11, 379–403 (1994)
[5] Fernau, H., Kaufmann, M., Poths, M.: Comparing trees via crossing minimization. J. of Computer and System Sciences (2009) (in press)
[6] Hafner, M.S., Sudman, P.D., Villablanca, F.X., Spradling, T.A., Demastes, J.W., Nadler, S.A.: Disparate rates of molecular evolution in cospeciating hosts and parasites. Science 265, 1087–1090 (1994)
[7] Holten, D.: Personal communication (2009)
[8] ILOG, Inc. ILOG CPLEX 11.2 (2007), http://www.ilog.com/products/cplex
[9] Jünger, M., Mutzel, P.: 2-layer straightline crossing minimization: performance of exact and heuristic algorithms. J. Graph Algorithms Appl. 1, 1–25 (1997)
[10] Nöllenburg, M., Völker, M., Wolff, A., Holten, D.: Drawing binary tanglegrams: An experimental evaluation. In: Proc. of the Workshop on Algorithm Engineering and Experiments, ALENEX 2009, pp. 106–119. SIAM, Philadelphia (2009)

[11] Page, R.D.M.: Tangled Trees: Phylogeny, Cospeciation, and Coevolution. University of Chicago Press, Chicago (2002)
[12] Rendl, F., Rinaldi, G., Wiegele, A.: A branch and bound algorithm for max-cut based on combining semidefinite and polyhedral relaxations. In: Fischetti, M., Williamson, D.P. (eds.) IPCO 2007. LNCS, vol. 4513, pp. 295–309. Springer, Heidelberg (2007)
[13] Venkatachalam, B., Apple, J., St. John, K., Gusfield, D.: Untangling tanglegrams: Comparing trees by their drawings. In: Măndoiu, I., Narasimhan, G., Zhang, Y. (eds.) ISBRA 2009. LNCS, vol. 5542, pp. 88–99. Springer, Heidelberg (2009)

Bit-Parallel Search Algorithms for Long Patterns*

Branislav Ďurian[1], Hannu Peltola[2], Leena Salmela[3], and Jorma Tarhio[2]

[1] S&T Slovakia s.r.o., Priemyselná 2, SK-010 01 Žilina, Slovakia
branislav.durian@snt.sk
[2] Department of Computer Science and Engineering, Aalto University
P.O.B. 15400, FI-00076 Aalto, Finland
{hpeltola,tarhio}@cs.hut.fi
[3] Department of Computer Science, University of Helsinki, P.O.B. 68,
FI-00014 University of Helsinki, Finland
leena.salmela@cs.helsinki.fi

Abstract. We present three bit-parallel algorithms for exact searching of long patterns. Two algorithms are modifications of the BNDM algorithm and the third one is a filtration method which utilizes locations of q-grams in the pattern. Two algorithms apply a condensed representation of q-grams. Practical experiments show that the new algorithms are competitive with earlier algorithms with or without bit-parallelism. The average time complexity of the algorithms is analyzed. Two of the algorithms are shown to be optimal on average.

Keywords: Bit-parallel, pattern, q-gram, string matching.

1 Introduction

String matching [1,12] is a classical problem of computer science. The basic task is to find all the occurrences of a pattern string in a text string, where both of the strings are drawn from the same alphabet. There are several variations of the problem. In this paper we concentrate on exact matching of long patterns, which has recently gained attention [3,5,8,9,10,17].

BNDM (Backward Nondeterministic DAWG Matching) [11] is among the best string matching algorithms. It implements a bit-parallel simulation of a nondeterministic automaton. BNDM is known to be efficient for patterns of at most w characters, where w is the register size of the computer, typically 32 or 64. It is straightforward to extend BNDM to handle longer patterns by simulating a virtual long register with registers of size w, but the resulting algorithms are not very efficient. Long BNDM [11], LBNDM [13], BLIM [8], and SABP [17] are faster bit-parallel solutions than the trivial one. However these algorithms are clearly slower than the best solutions (e.g. Lecroq's algorithm [10]) which do not apply bit-parallelism.

In this paper, we present three new bit-parallel algorithms BXS, BQL, and QF, which are in most cases faster than the previous bit-parallel algorithms

* Supported by Academy of Finland grants 118653 (ALGODAN) and 134287 (IASMP).

for patterns longer than the register size. Our algorithms are also competitive with earlier algorithms without bit-parallelism. Our algorithms apply *q-grams*, i.e. q consecutive characters together. Two of the algorithms are partly based on recent q-gram variations [2] of BNDM for short patterns. The third one is based on checking an alignment q-gram by q-gram introduced by Fredriksson and Navarro [4]. Two of the algorithms, BQL and QF, use a condensed representation of q-grams [10,15] that enables reasonable space requirements. We also analyze the time complexity of the algorithms. BQL and QF are shown to be optimal on average.

We use the following notations. Let $T = t_1t_2 \ldots t_n$ and $P = p_1p_2 \ldots p_m$ be two strings over a finite alphabet Σ of size σ. The task of exact string matching is to find all occurrences of the pattern P in the text T. Formally we search for all positions i such that $t_it_{i+1} \ldots t_{i+m-1} = p_1p_2 \ldots p_m$. In the algorithms we use C notations: '|', '&', and '<<' represent bitwise operations OR, AND, and left shift, respectively.

2 Previous Algorithms

Because two of our algorithms are partly based on BNDM, we introduce the code of BNDM. After that we will shortly explain the principles of five earlier algorithms for long patterns.

2.1 BNDM

The key idea of BNDM [11] is to simulate a nondeterministic automaton recognizing all the prefixes of the pattern. The automaton is simulated with bit-parallelism even without constructing it.

In BNDM (see Alg. 1) the precomputed table B associates each character with a bit mask expressing its occurrences in the pattern. At each alignment of the pattern, the algorithm reads the text from right to left until the whole pattern is recognized or the processed text string is not any substring of the pattern. Between alignments, the algorithm shifts the pattern forward to the start position of the longest found prefix of the pattern, or if no prefix is found, over the current alignment. With the bit-parallel Shift-AND technique the algorithm maintains a state vector D, which has one in each position where a substring of the pattern starts such that the substring is a suffix of the processed text window. The standard BNDM works only for patterns which are not longer than w.

The inner while loop of BNDM checks one alignment of the pattern from right to left. In the same time the loop recognizes prefixes of the pattern. The leftmost one of the found prefixes determines the next alignment of the algorithm.

2.2 Algorithms for Long Patterns

We consider five earlier algorithms. The first one is a modification of BNDM by Navarro and Raffinot [11]. We call it Long BNDM. In this algorithm, a prefix of

Algorithm 1. BNDM($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
    /* Preprocessing */
 1: for all $c \in \Sigma$ do $B[c] \leftarrow 0$
 2: for $j \leftarrow 1$ to $m$ do
 3:     $B[p_j] \leftarrow B[p_j] \mid (1 << (m - j))$
    /* Searching */
 4: $i \leftarrow 0$
 5: while $i \leq n - m$ do
 6:     $j \leftarrow m$; $last \leftarrow m$; $D \leftarrow (1 << m) - 1$
 7:     while $D \neq 0$ do
 8:         $D \leftarrow D \,\&\, B[t_{i+j}]$; $j \leftarrow j - 1$
 9:         if $D \,\&\, (1 << (m - 1)) \neq 0$ then
10:             if $j > 0$ then
11:                 $last \leftarrow j$
12:             else
13:                 report occurrence at $i + 1$
14:         $D \leftarrow D << 1$
15:     $i \leftarrow i + last$
```

w characters is searched with the standard BNDM and in the case of a match of that prefix, the rest of the alignment is verified in the BNDM manner in pieces of w characters. The maximum shift is w.

In LBNDM by Peltola and Tarhio [13], the pattern of length m is partitioned into $\lfloor \frac{m}{k} \rfloor$ consecutive pieces, each consisting of $k = \lfloor \frac{m-1}{w} \rfloor + 1$ characters. This division implies k subsequences of the pattern such that the ith sequence takes the ith character of each piece. The idea is to search first the superimposed pattern of these sequences so that only every kth character is examined. This filtration phase is done with the standard BNDM. Each occurrence of the superimposed pattern is a potential match of the original pattern and thus must be verified. The shift of LBNDM is a multiple of k and at most m. LBNDM works efficiently only for large alphabets.

Külekci [8] introduced BLIM which checks w alignments simultaneously. Starting with a vector of ones of length w, the vector is updated with the AND operation with the mask of a text character in turn until the vector becomes zero. The shifting is based on the character immediately following the window. The maximum shift of BLIM is $w + m$. SABP by Zhang et al. [17] is related to BLIM. In SABP, bitvectors are preprocessed in a so called matching matrix.

The Wide Window algorithm (WW) [5] applies two automata in a window of size $2m - 1$. WW is not a bit-parallel algorithm like the others in this section. The search begins from the middle of the window. The window is moved m positions forward until a character occurring in the pattern is found and a forward suffix automaton can start. Then the rest of the match is verified with a reverse prefix automaton. Finally the start position is moved past the current window.

3 New Algorithms

In this section we will present our new algorithms BXS, BQL, and QF. All the algorithms use q-grams, and we present the pseudocodes for $q = 3$. The value $q = 3$ has been selected only for the clarity of presentation. In the rest of the paper the variable w' holds the minimum of m and w.

3.1 BXS

Our first algorithm is BXS (BNDMq with eXtended Shift). We first cut the pattern into $\lceil m/w' \rceil$ consecutive pieces of length w' except for the rightmost piece which may be shorter. Then we superimpose these pieces getting a superimposed pattern of length w'. In each position of the *superimposed pattern* a character from any piece (in corresponding position) is accepted. We then use the following modified version of BNDM to search for consecutive occurrences of the superimposed pattern using bit vectors of length w' but still shifting the pattern by up

Algorithm 2. BXS$_3(P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n)$

```
Require: $m \geq q$
    /* Preprocessing */
 1: for all $c \in \Sigma$ do $B[c] \leftarrow 0$                          /* $0^w$ */
 2: $w' \leftarrow \min(m, w)$; $x \leftarrow m - (m \bmod w') + w'$
 3: for $j \leftarrow 1$ to $m$ do
 4:     $B[p_j] \leftarrow B[p_j] \mid (1 << ((x - j) \bmod w'))$        /* $0^*10^{(x-j) \bmod w'}$ */
 5: for $c \in \Sigma$ do
 6:     $B1[c] \leftarrow (B[c] << 1) \mid 1$
 7:     $B2[c] \leftarrow (B[c] << 2) \mid 3$                          /* $(B << 2) \mid 0^{w-2}1^2$ */
    /* Searching */
 8: $i \leftarrow mq1 \leftarrow m - q + 1$                                /* now $q = 3$ */
 9: while $i \leq n - q + 1$ do
10:     $D \leftarrow B2[t_{i+2}]$ & $B1[t_{i+1}]$ & $B[t_i]$
11:     if $D \neq 0$ then
12:         $j \leftarrow i$; $first \leftarrow i - mq1$
13:         repeat
14:             $j \leftarrow j - 1$
15:             if $D \geq (1 << (w' - 1))$ then /* is highest bit set */
16:                 if $j > first$ then
17:                     $i \leftarrow j$      /* possible prefix found; sliding backward */
18:                 else /* verify whole match */
19:                     if $t_{first+1}t_{first+2} \cdots t_{first+m} = p_1p_2 \cdots p_m$ then
20:                         report an occurrence at $first + 1$
21:                 $D \leftarrow (D << 1 \mid 1)$ & $B[t_j]$      /* rotating set highest bit */
22:             else
23:                 $D \leftarrow (D << 1)$ & $B[t_j]$
24:         until $D = 0$ or $j \leq first$
25:     $i \leftarrow i + mq1$
```

to m positions. We first initialize the B vectors as if we were searching with the standard BNDM for the superimposed pattern. When searching we rotate the bits in D rather than just shifting them to the left as in the standard BNDM. In the standard BNDM the D vector is guaranteed to die (i.e. all bits are 0) after at most m characters are read because the shift operation inserts zeroes to the right. Now we no longer have this guarantee because of rotating bits in D. Therefore we also need to check that we will not read more than m characters in a window and exit the inner loop of BNDM if this is the case. We further note that the w':th bit of D is set whenever the processed suffix of the current alignment matches a prefix of the original pattern. However, it is also set if the suffix of the alignment matches a prefix of a power of the superimposed pattern even if it does not match a prefix of the original pattern. Thus the shifts of the alignment can be unnecessarily short, and if the w':th bit in D is set after reading m characters, we need to verify for an occurrence of the original pattern.

In practise BXS is faster if we utilize q-grams as in BNDMq [2]. In each alignment we first read the last q characters and update D accordingly. To do this efficiently we store shifted values of B into tables Bi. This reduces the maximum shift length to $m-q+1$. Algorithm 2 shows the pseudo code for BXS with this modification. The computation of D on line 10 is different for each q as well as the computation of Bi tables on lines 5–7. Each B or Bi table needs $\sigma \cdot w$ bits.

BXS does not work well when the superimposed pattern is not sensitive enough, i.e. too many different characters are accepted at the same position. This happens when the alphabet is too small or the pattern is too long. Increasing the value of q can help, and another solution is to use only a substring of the pattern when constructing the superimposed pattern. Of course this limits the maximum shift length. It is also possible to relieve this problem by considering a condensed representation of q-grams introduced in the next section.

3.2 BQL

BQL (BNDMq Long) is our second algorithm. BQL increases the effective alphabet size by using overlapping q-grams, e.g. when using 3-grams the pattern “ACCTGGT” is processed as “ACC-CCT-CTG-TGG-GGT”. Thus we effectively search for a pattern of $m-q+1$ overlapping q-grams. Similar to BXS we cut the q-gram pattern into $\lceil (m-q+1)/w' \rceil$ pieces and superimpose them. The B vectors of BNDM are then initialized for the superimposed q-gram pattern.

In the search phase we use a modification of Simplified BNDM [13], which allows us to always shift by $m-q+1$ but still only use w' bits for the bit vectors B and D. We divide the text into nonoverlapping windows of length $m-q+1$ and in each window we do a BNDM like scan from right to left. Whenever the highest bit in D is set, we verify all such alignments of the pattern with the text that the prefix of one of the superimposed pieces is aligned with the processed suffix of the window. When the D vector dies, we shift always by $m-q+1$ to move to the next text window. Algorithm 3 shows the pseudo code of the algorithm. The computation of ch on line 12 is different for each q.

Algorithm 3. $\mathbf{BQL}_{3,\boldsymbol{s}}(P = p_1 p_2 \cdots p_m, T = t_1 t_2 \cdots t_n)$

```
Require: $m > q$ and theoretically $q \cdot s < w$
    /* Preprocessing */
 1: for $ch \leftarrow 0$ to $(2^{q \cdot s} - 1)$ do
 2:     $B[ch] \leftarrow 0$                                  /* $0^w$ */
 3: $w' \leftarrow \min(m, w)$; $mq1 \leftarrow m - q + 1$; $x \leftarrow m - (m \bmod w') + w'$
 4: $mask \leftarrow (1 << (q \cdot s)) - 1$                    /* $0^{w'-q \cdot s} 1^{q \cdot s}$ */
 5: $ch \leftarrow 0$
 6: for $i \leftarrow m$ downto 1 do
 7:     $ch \leftarrow ((ch << s) + p_i)\ \&\ mask$
 8:     if $i \leq mq1$ then
 9:         $B[ch] \leftarrow B[ch] \mid (1 << ((x - i) \bmod w'))$
    /* Searching */
10: $i \leftarrow mq1$
11: while $i \leq n - q + 1$ do
12:     $ch \leftarrow (((((t_{i+2}) << s) + t_{i+1}) << s) + t_i)\ \&\ mask$
13:     if $(D \leftarrow B[ch]) \neq 0$ then
14:         $j \leftarrow i$
15:         repeat
16:             $j \leftarrow j - 1$
17:             if $D \geq (1 << (w' - 1))$ then /* is highest bit set */
18:                 for $k \leftarrow j$ step down $w'$ while $k \geq i - mq1$ do
19:                     if $t_{k+1} t_{k+2} \cdots t_{k+m} = p_1 p_2 \cdots p_m$ then /* verify match */
20:                         if $k + m \leq n$ then
21:                             report an occurrence at $k + 1$
22:             $ch \leftarrow ((ch << s) + t_j)\ \&\ mask$
23:             $D \leftarrow (D << 1)\ \&\ B[ch]$
24:         until $D = 0$
25:     $i \leftarrow i + mq1$
```

We use the following condensed representation of q-grams to reduce the space usage of the B vectors. The parameter s regulates the number of bits reserved for each character in a q-gram, and q-grams are encoded as the $s \cdot q$ lowest bits of shifted sum of bit representations of ASCII characters, see line 12 of Algorithm 3. The vector table B thus needs $2^{s \cdot q} \cdot w$ bits. Roughly the value $s = 1$ is suitable for the binary alphabet, $s = 2$ is good for DNA and natural language, and $s \geq 5$ is good for random data of alphabet of 256 characters. A similar representation has earlier been used by Lecroq [10] and in the code of agrep [15].

In the experiments of Section 5, we used the following modification of line 19 before entering the inner verification loop: We made a guard check of the last 2-gram of the pattern. This modification makes the algorithm faster especially on small alphabets or with long patterns.

3.3 QF

Our third algorithm, QF (Q-gram Filtering), is similar to the approximate string matching algorithm by Fredriksson and Navarro [4], which is not a BNDM based

Algorithm 4. $\mathbf{QF}_{3,\boldsymbol{s}}(P = p_1p_2\cdots p_m, T = t_1t_2\cdots t_n)$

```
Require: $m > q$ and theoretically $q \cdot s < w$
    /* Preprocessing */
 1: for $ch \leftarrow 0$ to $(2^{q\cdot s} - 1)$ do /* note that $2^{q\cdot s} = (1 << (q \cdot s))$ */
 2:     $B[ch] \leftarrow 0$                                    /* only $0^q$ needed */
 3: $mq1 \leftarrow m - q + 1$; $ch \leftarrow 0$; $mask \leftarrow (1 << (q \cdot s)) - 1$      /* $0^{w-q\cdot s}1^{q\cdot s}$ */
 4: for $i \leftarrow m$ downto 1 do
 5:     $ch \leftarrow ((ch << s) + p_i)$ & $mask$
 6:     if $i \le mq1$ then
 7:         $B[ch] \leftarrow B[ch] \mid (1 << ((m - i) \bmod q))$          /* here $q = 3$ */
    /* Searching */
 8: $i \leftarrow mq1$
 9: while $i \le n - q + 1$ do
10:     $D \leftarrow B[((((t_{i+2} << s) + t_{i+1}) << s) + t_i)$ & $mask]$
11:     if $D \ne 0$ then
12:         $j \leftarrow i - mq1 + q$     /* end of the leftmost $q$-gram of an alignment */
13:         repeat
14:             $i \leftarrow i - q$
15:         until $i \le (j - q)$   or
                 $(D \leftarrow (D$ & $B[((((t_{i+2} << s) + t_{i+1}) << s) + t_i)$ & $mask])) = 0$
16:         if $i < j$ then
17:             $i \leftarrow j$
18:             for $k \leftarrow j - q + 1$ to $j$ do
19:                 if $t_kt_{k+1}t_{k+2}\cdots t_{k+m-1} = p_1p_2p_3\cdots p_m$ then
20:                     report an occurrence at $k$
21:     $i \leftarrow i + mq1$
```

algorithm. As preprocessing we store for each *phase* i, $0 \le i < q$, which q-grams occur in the pattern in that phase, i.e. start at position $i + j \cdot q$ for any j. To store this information we initialize a vector B for each q-gram where the i:th bit is set if the q-gram occurs in phase i in the pattern.

During searching we read consecutive q-grams in a window and keep track of *active* phases, i.e. such phases that all read q-grams occur in that phase in the pattern. This can be done conveniently with bit parallelism. We maintain a vector D where the i:th bit is set if the i:th phase is active. Initially all phases are active and after reading a q-gram G the vector D can be updated using the preprocessed B vectors: $D = D$ & $B[G]$. If we have read all the q-grams of the window and some phase is still active, we must verify for an occurrence of the pattern. When the vector D becomes inactive or after verification, we can shift the alignment past the last read q-gram.

To reduce space usage QF applies the same condensed representation of q-grams as BQL. The pseudocode of QF is shown as Algorithm 4. The **or** operator on line 15 is short-circuit OR. The computation of index expression of B on lines 10 and 15 is different for each q. The vector table B needs $q \cdot 2^{q\cdot s}$ bits. This can be a considerably enhancement compared to BQL, especially if B becomes small enough compared to the data cache.

D actually contains the information describing which *phases* are potential, so we would not need to check them all. Use of that information did not improve performance in practice. If tests on line 15 could be made separately with **goto**s, the test in the **if** statement on the next line would become unnecessary.

4 Analysis

The worst case complexity of our algorithms is $\mathcal{O}(mn)$, and the best case complexity is $\mathcal{O}(nq/(m-q))$.

When analyzing the average case complexity of the algorithms, we assume that the characters are statistically independent of each other and the distribution of characters is discrete uniform. Furthermore, we assume that the alphabet is such that the condensed representation of q-grams in BQL and QF produces a uniform distribution. For simplicity we assume in the analysis that w divides m.

When analyzing the average case complexity of BXS, we assume that $q = 1$. The parameter q in BXS is used to gain a practical speedup but it does not affect the asymptotic complexity of the algorithm. On the other hand, we will see that in the BQL and QF algorithms the value of q has a crucial impact on the average case complexity.

BXS. Let $P = p_1 p_2 \dots p_m$ be the pattern. Let us then construct a pattern

$$P' = ([p_1, p_{1+w}, p_{1+2w}, \dots][p_2, p_{2+w}, p_{2+2w}, \dots] \dots)^{\frac{m}{w}},$$

where the square brackets denote a class of characters and exponentiation the repetition of an element. If we now run the standard BNDM algorithm with the pattern P' on a machine where the length of the computer word is long enough, it will read exactly the same characters and perform exactly the same shifts as BXS with the original pattern P run on a machine with word length w.

The average case complexity of BNDM with a pattern containing classes of characters is $\mathcal{O}(n \log_{\bar{\sigma}} m/m)$ where $\bar{\sigma}$ is the inverse of the probability of a class of characters matching a random character. If there are at most m/w characters in a class then this probability is bounded by $m/(w\sigma)$. Note that this bound should be smaller than 1 and thus $m < w\sigma$ must hold. Now the average case complexity of the algorithm becomes

$$\mathcal{O}(n \log_{w\sigma/m} m/m) = \mathcal{O}\left(\frac{n}{m} \cdot \frac{\log_\sigma m}{1 - \log_\sigma \frac{m}{w}}\right).$$

The above result holds for a random pattern and a random text. However, our pattern P' has a repetitive structure with period m/w and is thus not completely random. Still if the text is random, the algorithm actually cannot perform worse with a repetitive pattern than with a random pattern because the probability of a random text substring matching the pattern in any position is in fact lower for the repetitive pattern as it contains fewer unique substrings. Thus the average case complexity of BXS is

$$\mathcal{O}\left(\frac{n}{m} \cdot \frac{\log_\sigma m}{1 - \log_\sigma \frac{m}{w}}\right).$$

An optimal string matching algorithm has the average case time complexity $\mathcal{O}(n \log_\sigma m/m)$ [16] so BXS is worse than optimal by a factor of $1/(1-\log_\sigma(m/w))$.

BQL. BQL processes the text in windows. There are $n/(m-q+1)$ windows. In each window the algorithm first reads the last q-gram of the window. Let us call a window good if the last q-gram of the window does not match the pattern in any position and let all other windows be called bad. In a good window the algorithm reads q characters and then moves on to the next window. Thus the work done by the algorithm in a good window is $\mathcal{O}(q)$. In a bad window the highest bit of the vector D can be set at most w times triggering m/w verifications each time. Each verification can be performed in $\mathcal{O}(m)$ time. Thus the work done by the algorithm in a bad window can be bounded by $\mathcal{O}(w \cdot m/w \cdot m) = \mathcal{O}(m^2)$. The probability that a window is bad is at most $m/2^{sq}$ and therefore the average complexity of the algorithm can be bounded by

$$\mathcal{O}\left(\frac{n}{m-q+1}\left(q + \frac{m}{2^{sq}} \cdot m^2\right)\right) = \mathcal{O}\left(\frac{nq}{m-q+1} + \frac{n}{m-q+1} \cdot \frac{m^3}{2^{sq}}\right).$$

Let us then choose $q = 3\log_{2^s} m$. Then

$$\mathcal{O}\left(\frac{nq}{m-q+1} + \frac{n}{m-q+1} \cdot \frac{m^3}{2^{sq}}\right) = \mathcal{O}\left(\frac{n\log_{2^s} m}{m}\right).$$

If we further choose s so that $2^s = \Theta(\sigma)$, then $\mathcal{O}\left(n\log_{2^s} m/m\right) = \mathcal{O}\left(n\log_\sigma m/m\right)$ and therefore BQL is optimal on average for an appropriate choice of q and s.

QF. The algorithm by Fredriksson and Navarro [4] (FN for short) is designed for multiple approximate string matching. FN is similar to our QF when we set the number of differences $k = 0$ and the number of patterns $r = 1$. There are two differences between the algorithms. QF counts the occurrences of the q different phases of the pattern separately, while FN disregards the phases and only counts how many differences are at least needed to align the read q-grams with the pattern somehow. Secondly, QF uses a condensed representation of the q-grams, while FN uses plain q-grams.

The condensed representation of the q-grams reduces the alphabet size to 2^s. If we assume that the alphabet size is 2^s, then QF never reads more characters in a window than FN. QF stops handling a window when the read q-grams do not match the pattern q-grams in the same phase. FN cannot stop sooner than QF because the read q-grams can be aligned with the pattern with 0 differences if QF has not stopped reading. Both of the algorithms shift the pattern so that the new window is shifted just past the last read q-gram. Because QF never reads less q-grams in a window than FN, it always makes a shift that is at least as long as in FN. Therefore, the average case complexity of QF cannot be worse than the average case complexity of FN, $\mathcal{O}(n\log_\sigma m/m)$ for $k = 0$ and $r = 1$ when $q = \Theta(\log_\sigma m)$. As the alphabet size in QF is 2^s, the average complexity of QF is

$\mathcal{O}(n \log_{2^s} m/m)$ when $q = \Theta(\log_{2^s} m)$. Thus the average complexity of QF is $\mathcal{O}(n \log_\sigma m/m)$ if $2^s = \Theta(\sigma)$ and $q = \Theta(\log_{2^s} m) = \Theta(\log_\sigma m)$. This complexity is optimal for exact matching of a single pattern, and thus the analysis gives a tight bound.

5 Experimental Comparison

The tests were run on a 2.8 GHz Pentium D (dual core) CPU with 1 GB of memory. Both cores have 16 KB L1 data cache and 1024 KB L2 cache. The computer was running Fedora 8 Linux. All the algorithms were tested in the testing framework of Hume and Sunday [7]. All programs were written in C and compiled with the gcc compiler 4.1.2 producing x86_64 "32-bit" code and using the optimization level `-O3`.

We used four texts of 2 MB in our tests: binary, DNA, English, and uniformly random of 254 characters[1]. The English text is the beginning of the KJV bible. The DNA text is from the genome of fruitfly (*Drosophila Melanogaster*). The binary and random texts were generated randomly. For each text we have pattern sets of lengths 25, 50, 100, 200, 400, 800, and 1600. The 200 patterns in each pattern set are picked randomly from the same data source as the text. Roughly more than half of the patterns appear in the text. The patterns in each pattern set are from non-overlapping positions.

We compared our algorithms with the following algorithms: Long BNDM [11] (the times for $m = 25$ were run with the standard BNDM), LBNDM [13] for English and random, BLIM [8], WW [5], A4 [14] for DNA, Lecroq [10], and EBOM [3]. We made also preliminary tests on SABP [17]. Its performance seems to be similar to that of BLIM. Lecroq's algorithm and A4 are q-gram variants of the Boyer–Moore–Horspool algorithm [6]. EBOM is an efficient implementation of the oracle automaton utilizing 2-grams. Because Lecroq's algorithm (as described in [10]) has at most 256 values of shift, it is not competitive for long patterns. Therefore we implemented another version called Lecroq2 which has 4096 values of shift. Obviously tests with long patterns were done with such a version in [10].

Table 1 shows average times of 200 runs in milliseconds. (To get more accuracy the runs with search times less than 10 ms were repeated 3000 times.) The best times have been boxed. The best values of parameters for each algorithm are given for each pattern set. Generally QF was the fastest and BQL was the second best—especially on longer patterns. In most cases the best value of q for BQL was bigger or equal than for QF. BLIM would work faster with $w = 64$ (i.e. using "64-bit" code) except on binaries and DNA for $m = 25$. Lecroq2 is a considerable improvement compared to the basic version on other data sets than binary when $m \geq 400$. The relatively good performance of Long BNDM on Random$_{254}$ seems to be due to a skip loop. Also WW has a related structure.

[1] Our testing environment allows an alphabet of at most 254 characters. So this is not a limitation of the algorithms.

Table 1. Search times for 200 patterns in milliseconds

	Algorithm	par.	25	par.	50	par.	100	par.	200	par.	400	par.	800	par.	1600
Binary data	Long BNDM		786		526		526		529		529		531		532
	BLIM		875		534		535		536		536		539		542
	WW		1384		801		468		354		283		210		172
	Lecroq	6	304	7	188	8	125	6	137	6	138	8	110	8	102
	Lecroq2	6	313	6	201	8	143	6	139	6	140	8	116	8	111
	EBOM		784		502		314		241		184		91		79
	BXS	5	708	12	507	14	1022								
	BQL	9,1	322	9,1	173	11,1	117	9,1	148	12,1	69	15,1	16	15,1	8.8
	QF	8,1	$\boxed{282}$	9,1	$\boxed{145}$	9,1	$\boxed{107}$	9,1	$\boxed{133}$	12,1	$\boxed{68}$	13,1	$\boxed{14}$	15,1	$\boxed{8.2}$
DNA data	Long BNDM		455		298		298		297		298		300		300
	BLIM		518		356		357		359		357		359		363
	WW		679		389		226		200		126		79		70
	Lecroq	3	215	4	147	4	111	3	124	3	131	7	106	6	90
	Lecroq2	4	223	4	149	5	110	4	164	8	89	8	18	8	9.8
	A4	4	228	4	156	4	113	6	183	6	83	6	18	6	10
	EBOM		411		262		160		132		80		44		47
	BXS	4	348	7	225	8	211	14	273						
	BQL	5,3	219	5,3	135	7,2	108	7,2	126	7,2	44	8,2	13	8,2	8.0
	QF	4,3	$\boxed{165}$	5,3	$\boxed{110}$	8,2	$\boxed{101}$	5,3	$\boxed{105}$	7,2	$\boxed{40}$	8,2	$\boxed{9.6}$	8,2	$\boxed{5.8}$
English data	Long BNDM		309		247		248		249		249		251		253
	LBNDM		389		238		168		144		123		113		138
	BLIM		371		224		207		191		183		176		165
	WW		406		242		162		152		91		47		44
	Lecroq	3	189	3	135	3	106	3	174	4	135	4	85	3	54
	Lecroq2	3	202	3	142	4	107	8	175	8	91	6	17	8	9.5
	EBOM		213		163		126		101		60		33		28
	BXS	3	211	4	129	6	119	3	145	5	70	9	34	12	31
	BQL	4,3	207	4,3	133	7,2	109	4,3	112	7,2	49	7,2	14	14,1	8.5
	QF	3,4	$\boxed{143}$	3,4	$\boxed{104}$	8,2	$\boxed{102}$	3,5	$\boxed{92}$	4,3	$\boxed{34}$	8,2	$\boxed{10}$	8,2	$\boxed{5.8}$
Random$_{254}$ data	Long BNDM		108		102		101		102		102		103		105
	LBNDM		124		106		102		84		37		11		8.0
	BLIM		164		114		104		135		138		120		120
	WW		117		103		103		100		54		16		9.5
	Lecroq	3	179	3	129	3	103	3	175	3	135	4	102	3	94
	Lecroq2	3	192	3	136	3	104	3	174	5	75	4	12	5	8.5
	EBOM		106		99		97		$\boxed{63}$		22		10		8.9
	BXS	2	99	2	$\boxed{93}$	1	$\boxed{94}$	2	74	2	17	4	7.8	4	6.4
	BQL	2,5	145	2,6	100	2,7	99	2,6	94	2,6	25	2,7	7.8	2,7	4.8
	QF	2,6	$\boxed{98}$	3,4	95	2,8	96	2,8	70	2,6	$\boxed{16}$	2,8	$\boxed{5.0}$	2,8	$\boxed{3.1}$

The times in Table 1 do not include preprocessing based on the patterns. The preprocessing times were unessential for all other algorithms except BLIM, WW, and EBOM. E.g. for English, their preprocessing times grew (according to pattern length) as follows: BLIM from 94 to 6512, WW from 3 to 675, and EBOM from 15 to 214 milliseconds per pattern set.

6 Concluding Remarks

We have presented three bit-parallel q-gram algorithms for searching long patterns. The new algorithms are efficient—both in theory and practice. Our experiments show that the new algorithms are in most cases faster than previous bit-parallel algorithms for long patterns. Our algorithms are also competitive with earlier algorithms without bit-parallelism. QF is the best of the algorithms. We showed that QF and BQL are optimal on average.

References

1. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific Publishing Company, Singapore (2002)
2. Ďurian, B., Holub, J., Peltola, H., Tarhio, J.: Tuning BNDM with q-grams. In: Proc. ALENEX 2009, the Tenth Workshop on Algorithm Engineering and Experiments, pp. 29–37 (2009)
3. Faro, S., Lecroq, T.: Efficient variants of the backward-oracle-matching algorithm. In: Proc. PSC 2008, The 13th Prague Stringology Conference, pp. 146–160 (2008)
4. Fredriksson, K., Navarro, G.: Average-optimal single and multiple approximate string matching. ACM Journal of Experimental Algorithmics 9(1.4), 1–47 (2004)
5. He, L., Fang, B., Sui, J.: The wide window string matching algorithm. Theoretical Computer Science 332(1-3), 391–404 (2005)
6. Horspool, R.N.: Practical fast searching in strings. Software – Practice and Experience 10(6), 501–506 (1980)
7. Hume, A., Sunday, D.M.: Fast string searching. Software – Practice and Experience 21(11), 1221–1248 (1991)
8. Külekci, M.O.: A method to overcome computer word size limitation in bit-parallel pattern matching. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 496–506. Springer, Heidelberg (2008)
9. Külekci, M.O.: Filter based fast matching of long patterns by using SIMD instructions. In: Proc. of the Prague Stringology Conference 2009, pp. 118–128 (2009)
10. Lecroq, T.: Fast exact string matching algorithms. Information Processing Letters 102(6), 229–235 (2007)
11. Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. ACM Journal of Experimental Algorithmics 5(4), 1–36 (2000)
12. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, New York (2002)
13. Peltola, H., Tarhio, J.: Alternative algorithms for bit-parallel string matching. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 80–93. Springer, Heidelberg (2003)
14. Tarhio, J., Peltola, H.: String matching in the DNA alphabet. Software – Practice and Experience 27(7), 851–861 (1997)
15. Wu, S., Manber, U.: Agrep – a fast approximate pattern searching tool. In: Proceedings of the Winter USENIX Technical Conference, pp. 153–162 (1992)
16. Yao, A.C.-C.: The complexity of pattern matching for a random string. SIAM Journal on Computing 8(3), 368–387 (1979)
17. Zhang, G., Zhu, E., Mao, L., Yin, M.: A bit-parallel exact string matching algorithm for small alphabet. In: Deng, X., Hopcroft, J.E., Xue, J. (eds.) FAW 2009. LNCS, vol. 5598, pp. 336–345. Springer, Heidelberg (2009)

Fast FPT Algorithms for Computing Rooted Agreement Forests: Theory and Experiments (Extended Abstract[⋆])

Chris Whidden[⋆⋆], Robert G. Beiko[⋆⋆⋆], and Norbert Zeh[†]

Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia, Canada
{whidden,beiko,nzeh}@cs.dal.ca

Abstract. We improve on earlier FPT algorithms for computing a rooted maximum agreement forest (MAF) or a maximum acyclic agreement forest (MAAF) of a pair of phylogenetic trees. Their sizes give the subtree-prune-and-regraft (SPR) distance and the hybridization number of the trees, respectively. We introduce new branching rules that reduce the running time of the algorithms from $O(3^k n)$ and $O(3^k n \log n)$ to $O(2.42^k n)$ and $O(2.42^k n \log n)$, respectively. In practice, the speed up may be much more than predicted by the worst-case analysis. We confirm this intuition experimentally by computing MAFs for simulated trees and trees inferred from protein sequence data. We show that our algorithm is orders of magnitude faster and can handle much larger trees and SPR distances than the best previous methods, **treeSAT** and **sprdist**.

1 Introduction

Phylogenetic trees are used to represent the evolution of a set of species (taxa) [13]. In addition to 'vertical' inheritance from parent to offspring, genetic material can be exchanged between contemporary organisms via lateral gene transfer, recombination and hybridization. These processes enable the rapid spread of antibiotic resistance and other harmful traits in pathogenic bacteria, and more generally allow species to rapidly adapt to new environments. Untangling vertical and lateral evolutionary histories requires the comparison of phylogenetic trees, and metrics that model reticulation events using subtree prune-and-regraft (SPR) [11] or hybridization [1] permutations are of particular interest, since the resulting series of permutations has a direct evolutionary interpretation [14,15].

Although these distance metrics are biologically meaningful, they are NP-hard to compute [8,10,12]. This has led to significant effort to develop approximation [5,7,16] and fixed-parameter (FPT) algorithms [7,9], as well as heuristic approaches [3,12], for computing these distances. The main tool of most such algorithms is the notion of a maximum agreement forest [1,8,11]. Recently,

[⋆] Details can be found in [17].

[⋆⋆] Supported by an NSERC PGS-D graduate scholarship.

[⋆⋆⋆] Canada Research Chair; supported in part by NSERC and Genome Atlantic.

[†] Canada Research Chair; supported in part by NSERC.

Whidden and Zeh [18] presented a unified view of previous methods for computing agreement forests and introduced improvements that led to the theoretically fastest approximation and FPT algorithms for computing such forests so far. The FPT algorithms for SPR distance and hybridization number use a bounded search tree approach and take $O(3^k n)$ and $O(3^k n \log n)$ time, respectively ($O(3^k k + n^3)$ and $O(3^k k \log k + n^3)$ using standard kernelizations [8,9].)

In this paper, we introduce improved branching rules to be used in the algorithms of [18]. These branching rules reduce the running times of the algorithms for SPR distance and hybridization number to $O(2.42^k n)$ and $O(2.42^k n \log n)$, respectively. Using the same kernelizations as before, the running times can be reduced further to $O(2.42^k k + n^3)$ and $O(2.42^k k \log k + n^3)$, respectively.

While these theoretical improvements are valuable in their own right, our main contribution is to evaluate the practical performance of the algorithm of [18] and the impact of our improved branching rules. An additional optimization we apply is to use the linear-time 3-approximation algorithm for SPR distance of [16,18] to prune branches in the search tree that are guaranteed to be unsuccessful. This reduces the size of the search tree substantially and leads to a corresponding decrease in running time. We demonstrate that each of the improved branching rules and the pruning of unsuccessful branches have a marked and distinct effect on the performance of the algorithm. Our experiments confirm that our algorithm is orders of magnitude faster than the currently best exact alternatives [4,20] based on reductions to integer linear programming and satisfiability testing, respectively. The largest distances reported using implementations of previous methods are a hybridization number of 14 on 40 taxa [6] and an SPR distance of 19 on 46 taxa [20]. In contrast, our method took less than 5 hours to compute SPR distances of up to 46 on trees with 144 taxa and 99 on synthetic 1000-leaf trees. This represents a major step forward towards tools that can infer reticulation scenarios for the hundreds of genomes that have been fully sequenced to date.

The rest of this paper is organized as follows. Section 2 introduces the necessary terminology and notation. Section 3 presents our FPT algorithms using the improved branching rules. Section 4 presents our experimental results.

2 Preliminaries

Throughout this paper, we mostly use the definitions and notation from [7,8,9, 16,18]. A *(rooted binary phylogenetic) X-tree* is a rooted tree T whose leaves are the elements of a label set X and whose non-root internal nodes have two children each; see Figure 1(a). The root of T has label ρ and has one child. Throughout this paper, we consider ρ to be a member of X. For a subset V of X, $T(V)$ is the smallest subtree of T that connects all nodes in V; see Figure 1(b). The *V-tree induced by T* is the tree $T|V$ obtained from $T(V)$ using *forced contractions*, each of which removes an unlabelled node v with only one child and its incident edges. If v was the root of the current tree, its child becomes the new root; otherwise an edge is added between v's parent and child. See Figure 1(c).

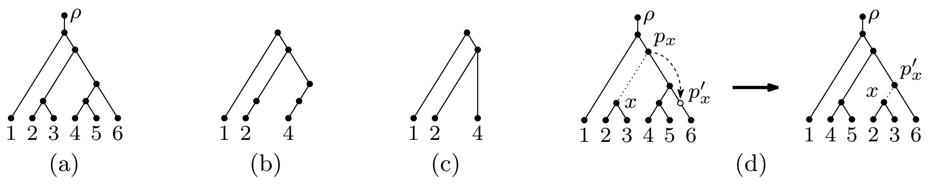


Fig. 1. (a) An X-tree T. (b) The subtree $T(V)$ for $V = \{1, 2, 4\}$. (c) $T|V$. (d) Illustration of an SPR operation.

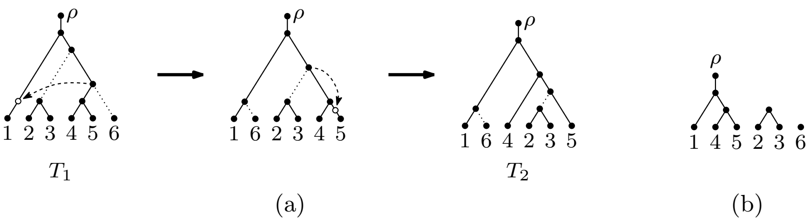


Fig. 2. (a) SPR operations transforming T_1 into T_2. Each operation changes the top endpoint of one of the dotted edges. (b) The corresponding agreement forest, which can be obtained by cutting the dotted edges in both trees.

A *subtree-prune-and-regraft* (SPR) operation on an X-tree T cuts an edge $e_x := xp_x$, where p_x denotes the parent of x. This divides T into subtrees T_x and T_{p_x} containing x and p_x, respectively. Then it introduces a node p'_x into T_{p_x} by subdividing an edge of T_{p_x} and adds an edge xp'_x, making x a child of p'_x. Finally, p_x is removed using a forced contraction. See Figure 1(d).

The distance measure $d_{spr}(T_1, T_2)$ between X-trees is the minimum number of SPR operations required to transform T_1 into T_2. A related distance measure is the *hybridization number*, $hyb(T_1, T_2)$, which is defined in terms of hybrid networks. A *hybrid network* of T_1 and T_2 is a directed acyclic graph H such that both trees, with their edges directed away from the root, can be obtained from H by forced contractions and edge deletions. For a node $x \in H$, let $\deg_{\text{in}}(x)$ be its in-degree and $\deg^-_{\text{in}}(x) = \max(0, \deg_{\text{in}}(x) - 1)$. Then $hyb(T_1, T_2) = \min_H \sum_{x \in H} \deg^-_{\text{in}}(x)$, taking the minimum over all hybrid networks H of T_1 and T_2. These metrics are related to the sizes of appropriately defined agreement forests. To define these, we first introduce some terminology.

For a forest F whose components $T_1, T_2, \ldots, T_k$ have label sets $X_1, X_2, \ldots, X_k$, we say F *yields* the forest with components $T_1|X_1, T_2|X_2, \ldots, T_k|X_k$; if $X_i = \emptyset$, then $T_i(X_i) = \emptyset$ and, hence, $T_i|X_i = \emptyset$. For a subset E of edges of G, we use $F - E$ to denote the forest obtained by deleting the edges in E from F, and $F \div E$ to denote the forest yielded by $F - E$. We say that $F \div E$ is a *forest of* F.

Given X-trees T_1 and T_2 and forests F_1 of T_1 and F_2 of T_2, a forest F is an *agreement forest* (AF) of F_1 and F_2 if it is a forest of both F_1 and F_2; see Figure 2. F is a *maximum agreement forest* (MAF) of F_1 and F_2 if there is no AF of F_1 and F_2 with fewer components. We denote the number of components in an MAF of F_1 and F_2 by $m(F_1, F_2)$, and the size of the smallest edge set E such that $F \div E$ is an AF of F_1 and F_2 by $e(F_1, F_2, F)$, where F is a forest of F_2. Bordewich and Semple [8] showed that $d_{spr}(T_1, T_2) = e(T_1, T_2, T_2) = m(T_1, T_2) - 1$.

Hybridization numbers correspond to MAFs with an additional constraint. For two forests F_1 and F_2 of T_1 and T_2 and an AF F of F_1 and F_2, each node x in F can be mapped to nodes $\phi_1(x)$ in T_1 and $\phi_2(y)$ in T_2 by defining X^x to be the set of labelled descendants of x in F and $\phi_i(x)$ to be the lowest common ancestor in T_i of all nodes in X^x. We say that F contains a *cycle* if there exist nodes x and y (a *cycle pair* (x, y)) that are roots of trees in F and such that $\phi_1(x)$ is an ancestor of $\phi_1(y)$ and $\phi_2(y)$ is an ancestor of $\phi_2(x)$. Otherwise, F is an *acyclic agreement forest* (AAF). A *maximum acyclic agreement forest* (MAAF) of F_1 and F_2 is an AAF with the minimum number of components among all AAFs of F_1 and F_2. We denote its size by $\tilde{m}(F_1, F_2)$ and the number of edges in a forest F of F_2 that must be cut to obtain an AAF of F_1 and F_2 by $\tilde{e}(F_1, F_2, F)$. Baroni et al. [1] showed that $hyb(T_1, T_2) = \tilde{e}(T_1, T_2, T_2) = \tilde{m}(T_1, T_2) - 1$.

We write $a \sim_F b$ when there exists a path between two nodes a and b of a forest F. For a node $x \in F$, F^x denotes the subtree of F induced by all descendants of x, inclusive. For forests F_1 and F_2 and nodes $a, b \in F_1$ with a common parent, we say (a, b) is a *sibling pair* of F_1 if there exist nodes $a', b' \in F_2$ such that $F_1^a = F_2^{a'}$ and $F_1^b = F_2^{b'}$. We refer to a' and b' as a and b for simplicity.

3 The Algorithms

In this section, we present our improved FPT algorithms for computing an MAF or MAAF of two phylogenies. As is customary for FPT algorithms, we focus on the decision version of the problem: "Given two X-trees T_1 and T_2, a distance measure $d(\cdot, \cdot)$, and a parameter k, is $d(T_1, T_2) \leq k$?" To compute the distance between two trees, we start with $k = 0$ and increase it until we receive an affirmative answer. This does not asymptotically increase the running time of the algorithm, as the dependence on k is exponential.

We begin with the MAF algorithm. The algorithm is recursive. Each invocation takes as input two forests F_1 and F_2 of T_1 and T_2 and a parameter k, and decides whether $e(T_1, T_2, F_2) \leq k$. We denote such an invocation by $\textsc{Maf}(F_1, F_2, k)$. The forest F_1 is the union of a tree $\dot{T}_1$ and a forest F, while F_2 is the union of the same forest F and another forest $\dot{F}_2$ with the same label set as $\dot{T}_1$. We maintain two sets of labelled nodes: R_d contains the roots of F, and R_t contains roots of subtrees that agree between $\dot{T}_1$ and $\dot{F}_2$. We refer to the nodes in these sets by their labels. For the top-level invocation, $F_1 = \dot{T}_1 = T_1$, $F_2 = \dot{F}_2 = T_2$, and $F = \emptyset$; R_d is empty, and R_t contains all leaves of F_2.

$\textsc{Maf}(F_1, F_2, k)$ identifies a small collection $\{E_1, E_2, \dots, E_q\}$ of subsets of edges of $\dot{F}_2$ such that $e(T_1, T_2, F_2) \leq k$ if and only if $e(T_1, T_2, F_2 - E_i) \leq k - |E_i|$, for at least one $1 \leq i \leq q$. It makes a recursive call $\textsc{Maf}(F_1, F_2 - E_i, k - |E_i|)$, for each subset E_i, and returns "yes" if and only if one of these calls does. The steps of this procedure are as follows.

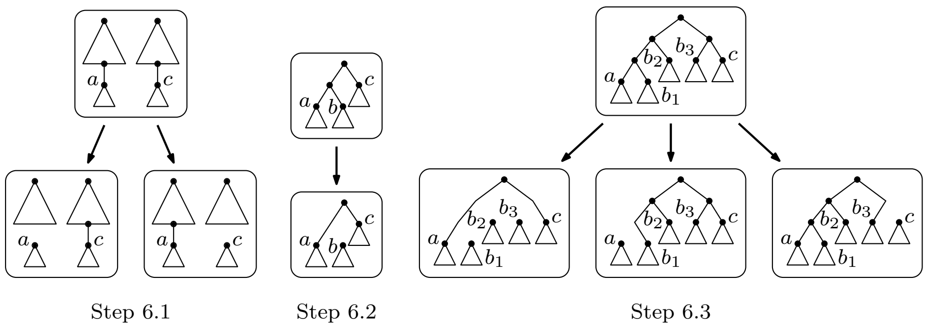


Fig. 3. The cases in Step 6 of the rooted MAF algorithm. Only $\dot{F}_2$ is shown. Each box represents a recursive call.

1. (Failure) If $k < 0$, there is no subset E of at most k edges of F_2 such that $F_2 - E$ yields an AF of T_1 and T_2. Return "no" in this case.
2. (Success) If $|R_t| \le 2$, then $\dot{F}_2 \subseteq \dot{T}_1$. Hence, $\dot{F}_2 \cup F$ is an AF of F_1 and F_2 and, thus, also of T_1 and T_2. Return "yes" in this case.
3. (Prune maximal agreeing subtrees) If there is no node $r \in R_t$ that is a root in $\dot{F}_2$, go to Step 4. Otherwise remove r from R_t and add it to R_d, thereby moving the corresponding subtree of $\dot{F}_2$ to F. Cut the edge e_r in $\dot{T}_1$ and apply a forced contraction to remove r's parent from $\dot{T}_1$. This does not alter F_2 and, thus, neither $e(T_1, T_2, F_2)$. Return to Step 2.
4. Choose a sibling pair (a, c) in $\dot{T}_1$ such that $a, c \in R_t$.
5. (Grow agreeing subtrees) If (a, c) is not a sibling pair of $\dot{F}_2$, go to Step 6. Otherwise remove a and c from R_t, label their parent in both trees with (a, c), and add it to R_t. Return to Step 2.
6. (Cut edges) Distinguish three cases (see Figure 3).
   6.1. If $a \nsim_{F_2} c$, make two recursive calls $\textsc{Maf}(F_1, F_2 \div \{e_a\}, k-1)$ and $\textsc{Maf}(F_1, F_2 \div \{e_c\}, k-1)$.
   6.2. If $a \sim_{F_2} c$ and the path from a to c in $\dot{F}_2$ has one pendant node b, make one recursive call $\textsc{Maf}(F_1, F_2 \div \{e_b\}, k-1)$.
   6.3. If $a \sim_{F_2} c$ and the path from a to c in $\dot{F}_2$ has $q \ge 2$ pendant nodes $b_1, b_2, \dots, b_q$, make three calls $\textsc{Maf}(F_1, F_2 \div \{e_{b_1}, e_{b_2}, \dots, e_{b_q}\}, k-q)$, $\textsc{Maf}(F_1, F_2 \div \{e_a\}, k-1)$, and $\textsc{Maf}(F_1, F_2 \div \{e_c\}, k-1)$.

   Return "yes" if one of the recursive calls does; otherwise return "no".

The above algorithm is identical to the one presented in [18], with the exception of Step 6. In this step, the algorithm of [18] chooses a and c so that the distance of a from the root of T_2 is no less than that of c, identifies the sibling b of a in $\dot{F}_2$ and then makes three recursive calls $\textsc{Maf}(F_1, F_2 \div \{e_a\}, k-1)$, $\textsc{Maf}(F_1, F_2 \div \{e_b\}, k-1)$, and $\textsc{Maf}(F_1, F_2 \div \{e_c\}, k-1)$. Next we show that by distinguishing between Cases 6.1–6.3 we achieve an improved running time.

Theorem 1. *For two rooted X-trees T_1 and T_2 and a parameter k, it takes $O((1+\sqrt{2})^k n) = O(2.42^k n)$ time to decide whether $e(T_1, T_2, T_2) \le k$.*

Using the same data structures as in the algorithm of [18], each recursive call takes $\mathrm{O}(n)$ time. Thus, the running time claimed in Theorem 1 follows if we can bound the number of recursive calls by $\mathrm{O}((1+\sqrt{2})^k)$. The number of recursive calls spawned by an invocation depends only on k with the following recurrence

$$I(k) = \begin{cases} 1 + 2I(k-1) & \text{Case 6.1} \\ 1 + I(k-1) & \text{Case 6.2} \\ 1 + 2I(k-1) + I(k-q) & \text{Case 6.3} \end{cases}$$
$$\leq 1 + 2I(k-1) + I(k-2)$$

because Case 6.3 dominates the other two cases and $q \geq 2$ in this case. Simple substitution shows that this recurrence solves to $I(k) = \mathrm{O}((1+\sqrt{2})^k)$. It remains to prove the correctness of the algorithm. Our strategy is as follows. Consider an edge set E of size $e(T_1, T_2, F_2)$ and such that $F_2 \div E$ is an AF of T_1 and T_2. $F_2 \div E$ is also an AF of F_1 and F_2. Now we consider each of the three cases of Step 6. We show that, in each case, there exists a set E as above and a recursive call $\textsc{Maf}(F_1, F_2 \div E_i, k - |E_i|)$ made in this case such that $E_i \subseteq E$. This implies by induction that (1) none of the other recursive calls we make returns "yes" unless $e(T_1, T_2, F_2) \leq k$ (because each recursive call $\textsc{Maf}(F_1, F_2 \div E_j, k')$ satisfies $k' = k - |E_j|$) and (2) the recursive call $\textsc{Maf}(F_1, F_2 \div E_i, k - |E_i|)$ returns "yes" if and only if $e(T_1, T_2, F_2) = k$. Thus, the current invocation gives the correct answer. Each of the following three lemmas considers one case. Due to the lack of space, proofs are omitted but can be found in [17].

Lemma 1 (Case 6.1). *If $a \nsim_{F_2} c$, there exists an edge set E of size $e(T_1, T_2, F_2)$ (resp. $\tilde{e}(T_1, T_2, F_2)$) such that $F_2 \div E$ is an AF (resp. AAF) of T_1 and T_2 and $E \cap \{e_a, e_c\} \neq \emptyset$.*

Lemma 2 (Case 6.2). *If $a \sim_{F_2} c$ and the path from a to c in F_2 has only one pendant node b, there exists an edge set E of size $e(T_1, T_2, F_2)$ such that $F_2 \div E$ is an AF of T_1 and T_2 and $e_b \in E$.*

Lemma 3 (Case 6.3). *If $a \sim_{F_2} c$ and the path from a to c in F_2 has $q \geq 2$ pendant nodes $b_1, b_2, \dots, b_q$, there exists an edge set E of size $e(T_1, T_2, F_2)$ (resp. $\tilde{e}(T_1, T_2, F_2)$) such that $F_2 \div E$ is an AF (resp. AAF) and either $E \cap \{e_a, e_c\} \neq \emptyset$ or $\{e_{b_1}, e_{b_2}, \dots, e_{b_q}\} \subseteq E$.*

As shown in [18], an algorithm for deciding whether T_1 and T_2 have a maximum *acyclic* agreement forest of size at most $k+1$ can be obtained using a two-phased approach. In the first phase, we employ the MAF algorithm. Whenever the MAF algorithm would return "yes" in Step 2, however, we invoke a second algorithm that tests whether all cycles in the obtained agreement forest can be eliminated by cutting at most k edges:

2′. If $|R_t| \leq 2$, then $F_2 = \dot{F}_2 \cup F$ is an AF of T_1 and T_2. Now invoke an algorithm $\textsc{Maaf}(F_2, k)$ that decides whether all cycles in F_2 can be eliminated by cutting at most k edges.

Such an algorithm $\textsc{Maaf}(F, k)$ with running time $\mathrm{O}(2^k n \log n)$ time is presented in [18]. The correctness of this two-phased MAAF procedure follows if we can show that in each of the three cases in Step 6, there exists a recursive call $\textsc{Maf}(F_1, F_2 \div E_i, k - |E_i|)$ such that E_i is a subset of a set E of size $\tilde{e}(T_1, T_2, F_2)$ and such that $F_2 \div E$ is an AAF of T_1 and T_2. For Cases 6.1 and 6.3, Lemmas 1 and 3 state that this is the case. In Case 6.2, however, edge e_b may not belong to such a set. To fix this, we replace it with the following case when computing an MAAF:

6.2′. If the path from a to c in F_2 has one pendant node b, make two recursive calls $\textsc{Maf}(F_1, F_2 \div \{e_b\}, k - 1)$ and $\textsc{Maf}(F_1, F_2 \div \{e_c\}, k - 1)$.

Theorem 2. *For two rooted X-trees T_1 and T_2 and a parameter k, it takes $\mathrm{O}((1 + \sqrt{2})^k n \log n) = \mathrm{O}(2.42^k n \log n)$ time to decide whether $\tilde{e}(T_1, T_2, T_2) \leq k$.*

The correctness proof of the MAAF algorithm obtained from our MAF algorithm using the above modifications is identical to the correctness proof of the MAF algorithm; however, we use Lemma 4 below instead of Lemma 2 to show that we cut the right edges in Case 6.2′. To bound the running time of the algorithm, we observe that the recurrence for the number of recursive calls in Case 6.2′ is $I(k) = 1 + 2I(k - 1)$, which is still dominated by the recurrence for Case 6.3. Using that the running time of the MAAF procedure is $T(n, k) = \mathrm{O}(2^k n \log n)$, substitution yields the claimed bound. Again, a proof of this lemma can be found in [17].

Lemma 4 (Case 6.2′). *If $a \sim_{F_2} c$ and the path from a to c in F_2 has only one pendant node b, there exists an edge set E of size $\tilde{e}(T_1, T_2, F_2)$ and such that $F_2 \div E$ is an AAF of T_1 and T_2 and either $e_b \in E$ or $e_c \in E$.*

As in [18], Theorems 1 and 2 along with known kernelizations [8,9] imply the following corollary.

Corollary 1. *For two rooted X-trees T_1 and T_2 and a parameter k, it takes $\mathrm{O}(2.42^k k + n^3)$ time to decide whether $e(T_1, T_2, T_2) \leq k$ and $\mathrm{O}(2.42^k k \log k + n^3)$ time to decide whether $\tilde{e}(T_1, T_2, T_2) \leq k$.*

4 Evaluation of SPR Distance Algorithms

In this section, we present an experimental evaluation of our MAF (SPR distance) algorithm that compares the algorithm's performance to that of two competitors using the protein tree data set examined in [2,3] and using synthetic trees. We also investigate the impact of the improved branching rules in Step 6 on the performance of the algorithm. Our competitors were **sprdist** [20] and **treeSAT** [4], which reduce the problem of computing SPR distances to integer linear programming and satisfiability testing, respectively. We do not provide a comparison with **EEEP** [3] because **sprdist** outperformed it and other heuristics at finding the exact SPR distance between binary rooted phylogenies [20].

For **sprdist** and **treeSAT**, we used publicly available implementations of these algorithms. For our own algorithm, we developed an implementation in C++ that allowed us to individually turn the optimized branching rules in Step 6 on and off. When the optimized branching rule in one of the cases is turned off, the algorithm uses the 3-way branching of [18] described on page 145 in this case. In particular, with all optimizations off, the algorithm is the one of [18]. Source code for our algorithm is available at [19].

We also implemented the linear-time 3-approximation algorithm for MAF of [18] and used it to implement two additional optimizations of our FPT algorithm. The FPT algorithm searches for the correct value of $e(T_1, T_2, T_2)$ by starting with a lower bound k of $e(T_1, T_2, T_2)$ and incrementing k until it determines that $k = e(T_1, T_2, T_2)$. If the 3-approximation algorithm returns a value of k', then $e(T_1, T_2, T_2) \geq \lceil k'/3 \rceil$; by using this as the starting value of our search, we can skip early iterations of the algorithm and thereby obtain a small improvement in the running time. The same approach can be used in a branch-and-bound strategy that prunes unsuccessful branches from the search tree. In particular, we extended Step 1 of the FPT algorithm as follows:

1′. (Failure) If $k < 0$, return "no". Otherwise compute a 3-approximation k' of $e(T_1, T_2, F_2)$. If $k' > 3k$, then $e(T_1, T_2, F_2) > k$; return "no" in this case.

We allowed this optimization of Step 1 to be turned on or off in our algorithm to investigate its effect on the running time, but our implementation always uses the 3-approximation algorithm to provide a starting guess of $e(T_1, T_2, T_2)$.

4.1 Data Sets

The protein tree data set of [2,3] contains 5689 protein trees with 10 to 144 leaves (each corresponding to a different microbial genome); each of these was compared in turn to a rooted reference tree covering all 144 genomes. The protein trees were unrooted, so we selected a rooting for each tree that gave the minimum SPR distance according to the 3-approximation algorithm of [18].

The synthetic tree pairs were created by first generating a random tree T_1 and then transforming it into a second tree T_2 using a known number of random SPR operations. Note that the SPR distance may be lower because the sequence of SPR operations we generated may not be the shortest such sequence. For n taxa, the label set of T_1 was represented using integers 1 through n, and T_1 was generated by splitting the interval $[1, n]$ into two sub-intervals uniformly at random, recursively generating two trees with these two intervals as label sets and then adding a root to merge these trees. Random SPR operations were generated by choosing an edge xp_x to cut uniformly at random and then choosing the new parent p'_x of x uniformly at random from among all valid locations of p'_x. We constructed pairs of 100-leaf trees with 1–20 SPR operations and with $25, 30, \ldots, 50$ SPR operations. We also constructed pairs of 1000-leaf trees with 1–20 SPR operations and with $25, 30, \ldots, 100$ SPR operations. For each tree size and number of SPR operations we generated ten pairs of trees.

4.2 Results

Our experiments were performed on a 3.16Ghz Xeon system with 4GB of RAM and running CentOS 2.6 Linux in a Rocks 5.1 cluster. Our code was compiled using gcc 4.4.3 and optimization -O2. Each run of an algorithm was limited to 5 hours of running time. If it did not produce an answer in this time limit, we say the algorithm did not solve the given input instance in the following discussion. We refer to the FPT algorithm with all optimizations off as **fpt**, and with only the branch-and-bound optimization turned on as **bb**. The activation of the improved branching rules in Step 6 is indicated using suffixes **sc** (Case 6.1: separate components), **cob** (Case 6.2: cut only b), and **cab** (Case 6.3: cut a or b). Thus, the algorithm with all optimizations on is labelled **bb_cob_cab_sc**.

Number of instances solved. Figure 4 shows the number of solved protein tree instances for the given ranges of tree sizes. Our experiments showed that the average SPR distance for trees of the same size ranged between one sixth and one third of the number of leaves. All of the algorithms solved all instances with 20 or fewer leaves and only **treeSAT** did not solve all instances with 40 or fewer leaves. **sprdist** solved most of the instances with 41-50 leaves, and half of the instances with 51-100 leaves, but very few of the larger instances. **fpt** performed similarly to **sprdist** but solved all of the instances with 41-50 leaves and more of the larger instances than **sprdist**. **bb** improved upon this somewhat. However, adding our new branching rules improved the results greatly. In particular, **bb_cob_cab_sc** solved all of the instances in this data set.

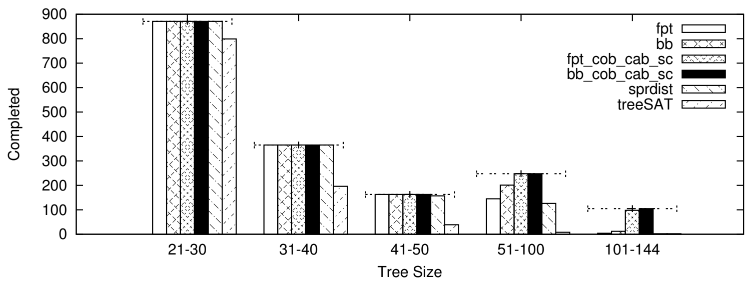


Fig. 4. Completion on the protein tree runs grouped by tree size. Abbreviations are defined in the text.

Figure 5 shows the number of protein trees found with a given SPR distance from the reference tree. The "number solved" axis is a $\log_3$-scale to allow easy comparison of the trees with small and large SPR distances, as the majority had small SPR distances. **treeSAT** was unable to solve any instances with SPR distance greater than 8. **sprdist** and **fpt** solved instances with a distance as large as 20. Since **bb_cob_cab_sc** solved all the instances in this data set, including instances with an SPR distance of 46, we were able to verify that **sprdist** and **fpt** solved all instances with SPR distance up to 15 and 18, respectively.

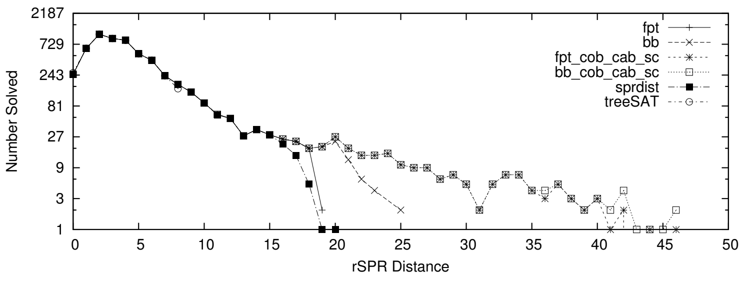


Fig. 5. Number of protein trees solved by rSPR distance

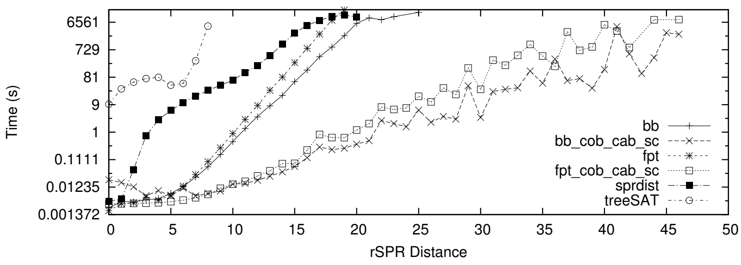


Fig. 6. Mean running time of the FPT, sprdist, and treeSAT protein tree runs

Running time. Figure 6 shows the mean running time of the algorithms on *solved* protein tree instances with the given SPR distance. The time axis here and in the following figures is a $\log_3$-scale to highlight the exponential running time of the algorithms and to allow easy comparison of the runs. The curves for some of the algorithms 'dip" for higher distance values, which is a result of taking the average running time only over solved instances. The slope of the curve for **fpt** is close to 1, indicating that the algorithm is close to its worst-case running time of $O(3^k n)$. **bb** shows a marked improvement over **fpt**; however, the improvement achieved using the new branching rules is much more dramatic. **treeSAT** was much slower than all the other algorithms and although **sprdist** solved a similar number of instances as **fpt**, as shown in Figure 5, it took much longer to solve them on average. The two instances that **sprdist** solved with an SPR distance of 19 and 20 are an exception to this, but that is likely an artifact of considering only solved instances. **bb_cob_cab_sc** solved all input instances with SPR distance of up to 20 in 5.5 seconds or less, and solved instances with SPR distance up to 46 in well under 2 hours, while none of the previous methods was able to solve instances with SPR distance greater than 20 in under 5 hours.

Figure 7 shows the mean running time of the fixed-parameter algorithms on the random data set. As expected, **fpt** took 10 times longer on average for the 1000-leaf trees as for the 100-leaf trees, given the same SPR distance. **fpt_cob_cab_sc** did not show this difference, which suggests that the improved branching rules have a more pronounced impact on larger trees. **bb_cob_cab_sc**

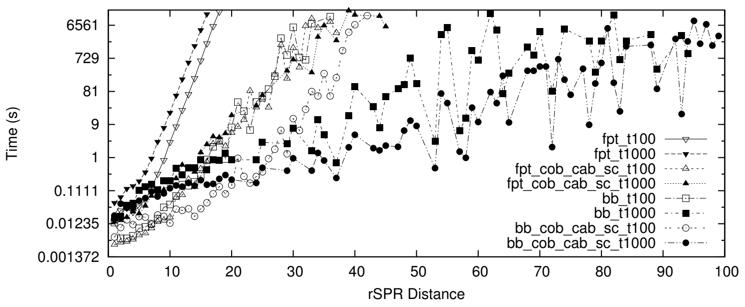


Fig. 7. Mean running time of the FPT algorithm on the data set of randomly permuted trees. The trees with 100 and 1000 leaves are shown separately.

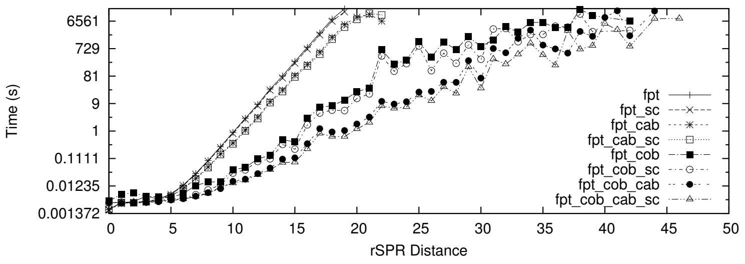


Fig. 8. Mean running time for combinations of the new cases on the protein tree runs

was able to solve instances with SPR distances up to 99 on the 1000-leaf trees, while a distance of 42 was the limit on 100-leaf trees. We believe that, since the proportion of SPR operations to the number of leaves is smaller for the bigger trees, the randomly generated SPR operations are more likely to operate on independent subtrees, which brings the approximation ratio of the approximation algorithm closer to its worst-case bound of 3 on these inputs. In our case, this provides better lower bounds on the true SPR distance and, thus, allows us to prune more branches in the search tree than is the case for the smaller trees.

Figure 8 shows the mean running time of the fixed-parameter algorithms without branch-and-bound on the protein tree data set and using only some of the improved branching rules. Case 6.1, Case 6.3, and Case 6.2 provide small, moderate and large improvements, respectively. Using all of the cases gives another large improvement, since each occurs under different conditions.

5 Conclusions

Our theoretical results improve on previous work, and our experiments confirm that these improvements have a tremendous impact in practice. Our algorithm efficiently solves problems with up to 144 leaves and an SPR distance of 20 in less than a second on average; for distance values up to 46, the running time was less than two hours. Our branch-and-bound approach showed a marked improvement on larger trees, allowing us to compute distance values up to 42 on 100-leaf synthetic trees and 99 on 1000-leaf synthetic trees.

We expect experimental results using an implementation of the hybridization algorithm would be similar, as only Case 6.2 is more costly than in the SPR algorithm. Thus, our hybridization algorithm should also be able to solve instances beyond the reach of current hybridization approaches. Producing an implementation of the hybridization algorithm is future work. Other open problems include extending our results to multifurcating trees or the related problem of finding maximum agreement forests of multiple trees.

References

1. Baroni, M., Grünewald, S., Moulton, V., Semple, C.: Bounding the number of hybridisation events for a consistent evolutionary history. J. Math. Biol. 51(2), 171–182 (2005)
2. Beiko, R.G., Harlow, T.J., Ragan, M.A.: Highways of gene sharing in prokaryotes. P. Natl. Acad. Sci. USA 102(40), 14332–14337 (2005)
3. Beiko, R.G., Hamilton, N.: Phylogenetic identification of lateral genetic transfer events. BMC Evol. Biol. 6(1), 15 (2006)
4. Bonet, M.L., John, K.S.: Efficiently Calculating Evolutionary Tree Measures Using SAT. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 4–17. Springer, Heidelberg (2009)
5. Bonet, M.L., John, K.S., Mahindru, R., Amenta, N.: Approximating subtree distances between phylogenies. J. Comp. Biol. 13(8), 1419–1434 (2006)
6. Bordewich, M., Linz, S., John, K.S., Semple, C.: A reduction algorithm for computing the hybridization number of two trees. Evol. Bioinform. 3, 86–98 (2007)
7. Bordewich, M., McCartin, C., Semple, C.: A 3-approximation algorithm for the subtree distance between phylogenies. J. Disc. Alg. 6(3), 458–471 (2008)
8. Bordewich, M., Semple, C.: On the computational complexity of the rooted subtree prune and regraft distance. Annals of Comb. 8(4), 409–423 (2005)
9. Bordewich, M., Semple, C.: Computing the hybridization number of two phylogenetic trees is fixed-parameter tractable. IEEE/ACM T. Comp. Biol. 4(3), 458–466 (2007)
10. Bordewich, M., Semple, C.: Computing the minimum number of hybridization events for a consistent evolutionary history. Disc. Appl. Math. 155(8), 914–928 (2007)
11. Hein, J., Jiang, T., Wang, L., Zhang, K.: On the complexity of comparing evolutionary trees. Disc. Appl. Math. 71(1-3), 153–169 (1996)
12. Hickey, G., Dehne, F., Rau-Chaplin, A., Blouin, C.: SPR distance computation for unrooted trees. Evol. Bioinform. 4, 17–27 (2008)
13. Hillis, D.M., Moritz, C., Mable, B.K. (eds.): Molecular Systematics. Sinauer Associates (1996)
14. Maddison, W.P.: Gene trees in species trees. Syst. Biol. 46(3), 523–536 (1997)
15. Nakhleh, L., Warnow, T., Lindner, C.R., John, K.S.: Reconstructing reticulate evolution in species—theory and practice. J. Comp. Biol. 12(6), 796–811 (2005)
16. Rodrigues, E.M., Sagot, M.F., Wakabayashi, Y.: The maximum agreement forest problem: Approximation algorithms and computational experiments. Theor. Comp. Sci. 374(1-3), 91–110 (2007)

17. Whidden, C., Beiko, R.G., Zeh, N.: Fast FPT algorithms for computing rooted agreement forests: Theory and experiments. Tech. Rep. CS-2010-03, Faculty of Computer Science, Dalhousie University (2010)
18. Whidden, C., Zeh, N.: A unifying view on approximation and FPT of agreement forests. In: Salzberg, S.L., Warnow, T. (eds.) WABI 2009. LNCS, vol. 5724, pp. 390–401. Springer, Heidelberg (2009)
19. Whidden, C.: rSPR FPT Software, `http://kiwi.cs.dal.ca/Software/RSPR`
20. Wu, Y.: A practical method for exact computation of subtree prune and regraft distance. Bioinformatics 25(2), 190–196 (2009)

Experimental Evaluation of Approximation and Heuristic Algorithms for Sorting Railway Cars*

Alain Hauser[1] and Jens Maue[2]

[1] Seminar for Statistics, ETH Zurich, Switzerland
hauser@stat.math.ethz.ch
[2] Institute of Theoretical Computer Science, ETH Zurich, Switzerland
jens.maue@inf.ethz.ch

Abstract. In this paper we consider a sorting problem from railway optimization called train classification, which is NP-hard in general. We introduce two new variants of an earlier developed 2-approximation as well as a new heuristic for finding feasible classification schedules, i.e. solutions for the train classification problem. We evaluate the four algorithms experimentally using various synthetic and real-world traffic instances and further compare them to an exact IP approach. It turns out that the heuristic matches up to the basic approximation, but both are clearly outperformed by our heuristically improved 2-approximations. Finally, with an average objective value of only 5.4 % above optimal, the best algorithm gets close to the real-world schedules of the IP approach, so we obtain very satisfactory practical schedules extremely quickly.

Keywords: railway optimization, sorting, approximation algorithms, heuristics, experimental evaluation.

1 Introduction

In everyday railway operation, freight trains are split up into their cars and reassembled to build new trains, a procedure that is called *train classification* and presents the optimization problem dealt with in this paper. The classification process takes place in a special installation of tracks and switches called *classification yard*. Inbound trains arrive on a *hump track* where their cars are decoupled and pushed over a *hump* by a shunting engine. The cars then accelerate by gravity and roll through a tree of switches by which every car can be guided separately to some *classification track*; we call this process a *roll-in* operation. In a *pull-out* operation an engine pulls all the cars of some classification track back over the hump to perform another roll-in. A pair of a pull-out and subsequent roll-in is called a (*sorting*) *step*, an initial roll-in followed by a sequence of h sorting steps a *classification schedule*, and h its *length*.

There are ℓ *inbound trains* which, concatenated in the order they arrive at the yard, form the *inbound train sequence*. Further, there are m order specifications

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

for outbound trains. We search a *feasible* classification schedule, i.e. a schedule which, if applied to the inbound sequence, yields the correctly ordered outbound trains, each on a separate classification track. The number of steps h essentially determines the time to perform the schedule. This is the main objective, while the total number w of cars rolled in presents a secondary objective.

Related Work. The classification method above is known as *multistage sorting* and is applied to accomplish tight sorting requirements for outbound trains. In the field of railway engineering, Krell [10] analyzes different multistage methods with regard to several objectives and resource requirements without proving optimality. There are earlier appearances of some of these methods with Flandorffer [5] and Pentinga [14], while Siddiqee [15] and Daganzo et al. [2] give more recent descriptions. For the practical case of classification tracks of bounded length, minimizing the number of sorting steps is shown to be an NP-hard problem in [8] by introducing an efficient encoding of classification schedules. This encoding is applied in [9] to obtain a 2-approximation for the same setting and in [12] for an exact integer programming approach. A related algorithmic multistage sorting problem is considered by Dahlhaus et al. [3], who also give a systematic framework for order requirements of outbound trains. These requirements are summarized by Hansmann et al. in [7], which also contains a wide framework of multistage sorting schemes and the so-called *single-stage method*, which is a related method suitable for simpler sorting specifications. Further overviews of train classification can be found in [4] and recently [6].

Our Contribution. For the NP-hard optimization problem of train classification with limited track capacities, we introduce two advanced variants of a previous 2-approximation and a new heuristic algorithm (Sect. 3), and we further implement the three new methods as well as the original approximation. Using a large set of realistic instances from railway practice as well as more complex synthetically derived problems (Sect. 4), we extensively evaluate the four algorithms including a comparison to a previous exact IP approach (Sect. 5). Analyzing the different algorithms and results yields interesting insights that also indicate directions for future research (Sect. 6).

2 Encoding Classification Schedules

Terminology and Notation. In addition to the concepts of Sect. 1, we use the notation of [8]: every car τ is represented by a positive integer $\tau \in \mathbb{N}$, a train is defined as an ordered sequence $T = (\tau_1, \ldots, \tau_k)$ of cars τ_i, $i = 1, \ldots, k$, and the number of cars k of T is called the *length* of T. The total volume of cars is denoted by n, and we assume the inbound train sequence to be a permutation of $(1, \ldots, n)$. The number of outbound trains is denoted by m and the length of the ith outbound train by n_i, $i = 1, \ldots, m$, so $\sum_{i=1}^{m} n_i = n$. W.l.o.g. we assume the specification of the first outbound train is $(1, \ldots, n_1)$, the second $(n_1+1, \ldots, n_1+n_2)$, etc. (There is no implied order of the outbound trains as explained in the next paragraph.) A pair of cars τ, $\tau+1$ of the same outbound train is called a *break* if the cars occur in reversed order in the inbound sequence.

During the classification process the cars for different outbound trains are sorted simultaneously on the same set of classification tracks, called the *sorting tracks*. The number of available sorting tracks is unbounded, and the number actually used corresponds to the number of pull-out operations, i.e. to the schedule length. Each sorting track is pulled out once; we will refer to the track pulled out in the kth step by θ_k, $k = 0, \ldots, h-1$. The maximum number C of cars that fit on any sorting track is called the *capacity* of the tracks. The outbound trains are finally formed on a separate track each, called *destination tracks*.

Our optimization problem is now defined as follows: Given an inbound train sequence of n cars and m order specifications of outbound trains, find a feasible classification schedule of minimum length.

Representation. Any classification schedule of length h can be encoded by an assignment of cars to bitstrings of length h: the course of the jth car is represented by $b^j = b^j_{h-1} \ldots b^j_0$ with $b^j_k = 1$ iff the car visits track θ_k pulled in step k, $k = 0 \ldots h-1$. If this car is pulled in the kth pull-out, it is rolled in to θ_ℓ with $\ell = \min\{k < i \leq h{-}1 \mid b^j_i{=}1\}$. If $b^j_i = 0$ for all $i > k$, it is sent to the destination track of its outbound train. The bitstrings $b^1, \ldots, b^n$ form the rows of a matrix B of width h, and $b_0, \ldots, b_{h-1}$ denote its columns from right to left.

The total number of cars $w(B) := \sum_{i=0}^{h-1} \sum_{j=1}^{n} b^j_i$ rolled in is called the *weight* of B, the number $\ell(b_i) = \sum_{j=1}^{n} b^j_i$ rolled in in the ith step the *load* of b_i; further, if $b_i = \mathbf{0}$ for all $i = h', \ldots, h{-}1$ and $b_{h'-1} \neq \mathbf{0}$, then h' is called the *actual length* of B, i.e., removing all leading zero-columns gives the actual length.

The encoding can also be applied to construct feasible schedules as formalized in Thrm. 1 below. (Note that the formulation differs from that of [9].) Equation (1) will be referred to by the *order constraint* and applies to all feasible schedules independently of the respective yard size or layout.

Theorem 1 ([9], Thrm. 1). *For a classification problem of n cars, let $F \subseteq [n]$ be the set of cars that are the first of their resp. outgoing trains, and let* $\mathrm{rev}(i,j)$ *be an indicator function with* $\mathrm{rev}(i,j) = 1$ *iff the ith and jth car are in reversed order in the inbound sequence. Then, a schedule B of length h is feasible iff:*

$$\sum_{0 \leq i < h} 2^i b^j_i \geq \mathrm{rev}(j, j{-}1) + \sum_{0 \leq i < h} 2^i b^{j-1}_i \quad \textit{for all } j \in \{1, \ldots, n\} \setminus F \ . \tag{1}$$

Note that $\sum_{0 \leq i < h} 2^i b^j_i$ is the integer represented by the bitstring b^j. Basically, if two consecutive cars j, $j{+}1$ of the same outbound train arrive at the yard in reversed order, j must get a smaller bitstring than $j{+}1$. (See [9] for an example.)

3 Algorithms

3.1 Optimal Classification Schedules

As mentioned before, the train classification problem with limited track capacities is NP-hard. To assess the algorithms of Sect. 3.2–3.4, we tried to derive optimal schedules in Sect. 5 with the integer programming approach of [12].

In the basic form of this IP model, the encoding of a schedule B is represented by binary variables b_i^j, $j = 1, \ldots, n$, $i = 0, \ldots, h-1$, for the jth car in the ith sorting step as explained in Sect. 2. Recall that the capacity of a track is denoted by C. The IP constraints are given by (1) and the following equation for the restricted capacity of the classification tracks, called the *capacity constraint*:

$$\sum_{1 \leq j \leq n} b_i^j \leq C \quad \text{for all } i \in \{0, \ldots, h-1\} \ . \tag{2}$$

The model's objective minimizes the weight $w(B)$ of B for a fixed length h, which presents a secondary objective as mentioned in Sect. 1. To minimize the primary objective h, a short sequence of integer programs is solved with increasing values for h, corresponding to the approach in [12] further explained in Sect. 4.

3.2 Basic 2-Approximation

A polynomial time 2-approximation was introduced in [9] as summarized in this section. Consider the following constraint on the number of cars rolled in for a schedule B of length h, which we will also call the *weight constraint*:

$$w(B) \leq Ch \ . \tag{3}$$

Every schedule satisfying (2) satisfies (3) as well, which is used in the following algorithm to achieve the 2-approximation in two stages. First, a feasible schedule $\tilde{B}$ satisfying (3) of minimum length $\tilde{h}$ is derived by dynamic programming ([9], Lemma 6): for increasing values of $h' = 0, 1, 2, \ldots$, the algorithm iteratively estimates the schedule B' that has minimum weight among all schedules of this length h' until the first such schedule meets (3). In case of m outbound trains, there is one dynamic programming table for every train, and these tables are computed independently of each other while h' is increased simultaneously for all tables. Each iteration results in m schedules $B'_1, \ldots, B'_m$ of the same length h', for which $w(B') \leq h'C$ is checked for the "vertical" concatenation B'. We will call a schedule produced by this 1st stage an *intermediate* schedule.

In the 2nd stage, every column of $\tilde{B}$ violating (2) is distributed over several newly introduced columns meeting (2) without loosing feasibility ([9], Thrm. 8). For every column of $\tilde{B}$, at most one column with load $\ell < C$ is produced, and the result has at most $\tilde{h}$ columns with load $\ell = C$ by (3). Hence, the resulting length h satisfies $h \leq 2\tilde{h} \leq 2h^*$, where h^* is the length of an optimal schedule.

3.3 Improved 2-Approximations

In the basic approximation algorithm BASE just described, there are m partial schedules $\tilde{B}_1, \ldots, \tilde{B}_m$ of length $\tilde{h}$ at the end of the 1st stage, one for each outbound train. It may happen that the dynamic programming table of the jth train found a feasible schedule B'_j with weight $w(B'_j) = w(\tilde{B}_j)$ in an earlier iteration corresponding to some $h' < \tilde{h}$. If (3) is not satisfied yet at this point, one reason for which may be high weights of other partial schedules, the basic algorithm continues with appending a leading zero-column $b_{i-1} = \mathbf{0}$ to B'_j in all subsequent

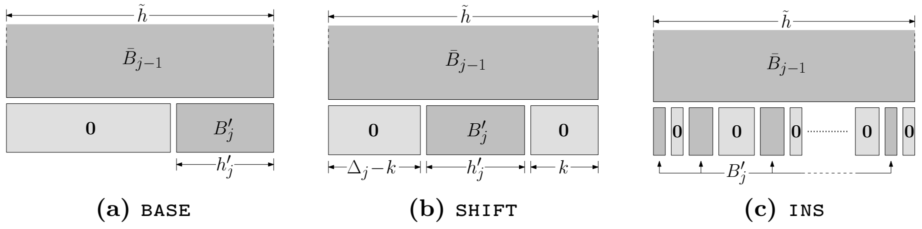


Fig. 1. Three different ways to combine the partial schedules to an intermediate schedules at the end of the 1st stage of the approximation algorithms

steps $i = h'+1, \dots, \tilde{h}$, so that the actual length of $\tilde{B}_j$ is $h' < \tilde{h}$. Besides, the algorithm is designed in a way not to yield any zero-columns which are not leading. In this way, schedules derived by BASE tend to have many columns of low index violating (2), while the load remains far behind C for higher indices.

The improved algorithms SHIFT and INS are based on the following theorem:

Theorem 2. *For any feasible schedule $B = b_{h-1} \dots b_0$ of length h, inserting a zero-column $b_k = \mathbf{0}$ at any position $k \in \{0, \dots, h\}$ of B yields a feasible schedule of length $h+1$.*

Consequently, inserting $\tilde{h} - h'_j$ zero-columns between any columns of B'_j yields a feasible (partial) schedule of length $\tilde{h}$. We now make use of this fact to obtain two new variants of the basic approximation algorithm, called SHIFT and INS, that effectively reduce the scale and number of capacity violations. Both variants apply the dynamic program like BASE but without adding leading zero-columns. Instead, two heuristics are applied that distribute the columns of partial schedules with a small actual length in more sophisticated ways, which we describe in the following. The 2nd stage is the same again for all BASE, SHIFT, and INS.

After the 1st stage, SHIFT and INS continue with sorting the partial schedules in decreasing order of their actual lengths. Let $B'_1, \dots, B'_m$ be the ordered partial schedules with *actual* length values $h'_1 \geq \dots \geq h'_m$. Both variants start with an empty schedule $\bar{B}_0$ of length $\tilde{h}$ and successively append the partial schedules to obtain concatenations $\bar{B}_j$, $j = 1, \dots, m$, as shown in Fig. 1, where $\bar{B}_m$ finally presents the intermediate schedule $\tilde{B}$ satisfying (3) passed to the 2nd stage.

In the jth step of SHIFT, $j = 1, \dots, m$, let $\Delta_j := \tilde{h} - h'_j$. According to Thrm. 2, for any $k = 0, \dots, \Delta_j$, we can add k consecutive trailing zero-columns to B'_j and $\Delta_j - k$ consecutive leading zero-columns to obtain a partial schedule of length $\tilde{h}$, which can be appended "vertically" to $\bar{B}_{j-1}$ to obtain $\bar{B}_j$ as shown in Fig. 1b. Now, SHIFT obtains $\bar{B}_j$ by choosing the $k \in \{0, \dots, \Delta_j\}$ that minimizes the value of $\sum_{i=0}^{\tilde{h}-1} (C - (\ell(b_i) \bmod C))$, where b_i is the ith column *after* appending B'_j to $\bar{B}_{j-1}$. This objective is motivated as follows: if $\ell(b_i) > C$ for some column i, an additional column will certainly be introduced in the 2nd stage as described above; hence, we try to further fill this column just below the value of $2C$, which triggers no further columns in the 2nd stage. The number of "1"s that can be added to b_i until the next multiple of C is reached is given by $(C - (\ell(b_i) \bmod C))$, which explains the objective. In contrast, BASE implicitly puts $k = 0$ in every step which yields the picture shown in Fig. 1a.

In the jth step of INS, $j = 1, \ldots, m$, we determine $(\ell(b_i) \bmod C)$ for all columns b_i of $\bar{B}_{j-1}$ and append the columns of B'_j to the h'_j columns of $\tilde{B}_{j-1}$ that have the smallest values of $(\ell(b_i) \bmod C)$, where here b_i is the ith column of $\bar{B}_{j-1}$. Then, appending zero-columns to the remaining $\tilde{h} - h'_j$ columns of $\bar{B}_{j-1}$ following Thrm. 2 yields $\bar{B}_j$ for the next iteration as illustrated in Fig. 1c. This choice of the h'_j columns is motivated similarly to the objective of SHIFT.

3.4 Heuristic Schedules

In order to achieve their approximation factors, the algorithms just described all calculate an intermediate schedule that is optimal w.r.t. (3). In contrast, our new heuristic approach HEUR first calculates the shortest feasible schedule—without regard to (2) or (3)—using the approach from [8]: if β_k, $k = 1, \ldots, m$, denotes the number of breaks in the kth outbound train of a classification instance, we assign the β+1 bitstrings representing the numbers $0, \ldots, \beta$ to the cars such that $b^j < b^{j+1}$ if j and $j+1$ form a break and $b^j = b^{j+1}$ otherwise. In this way, we obtain m partial schedules of respective actual length $h'_k = \lceil \log_2 \beta_k \rceil$, $k = 1, \ldots, m$, which are concatenated in the same way as for BASE. This yields an intermediate schedule $\tilde{B}$ of length $\tilde{h} = \max_k h'_k$ not necessarily satisfying (3).

Now, HEUR applies the 2nd stage of the approximation algorithm to $\tilde{B}$ to obtain a schedule satisfying (2). This approach does not guarantee the approximation factor but has a very simple implementation; with regard to the experimental evaluation in Sect. 5, the more sophisticated approximation algorithms are thus only useful if they perform significantly better than HEUR.

4 Experimental Setup

Test Instances. We extracted five real-world instances corresponding to five days from the complete traffic data of a whole week in 2005 for the Swiss classification yard Lausanne Triage. The instances have a total volume of cars ranging from 310 to 486, between 24 and 27 outbound trains, and numbers of breaks between 24 and 28. This yard has a track capacity of about $C = 40$, and we chose to combine our five instances with similar track capacities $C \in \{30, \ldots, 50\}$, obtaining 105 classification problems in total.

To test the approaches for more complex sorting problems, we used the synthetic instances of [13]: they have volumes of $n \in \{200, 400, 800\}$, outbound train lengths uniformly drawn from the intervals $[2 \ldots 20]$, $[2 \ldots 40]$, or $[2 \ldots 60]$, and numbers of breaks geometrically distributed with three different parameter values; with four instances for every combination of parameters, there are 108 instances in total. For the IP approach of the following section, we combined every such problem with $C \in \{10, 20, \ldots, 60\}$, yielding 648 synthetic classification problems. The quicker computation enabled us to evaluate the algorithmic approaches for a bigger set of problems: we combined the synthetic instances with $C \in \{10, 20, 30, \ldots, 10\lfloor \frac{n}{30} \rfloor\}$, obtaining 1'620 synthetic instances.

Schedule Computation. The IP model of Sect. 3.1 contains h as a parameter and searches a schedule of minimum weight for fixed h. To minimize h, we solve a sequence of IP models starting from $h = 1$ as mentioned before. We bound the number of iterations by $h = 8$ and the time for each iteration by one hour.

Let now B be a computed feasible schedule of length h. If B has been proven optimal for the IP model of the hth iteration and the same problem has been shown infeasible for $h-1$, then B is called *weight-optimal.* If the problem is infeasible for $h-1$ but B could not be proven optimal, B is called *length-optimal.* As a third possibility, the IP solver may find a feasible solution for some value h but not $h-1$ within the time limit. Finally, no solution is found if the eighth iteration ends with a timeout or infeasibility. We found weight-optimal schedules for all real-world problems, and the longest such schedule had $h = 8$. All IP computations presented in Sect. 5 were performed with ILOG Studio 3.7 featuring CPLEX 9.0 on an Intel Xeon CPU with 2.80 GHz and 2 GB main memory.

For every classification problem we computed a schedule with each BASE, SHIFT, INS, and HEUR, resp., which we compare to each other and—wherever possible—to the IP schedules in Sect. 5. In the evaluation we will refer to the shortest of the schedules returned by BASE, SHIFT, and INS for a problem by MIN, which can be regarded as another variant. All approximations and HEUR were implemented in C++, compiled with the GNU compiler g++ 4.1.2, and run on an Intel Pentium IV CPU with 2.80 GHz and 1.5 GB RAM.

Measuring Objectives. The lengths of the approximate and heuristic schedules apparently present the most important performance measure. In particular, we want to know how far from optimal these schedules are, and the notion of *relative excess* relates the length of a schedule to the optimal length: let B be a feasible schedule of length h and h^* the length of a shortest feasible schedule, both satisfying (2); then, the *relative excess* $\eta(B)$ of B is defined as $\eta(B) := \frac{h-h^*}{h^*}$. Clearly, $\eta(B^*) = 0$ for every optimal schedule B^*.

In order to assess derived schedules for problems the optimal solution of which is unknown, we establish an upper bound on the relative excess for approximate schedules: if $\tilde{B}$ is optimal w.r.t. (3), $\tilde{h}$ denotes its length, and B is feasible and satisfies (2), then the fraction $\ell(B) := \frac{h-\tilde{h}}{\tilde{h}}$ is called the *relative extension* of B. Since $h^* \geq \tilde{h}$ and $\tilde{h} \leq h$, we obtain $0 \leq \eta(B) \leq \ell(\tilde{B})$, so the extension of $\tilde{B}$ bounds the excess of B. Indeed, the relative extension yields a sufficient (yet not necessary) optimality condition: if $\ell(\tilde{B}) = 0$, then $\eta(B) = 0$, so B is optimal.

5 Experimental Results and Discussion

Length Values. As mentioned before, the IP approach found weight-optimal schedules for all real-world instances. Of all algorithms tested, BASE performed

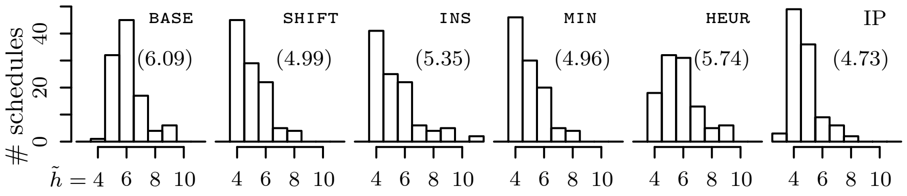


Fig. 2. Length distributions of the real-world schedules produced by the different approximation algorithms, HEUR, and IP; mean values in parentheses

worst with an average length of 6.09 for the real-world instances and was even beaten by HEUR (Fig. 2). However, the improved approximations SHIFT and INS performed considerably better than HEUR. Schedules produced by INS were on average 13.1 % longer than the optimum, and those produced by SHIFT even only 5.4 %. Of the 105 classification problems, SHIFT yielded the best result of the three approximation algorithms in 102 cases, so SHIFT and MIN basically yield the same picture in Fig. 2. The approximate schedule of minimal length, i.e. MIN, was on average only 4.8 % longer than the optimal IP schedule. Furthermore, only twelve schedules of BASE were shorter than the corresponding ones of INS, and SHIFT was beaten by BASE in one case only.

These numbers already show that BASE is not on par even with the simple HEUR, which emphasizes the demand for enhancement. Both SHIFT and INS indeed improved the basic algorithm, and the considerably shorter schedules show that the ideas behind the two improvements were actually fruitful.

Comparing HEUR to the other approaches yields a similar result for real-world instances (Tab. 1, upper block): HEUR got schedules of minimum length in 26 cases and shorter schedules than BASE for 50 problems. But, HEUR beat SHIFT and INS only three and seven times, resp., and MIN was beaten in only one case.

Surprisingly, the seemingly more sophisticated INS did not surpass SHIFT (Fig. 2 and Tab. 1). Thus, the strength of SHIFT, i.e. taking into account *resulting* column loads for the choices when merging the partial schedules, outpasses the strength of INS, i.e. the higher flexibility in distributing the columns of partial schedules to different positions in the intermediate schedule.

Table 1. Number of problems where HEUR yielded a shorter, equal, or higher length than the other approaches for real-world (upper block) and synthetic problems (lower)

HEUR	IP	BASE	SHIFT	INS	MIN
shorter	0 (0.0 %)	50 (47.6 %)	3 (2.9 %)	7 (6.7 %)	1 (1.0 %)
same	26 (24.8 %)	41 (39.0 %)	39 (37.1 %)	55 (52.4 %)	40 (38.1 %)
longer	79 (75.2 %)	14 (13.3 %)	63 (60.0 %)	43 (41.0 %)	64 (61.0 %)
shorter		130 (8.0 %)	8 (0.5 %)	60 (3.7 %)	1 (0.1 %)
same		314 (19.4 %)	118 (7.3 %)	128 (7.9 %)	111 (6.9 %)
longer		1'176 (72.6 %)	1'494 (92.2 %)	1'432 (88.4 %)	1'508 (93.1 %)

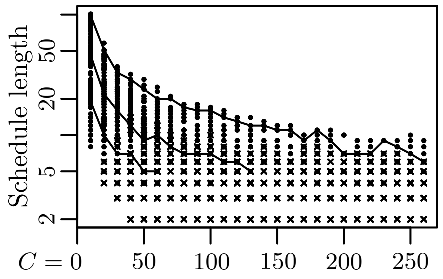


Fig. 3. Lengths of schedules produced by SHIFT as a function of capacity for the synthetic problems; $\times$: provably optimal schedules with $\ell = 0$, $\bullet$: other schedules. As an example, the three lines connect the data points for three instances with different car numbers. Schedules produced by other approximations show a similar trend.

Since a feasible schedule for a specific instance satisfying (2) for some capacity C also satisfies it for every $C' > C$, the length h of schedules should be non-increasing for growing C. This is mostly the case for the synthetic instances, yet not always: where the lines plotted in Fig. 3 increase, it would be beneficial to reuse a schedule of lower capacity instead of producing a new one.

In the case of the synthetic classification instances, the IP approach found feasible schedules for 245 of the 648 classification problems; 165 of them were weight-optimal, 49 length-optimal. The majority of feasible schedules, 153, was found for smaller problems ($n = 200$). Because of the limited number of iterations and the time limit (Sect. 4), we obtained no IP solutions for a big number of problems. Hence a direct comparison of length values of IP and approximate schedules would not be meaningful, and we compare the algorithms for the synthetic instances in terms of their relative extension in the following section.

Extension Values. With a value of 0.28, the average extension of BASE schedules for the synthetic problems was approximately the same as for the real-world schedules. In contrast, HEUR performed markedly worse than BASE on the synthetic instances (Fig. 4). The extensions of schedules produced by SHIFT and INS were half as large as for BASE, and MIN still performed slightly better. A similar picture is given by Tab. 1 for the synthetic problems: INS was beaten by HEUR in 3.7 % of the cases, SHIFT in 0.5 %, and MIN in a single case only. All in all, SHIFT and INS improved on BASE significantly for the synthetic problems as well, and HEUR was almost completely ruled out by MIN.

As explained in Sect. 4, if a schedule B has a relative extension $\ell(B) = 0$, it has optimal length w.r.t. (2). Of the three approximation algorithms and the heuristic, SHIFT produced zero-extension schedules the most frequently, followed by INS, both for the real-world and the synthetic classification problems (Tab. 2). MIN found a zero-extension schedule for more than three quarters of all real-world problems, whereas this was only the case for half of the synthetic problems. The latter value increased when restricting the analysis to classification problems of higher capacity, e.g. to 70 % for problems with $C \geq 100$.

For the best approximation algorithm, SHIFT, only 2.4 % of the schedules have a relative extension of $\ell > 0.5$ (Fig. 5); they were all generated from a relatively short intermediate schedule of length $\tilde{h} \leq 20$. On the other hand, all the 735 schedules with $\ell = 0$ are rather short ($h \leq 14$) too.

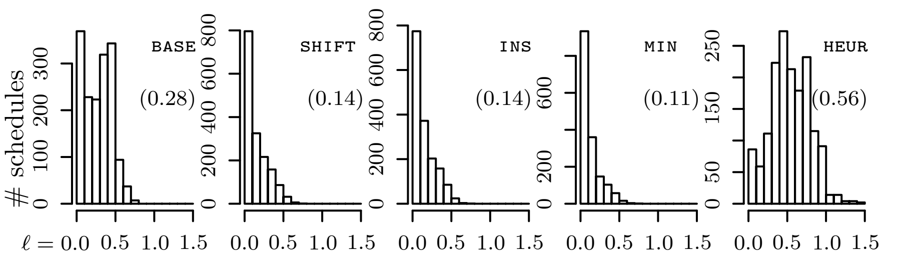


Fig. 4. Distributions of extension values for the schedules of the synthetic problems generated by the different approximation algorithms. Mean values in parentheses.

Table 2. Numbers of provably optimal schedules ($\ell = 0$) for the different algorithms

	#prob.	BASE	SHIFT	INS	MIN	HEUR
real-world	105	6 (5.7 %)	80 (76.2 %)	53 (50.5 %)	81 (77.1 %)	26 (24.8 %)
synthetic	1’620	333 (20.6 %)	735 (45.4 %)	713 (44.0 %)	810 (50.0 %)	79 (4.9 %)

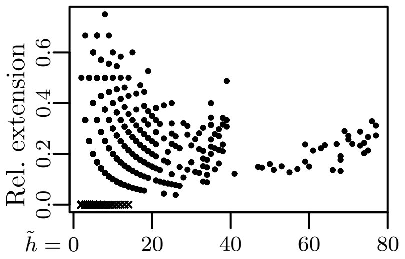


Fig. 5. Relative extension of schedules produced by SHIFT as a function of the length of the intermediate schedule $\tilde{h}$. $\times$: $\ell = 0$, $\bullet$: $\ell > 0$. Intermediate schedules of length $\tilde{h} > 40$ were produced only for classification problems with $(n, C) = (800, 10)$. The plot looks similar for the other variants of the 2-approximation.

Intermediate Schedule Lengths. With only one exception, for all real-world and synthetic problems for which we found length- or weight-optimal IP solutions, the length of the intermediate schedule of the approximation equalled the length of the corresponding IP schedule. Thus, the relative excess length corresponds to the relative extension for these problems; in particular, this means that the numbers of zero-extension real-world schedules in Tab. 2 are the *exact* numbers of optimal schedules found by the different algorithms. As mentioned in Sect. 3.2, an optimal schedule satisfying (2) is in general longer than the optimal one for (3), so the above result was rather unexpected.

The DP algorithm for the intermediate schedule assigns the same bitstring to consecutive cars of the same train unless they form a break ([9]). Hence, if the intermediate schedule violates (2) but not (3) for its length h, it usually has long sequences of consecutive cars without breaks. The IP solver, however, may very well assign different bistrings to two consecutive cars that do not form a break. Compared to the intermediate schedule, which uses between $2^{h-1}+1$ and 2^h different bitstrings, the required number of different bitstrings thus increases. But, the IP schedule length will not increase unless this number of needed bitstrings exceeds 2^h. All in all, the IP approach makes up for the stronger constraint (2) by more flexibility to assign the 2^h bitstrings available at length h.

Running Times. For the 648 real-world problems, the IP solver calculated all schedules in 103 seconds in total. This covers the 105 successful iterations but not the 392 attempts proven infeasible. The approximation algorithms and HEUR needed less than one second each for the complete set of real-world problems.

For the whole set of synthetic instances, the IP approach required more than 98 hours in total and did not find a feasible solution for 403 problems (cf. Sect. 4). The three approximation algorithms and HEUR each needed less than one minute to compute the complete set of 1'620 synthetic schedules.

6 Conclusion

We developed, implemented, and evaluated three new algorithms for the railway optimization problem of train classification, including a thorough comparison with a previous algorithm and an exact integer program. All in all, the results of Sect. 5 show that all three approximation variants perform much better in practice than their approximation ratio suggests. Even the basic algorithm BASE from [9] quickly yielded a feasible schedule when the IP approach needed a big number of iterations. However, the quality of BASE is not acceptable with an excess of 30.2 % for the real-world instances, and applying this involved algorithm is not justified since our new, much simpler heuristic HEUR rarely yielded longer schedules. This motivates refining BASE, and our heuristic improvements in 3.3 turned out to be very fruitful: as the best algorithm, SHIFT yielded an average excess of only 5.4 % over the optimum for the real-world instances, which was even better than the 13.1 % of the seemingly more sophisticated INS. Moreover, analyzing the approximation technique in Sect. 4 yielded an interesting optimality condition inherent in all three approximations, which applied to 76.2 % of the real-world problems for SHIFT. Summarizing, our new approximation algorithms present a fast approach to deriving capacity-restricted classification schedules with a highly competitive solution quality.

Future Work. First of all, the tests with the synthetic instances suggest an improvement also for higher volumes of traffic, so it would be interesting to verify the results of Sect. 5 for larger real-world instances. Second, an exact algorithm might improve over the approximation through greater freedom to assign bitstrings to cars and may still yield acceptable running times for small real-world instances; an improvement based on same idea might also be achieved by a heuristic obtained by modifying the first phase of the approximation of Sect. 3.2. We used the IP approach of [12] to asses the algorithms of Sect. 3.2, but the IP approach may also be improved by the approximation: for an intermediate schedule of length $\tilde{h}$ and weight w, the value of w is a sophisticated lower bound for the IP objective of a solution attempt corresponding to $\tilde{h}$. Since the IP solver usually found solutions of length $\tilde{h}$ for the basic IP model (Sect. 5), this lower bound may significantly accelerate finding weight-optimal IP solutions. Finally, as mentioned in [12], the approaches presented in Sect. 3 are based on complete knowledge of the order of the inbound car sequence. As trains may be delayed, the actual order may differ from the expected order and invalidate a computed schedule, which requires algorithms that are robust against realistic scenarios of delay [11]. For the special case of unrestricted track capacities, a first such attempt was made in [1].

Acknowledgments. Many thanks to S. Leber (SBB Infrastruktur) for the valuable traffic data and to P. Widmayer for his support at several project stages.

References

1. Cicerone, S., D'Angelo, G., Di Stefano, G., Frigioni, D., Navarra, A., Schachtebeck, M., Schöbel, A.: Recoverable robustness in shunting and timetabling. In: Ahuja, R., Möhring, R., Zaroliagis, C. (eds.) Robust and Online Large-Scale Optimization. LNCS, vol. 5868, pp. 28–60. Springer, Heidelberg (2009)
2. Daganzo, C.F., Dowling, R.G., Hall, R.W.: Railroad classification yard throughput: The case of multistage triangular sorting. Transp. Res. 17A(2), 95–106 (1983)
3. Dahlhaus, E., Manne, F., Miller, M., Ryan, J.: Algorithms for combinatorial problems related to train marshalling. In: Proc. of the 11th Australasian Workshop on Combinatorial Algorithms (AWOCA 2000), pp. 7–16 (2000)
4. Di Stefano, G., Maue, J., Modelski, M., Navarra, A., Nunkesser, M., van den Broek, J.: Models for rearranging train cars. Tech. Rep. TR-0089, ARRIVAL (2007)
5. Flandorffer, H.: Vereinfachte Güterzugbildung. ETR RT 13, 114–118 (1953)
6. Gatto, M., Maue, J., Mihalak, M., Widmayer, P.: Shunting for dummies: An introductory algorithmic survey. In: Ahuja, R., Möhring, R., Zaroliagis, C. (eds.) Robust and Online Large-Scale Optimization. LNCS, vol. 5868, pp. 310–337. Springer, Heidelberg (2009)
7. Hansmann, R.S., Zimmermann, U.T.: Optimal sorting of rolling stock at hump yards. In: Mathematics - Key Technology for the Future: Joint Projects Between Universities and Industry, pp. 189–203. Springer, Heidelberg (2007)
8. Jacob, R., Márton, P., Maue, J., Nunkesser, M.: Multistage methods for freight train classification. In: Liebchen, C., Ahuja, R.K., Mesa, J.A. (eds.) ATMOS 2007, pp. 158–174. IBFI Schloss Dagstuhl, Wadern (2007)
9. Jacob, R., Márton, P., Maue, J., Nunkesser, M.: Multistage methods for freight train classification. NETWORKS—Special Issue: Optimization in Scheduled Transportation Networks (to appear, 2010)
10. Krell, K.: Grundgedanken des Simultanverfahrens. ETR RT 22, 15–23 (1962)
11. Liebchen, C., Lübbecke, M.E., Möhring, R.H., Stiller, S.: The concept of recoverable robustness, linear programming recovery, and railway applications. In: Ahuja, R., Möhring, R., Zaroliagis, C. (eds.) Robust and Online Large-Scale Optimization. LNCS, vol. 5868, pp. 1–27. Springer, Heidelberg (2009)
12. Márton, P., Maue, J., Nunkesser, M.: An improved classification procedure for the hump yard Lausanne Triage. In: Clausen, J., Di Stefano, G. (eds.) ATMOS 2009. IBFI Schloss Dagstuhl, Wadern (2009)
13. Maue, J., Nunkesser, M.: Evaluation of computational methods for freight train classification schedules. Tech. Rep. TR-0184, ARRIVAL (2009)
14. Pentinga, K.: Teaching simultaneous marshalling. Railway Gaz, pp. 590–593 (1959)
15. Siddiqee, M.W.: Investigation of sorting and train formation schemes for a railroad hump yard. In: Proc. of the 5th Int. Symposium on the Theory of Traffic Flow and Transportation, pp. 377–387 (1972)

Time-Dependent Contraction Hierarchies and Approximation[⋆]

Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
{batz,geisberger,sanders}@kit.edu

Abstract. Time-dependent Contraction Hierarchies provide fast and exact route planning for time-dependent large scale road networks but need lots of space. We solve this problem by the careful use of approximations of piecewise linear functions. This way we need about an order of magnitude less space while preserving exactness and accepting only a little slow down. Moreover, we use these approximations to compute an exact travel time profile for an entire day very efficiently. In a German road network, e.g., we compute exact time-dependent routes in less than 2 ms. Exact travel time profiles need about 33 ms and about 3 ms suffice for an inexact travel time profile that is just 1 % away from the exact result. In particular, time-dependent routing and travel time profiles are now within easy reach of web servers with massive request traffic.

1 Introduction

In recent years, there has been considerable work on routing in road networks (see [1] for an overview). For the special case of constant edge weights (usually highly correlated with travel time) it is now possible to compute optimal paths orders of magnitude faster than with Dijkstra's algorithm. Such algorithms are now in wide-spread use in server based route-planning systems. There, 10 ms query time are acceptable but decreasing this to 1 ms is still desirable.

Recently, the more realistic *time-dependent* edge weights, which can model congestions during rush-hour and similar effects, have gained considerable interest. For example, time-dependent contraction hierarchies [2,3] can compute optimal earliest arrival[1] (EA) routes in a German road network with midweek-traffic in about a millisecond. However, it requires a lot of space – too much for current low cost servers. Moreover, in a time-dependent setting, we may not only be interested in the best route for a given departure time, but also in a travel time profile over a long interval of potential departure times, e.g., in order to choose a good departure time. We are not aware of previous solutions that allow such profile queries in time suitable for current systems. In this paper, we address both issues: the *reduction of space requirements* and the *efficient computation of*

⋆ Partially supported by DFG grant SA 933/5-1 and the 'Concept for the Future' of KIT within the framework of the German Excellence Initiative.

1 For start, destination, and departure time compute the earliest possible arrival time.

travel time profiles. It turns out that the key to the solution of *both* problems is to approximate the piecewise linear functions used to describe time-dependent edge weights. Interestingly, this can be done without sacrificing exactness.

Our Contributions in More Detail. Time-dependent contraction hierarchies (TCHs) make intense use of *shortcut* edges. The time-dependent edge weights of the shortcuts contain lots of redundant information. This is where we attack.

We reduce the memory usage of TCHs greatly while accepting only a moderate slowdown of the runtime for the EA problem. Although we (partly) use approximated data, the result of our computation is still exact. The main idea behind this is that shortcuts get *approximated* and non-shortcuts get *exact* time-dependent edge weights. A bidirectional search in such an *approximated TCH* (ATCH) then yields a *corridor* of shortcuts. After unpacking these shortcuts, we can perform a time-dependent search in the *unpacked corridor* (Section 3.1).

TCHs can be used to compute exact travel time profiles in a straightforward but expensive way. However, computing a corridor of shortcuts based on upper and lower bounds first brings better runtimes. Still, the result of our computation remains exact (Section 3.2). But exact computations of this kind are also possible with the space saving ATCHs: We again compute a corridor of shortcuts based on upper and lower bounds. Now, we unpack this corridor completely. However, a profile search in the unpacked corridor is straightforward but slow. Performing a *corridor contraction* instead yields *very good* performance (Section 3.3).

Our techniques provide an accuracy that may not be necessary at all in practice. Using *solely* approximated edge weights yields only a small error but saves lots of memory and provides nearly full runtime performance for the EA problem. Moreover, accepting a small error when computing travel time profiles we get a great speedup compared to the exact computation (Section 3.4).

We have implemented all of the above techniques and performed several experiments to support our claims (see Section 4).

More Related Work. Contraction hierarchies [4] are the basis of TCHs. Time-dependent route planning itself started with classical results (e.g. [5]) showing that a generalization of Dijkstra's unidirectional algorithm works for time-dependent networks and that a small modification yields a (fairly expensive) means of profile search. Some TomTom car navigation systems allow a kind of time-dependent routing. However, the method used is unpublished and probably not able to guarantee optimal routes. A successful approach to fast EA routing is to combine a simpler form of contraction with goal directed techniques [6,7,8]. In particular, a combination with the arc flag technique (TD-SHARC [7]) yields good speedups, yet has problems when time-dependence is strong – then, either preprocessing becomes prohibitive or loses so much precision that query times get fairly high. However, for *inexact* EA queries it runs very fast (though preprocessing takes fairly long) [7,8]. A combination with landmark A^* (ALT) works surprisingly well (TD-CALT [6]). We take this as an indication, that a combination with ALT could further improve the performance of TCHs.

2 Preliminaries

2.1 Time-Dependent Road Networks

Given a directed graph $G = (V, E)$ representing a road network[2]. Each edge $(u, v) \in E$ has a function $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ assigned as edge weight. This function f specifies the time $f(\tau)$ needed to reach v from u via edge (u, v) when starting at *departure time* τ. Such edge weights are called *travel time functions* (TTFs).

In road networks we usually do not arrive earlier when we start later. This is reflected by the fact, that all TTFs f fulfill the *FIFO-property*: $\forall \tau' > \tau : \tau' + f(\tau') \geq \tau + f(\tau)$. In this work all TTFs are piecewise linear functions.[3] With $|f|$ we denote the *complexity* (i.e., the number of points) of f.

For TTFs we need the three operations: (1) *Evaluation*: Given a TTF f and a departure time τ we want to compute $f(\tau)$. Using a bucket structure this runs in constant average time. (2) *Linking*: Given two adjacent edges $(u, v), (v, w)$ with TTFs f, g we want to compute the TTF of the whole path $\langle u \to_f v \to_g w\rangle$. This is the TTF $g * f : \tau \mapsto g(f(\tau) + \tau) + f(\tau)$ (meaning g "after" f). It can be computed in $\mathrm{O}(|f| + |g|)$ time and $|g * f| \in \mathrm{O}(|f| + |g|)$ holds. Linking is an associative operation, i.e., $f * (g * h) = (f * g) * h$ for TTFs f, g, h. (3) *Minimum*: Given two parallel edges e, e' from u to v with TTFs f, f', we want to *merge* these edges into one while preserving all shortest paths. The resulting single edge e'' from u to v gets the TTF $\min(f, f')$ defined by $\tau \mapsto \min\{f(\tau), f'(\tau)\}$. It can be computed in $\mathrm{O}(|f| + |f'|)$ time and $|\min(f, f')| \in \mathrm{O}(|f| + |f'|)$ holds.

In a time-dependent road network, shortest paths depend on the departure time. For fixed start and destination nodes s and t and different departure times there might be different shortest paths with different arrival times. The minimal travel times from s to t for all departure times τ also form a TTF which we call the *travel time profile* (TTP) from s to t. Each TTF f implicitly defines an *arrival time function* $\mathrm{arr}f : \tau \mapsto f(\tau) + \tau$ that yields the arrival time for a given departure time. Analogously, the *departure time function* $\mathrm{dep}f := (\mathrm{arr}f)^{-1}$ yields the departure time for a given arrival time – provided that $\mathrm{arr}f$ is a one-to-one mapping. Otherwise, $\mathrm{dep}f(\tau)$ is the set of *possible* departure times.

2.2 Algorithmic Ingredients

In addition to TCHs three known modifications of Dijkstra's algorithm are crucial for this work: *Time-dependent Dijkstra* and and *profile search* are well known. *Interval search* has already been used in the precomputation of TCHs [2].

Time-Dependent Dijkstra. The time-dependent version of Dijkstra's algorithm solves the EA problem. It works exactly like the original except for the relaxation of edges (u, v) with TTFs f_{uv}. Let the label of node u be $d_s(u)$. The old label $d_s(v)$ of the node v is updated by $\min\{d_s(v), \mathrm{arr}f_{uv}(d_s(u))\}$. The initial node label of the start node is the departure time instead of 0.

[2] Nodes represent junctions and edges represent road segments.

[3] Here, all TTFs have period 24 h. Using non-periodic TTFs makes no real difference.

Profile Search. A label correcting modification of Dijkstra's algorithm. It computes the TTPs of all reached nodes for a given start node. Thus, node labels are TTPs. The initial node label of the start node is the TTP which is constant 0. We relax an edge (u, v) with TTF f_{uv} as follows: If f_u is the label of node u, we update the label f_v of node v by computing the minimum TTP $\min(f_v, f_{uv} * f_u)$.

Interval Search. Profile search is a very expensive algorithm. *Interval search* runs much faster with a runtime similar to Dijkstra's algorithm. Instead of TTPs it computes intervals containing all possible arrival times. So, the labels are intervals $[a, b] \subset \mathbb{R}_{\geq 0}$. The initial label of the start node is $[0, 0]$. We relax an edge (u, v) with TTF f_{uv} as follows: If $[a_u, b_u]$ is the label of node u, we update the label $[a_v, b_v]$ of node v with $[\min\{a_v, a_u + \min f_{uv}\}, \min\{b_v, b_u + \max f_{uv}\}]$.

Corridors. Given a start node s and a destination node t. A subgraph C of G containing s and t where t is reachable from s is a *corridor*. Corridors help to speed up profile searches very much: The expensive profile search is performed only in the previously computed corridor (as applied very successfully in the precomputation of TCHs [2]). Corridors can also be used to enable exact computations in the presence of approximated data (see Section 3).

TCHs. In a *time-dependent contraction hierarchy* [2] all nodes of G are *ordered* by increasing *importance*. The TCH (as a structure) is constructed by *contracting* the nodes in the above order. Contracting a node v means removing v from the graph without changing shortest path distances between the remaining (more important) nodes. The shortest path distances are preserved by introducing *shortcut edges* when necessary. This way we construct the next higher *level* of the hierarchy from the current one. The node ordering and the construction of the TCH are performed in a precomputation.

EA Queries on TCHs. To answer *EA queries* given by users we first perform a bidirectional search in the TCH. The forward search is a time-dependent Dijkstra, the backward search is an interval search. Both searches go *upward* – meaning that only edges leading to more important nodes are used. The meeting points of the searches are called *candidate* nodes. The final step is the *downward search*: a time-dependent Dijkstra that only uses edges touched by the backward search. It starts from the candidate nodes where the arrival times computed by the forward search are used as initial node labels. During the bidirectional search we perform *stall-on-demand* [4,2]: The search stops at nodes when we already found a better route coming from a higher level.

Unpacking Time-Dependent Shortcuts. A TCH contains two kinds of edges: *shortcut edges* representing paths $\langle u \to v \to w \rangle$ and *original edges* stemming from the original road network. Usually a shortest path computed by TCHs contains shortcuts which have to be *unpacked*. As the path represented by a shortcut may again contain shortcuts, we do this in a recursive manner. In time-dependent case a shortcut might represent different paths for different departure times. For some departure times it might even represent an original edge.

3 Applying Approximation

Approximation helps to save memory and to speed up computations. To save memory we use an approximated version of the TCH structure.

Approximated TCHs. An *approximated TCH* (ATCH) with relative error $\varepsilon \in [0,1]$ arises from a given TCH as follows: For all edges that represent an original edge for at least one departure time nothing happens. For all other edges the TTF f is replaced by an *upper bound* $f^{\uparrow}$ with $\forall\tau : f(\tau) \leq f^{\uparrow}(\tau) \leq (1+\varepsilon)f(\tau)$. Implicitly, $f^{\uparrow}$ also represents a *lower bound* $f^{\downarrow} : \tau \mapsto f^{\uparrow}(\tau)/(1+\varepsilon)$. For edges e with *exact* TTF f_e we have $f_e^{\uparrow} = f_e^{\downarrow} = f_e$. Usually $|f^{\uparrow}|$ is considerably smaller than $|f|$. Thus, an ATCH needs considerably less memory than the respective TCH (see Section 4). To compute $f^{\uparrow}$ from an exact TTF f we use an implementation (see Neubauer [9]) of an efficient geometric algorithm described by Imai and Iri [10]. It yields an $f^{\uparrow}$ of minimal $|f^{\uparrow}|$ for ε in time $\mathrm{O}(|f|)$.

Min-Max-TCHs. An extreme case of an approximated TCH is a *Min-Max-TCH.* For a shortcut, that never represents an original edge for any departure time, we only store the pair of numbers $(\min f, \max f)$ instead of the TTF f. Min-Max-TCHs need even less memory than ATCHs (see Section 4).

3.1 Exact Earliest Arrival Queries with Approximated TCHs

Our method for exact EA queries with ATCHs uses the two following new algorithmic ingredients. For their correctness the FIFO-property is required.

Arrival Interval Search. Is similar to interval search and computes an approximated solution of the EA problem, i.e., an interval containing the exact value, which is the best we can do for ATCHs. We relax an edge (u,v) with upper bound $f_{uv}^{\uparrow}$ and lower bound $f_{uv}^{\downarrow}$ as follows: If $[a_u, b_u]$ is the label of u, we update the label $[a_v, b_v]$ of v with $[\min\{a_v, a\}, \min\{b_v, b\}]$ where $[a,b] := [\mathrm{arr} f_{uv}^{\downarrow}(a_u), \mathrm{arr} f_{uv}^{\uparrow}(b_u)]$. The initial label of the start node is $[\tau_0, \tau_0]$ for the departure time τ_0.

Backward Travel Time Interval Search. Is dual to arrival interval search. Given a destination node t and an interval $[\sigma, \sigma']$ it computes intervals containing the possible times needed for traveling to t if the arrival time lies in $[\sigma, \sigma']$. The algorithm runs *backward* starting from t with $[0,0]$ as initial node label. Consider the (backward) relaxation of an edge (u,v) with upper bound $f_{uv}^{\uparrow}$ and lower bound $f_{uv}^{\downarrow}$. Let the label of node v be $[p_v, q_v]$. The old label $[p_u, q_u]$ of node u is updated with $[\min\{p_u, p\}, \min\{q_u, q\}]$ where $[p,q] := [p_v + \min f^{\downarrow}|_I, q_v + \max f^{\uparrow}|_I]$ and $I := [\max \mathrm{dep} f_{uv}^{\uparrow}(\sigma - q_v), \min \mathrm{dep} f_{uv}^{\downarrow}(\sigma' - p_v)]$.

Queries. Having defined all necessary ingredients, we are able to specify an algorithm for *exact* EA queries on ATCHs and Min-Max-TCHs: Given a start node s, a destination node t, and a departure time τ_0 proceed as follows:

– *Phase 1 (bidirectional upward search).* Perform a bidirectional search using solely *upward* edges with stall-on-demand. The forward search is an arrival

interval search from s with initial label $[\tau_0, \tau_0]$ that computes intervals containing *arrival times*. The backward search is an interval search from t with initial label $[0, 0]$ that computes intervals containing *travel times*. The meeting points of the searches are *candidate* nodes. For a candidate node c with forward label $[\tau, \tau']$ and backward label $[\sigma, \sigma']$ the interval $[\tau + \sigma, \tau' + \sigma']$ contains an arrival time for traveling from s to t via c for departure time τ_0.
- *Phase 2 (forward/downward search).* Perform a forward arrival interval search starting from the candidates, that only uses edges touched by the backward search of Phase 1. The initial node labels of the candidates are the arrival time intervals computed by the forward search in Phase 1.
- *Phase 3 (backward/upward search).* Phase 2 yields an interval $[a_t, b_t]$ containing the EA time for t. Now, we perform a backward travel time interval search starting from t with initial label $[a_t, b_t]$. The search runs *backward* and uses only *upward* edges touched by Phase 2. When we reach a node, that has also been reached by the forward search of Phase 1, the node is again a candidate.
- *Phase 4 (unpacking and Dijkstra).* From the candidates provided by Phase 3 perform a forward and a backward BFS on the edges touched by Phases 3 and 1 respectively. Unpacking all shortcuts touched by these BFSs yields a corridor C whose edges have only exact TTFs. A time-dependent Dijkstra in C from s with departure time τ_0 yields the sought-after exact arrival time.

As C contains rather few edges, the time-dependent Dijkstra in Phase 4 does not need much time. As a result, the runtimes are only moderately worse than with exact TCHs. Note, that we could perform Phase 4 directly after Phase 1. But the Phases 2 to 3 help to reduce the candidate set and thus the corridor. An improvement to Phase 4 is to unpack the shortcuts only when they are needed by the time-dependent Dijkstra – this is to say “on demand”.

3.2 Exact Profile Queries with Exact TCHs

Computing TTPs using exact TCHs is straightforward: For a start node s and a destination node t just perform a bidirectional profile search, that only uses upward edges while using stall-on-demand based on global minima and maxima. Again, the meeting points of the bidirectional search are *candidate* nodes, each of them representing a TTP (though not necessary an optimal one). Now, merge all these TTPs using the minimum operation, which yields the sought-after TTP.

As this is quite time consuming, we propose a great improvement: Perform a bidirectional upward interval search first. Again, the meeting points are candidate nodes. Similar to Phase 4 in Section 3.1 perform a forward and a backward BFS starting from the candidates – but do *not* unpack the shortcuts (all edges have exact TTFs this time). Now, perform the bidirectional upward profile search only in the resulting corridor C, which makes the search strongly directed. Again, merge the candidate TTPs, which yields the sought-after TTP.

3.3 Exact Profile Queries with ATCHs

Exact profile queries can also be answered using ATCHs. This is possible by adapting the method described in Section 3.2 in a straightforward way: Unpack all shortcuts in the corridor C and perform profile search in the resulting corridor C'. However, this takes some time. The reason is, that C' usually contains many more edges than C. During a profile search the points of the TTFs of these edges are processed again and again and again. Assume, for example, that C' is a path of ℓ edges and that all TTFs have k points. Then, a profile search in C' processes $\Theta(k\ell^2)$ points in the worst case. Here, we propose *corridor contraction* – a much faster algorithm reducing this to $\Theta(k\ell \log \ell)$ points. It exploits the fact that linking is an *associative* operation on TTFs, which enables us to alter the order of link operations without altering the result.

Corridor Contraction. Our algorithm uses a priority queue (PQ) to control the order of performed operations. The elements of the PQ are nodes. As key we use the estimated effort needed to contract each node. First, we insert all nodes in C' except for s, t into the PQ. Then, we *contract* C': While the PQ is not empty, we delete the minimal node v from the PQ and contract it completely. That is, for all paths $\langle u \rightarrow_f v \rightarrow_g w \rangle$ we add an edge (u, w) to C' with TTF $h := g * f$ and remove all edges incident to v from C'. Also we update the keys of u and w. If an edge (u, w) already exists in C', we merge its TTF with h. After termination only an edge (s, t) is left in C'. Its TTF is the sought-after TTP.

As an optimization we thin out C' by a preceding bidirectional approximate upward profile search which computes approximate TTPs that are upper and lower bounds of the exact ones. This makes our method even faster than the one in Section 3.2. However, for Min-Max-TCHs this is not possible of course.

3.4 Inexact Earliest Arrival and Profile Queries

In practice exact results may be not necessary, or the accuracy of the TTFs may be arguable. In such cases, small errors are allowed and all computations can be performed using an *inexact TCH*, which can be obtained from an exact TCH by replacing every TTF f by an inexact TTF $f^{\updownarrow}$ with $(1+\varepsilon)^{-1} f(\tau) \leq f^{\updownarrow}(\tau) \leq (1+\varepsilon) f(\tau)$. Additionally, we store the *conservative bounds* $\min\{\min f, \min f^{\updownarrow}\}$ and $\max\{\max f, \max f^{\updownarrow}\}$ with every edge. Inexact TCHs save lots of memory. But when we compute TTPs they even gain an enormous speedup. This is because the processed TTFs have much less points.

To preserve correctness we always perform a preceding bidirectional upward interval search that employs stall-on-demand using only the conservative bounds. All further passes of any query avoid stall-on-demand and only relax edges touched by this initial phase. For EA queries we have two further passes: the forward and the downward search as described in Section 2.2 (without backward search this time). In this way we get a corridor-based variant of the original TCH query which also works with exact TCHs. For profile queries the only further pass is a bidirectional upward profile search. So, we actually have the method from Section 3.2, but this time applied to inexact TCHs.

4 Experiments

Inputs and Setup. As inputs we use two road networks of Germany and Western Europe, both provided by PTV AG for scientific use. Germany has 4.7 million nodes, 10.8 million edges, and time-dependent edge weights reflecting the midweek (Tuesday till Thursday) traffic collected from historical data, i.e., a high traffic scenario with about 8 % time dependent edges. Western Europe has about 18 million nodes and 42.6 million edges. It has been augmented with synthetic time-dependent travel times as in [11] using a high amount of traffic where all edges but local and rural roads have time-dependent edge weights.

The experimental evaluation was done on a machine with four Core i7 Quad-Cores (2.67 Ghz) with 48 GiB of RAM running SUSE Linux 11.1. All programs were compiled by GCC 4.3.2 with optimization level 3. Running times were always measured using one single thread. All figures refer to the scenario that only the EA times and the TTPs have to be determined, without outputting complete path descriptions. However, when reporting memory consumption, we include the space needed to allow fast path reporting. The memory usage is given in terms of the average *total* space usage of a node (not the overhead) in byte per node. We also report the growth factor of the memory usage compared to the *original graph*, i.e., the graph used for time-dependent Dijkstra. For Germany this graph needs 95 byte per node, for Europe 76 byte per node.

We measured the average performance of EA and profile queries for 1 000 randomly selected start and destination pairs. For EA queries the departure time is randomly selected from $[0\text{h}, 24\text{h})$ each. To measure the errors we used many more test cases: 1 000 000 EA queries and 10 000 profile queries, where the error of profile queries was measured for 100 random departure times each.

We also measured the machine-independent behaviour of our algorithms: In all cases we count the number of deleteMin-Operations and touched edges (which is identical to the number of relaxed edges for time-dependent Dijkstra). For EA queries we also count how often TTFs are evaluated (including similar operations like, e.g., computing $\max \operatorname{dep} f^{\uparrow}_{uv}(\sigma - q_v)$ in Section 3.1). For profile queries we count the points of the TTFs processed by link and minimum operations.

Results. Table 1 evaluates EA queries for different approaches. Inexact TCHs (Section 3.4) save space and are at most 2 times slower than exact TCHs. For Germany there is no slow down for small ε. The maximum errors are small for small ε, the average errors are even smaller. However, in theory one can easily construct inputs where errors could get much larger than ε. ATCHs (Section 3.1) run slower by a factor of 1.5–3.3 for Germany and 1.2–8.2 for Europe. However, ATCHs save memory: 3.2–8.4 times less space than exact TCHs for Germany and 2.3–6.0 for Europe. Note, that the speed difference would further decrease when shortest paths were to be computed since this is done anyway for ATCHs.

For profile queries look at Table 2. Exact TCHs, using the straightforward method (Section 3.2), take about 1.1 s for Germany and 4.2 s for Europe – far too slow for a server scenario. Restricting profile search to a corridor (also Section 3.2) helps, but ATCHs with corridor contraction (Section 3.3) work

Table 1. Behaviour of EA queries using different methods. ATCH with $\varepsilon = \infty$ denotes Min-Max-TCHs. TCH with $\varepsilon \neq 0$ denotes inexact queries on inexact TCHs, cor.= corridor, UoD= shortcut **U**npacking-**o**n-**D**emand, SPD= speedup of time-dependent Dijkstra, GRO= growth of space usage compared to the original graph, MAX and AVG are maximum and average relative errors.

	ε	space		time		delMin		edges		evals		error [%]	
method	[%]	[B/n]	GRO	[ms]	SPD	#	SPD	#	SPD	#	SPD	MAX	AVG
Germany midweek													
TCH	–	994	10.4	0.72	1 440	520	4 616	5 813	951	1 269	162	0.00	0.00
TCH (cor.)	0.0	994	10.4	0.74	1 401	639	3 756	7 092	780	76	2 704	0.00	0.00
	0.1	286	3.0	0.71	1 460	642	3 739	7 128	770	77	2 669	0.10	0.02
	1.0	214	2.3	0.72	1 440	654	3 670	7 262	762	84	2 446	1.01	0.27
	10.0	113	1.2	1.03	1 006	897	2 676	10 096	548	223	921	9.75	3.84
ATCH (UoD)	0.1	308	3.2	1.10	942	554	4 332	7 734	715	3 080	67	0.00	0.00
	1.0	239	2.5	1.27	816	582	4 124	8 338	664	3 347	61	0.00	0.00
	10.0	163	1.7	2.40	432	824	2 913	21 036	263	7 486	27	0.00	0.00
	∞	118	1.2	1.45	714	698	3 439	20 116	275	3 153	65	0.00	0.00
Europe high traffic													
TCH	–	589	7.8	1.89	1 807	986	9 161	13 003	1 665	2 370	289	0.00	0.00
TCH (cor.)	0.0	589	7.8	3.19	1 071	1 653	5 464	23 031	929	1 412	484	0.00	0.00
	0.1	237	3.1	3.67	931	1 661	5 438	23 142	924	1 427	479	0.14	0.02
	1.0	193	2.5	2.85	1 199	1 716	5 264	24 036	890	1 544	443	1.46	0.20
	10.0	143	1.9	2.68	1 275	1 726	5 233	24 221	883	1 583	432	15.34	2.85
ATCH (UoD)	0.1	256	3.4	2.25	1 518	1 032	8 752	17 894	1 195	5 382	127	0.00	0.00
	1.0	207	2.7	2.47	1 396	1 104	8 152	22 683	943	6 362	108	0.00	0.00
	10.0	164	2.2	7.37	463	1 771	5 100	137 221	156	23 949	29	0.00	0.00
	∞	99	1.3	15.43	221	2 196	4 113	448 360	48	42 939	16	0.00	0.00

better. For Germany Min-Max-TCHs also work well. For Europe they do not show acceptable running times. However, for really fast profile queries we need inexact TCHs. This works especially well for Germany.

Figure 1 shows the distribution of running times of profile queries on Germany: For $i = 5..22$ we look at 100 queries with the property that Dijkstra's Algorithm settles 2^i nodes (2^i is called the *Dijkstra rank*). A profile query in a middle-sized German town (rank 2^{12}) needs less than 1 ms on ATCHs with corridor contraction. Plain profile search (Section 2.2) is much slower. We stopped when the average running time exceeded 10 s. In Section 3.3 we claim that corridor contraction brings a considerable additional speedup. Indeed, when we replaced corridor contraction by a profile search in the precomputed corridor it ran considerably slower (also Figure 1).

For the comparison with (the best) other techniques look at Table 3. For EA queries we only compare speedups of time-dependent Dijkstra – absolute query times would be unreliable as different machines are used. As plain profile search takes too long, we are not able to report speedups for profile queries. Instead, we also compare the running time of profile queries with time-dependent Dijkstra. This way we get a *relative speed*. As our preprocessing works in two phases (*node*

Table 2. Profile queries using different methods. CC= corridor contraction, the rest of the nomenclature is the same as in Table 1.

method	ε [%]	space [B/n]	space GRO	time [ms]	delMin #	edges #	points #	error [%] MAX	error [%] AVG
Germany midweek									
TCH	–	994	10.4	1 112.02	570	6 796	20 623 155	0.00	0.00
TCH (cor.)	0.0	994	10.4	88.87	646	7 170	1 437 892	0.00	0.00
	0.1	286	3.0	6.13	650	7 208	86 391	0.10	0.02
	1.0	214	2.3	2.94	662	7 348	35 769	1.03	0.27
	10.0	113	1.2	2.48	923	10 361	23 010	9.69	3.84
ATCH (CC)	0.1	308	3.2	36.22	650	29 551	576 099	0.00	0.00
	1.0	239	2.5	32.75	675	32 131	531 795	0.00	0.00
	10.0	163	1.7	105.45	889	92 740	1 731 359	0.00	0.00
	∞	118	1.2	76.58	578	59 368	1 278 095	0.00	0.00
Europe high traffic									
TCH	–	589	7.8	4182.43	1 090	17 234	70 937 950	0.00	0.00
TCH (cor.)	0.0	589	7.8	2 016.86	1 797	25 486	30 734 960	0.00	0.00
	0.1	237	3.1	198.00	1 813	25 655	3 371 555	0.13	0.02
	1.0	193	2.5	105.72	1 882	26 796	1 741 315	1.27	0.20
	10.0	143	1.9	36.75	1 889	26 977	755 646	14.65	2.85
ATCH (CC)	0.1	256	3.4	565.28	1 806	169 378	8 200 162	0.00	0.00
	1.0	207	2.7	382.12	1 887	199 551	5 448 190	0.00	0.00
	10.0	164	2.2	2 306.11	2 429	1 259 891	35 330 837	0.00	0.00

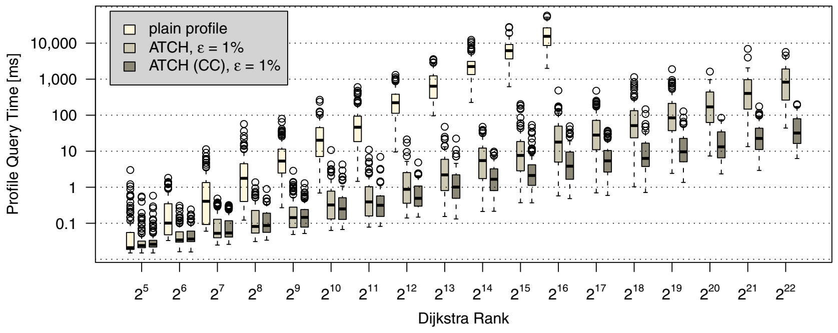


Fig. 1. Profile query times over the Dijkstra rank for different methods. CC means corridor contraction is used, otherwise a profile search in the corridor is performed.

ordering and *contraction*) there are preprocessing times like 0:28+0:09 for TCH based techniques (28 min node ordering and 9 min contraction). Node orders can be reused for different traffic scenarios, i.e., they need not to be recomputed. However, this might slow down the query time a bit [2].

For exact queries ATCHs dominate TD-SHARC [7] in all respects. However, TD-CALT [6,7] has much better preprocessing time. For Europe the advantage of TCH based techniques over TD-CALT with respect to query time becomes

Table 3. Comparison of different algorithms for exact and inexact time-dependent EA and profile (TTP) queries. Memory usage is given as overhead, errors are max. rel. errors. REL= relative speed of profile queries compared to time-dependent Dijkstra, SH= SHARC, inex= inexact, app= approximate, heu= heuristic, spc eff= space efficient.

	ε	prepro.	ovh.	EA	TTP	err.	prepro.	ovh.	EA	TTP	err.
method	[%]	[h:m]	[B/n]	SPD	REL	[%]	[h:m]	[B/n]	SPD	REL	[%]
		Germany midweek					Europe high traffic				
TCH	–	0:28+0:09	899	1 440	11.66	0.00	3:45+0:59	513	1 807	1.69	0.00
ATCH	1	0:28+0:09	144	816	31.65	0.00	3:45+0:59	131	1 396	9.94	0.00
ATCH	∞	0:28+0:09	23	714	13.49	0.00	3:45+0:59	23	221	–	0.00
CALT	–	0:09	50	280	–	0.00	1:00	61	47	–	0.00
SH	–	1:16	155	60	0.02	0.00	6:44	134	70	–	0.00
L-SH	–	1:18	219	238	–	0.00	6:49	198	150	–	0.00
inex TCH	1	0:28+0:09	119	1 440	352.59	1.03	3:45+0:59	117	1 199	32.31	1.46
inex TCH	10	0:28+0:09	18	1 006	417.99	9.75	3:45+0:59	67	1 275	92.95	15.34
app CALT	–	0:09	50	804	–	13.84	1:00	61	624	–	8.69
heu SH	–	3:26	137	2 164	1.40	0.61	22:12	127	1 958	–	1.60
heu L-SH	–	3:28	201	3 915	–	0.61	22:17	191	2 703	–	1.60
spc eff SH	–	3:48	68	1 177	–	0.61	–	–	–	–	–
spc eff SH	–	3:48	14	491	–	0.61	–	–	–	–	–

much larger. This is an indication that TCH combined with ALT will not scale well with the input size. For inexact EA queries approximate TD-CALT works much better. But again, it is outperformed by inexact TCHs except for preprocessing. Heuristic TD-SHARC has better speedups for EA queries but worse memory usage and preprocessing times than inexact TCHs. For *space efficient* TD-SHARC [8] the memory usage is very good but the speedups are worse than for inexact TCHs. Regarding profile search, TCH based techniques come off as a clear winner as they run up to three orders of magnitude faster in the exact and about 250–300 times faster in the in the inexact setting.

5 Conclusions and Future Work

We have demonstrated that using approximations of travel time functions greatly reduces the space consumption of time-dependent contraction hierarchies. By using these approximation only for obtaining a corridor of possibly useful roads, we can still obtain exact results. We have also explained how travel time profiles can be computed very efficiently – fast enough for current server systems.

The achieved space reduction by up to an order of magnitude may not be the end of the story because we can possibly come up with much more compact representations of the approximations than the piecewise linear functions currently used. It might be possible to represent the TTFs with some tabulated patterns and then just store references and scaling factors. This way (near exact) time-dependent route planning may even be possible on mobile devices.

Future work will have to allow even more realistic modelling, in particular, incorporating traffic jams, and allowing additional objective functions.

Acknowledgements. We thank Christian Vetter for important contributions to the TCH code. There were many fruitful discussions with Daniel Delling.

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. Batz, G.V., Delling, D., Sanders, P., Vetter, C.: Time-Dependent Contraction Hierarchies. In: Finocchi, I., Hershberger, J. (eds.) ALENEX 2009. SIAM, Philadelphia (2009)
3. Vetter, C.: Parallel Time-Dependent Contraction Hierarchies (2009), Student Research Project, http://algo2.iti.kit.edu/download/vetter_sa.pdf
4. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: [12], pp. 319–333
5. Orda, A., Rom, R.: Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. Journal of the ACM 37(3), 607–625 (1990)
6. Delling, D., Nannicini, G.: Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 812–823. Springer, Heidelberg (2008)
7. Delling, D.: Time-Dependent SHARC-Routing. Algorithmica (2009); In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 332–343. Springer, Heidelberg (2008)
8. Brunel, E., Delling, D., Gemsa, A., Wagner, D.: Space-Efficient SHARC-Routing. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 47–58. Springer, Heidelberg (2010)
9. Neubauer, S.: Space Efficient Approximation of Piecewise Linear Functions (2009), Student Research Project, http://algo2.iti.kit.edu/download/neuba_sa.pdf
10. Imai, H., Iri, M.: An optimal algorithm for approximating a piecewise linear function. Journal of Information Processing 9(3), 159–162 (1987)
11. Nannicini, G., Delling, D., Liberti, L., Schultes, D.: Bidirectional A* Search for Time-Dependent Fast Paths. In: [12], pp. 334–346
12. McGeoch, C.C. (ed.): WEA 2008. LNCS, vol. 5038. Springer, Heidelberg (2008)

A New Combinational Logic Minimization Technique with Applications to Cryptology

Joan Boyar[1,⋆] and René Peralta[2]

[1] Department of Mathematics and Computer Science,
University of Southern Denmark
joan@imada.sdu.dk

[2] Information Technology Laboratory, NIST
rene.peralta@nist.gov

Abstract. A new technique for combinational logic optimization is described. The technique is a two-step process. In the first step, the non-linearity of a circuit – as measured by the number of non-linear gates it contains – is reduced. The second step reduces the number of gates in the linear components of the already reduced circuit. The technique can be applied to arbitrary combinational logic problems, and often yields improvements even after optimization by standard methods has been performed. In this paper we show the results of our technique when applied to the S-box of the Advanced Encryption Standard (AES [6]). This is an experimental proof of concept, as opposed to a full-fledged circuit optimization effort. Nevertheless the result is, as far as we know, the circuit with the smallest gate count yet constructed for this function. We have also used the technique to improve the performance (in software) of several candidates to the Cryptographic Hash Algorithm Competition. Finally, we have experimentally verified that the second step of our technique yields significant improvements over conventional methods when applied to randomly chosen linear transformations.

Keywords: Circuit complexity, multiplicative complexity, linear component minimization, AES, S-box.

1 Introduction

Constructing optimal combinational circuits is an intractable problem under almost any meaningful metric (gate count, depth, energy consumption, etc.). In practice, no known techniques can reliably find optimal circuits for functions with as few as eight Boolean inputs and one Boolean output.

For example, the multiplicative complexity[1] of the Boolean function E_4^8, which is true if and only if exactly four of its eight input bits are true, is unknown [2]. In

⋆ Partially supported by the Danish Natural Science Research Council (FNU). Some of this work was done while visiting the University of California, Irvine.

1 The multiplicative complexity of a function is the number of GF(2) multiplications necessary and sufficient to compute it.

practice, we build circuit implementations of functions using a variety of heuristics. Many of these heuristics have exponential time complexity and thus can only be applied to small components of a circuit being built. This works reasonably well for functions that naturally decompose into repeated use of small components. Such functions include arithmetic functions (which we often build using full adders), matrix multiplication (which decomposes into multiplication of small submatrices), and more complex functions such as cryptographic functions (which are commonly based on multiple iterations of an algorithm containing linear steps and one non-linear step).

This work presents a new technique for logic synthesis and circuit optimization. The technique can be applied to arbitrary functions, and yields improvements even on programs/circuits that have already been optimized by standard methods. We apply our technique to the S-box of AES[2], which, in addition to being used in AES, has been used in several proposals for a new hash function standard[3]. The result is, as far as we know, the smallest circuit yet constructed for this function. The circuit contains 32 AND gates and 83 XOR/XNOR gates for a total of 115 gates. We have also applied these techniques to the logic embedded in the non-linear components of several candidates to the SHA-3 competition. The improvements in software performance were significant.

Our circuits are over the basis $\{\oplus, \wedge, 1\}$. This basis is logically complete: any Boolean circuit can be transformed into this form using only local replacements. The circuit operations can be viewed either as performing Boolean logic or arithmetic modulo 2 (when viewing it the latter way, we will write outputs to be computed as polynomials with multiplication replacing $\wedge$ and addition replacing $\oplus$). The number of $\wedge$ gates is called the *multiplicative complexity* of the circuit. Connected components of the circuit containing $\wedge$ gates are called *non-linear*. Components free of $\wedge$ gates are called *linear*. Circuits and programs for computing Boolean functions can be defined using straight-line programs, where each statement defines the operation of a gate or a line in a program. Consider the examples in Fig. 1, defining two different circuits for computing the majority function of three inputs, a, b, and c:

$t_1 = a \wedge b$;	$t_2 = a \oplus b$;	$t_3 = t_2 \wedge c$;	$t_4 = t_1 \oplus t_3$;
$u_1 = a \oplus b$;	$u_2 = b \oplus c$;	$u_3 = u_1 \wedge u_2$;	$u_4 = u_3 \oplus b$;

Fig. 1. Two circuit definitions for MAJ(a, b, c)

2 Combinational Circuit Optimization

The techniques described here would generally be applied to subcircuits of a larger circuit, such as an S-box in a cryptographic application, which have

[2] Our circuit for the AES S-box has already been used as the basis of a software bitsliced implementation of AES in counter mode [8].

[3] See http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

relatively few inputs and outputs connecting them to the remainder of the circuit. The key observation that led us to our techniques is that circuits with low multiplicative complexity will naturally have large sections which are purely linear (i.e. contain only $\oplus$ gates). Thus

it is plausible that a two-step process, which first reduces multiplicative complexity and then optimizes linear components, leads to small circuits.

We have, of course, no way of proving this hypothesis. But the experiments reported here support it.

First step: The first step of our technique consists of identifying non-linear components of the subcircuit to be optimized and reducing the number of $\wedge$ gates. This reduction is not easy to do. For example, it is not obvious how to algorithmically transform one of the circuits defined in Fig. 1 into the other. Finding circuits with minimum multiplicative complexity is, in all likelihood, a highly intractable problem. However, recent work on multiplicative complexity contains an arsenal of reduction techniques that in practice yield circuits with small, and often optimal, multiplicative complexity [2]. That work focuses exclusively on symmetric functions (those whose value depends only on the Hamming weight of the input). In this paper we use ad-hoc heuristics to construct a circuit with low multiplicative complexity for inversion in $GF(2^4)$. (In general, $GF(2^n)$ is the field with 2^n elements.) The technique is partially described in Section 3.

Second step: The second step of our technique consists of finding maximal linear components of the circuit and then minimizing the number of XOR gates needed to compute the target functions computed in these linear components. A new heuristic for this computationally intractable problem is described in Section 4.

3 AES's S-Box

The non-linear operation in AES's S-box is to compute an inverse in the field $GF(2^8)$. A recursive method for building a circuit for inverses in $GF(2^{mn})$, given a circuit for inverses in $GF(2^m)$, is due to Itoh and Tsujii [7]. The circuits produced by this method are said to have a *tower fields architecture*. Since there are multiple possible representations for Galois fields, several authors have concentrated on finding representations that yield efficient circuits under the tower fields architecture. We use the same general technique for the reduction from inversion in $GF(2^8)$ to $GF(2^4)$ inversion, but we use a completely different technique for computing the inversion in $GF(2^4)$. We then place the optimized circuit for $GF(2^4)$ inversion in its appropriate place in AES's S-box and apply a novel optimization technique on the linear parts of the resulting circuit.

***GF*(2⁴) inversion – A Non-Linear Component.** The tower fields architecture for inversion in $GF(2^8)$ has (non-trivial) easily identifiable non-linear components corresponding to inversion in subfields. The first step in our method is to focus on one of these components and derive a circuit that uses few $\wedge$ gates. The component for inversion in $GF(2^2)$ is too small for us to benefit significantly from optimizing it. Instead we focus on inversion in $GF(2^4)$. There are many representations of $GF(2^4)$. We construct

- $GF(2^2)$ by adjoining a root W of x^2+x+1 over $GF(2)$;
- $GF(2^4)$ by adjoining a root Z of x^2+x+W^2 over $GF(2^2)$.

Following Canright [5], we represent $GF(2^2)$ using the basis (W, W^2) and $GF(2^4)$ using the basis (Z^2, Z^8). Thus, an element $\delta \in GF(2^4)$ is written as $\delta_1 Z^2 + \delta_2 Z^8$, where $\delta_1, \delta_2 \in GF(2^2)$. Similarly, an element γ in $GF(2^2)$ is written as $\gamma_1 W + \gamma_2 W^2$, where $\gamma_1, \gamma_2 \in GF(2)$. Since Z satisfies $x^2+x+W^2=0$ and W satisfies $x^2+x+1=0$, one can calculate that $Z^4 = Z^2+W$, $Z^8 = Z^2+1$, $Z^{10} = Z^4+Z^2$, $Z^{16} = Z^8+W$, $W^3 = W^2+W$, $W^4 = W$, and $W^5 = W^2$. These equations can be used to reduce expressions to check equalities.

Using this representation, an element of $GF(2^4)$ can be written as $\Delta = (x_1W + x_2W^2)Z^2 + (x_3W + x_4W^2)Z^8$, where $x_1, x_2, x_3, x_4 \in GF(2)$. The inverse of this element, $\Delta' = (y_1W + y_2W^2)Z^2 + (y_3W + y_4W^2)Z^8$, can then be calculated using the following polynomials over $GF(2)$:

- $y_1 = x_2x_3x_4 + x_1x_3 + x_2x_3 + x_3 + x_4$
- $y_2 = x_1x_3x_4 + x_1x_3 + x_2x_3 + x_2x_4 + x_4$
- $y_3 = x_1x_2x_4 + x_1x_3 + x_1x_4 + x_1 + x_2$
- $y_4 = x_1x_2x_3 + x_1x_3 + x_1x_4 + x_2x_4 + x_2$

The fact that Δ' is the inverse of Δ can be verified by multiplying the two elements together and reducing using the equations mentioned above (along with $x^2 = x$ and $x+x = 0$). The symbolic result is $(QW+QW^2)Z^2+(QW+QW^2)Z^8$, where $Q = x_1x_2x_3x_4 + x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4 + x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4 + x_1 + x_2 + x_3 + x_4$. The fact that the value of Q is 1 unless all four variables have the value 0, when it is 0, can be seen by observing that it is the symmetric function $\Sigma_4^4 + \Sigma_3^4 + \Sigma_2^4 + \Sigma_1^4$. If exactly four variables are set, then the first term gives the value 1 (and the others 0); if three are set, then the second, third and fourth terms give the value 1; if exactly two are set, then only the third gives the value 1; and if only one is set, then only the last gives the value 1. Hence, the result is 1, except for the zero input.[4]

Thus the task at hand is to construct a circuit with four inputs and four outputs that calculates the above system of equations using as few $\wedge$ gates as possible. Currently, our heuristic search programs can handle functions with one output and up to eight inputs. This means that we can directly construct optimal circuits for each of the four equations individually, but not for the system itself. For the full system we took the following approach:

[4] A circuit for finite field inversion must have some output for the non-invertible zero element. In the following constructions we follow the AES convention that the output on input zero is zero.

- pick an equation and construct an efficient circuit for it;
- store intermediate functions computed in the previous steps for possible use in constructing a circuit for the next equation to be tackled;
- iterate until all equations have been computed.

The first step is non-trivial even for predicates on few inputs. The heuristic we used is inspired by methods from automatic theorem proving. We omit its description here due to space constraints[5]. We can report, however, that we succeeded in determining the multiplicative complexity of all 2^{16} predicates on four bits. It turns out that 3 multiplications are enough to compute any predicate on four variables.[6] This is of interest to designers of cryptographic functions since many constructions have been proposed which use 4x4 S-boxes. We have not yet been able to do the same for all predicates on 5 bits.

We performed the three steps above for each of the twenty-four orderings of $\{y_1, y_2, y_3, y_4\}$. The ordering (y_4, y_2, y_1, y_3) gave the best results. The resulting circuit, expressed as a straight-line program over GF(2), is shown in Figure 2 (outputs are indicated by an (*)).

$t_1 = x_1 + x_2$	$t_2 = x_1 \times x_3$	$t_3 = x_4 + t_2$
$t_4 = t_1 \times t_3$	$y_4 = x_2 + t_4$ $(*)$	$t_5 = x_3 + x_4$
$t_6 = x_2 + t_2$	$t_7 = t_6 \times t_5$	$y_2 = x_4 + t_7$ $(*)$
$t_8 = x_3 + y_2$	$t_9 = t_3 + y_2$	$t_{10} = x_4 \times t_9$
$y_1 = t_{10} + t_8$ $(*)$	$t_{11} = t_3 + t_{10}$	$t_{12} = y_4 \times t_{11}$
$y_3 = t_{12} + t_1$ $(*)$		

Fig. 2. Inversion in $GF(2^4)$

This circuit contains 5 $\wedge$ gates and 11 $\oplus$ gates. It is a significant improvement over previous constructions, e.g. Paar's construction [10] has a gate count of 10 $\wedge$ gates and 15 $\oplus$ gates for the same function. It is harder to compare to Canright's construction [5]. In his original, he had 9 $\wedge$ gates (and NAND gates) and 14 $\oplus$ gates (and XNOR gates), but he optimized, allowing NOR gates. After this, he had 8 NAND gates, 2 NOR gates, and 9 XOR/XNOR gates.

The multiplicative complexity of a function is the number of GF(2) multiplications necessary and sufficient to compute it. Under the given representation for $GF(2^4)$, the multiplicative complexity of inversion is 5. This can be argued as follows: the upper bound is given by the construction. The four outputs that have to be computed all have degree 3. One $\wedge$ is needed to compute a polynomial of degree 2. Then, an additional $\wedge$ is necessary to produce each of the four linearly independent polynomials, since each is of degree 3.

[5] A description can be found in the patent application by NIST and the University of Southern Denmark ([4]).

[6] Lest the reader think this trivial, he/she may attempt to compute the function $f(x_1, x_2, x_3, x_4) = x_1x_2x_3x_4 + x_1x_2x_3 + x_1x_2x_4 + x_2x_3x_4 + x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_3x_4$ using only three multiplications.

A View of the Structure of AES's S-Box. In the previous section, using the tower fields architecture, we identified and optimized (with respect to multiplicative complexity) a major non-linear component in an implementation of the AES S-box. That completes the first step of our technique for circuit optimization, but in other circuits, one may be able to identify more non-linear components with few enough inputs that they can also be optimized before continuing. For the AES S-box, after optimizing the non-linear portion of the circuit, the resulting circuit contained large linear connected components. In fact, from a cryptanalyst's point of view, the topology of the resulting circuit is potentially of interest: the S-box of AES consists of an initial linear expansion U from 8 to 22 bits, followed by a non-linear contraction F from 22 to 18 bits, and ending with a linear contraction B from 18 to 8 bits. The U and B matrices are given in [3]. AES's S-box is $S(\mathbf{x}) = B \cdot F(U \cdot \mathbf{x}) + [11000110]^T$, where $\cdot$ is matrix multiplication and $\mathbf{x}$ is the 8-bit S-box input. Note that the initial linear expansion and the linear contraction were defined to contain as much of the circuit as possible while still being linear. Thus, the portion of the circuit defined by U, for example, overlaps with the $GF(2^8)$ inversion. The next step was to minimize the circuits for computing U and B.

4 Minimizing Linear Components

Gate optimization of circuits for linear functions has been extensively studied. It has been shown that the problem of linear-circuit optimization is NP-hard [1]. That paper further shows that unless P=NP, this problem does not even have efficient ϵ-approximation schemes. Thus, our goal in this research is restricted to improving on known heuristics. As far as we know, the most successful heuristics are variations on a greedy algorithm due to Paar [11]. We report significant improvements over the latter methods.

A linear straight-line program over a field F is a variation on a straight-line program which does not allow multiplication of variables. That is, every line of the program is of the form $u := \lambda v + \mu w$ where λ, μ are in F and v, w are variables. Constructing a linear circuit for a given function f is equivalent to constructing a linear straight-line program over GF(2) which computes f. (Note that, over GF(2) λ, and μ are always 1 and thus are never written explicitly.)

A linear straight-line program over GF(2) is said to be *cancellation-free* if, for every line of the program $u := v + w$, none of the variables in the expression for v are also present in the expression for w, i.e., there is no cancellation of variables in the computation.

Previous work on circuit minimization for AES S-boxes (e.g. [10,12,5]) only consider cancellation-free straight-line programs for producing a set of linear forms over GF(2). Some authors appear to make the incorrect assumption that there always exists a cancellation-free optimal linear program over GF(2). A small counter-example showing this is not the case is the following:

$$x_1 + x_2;\, x_1 + x_2 + x_3;\, x_1 + x_2 + x_3 + x_4;\, x_2 + x_3 + x_4.$$

It is not hard (although somewhat tedious) to see that the optimum cancellation-free straight-line program has length 5. A solution of length 4 which allows cancellations is

$$v_1 = x_1 + x_2;\ v_2 = v_1 + x_3;\ v_3 = v_2 + x_4;\ v_4 = v_3 + x_1.$$

In [1], we show that any algorithm for computing linear programs that is restricted to cancellation-free programs is at most $\frac{3}{2}$-approximating. Thus, even optimal cancellation-free circuits can be far from optimal in the unrestricted model. The heuristic we present below is not restricted to producing cancellation-free circuits. Furthermore, there appears to be little reason for restricting the search to cancellation-free circuits, as we have shown that finding an optimal cancellation-free circuit is NP-hard ([1]).

A New Heuristic. Let S be a set of linear functions. For any linear predicate f, we define the distance $\delta(S, f)$ as the minimum number of additions of elements from S necessary to obtain f.

The problem is to find a short linear program that computes $f(\mathbf{x}) = M\mathbf{x}$ where M is an $m \times n$ matrix over GF(2). The heuristic is as follows. We keep a "base" S of "known" functions. Initially S is just the set of variables $x_1, \ldots, x_n$. We maintain the vector $Dist[]$ of distances from S to the linear functions given by the rows of M. That is, $Dist[i] = \delta(S, f_i)$ where f_i is the i^{th} row of M multiplied by the input vector $\mathbf{x}$. Initially, $Dist[i]$ is just one less than the Hamming weight of row i. We then perform the following loop

– pick a new base element by adding two existing base elements;
– update $Dist[]$;

until $Dist[i] = 0$ for all i.

The current criterion for picking the new base element is

– pick one that minimizes the sum of new distances;
– resolve ties by *maximizing* the Euclidean norm of the vector of new distances.

This tie resolution criterion, which we term "Norm", may seem counter-intuitive. The basic idea is that we prefer a distance vector like 0,0,3,1 to one like 1,1,1,1. In the latter case, we would need 4 more gates to finish. In the former, 3 might do it.

The bulk of the time of the heuristic is spent on picking the new base element. Our experiments show that the following "pre-emptive" choice usually improves running time without increasing the size of the output circuit:

– if any two bases $S[i], S[j]$ are such that $S[i] \oplus S[j]$ is a row in M, then pick this sum as the new base element.

The tie resolution criterion is a critical part of the heuristic. It does well on most matrices we have tried, but we have found specific matrices for which other decision rules do better. Intuitively, no one simple rule should work for all matrices. The effectiveness of the heuristic most likely depends on the topology of the digraph represented by the input matrix. We have not pursued this line of inquiry. We have, however tested our heuristic with various tie resolution methods against Paar's algorithm [11]. On random matrices, our heuristic gives significant improvements under Norm as well as under three other tie-breaking rules (see Section 6),

The distance vector in our heuristics is computed by exhaustive search. The reason the heuristic is practical for moderate-size matrices is that the distance can only decrease. In fact, it can only decrease by 1. So when a new base is being considered, if a distance is d, then only combinations of exactly $d-1$ old base elements and the new base element need to be considered.

A Small Example Using the Heuristic. Suppose we need a circuit that computes the system of equations defined in Fig. 3, which is equivalent to finding a circuit for multiplication by the 6×5 matrix, M, given in the figure.

$$\begin{aligned} y_0 &= x_0 + x_1 + x_2 \\ y_1 &= x_1 + x_3 + x_4 \\ y_2 &= x_0 + x_2 + x_3 + x_4 \\ y_3 &= x_1 + x_2 + x_3 \\ y_4 &= x_0 + x_1 + x_3 \\ y_5 &= x_1 + x_2 + x_3 + x_4 \end{aligned} \qquad M = \begin{bmatrix} 1\,1\,1\,0\,0 \\ 0\,1\,0\,1\,1 \\ 1\,0\,1\,1\,1 \\ 0\,1\,1\,1\,0 \\ 1\,1\,0\,1\,0 \\ 0\,1\,1\,1\,1 \end{bmatrix}$$

Fig. 3. Example sequence of equations and corresponding matrix

The *target signals* to be computed are simply the rows of M. The initial base is $\{x_0, x_1, x_2, x_3, x_4\}$, which corresponds to

$$S = \{\begin{bmatrix}1\,0\,0\,0\,0\end{bmatrix}, \begin{bmatrix}0\,1\,0\,0\,0\end{bmatrix}, \begin{bmatrix}0\,0\,1\,0\,0\end{bmatrix},$$

$$\begin{bmatrix}0\,0\,0\,1\,0\end{bmatrix}, \begin{bmatrix}0\,0\,0\,0\,1\end{bmatrix}\}$$

The initial distance vector is $D = \begin{bmatrix}2\,2\,3\,2\,2\,3\end{bmatrix}$.

The heuristic must find two base vectors whose sum, when added to the base, minimizes the sum of the new distances. It turns out the right choice is to calculate $x_1 + x_3$. So the new base S is expanded to contain the signal

$$\begin{bmatrix}0\,1\,0\,1\,0\end{bmatrix} \quad = \quad \begin{bmatrix}0\,1\,0\,0\,0\end{bmatrix} \quad + \quad \begin{bmatrix}0\,0\,0\,1\,0\end{bmatrix}$$

The new distance vector is $D = \begin{bmatrix}2\,1\,3\,1\,1\,2\end{bmatrix}$.

The full run of the program is below. The tie breaking criteria is used in Step 2. If one had chosen $x_1 + x_2$ instead of $x_0 + t_5$, the new distance vector would be $[\ 1\ \ 1\ \ 3\ \ 1\ \ 1\ \ 2\ ]$, which has norm $\sqrt{17}$, while the one found has norm $\sqrt{19}$. Note that there is cancellation in the last step.

Step 1 : $t_5 = x_1 + x_3$. New D : [2 1 3 1 1 2].
Step 2 : $t_6 = x_0 + t_5$ (found target signal y_4 = [1 1 0 1 0]). New D : [2 1 3 1 0 2].
Step 3 : $t_7 = x_2 + t_5$ (found target signal y_3 = [0 1 1 1 0]). New D : [2 1 3 0 0 1].
Step 4 : $t_8 = x_4 + t_5$ (found target signal y_1 = [0 1 0 1 1]). New D : [2 0 3 0 0 1].
Step 5 : $t_9 = x_2 + t_8$ (found target signal y_5 = [0 1 1 1 1]). New D : [2 0 2 0 0 0].
Step 6 : $t_{10} = x_0 + x_1$. New D : [1 0 1 0 0 0].
Step 7 : $t_{11} = x_2 + t_{10}$ (found target signal y_0 = [1 1 1 0 0]) .
New D : [0 0 1 0 0 0].
Step 8 : $t_{12} = t_8 + t_{11}$ (found target signal y_2 = [1 0 1 1 1]).
New D : [0 0 0 0 0 0]. (DONE!)

Thus, after the x_i, which may be nonlinear functions of other variables, are computed, the y_i are computed by following the algorithm produced and, in this case, letting $y_0 = t_{11}$, $y_1 = t_8$, $y_2 = t_{12}$, $y_3 = t_7$, $y_4 - t_6$, $t_5 = t_9$.

Note that the optimization mentioned above with the pre-emptive choice for a new base element was not applied in this example. That optimization gives a less interesting ordering from what is shown here, though one still gets a circuit with eight gates.

5 A Circuit for the S-Box of AES

Our techniques yield a circuit for the AES S-box composed of three parts: a "top" linear transformation, U; a middle non-linear part; and a "bottom" linear transformation, B. See [3] for a definition of the circuit. For the matrix U, the smallest circuits we found had 23 $\oplus$ gates. Among the many such circuits, the shortest ones have depth 7. It is worthwhile to note that if 24 $\oplus$ gates are allowed, circuits with depth 4 exist for the matrix U. The non-linear middle part of the S-box circuit is a function from 22 to 18 bits. It contains 32 $\wedge$ gates and 30 $\oplus$ gates. For the matrix, B, the randomized version of our heuristic yields many circuits with 30 $\oplus$ gates. The heuristic is fast enough that we are able to pick a circuit which is both small and short, having depth 6.

6 Experiments with Different Tie–Breaking Methods

In order to compare the effects of using different tie-breakers, we tested our heuristics on matrices generated as follows

- We first chose a size (for example, 10×20 matrices, which represent 10 linear forms on 20 distinct variables);
- We then picked a *bias* ρ between 0 and 1;
- For each entry of the matrix, we set the bit to 1 with probability ρ and to 0 with probability $1 - \rho$. Thus ρ is the expected fraction of variables that appears in each linear form.
- Matrices with rows which are all zeros were discarded, as were matrices containing duplicate rows.

The testing was performed with a C++ program, compiled with g++ -O3, on a quadcore x86_64, running Ubuntu 9.10, with Intel Xenon 5150 processors running at 2.66 GHz, with 8 GB memory. There were no other users on the machine. The programs and matrices used can be found at www.imada.sdu.dk/ joan/xor/, though minor changes are necessary to run the programs with different files as input or to change the matrix size and bias for the matrix generator. We compared the different heuristics on sets of one hundred random matrices with different sizes and densities. The experiment showed that the heuristics were slower when the bias was larger. This was expected, since the initial "distances" (number of operations on the base vectors to obtain the target vectors) were then larger on average when there were more ones in the matrices.

The tie-breakers we compared were the following:

– **Norm:** maximizing the Euclidean norm
– **Norm-largest:** maximizing the square of the Euclidean norm minus the largest distance
– **Norm-diff:** maximizing the square of the Euclidean norm minus the difference of the largest two distances
– **Random:** In processing the possible new base vectors, if the current possible new base vector has the same sum of distances as the previous best (current choice), then flip an unbiased coin. If heads, then keep the current choice. If tails, then apply the Norm criterion. This heuristic may end up choosing a pair with non-maximum Euclidean norm. On the other hand, it allows substitution of one optimum (by sum-of-distances and Euclidean norm) pair by another found later in the search.

In all cases, except the "Random" one, when there were still ties after applying the "tie-breaker", the first pair with both the minimum sum of distances and the optimal value for the tie-breaker was chosen. This was the base pair with lexicographically minimum indices (i, j). Randomized tie-breaking allows running the heuristic several times and picking the best result. In our tests we ran the heuristic with "Random" tie-breaking three times.

We also compared these heuristics to Paar's heuristic [11] on the same matrices. Paar's heuristic repeatedly finds the most frequently occurring base pair and adds that as the next base pair. It is significantly faster than our heuristic, but it produces only cancellation-free circuits. Its performance, relative to the heuristics proposed here, decreases as the bias increases, using more than 30% extra gates when the bias is 3/4 (when the number of rows is at least 15) and 40% extra when the bias is 9/10.

Among the biases tried, the number of gates in the circuits found by our heuristics is similar with biases 1/2 and 3/4. It is not a strictly increasing function of the bias, since when nearly all of the variables are used in nearly all of the forms, the outputs from many of the gates can be reused for many targets. Thus, circuits with fewer gates were found when the bias was 9/10 than when it was 1/2 or 3/4. This was also true for Paar's heuristic, but less dramatically so.

All the tie resolution criteria performed fairly similarly, producing circuits of nearly the same size, with Random apparently doing slightly better (more often producing smaller circuits), presumably because it tries three different circuits and uses the best. Random also runs for about three times as long as the others. The results of these tests are presented in tables in [3]. For each heuristic, and all matrix sizes and biases, 100 randomly chosen matrices were tested.

For each tie-breaker rule and Paar's heuristic, for each matrix size and bias, the average number of gates in the circuits found and the number of matrices where that heuristic did not obtain the minimum value of all of the heuristics was computed, along with the running time in seconds. The Paar heuristic was beaten by at least one of the other heuristics on all 700 matrices except for 17 of the 100 with bias 1/4 (and there was only one matrix on which Paar's heuristic beat any of the other heuristics). In fact, for the tests with bias larger than 1/4, Paar's heuristic did worse than any of the other heuristic on every one of the matrices; usually the values obtained for the newer heuristics were similar, with Random possibly being marginally better, but with the value for Paar's heuristic being significantly larger.

Paar's heuristic (and, for matrices between size 4 and 10, a variant which does at most one gate better on average in the data presented) was tested [11] on square matrices of sizes 4×4 through 16×16 and the average number of XOR gates is presented, along with the relative improvement over the straightforward implementation. These square matrices came from applying Mastrovito's [9] matrix description of multiplication in $GF(2^n)$ to constant multiplication. Paar tries all possible constants in $GF(2^n)$ for n between 4 and 16, giving these square matrices. Since our heuristics are so much slower and the matrices in the cryptographic applications we are interested in do not necessarily have this form, we have not tested on all of these restricted matrices of those sizes, but rather on random matrices with different biases. For 15×15 matrices, Paar gets an average of 52.9 gates. This is similar to our results for Paar's algorithm with 15×15 matrices with biases 1/2 and 3/4, where the Paar heuristic gets averages of 51.7 and 53.3 gates, respectively. For bias 1/2, our deterministic heuristics get average gate counts between 44.21 and 44.28, while Random gets 43.81. For bias 3/4, our deterministic heuristics all get average count 40.82, while Random gets 40.38. Thus, our relative improvement over the Paar heuristic is between 17% and 32% for these types of matrices. Paar's result of 52.9 gates for 15×15 matrices is a relative improvement of 45.5% over the straightforward approach.

We also computed the sums of the values which are the minimum of those calculated by the different heuristics for each matrix. The tables in [3] show that for each of the tie-breakers, there are cases where it gets a worse result than at least one of the others.

7 Conclusions and Work in Progress

We tested new techniques for decreasing circuit size. The techniques were applied to the extensively studied AES S-box. We obtained the smallest circuit yet constructed for this function. The circuit contains 32 AND gates and 83 XOR/XNOR gates for a total of 115 gates. As by-products of the experiment, we obtained very small circuits for inversion in $GF(2^4)$ and $GF(2^8)$.

The experiments with linear circuit optimization indicate that our techniques are likely to be superior to previous techniques which produced only cancellation-free circuits. We expect this to be particularly useful for cryptographic applications, both hardware and software implementations, where many XOR operations are used, along with some AND operations to introduce nonlinearity.

It would be interesting to determine how close to optimal the circuits found by these techniques usually are and how much better they are than the optimal cancellation-free circuits. Finding even better techniques which are not restricted to finding cancellation-free circuits would also be very interesting, as would applying these techniques to other applications.

References

1. Boyar, J., Matthews, P., Peralta, R.: On the shortest linear straight-line program for computing linear forms. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 168–179. Springer, Heidelberg (2008)
2. Boyar, J., Peralta, R.: Tight bounds for the multiplicative complexity of symmetric functions. Theoretical Computer Science 396(1-3), 223–246 (2008)
3. Boyar, J., Peralta, R.: New logic minimization techniques with applications to cryptology. Cryptology ePrint Archive, Report 2009/191 (2009), http://eprint.iacr.org/
4. Boyar, J., Peralta, R.: Patent application number 61089998 filed with the U.S. Patent and Trademark Office. In: A new technique for combinational circuit optimization and a new circuit for the S-Box for AES (2009)
5. Canright, D.: A very compact Rijndael S-box. Technical Report NPS-MA-05-001, Naval Postgraduate School (2005)
6. FIPS. Advanced Encryption Standard (AES). National Institute of Standards and Technology (2001)
7. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. Inf. Comput. 78(3), 171–177 (1988)
8. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 1–17. Springer, Heidelberg (2009)
9. Mastrovito, E.: VLSI architectures for computation in Galois fields. PhD thesis, Linköping University, Dept. Electr. Eng., Sweden (1991)
10. Paar, C.: Some remarks on efficient inversion in finite fields. In: 1995 IEEE International Symposium on Information Theory, Whistler, B.C. Canada, p. 58 (1995)
11. Paar, C.: Optimized arithmetic for Reed-Solomon encoders. In: IEEE International Symposium on Information Theory, p. 250 (1997)
12. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact rijndael hardware architecture with S-box optimization. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 239–254. Springer, Heidelberg (2001)

Randomized Rounding for Routing and Covering Problems: Experiments and Improvements*

Benjamin Doerr[1], Marvin Künnemann[2], and Magnus Wahlström[1]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] Universität des Saarlandes, Saarbrücken, Germany

Abstract. We investigate how the recently developed different approaches to generate randomized roundings satisfying disjoint cardinality constraints behave when used in two classical algorithmic problems, namely low-congestion routing in networks and max-coverage problems in hypergraphs. Based on our experiments, we also propose and investigate the following new ideas. For the low-congestion routing problems, we suggest to solve a second LP, which yields the same congestion, but aims at producing a solution that is easier to round. For the max-coverage instances, observing that the greedy heuristic also performs very good, we develop hybrid approaches, in the form of a strengthened method of derandomized rounding, and a simple greedy/rounding hybrid using greedy and LP-based rounding elements. Experiments show that these ideas significantly reduce the rounding errors.

For an important special case of max-coverage, namely unit disk max-domination, we also develop a PTAS. However, experiments show it less competitive than other approaches, except possibly for extremely high solution qualities.

1 Introduction

Randomized rounding is one of the core primitives in randomized algorithmics. In contrast to many deep theoretical results, only very little experimental knowledge exists, and almost no fine-tuning and other implementation advice exists. Such results became even more interesting, since in the last ten years two substantially different methods [20,6,7,4] extending the classical approach of Raghavan and Thompson [19,18] were developed.

The only experimental work on either classical randomized rounding or the new approaches seems to be [5]. It compares the different methods on randomly generated rounding problems. The purpose of this work is to extend these results to two less artificial problem classes, namely routing and covering problems. These problems are among the first ones for which randomized rounding has been proven (by theoretical means) to lead to good algorithms.

Randomized Rounding: Given an arbitrary real number x, we say that (the random variable) y is a randomized rounding of x, if y equals $\lfloor x \rfloor + 1$ with probability $\{x\} := x - \lfloor x \rfloor$ and $\lfloor x \rfloor$ otherwise. In simple words, the closer x is to the next larger integer, the higher the chance of being rounded up.

* Supported by the German Science Foundation (DFG) via its priority program (SPP) 1307 "Algorithm Engineering", grant DO 749/4-2.

Randomized rounding builds on the simple observation that this keeps the expectation unchanged, that is, $E(y) = x$. This naturally extends to linear expressions. If $y_1, \ldots, y_n$ are randomized roundings of $x_1, \ldots, x_n$ and $f : \mathbb{R}^n \to \mathbb{R}$ is a linear function, then $E(f(y_1, \ldots, y_n)) = f(x_1, \ldots, x_n)$. If, in addition, the y_i are independent, then Chernoff bounds allow strong quantitative statements showing that with high probability, $f(y_1, \ldots, y_n)$ is not far from its expectation. These two key facts allow to use randomized rounding in connection with integer linear programming. Two examples of this are given in the following sections.

The new aspect of the works [20,6,7,4] is that they allow to generate randomized roundings that satisfy certain cardinality constraints with probability one. That is, for certain sets I, we can prescribe that $\sum_{i \in I} y_i = \sum_{i \in I} x_i$, provided the right-hand side is integral. This can be done without giving in with the other properties—both methods generate randomized roundings that admit the same Chernoff bounds as independent randomized rounding.

For reasons of space, we cannot describe these rounding algorithms here. However, since in this paper we are mainly comparing them experimentally, the reader may treat them as black-box, keeping in mind only that they generate randomized roundings that look independent, but satisfy cardinality constraints.

We shall concentrate on disjoint cardinality constraints. This is the most common form of cardinality constraints. Also, the comparison of the methods is more interesting here, since all have the same time complexity.

Our Results: The aim of this work is to find out how well the different rounding approaches are suited to solve classical problems that are often attacked with LP-based methods, but also to try to find fine-tunings and alternative approaches.

As underlying problems we chose the classical low-congestion routing problem and the max-coverage problem. They are different in flavor since in the first, randomized rounding is used with a focus of exploiting Chernoff bounds in linear constraints and objective function. In the second, since the right-hand side of the inequalities is one, we cannot do so, but resort to accepting that a certain fraction of the vertices covered in the relaxation are not covered after rounding.

All our results indicate that generally the derandomized algorithms yield superior results. The increase run-time over the randomized versions usually is still negligible compared to the complexity of solving the LPs involved.

For the low-congestion routing problem, we regard routing requests placed randomly on a two-dimensional grid. We regard instances small enough to compute the optimum solutions via solving an integer linear program. We observe that randomized rounding with cardinality constraints obtains reasonably good solutions. Surprisingly, unlike in previous experiments [5], we observe that the bit-wise randomized approach of [4] produces better results than the tree-based one of [20].

The gap to the optimum is roughly halved if we use a derandomization of randomized rounding. Here, the derandomization of [20] obtained in [5] proved to be superior.

In an attempt to fine-tune randomized rounding, we propose solving a second LP which gives the same congestion in the relaxation, but aims at making the solution easier to round. While this naturally does not give improved theoretical guarantees, it yields a good reduction of the rounding errors, in particular, in combination with the derandomization. This seems to be a fruitful approach whenever the additional cost of solving a second LP is admissible.

Our analysis of max-coverage shows that both randomized rounding and the greedy algorithm produce good results in general. However, for both there are instances showing the other behave much better. Analyzing the data produced in our experiments, we consider two paths to a hybrid approach. One way is to strengthen the derandomization to include a greedy component, as a gradient-based rounding; the other, complementary, is to spend part of the budget greedily, and solve the remaining instance via an LP- and randomized rounding-approach. Both hybrids perform better than either of the plain approaches; the gradient-based rounding performs particularly well. We also give a new way of rounding under weighted knapsack constraints, which is both significantly more practical and theoretically cleaner than the previously known method.

For a natural planar Euclidian version of the problem, we also give a PTAS. However, unlike for all other approaches used in this paper, the experimental results are not much better than the theoretical guarantees. Therefore, this is an alternative useful only if very good approximations are needed and if computation power is available plentiful.

2 Randomized Rounding for Low-Congestion Routing

The Low-Congestion Routing Problem in Networks. The low-congestion routing problem is one of the classical applications of randomized rounding [19]. In its simplest version, the objective is to route a number of requests through a given network, minimizing the maximum usage of an edge ("congestion"). Problems of this type found all kinds of applications, an early one being routing wires in gate arrays [9].

We regard the following basic variant, previously investigated in [19,18,11,20,7]. Given is a (directed) network $G = (V, E)$, together with k routing requests. Each consists of a source vertex s_i, a target vertex t_i and a demand r_i. The objective is to find, for each $i \in [k] := \{1, \ldots, k\}$, a flow from s_i to t_i having flow value $r_i \in \mathbb{N}$, such that the congestion, that is, the maximum total flow over an edge, is minimized. This problem is easily formulated as integer linear program (ILP): We minimize the congestion C subject to the constraints

$$\forall e \in E : \sum_{i=1}^{k} x_{ie} \leq C \tag{1}$$

$$\forall i \in [k] : \sum_{e=(s_i,v)\in E} x_{ie} - \sum_{e=(v,s_i)\in E} x_{ie} = r_i \tag{2}$$

$$\forall i \in [k]\, \forall v \in V \setminus \{s_i, t_i\} : \sum_{e=(w,v)\in E} x_{ie} = \sum_{e=(v,w)\in E} x_{ie} \tag{3}$$

$$\forall i \in [k]\, \forall e \in E : x_{ie} \in \{0, 1\}. \tag{4}$$

We should add that [19] only regard the special case of all demands r_i being one, since randomized rounding respecting cardinality constraints with right-hand side greater than one was not available at that time. For an application with particular need for larger r_i, see the failure restoration problem in optical networks described in [7]. Also, we should add that other authors in addition have edge capacities c_e and then minimize the relative congestion, but it is easily seen that this just replaces the C in the first type of constraints by $c_e C$.

Since already the case of unit demands is NP-complete, optimal solutions seem difficult to obtain. The common solution concept is to (i) solve the linear relaxation of the ILP and obtain a fractionally optimal solution (x^*, C^*); (ii) use *path stripping* to decompose each flow f_i encoded in $x^*_\cdot$ into a weighted sum $f_i = \sum_{P \in \mathcal{P}_i} y^*_{iP} f_P$, where $\mathcal{P}_i$ is a finite set of s_i–t_i paths, for each $P \in \mathcal{P}_i$, f_P is the flow that has exactly one unit on each eadge of P, and $y^*_{iP} \in [0,1]$—note that all this implies $\sum_{P \in \mathcal{P}_i} y^*_{iP} = r_i$; (iii) use randomized rounding to round all y^*_{iP} to $y_{iP} \in \{0,1\}$ in such a way that the cardinality constraints $\sum_{P \in \mathcal{P}_i} y_{iP} = r_i$ are maintained. Now $\sum_{P \in \mathcal{P}_i} y_{iP} f_P$ is a flow from s_i to t_i with flow value r_i. These flows form a solution having a congestion of $C = \max_{e \in E} \sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i, e \in P} y_{iP}$. Large deviation bounds show that this congestion is not far from the value C^* given by the relaxation [19,11]:

$$C = O\left(\frac{\log m}{\log(2 \log m / C^*)}\right), \text{ if } C^* \le \log m;$$
$$C = C^* + O(\sqrt{C^* \log m}), \text{ if } C^* > \log m.$$

Recall that C^* is a lower bound for the optimal solution. Hence if C^* is not too small compared to m, then this approach gives very good approximation factors.

Algorithms Used. To approximately or exactly solve our test instances, we used the following algorithms. Whenever run-times permitted, we used the exact ILP-Solver ILOG CPLEX 11.0 to directly solve the ILP given by (1) to (4). All other approaches involve solving the linear relaxation of the ILP (for which again we used CPLEX) and then different rounding methods.

Since the ILP contains hard cardinality constraints, we cannot use the classical independent approach of Raghavan and Thompson (we did so, though, ignoring the cardinality constraints, to see if the cardinality constraints make rounding more difficult). There are two approaches to generate randomized roundings respecting cardinality constraints due to Srinivasan [20] and the first author [4]. Both can be derandomized [5,4], so that in total we have four rounding methods available. See the original papers or [5] for a more detailed discussion of these methods. All algorithms different from CPLEX were implemented in C/C++.

Experimental Set-up. To analyze the questions discussed in the introduction, we regarded the following type of instances. Motivated by the fact that many routing problems have a two-dimensional flavor (e.g., the wire routing problem of [9]), we chose a finite two-dimensional bi-directed grid. Note that this simple graph is far from trivial for routing, see, e.g., the thrilling one-turn routing conjecture in [9], which is, to the best of our knowledge still open.

We choose routing requests randomly as follows. Both s_i and t_i are chosen uniformly at random from V. To reduce otherwise the influence of randomness, we choose all

Table 1. Congestions achieved by the 7 different approaches for a selection of 6 instances. The optimum was computed by solving the IP via CPLEX (not feasible for larger instances). For the other algorithms we state the relative increase of the congestion over the optimum.

	5×5, 10	10×10, 10	15×15, 10	5×5, 75	10×10, 75	15×15, 75
Optimum	3.37	2.19	1.98	14.76	7.76	5.39
RR [20]	+9.23%	+32.17%	+42.29%	+5.96%	+24.48%	+45.45%
RR [4]	+7.13%	+30.43%	+40.31%	+4.13%	+19.07%	+38.78%
RR+	+6.35%	+31.85%	+40.26%	+2.64%	+12.50%	+31.91%
DeRR [4]	+3.77%	+22.33%	+23.42%	+1.90%	+10.95%	+25.97%
DeRR [5]	+2.76%	+16.38%	+13.76%	+0.88%	+8.38%	+17.07%
DeRR+	+1.19%	+11.81%	+16.59%	+0.47%	+4.25%	+14.84%

Table 2. Run-times of the 5 algorithms in seconds. Given is the time for this particular step. For example, the run-time of what is called "DeRR+" in Table 1 is the sum of the values in lines "LP", "DeRR" and "Heur.".

	5×5, 10	10×10, 10	15×15, 10	5×5, 75	10×10, 75	15×15, 75
IP (CPLEX)	0.0270	0.368	5.72	0.776	61.52	7977
LP (CPLEX)	0.0227	0.235	1.61	0.697	34.02	2004
Heur.	0.0129	0.612	9.42	0.096	51.31	1755
DeRR [5]	0.0009	0.0061	0.0178	0.0028	0.018	0.07
DeRR [4]	0.0126	0.1062	0.3332	0.0589	0.459	1.61

demands as $r_i = 3$. We also tried placing the s_i, t_i uniformly at random on the outer border of the grid, but saw no significant differences.

The size n of the grid and the number of demands k was varied to create different instance sizes and densities. All numerical values reported are the averages over 100 runs. The times were measured on AMD dual processor 2.4 GHZ Opteron machines.

Analysis. A small subset of our results is presented in Table 1. For the grid sizes 5×5, 10×10 and 15×15 together with 10 and 75 demands, we state the average values of the congestion of an optimal solution (line 1 of the table) and the amounts by which a solution computed by one of the four randomized rounding approaches (lines 2, 3, 5, and 6) is worse (in percent). Line 4 and 7 of the table refer to an improvement discussed in the subsequent subsection.

Particularly for instances that do show some congestion, we see that randomized rounding yields quite good solutions, much better than what the theoretical bounds would predict. Derandomization is clearly worth the small extra effort (cf. the run-times in Table 2), reducing the gap to the optimum by roughly a half.

Comparing the different methods, surprisingly, our experiments generally show that the bit-wise randomized rounding approach of [4] (line 3) produced slightly better rounding errors than the tree-based one of [20] (line 2). We do not understand this phenomenon currently. Among the derandomizations, as expected and similarly as for random instances [5], the derandomization of the tree-based approach of [20] given in [5] is superior to the derandomization of the bit-wise one in [4]. This is due to the iterative nature of the latter, see again [5].

We also used classical independent randomized rounding. Clearly, this does not produce feasible solutions in most cases. However, even ignoring this issue, we also observed that we typically have slightly larger congestions (e.g. in the sparse instance of 10 demands in a 15×15 grid, independent rounding lead to a congestion of 3.19 compared to congestions of 2.80 and 2.85 for the randomized approaches of [4] and [20]).

Run-times consumed by the different stages are mainly given in Table 2. All randomized rounding stages for each instance took less than 0.02 seconds. Less than a tenth of this is the time needed for the path-stripping in each instance. Hence these numbers are not given in the table. From the table, we see that the bit-wise derandomization takes about 20 times longer than the tree-based one, but both numbers are greatly dominated by the times for solving the LP (ignore the "Heur." line for the moment).

A Heuristic Making Life Easier for Randomized Rounding. As can be seen from the results presented so far, the different randomized rounding approaches usually find solutions that are not far from the optimum. We now propose and analyze a heuristic way to improve the performance.

The rough idea is simple. Having solved the linear relaxation of the ILP, we know the optimal (relaxed) congestion C^* that can be achieved. The congestion we end up with stems from this C^* plus possible rounding errors inflicted in the congestion constraints (1). It is clear that randomized rounding has a higher change to increase the congestion if there are many congestion constraints satisfied with equality in the relaxation.

Therefore, the heuristic we suggest is to resolve the LP with the following modifications. Let $\delta \in [0, C^*]$ be a parameter open for fine-tuning. We replace the congestion constraints (1) by $\forall e \in E : \sum_{i=1}^{k} x_{ie} \leq C^* - \delta + z_e$, where C^* is the (fixed) optimal congestion obtained from the first LP and $z_e \in [0, \delta]$ are new variables. The new objective is to minimize $\sum_{e \in E} z_e$. Since the z_e are at most δ, the flow given by a solution of this new LP also yields a congestion of at most C^*. However, the new objective punishes edges with total flow exceeding $C^* - \delta$. In consequence, the solution we obtain is also a solution for the original LP, but one that in addition tries to keep some room in the congestion constraints.

For the few instances that space permits do give data, the experimental results are again presented in Table 1. Line 4 contains the results obtained by using randomized rounding as in [4] after applying the heuristic and line 7 does so with the derandomization of [20,5]. We did the same experiments with the other two rounding algorithms. Since the results were mainly inferior (to a similar extent as without the improvement), we omitted these numbers in the table. In all experiments, we chose $\delta = 1$.

The results clearly show that using this heuristic can be worth the extra effort of solving a second LP. Apart from two instances with very small objective values 1.98 and 2.19, the heuristic always gains us a significant improvement. Surprisingly, these gains tend to be higher when using the derandomized rounding algorithm.

It should be noted, though, that solving the second LP can be costly, as the numbers in the last line of Table 2 indicate.

3 Max Coverage: Greedy, Rounding, and Hybrid Approaches

Another problem where dependent rounding has found application is the max-coverage problem. In this problem, the input is a set $\{S_1, \dots, S_n\}$ of sets and a budget bound L. The task is to select a set of L sets to maximize the size of their union. Additionally, there can be costs c_i associated with the sets, and weights or profits w_i associated with the elements. In this case the task is to maximize the weighted sum of the covered elements, subject to the constraint that the total cost of the sets is at most L. Approximation algorithms for max-coverage have been devised using both greedy and rounding-based approaches; see [3,10,20,1] for details.

The rounding-based approximations start from the following LP-relaxation of the problem. Let n be the number of sets, and m the number of elements in the instance. Use variables $x \in [0,1]^m$ and $y \in [0,1]^n$; we maximize $\sum_{i=1}^m w_i x_i$ subject to (i) $\sum_{i=1}^n c_i y_i \le L$, and (ii) $x_i \le \sum_{j:i\in S_j} y_j$ for every $i \in [m]$. Let (x^*, y^*) be an optimal solution to this, of value W^*, and consider the expected outcome of independently rounding the variables y^*:

$$F(y^*) = \sum_{i=1}^{m} w_i (1 - \prod_{j:i\in S_j} (1 - y_j^*)). \tag{5}$$

In the unit-cost case, applying randomized rounding to y^* with a cardinality constraint preserving the sum $\sum_i y_i^*$ produces a rounding with expected value $F(y^*)$, due to the negative correlation properties of the rounding [20]; since $F(y^*) \ge (1 - 1/e)W^*$, we get a randomized $(1 - 1/e)$-approximation. The derandomized version works via the method of conditional expectation. As shown by Ageev and Sviridenko [1], we can use $F(y)$ directly as a guide for the derandomization, and produce a rounding $y \in \{0,1\}^n$ of y^*, such that $F(y) \ge F(y^*)$ with certainty.

For the case of weighted (knapsack) budget constraints, Srinivasan gives a rounding procedure (Lemma 3.1 in [20]) that approximately preserves the value of a weighted sum of the rounded variables, while guaranteeing negative correlation properties as in the unit-cost case. Combined with an enumeration and guessing phase, this provides a $(1 - 1/e + \varepsilon)$-approximation for any $\varepsilon > 0$ [20]. Unfortunately, due to the inexactness of the budget bound, this phase becomes very expensive; we complement this by a budget-preserving rounding procedure, described below.

Algorithms and Improvements. The algorithms we mainly compare will be the greedy algorithm, and variations on the rounding-based algorithms. The *greedy algorithm* repeatedly selects a set fitting in the budget that maximizes the ratio of the profit of the newly covered elements to the cost of the set. The *rounding-based* approach is outlined above. Recall that the expected value of a single randomized rounding equals $F(y^*)$, and can thus (unlike in Section 2) be computed exactly. We consider three ways of boosting this value. The first, *random-1000*, is to simply apply randomized rounding 1000 times and pick the best result; the second is *derandomization*, using Srinivasan-type rounding directly on $F(y)$, with arbitrary order of variable comparison. The third is *gradient-based rounding* which works as follows.

Recall that cardinality-preserving randomized rounding works by repeatedly considering pairs of non-integral variables and readjusting their values, maintaining the sum, such that one of them becomes integral (see e.g. [5]). By gradient-based rounding, we attempt to identify the best pair of variables to select for adjustment in each step. To truly find this pair would require $O(n^2)$ comparisons, each with cost $O(m)$, but we can approximate the selection by considering the gradient of $F(y)$. It is easy to show that if y_i and y_j are non-integral, and if $\frac{\partial F(y)}{\partial y_i} \geq \frac{\partial F(y)}{\partial y_j}$, then moving mass from y_j to y_i will keep the value of $F(y)$ non-decreasing. Thus we only need to compute and update the partial derivatives $\frac{\partial F(y)}{\partial y_i}$, which can be done analytically at a cost of $O(nm)$ per step, and we can in every step pair off the variables with largest and smallest values of partial derivative. While the total complexity of the rounding process becomes $O(n^2 m)$, as opposed to $O(nm)$ for standard derandomized rounding, the time requirement is small in practice (see experiments below).

Finally, we introduce the following *budget-preserving rounding*.

Theorem 1. *Let $y \in [0,1]^n$ such that $\sum_i c_i y_i = L$. In polynomial time, one can compute a rounding $\tilde{y} \in \{0,1\}^n$ such that $\sum_i c_i \tilde{y}_i \leq L + \max_i c_i$ and $F(\tilde{y}) \geq F(y)$.*

Proof sketch. Instead of maintaining the cardinality in each rounding step, change the variables to be rounded by unequal amounts, maintaining the weighted budget. It is easy to show that the convexity property used in [1] to derandomize the unit-cost case still holds for $F(y)$ over the new operation. □

For problems with a weighted budget, we use the same three rounding methods as for the unweighted case, with the random-1000 and the derandomized roundings using Lemma 3.1 of [20], and the gradient-based rounding using Theorem 1. Should a rounding exceed the budget constraint, we will greedily discard sets until the budget bound is reached.

A Unit Disk Maximum Domination PTAS. As a source of real-world instances, we consider a type of max-coverage instances derived from planar point set data. Given a set of points $P = \{p_1, \ldots, p_n\}$ in the plane and a *diameter* d, we define a graph (the unit disk intersection graph) by letting two points p_i, p_j be connected if and only if the Euclidean distance between them is at most d. In this graph, we consider the problem of max-domination, where selecting a vertex v covers v and all its neighbours. Interpreted as max-coverage, we thus get an instance where every vertex corresponds to one set and one element. All sets will have unit cost; the elements may have weights, interpreted as the profit of covering them.

This problem is NP-hard, as follows from the hardness of Minimum Dominating Set in unit disk graphs [15]. However, it has good approximation properties—using the grid-based shifting strategy of Arora [2], we are able to provide a polynomial-time approximation scheme (PTAS). This strategy was also applied to a related problem on the placement of wireless base stations by Glaßer et al. [8]. The proof is omitted for lack of space.

Theorem 2. *For any $\ell > 1$, the Max Domination problem on unit disk graphs with weighted vertices and unit costs admits a $(1 - 2/\ell)$-approximation in time $n^{O(\ell^2)}$.*

In our implementation, we replace certain steps by an MIP solver, specifically the exhaustive enumeration phase for the subproblems, and the dynamic programming knapsack step. With these modifications, execution of the algorithm for non-trivial approximation ratios becomes possible. Unfortunately, we will see that even with these modifications, the PTAS approach is inferior to the greedy and rounding algorithms for realistic approximation settings.

Experimental Setup. Our instances are of two types. For the unit disk max-domination problem, we use benchmark instances stemming from a real-world facility location problem, previously used in [17] and available at [14]. For these instances, a demand is provided with every point; we use these demands as profit values. To complement this, we use instances converted from facility location problems; all such instances are downloaded from UflLib [21]. In both cases, we convert the instances to max-coverage by selecting an appropriate distance threshold for membership. Though some instances are weighted, in all tests the individual set costs will be much smaller than our allocated budget, avoiding potential issues with approximations.

We use CPLEX for LP- and ILP-solving; all other algorithms were implemented by the authors in C.

Experiments. We now report the results of our experiments. To begin with, we show the running times of the different methods on various instances in Table 3. Note that the LP solver once again uses a significant fraction of our running time, and that performing many random roundings becomes more costly than derandomization, due to the need to evaluate the objective value for each solution. The low numbers for the gradient-based rounding, as compared to the derandomization, can partly be explained by the gradient-based rounding being problem-specific, while the derandomization uses general-purpose code.

Table 3. Running times for various instances and algorithms; the times for the rounding methods exclude the time for solving the relaxation

Instance	n	LP	Random-1000	Derand.	Grad. rounding	Greedy	IP
br818-400-30	818	0.36s	0.80s	0.11s	0.09s	0.33s	25s
kmed1-1k-37	1000	0.97s	1.46s	0.22s	0.25s	0.90s	$>$ 3600s
MR1-060-16.5	500	0.36s	0.74s	0.05s	0.04s	0.14s	$>$ 3600s

Table 4. Experiments on single instances. The data is averaged over 100 runs where the input data is randomly permuted. The column "LP: once" shows the expected outcome of a single randomized rounding; the following three columns show our three rounding methods. The optimum gives the best upper and lower bound achieved by an IP solver after one hour of running time.

Name	Size	Budget	Greedy	LP: once	1000	derand	gradient	Optimum
Chessboard	144	16	130	144	144	144	144	144
FPP ($k = 17$)	307	17	290	200	210	230	290	290
br818-400	818	30	28054	22157	26199	27397	28448	28709
kmed1-1k	1000	37	948	709	817	923	962	993–95
MR1-060	500	16500	1444	1179	1254	1402	1445	1462–94

Table 4 shows results for some individual instances (described below). The first we want to highlight are the Chessboard and Finite Projective Plane (FPP) instances [12], which will serve to reveal the differences between the different approximation approaches. The Chessboard instances are instances on a chessboard with side $3k$, and essentially correspond to a planar packing problem; the FPP instances are similarly a kind of packing problem, but on a graph with more complicated structure. Since all instances of these classes have equivalent combinatorial structure, we use only instance per class in our experiments. In both cases, the budget is set at the most difficult setting, which turns out to be just where the LP-relaxation can cover all or almost all elements.

The results immediately show the reason to pursue hybrid greedy/rounding algorithms. For the Chessboard instance, the LP-optimum is already integral, and thus every LP-rounding-based algorithm discovers an optimal solution. On the other hand, these instances are difficult for the greedy algorithm, as a few early mistakes, when all sets seem equivalent, will hurt the end tiling. In the FPP instances, however, we see the opposite effect. Here, upon inspection, we find that the LP-optimum is a useless mix of taking an equal, small amount of almost every variable, leaving all the work of finding a good integral solution up to the rounding. On the other hand, the greedy algorithm performs very well here; runs with an ILP-solver show that it is at most one step away from the optimum. We find that our proposed hybrid, the gradient-based rounding, produces consistent top-quality results in both test cases.

We now focus on the unit disk max-domination problem, with instances as described previously. We select the largest instance, with 818 points inscribed in a box of sides 6395 by 3975, and use a distance threshold of 400. This was chosen as a good balance, as too small or too large values (e.g., 100 resp. 800) creates too simple instances. Figure 1 shows the behaviour of the main algorithms (excluding the PTAS) for this instance, as depending on the budget. Observe that the LP-rounding approach is very powerful for small budgets (up to 20), while further guidance is needed for larger budgets. The gradient-based LP-rounding, providing just such guidance, produces top values throughout, frequently better than either the greedy or the standard rounding algorithms. This instance also appears in Tables 4 and 3 under the name br818-400 or br818-400-L, where L is the budget bound. Another instance class of the same type is the *k-median* instances [21]. Here we use the one named 1000-10, with a threshold of 1000, occurring in Tables 4 and 3 as kmed1-1k or kmed1-1k-L. For concerns of clutter, the PTAS is not included in the figure, but its data is given separately in Table 5. Note that for every feasible setting, the PTAS is both of lower quality and significantly slower than the alternatives. The k-median-instance is omitted, since we lack point data.

Table 5. PTAS performance compared to the greedy algorithm and IP solver

Instance	PTAS ($\ell = 3$)		PTAS ($\ell = 5$)		PTAS ($\ell = 7$)		Greedy	IP	
	Value	Time	Value	Time	Value	Time	Value	Value	Time
br818-400-20	20742	1.3s	22857	9s	24129	69s	25247	26192	0.5s
br818-400-25	23008	1.2s	24683	9s	25308	71s	26907	27670	9s
br818-400-30	23951	1.3s	24909	10s	27270	74s	28054	28709	25s

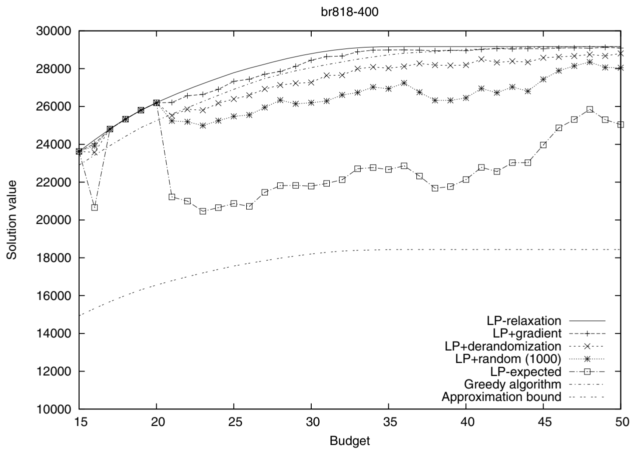


Fig. 1. Results for unit disk max-coverage instance br818-400. The plot shows the value of the LP-relaxation, the outcome of the three rounding methods, and the expected value of a single rounding against the greedy algorithm. The approximation bound shows $(1 - 1/e)$ times the LP optimum.

Finally, we also consider an instance class with weighted budgets, namely the M* instances proposed in [13]. These instances have facility costs scaled so that facilities close to many customers are more expensive, a situation the authors of [13] propose would arise in real-world situations. In the data from [21], the distances were pre-scaled by demand values, making the distances inappropriate for our use; we remove this scaling, and instead use the demands as profit values. Table 4 gives the results for instance MR1 with distance threshold 0.6, under the name MR1-060. In general for this class, we found that the greedy algorithm and the gradient-based rounding method produce practically identical results, while the other methods are inferior to this.

A Greedy/LP Hybrid. Motivated by our results, we consider a different, more general form of greedy/LP hybrid. Before we commence with the LP-rounding, we allocate some portion of the budget to greedy pre-selection, and apply the LP-relaxation and rounding using the remaining budget to the thus reduced problem. We find best results with a pre-selection of between ten and forty percent, finding that this hybrid can produce results superior to either the greedy or the derandomized rounding algorithm on their own. Combining greedy pre-selection with gradient-rounding can produce further improvement on some instances (e.g., kmed1-1k), but for others (e.g., br818-400-30), no clear benefit was found. This data has been omitted for lack of space. We further report that with a pre-selection fraction of 0.3, both the Chessboard and the FPP instances of Table 4 receive optimal solutions.

4 Conclusions

We tried different randomized rounding approaches for routing and covering problems. We find that randomized rounding, in particular in derandomized versions, works well also for such non-artificial instances, yielding much better results in practice than what the theoretical guarantees assure. In addition, relatively simple fine-tunings give additional gains. This indicates a fruitful direction for further research.

References

1. Ageev, A.A., Sviridenko, M.: Pipage rounding: A new method of constructing algorithms with proven performance guarantee. J. Comb. Optim. 8, 307–328 (2004)
2. Arora, S.: Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. J. ACM 45, 753–782 (1998)
3. Cornuejols, G., Fisher, M.L., Nemhauser, G.L.: Location of Bank Accounts to Optimize Float: An Analytic Study of Exact and Approximate Algorithms. Management Science 23, 789–810 (1977)
4. Doerr, B.: Generating randomized roundings with cardinality constraints and derandomizations. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 571–583. Springer, Heidelberg (2006)
5. Doerr, B., Wahlström, M.: Randomized rounding in the presence of a cardinality constraint. In: ALENEX 2009, pp. 162–174 (2009)
6. Gandhi, R., Khuller, S., Parthasarathy, S., Srinivasan, A.: Dependent rounding in bipartite graphs. In: FOCS 2002, pp. 323–332 (2002)
7. Gandhi, R., Khuller, S., Parthasarathy, S., Srinivasan, A.: Dependent rounding and its applications to approximation algorithms. J. ACM 53, 324–360 (2006)
8. Glaßer, C., Reith, S., Vollmer, H.: The complexity of base station positioning in cellular networks. Discrete Applied Mathematics 148, 1–12 (2005)
9. Karp, R.M., Leighton, F.T., Rivest, R.L., Thompson, C.D., Vazirani, U.V., Vazirani, V.V.: Global wire routing in two-dimensional arrays. Algorithmica 2, 113–129 (1987)
10. Khuller, S., Moss, A., Naor, J.S.: The budgeted maximum coverage problem. Information Processing Letters 70, 39–45 (1999)
11. Kleinberg, J.: Approximation Algorithms for Disjoint Paths Problems. PhD thesis, MIT (1996)
12. Kochetov, Y., Ivanenko, D.: Computationally difficult instances for the uncapacitated facility location problem. In: Proc. of the 5th Metaheuristic Conf., MIC 2003 (2003)
13. Kratica, J., Toic, D., Filipovi, V., Ljubi, I.: Solving the simple plant location problem by genetic algorithm. RAIRO Oper. Res. 35, 127–142 (2001)
14. L.A.N.Lorena-instancias, http://www.lac.inpe.br/~lorena/instancias.html
15. Masuyama, S., Ibaraki, T., Hasegawa, T.: Computational complexity of the m-center problems on the plane. Trans. of the Inst. of Electron. and Commun. Eng. of Japan. Sect. E 64, 57–64 (1981)
16. OR-Library, http://people.brunel.ac.uk/~mastjjb/jeb/info.html
17. Pereira, M.A., Lorena, L.A.N., Senne, E.L.F.: A column generation approach for the maximal covering location problem. Int. Trans. in Oper. Res. 14, 349–364 (2007)
18. Raghavan, P.: Probabilistic construction of deterministic algorithms: Approximating packing integer programs. J. Comput. Syst. Sci. 37, 130–143 (1988)
19. Raghavan, P., Thompson, C.D.: Randomized rounding: A technique for provably good algorithms and algorithmic proofs. Combinatorica 7, 365–374 (1987)
20. Srinivasan, A.: Distributions on level-sets with applications to approximations algorithms. In: FOCS 2001, pp. 588–597 (2001)
21. UflLib, http://www.mpi-inf.mpg.de/departments/d1/projects/benchmarks/UflLib/

The Time Dependent Traveling Salesman Problem: Polyhedra and Branch-Cut-and-Price Algorithm

Hernán Abeledo[1], Ricardo Fukasawa[2], Artur Pessoa[3], and Eduardo Uchoa[3]

[1] Department of Engineering Management and Systems Engineering, The George Washington University, Washington, DC, USA
[2] Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON, Canada
[3] Departamento de Engenharia de Produção, Universidade Federal Fluminense, Niterói, RJ, Brazil

Abstract. The Time Dependent Traveling Salesman Problem (TDTSP) is a generalization of the classical Traveling Salesman Problem (TSP), where arc costs depend on their position in the tour with respect to the source node. While TSP instances with thousands of vertices can be solved routinely, there are very challenging TDTSP instances with less than 60 vertices. In this work, we study the polytope associated to the TDTSP formulation by Picard and Queyranne, which can be viewed as an extended formulation of the TSP. We determine the dimension of the TDTSP polytope and identify several families of facet defining cuts. In particular, we also show that some facet defining cuts for the usual Asymmetric TSP formulation define low dimensional faces of the TDTSP formulation and give a way to lift them. We obtain good computational results with a branch-cut-and-price algorithm using the new cuts, solving several instances of reasonable size at the root node.

Keywords: Traveling salesman problem, integer programming, branch-cut-and-price.

1 Introduction

The Time-Dependent Traveling Salesman Problem (TDTSP) is a generalization of the Traveling Salesman Problem (TSP) where arc costs depend on their position in the tour. This work departs from a formulation by Picard and Queyranne [1], used earlier in [2] for the TSP, to define and study the TDTSP polytope. Our motivations are the following:

– The TDTSP itself is a rich problem, with a number of important applications. These include routing problems like the Traveling Deliveryman Problem (TDP), known also as the minimum latency problem, and scheduling problems such as the $1|s_{ij}|\sum C_j$.

– The formulation in [1], called here PQ, can be generalized to provide very effective formulations to be used in branch-cut-and-price algorithms for several Vehicle Routing Problem (VRP) variants [3] (including "nasty" cases, like the heterogenous fleet VRP [4]) and also complex single and multi-machine scheduling problems [5]. The TDTSP facet-defining inequalities studied in this paper can be readily generalized and used on those problems.
– The PQ formulation can be used for solving the TSP. Of course, every known valid inequality for the TSP could still be added to the PQ formulation. However, we verified that inequalities known to define facets of the TSP polytope [6] correspond to disappointingly low dimensional faces of the TDTSP polytope and are usually dominated by the newly proposed TDTSP inequalities. This means that adding TDTSP inequalities to the PQ formulation yields a TSP formulation that is potentially stronger than those usually used, at the expense of having n times more variables. Furthermore, we believe the TDTSP inequalities may be projected into more complex, yet unknown, valid inequalities for the TSP polytope. Our hope is supported by some precedents. For example, [7] derived new Symmetric TSP (STSP) facets from known Asymmetric TSP (ATSP) facets. Similarly, [8] provides another case where relatively simple facets of an extended formulation are combined and projected into new complex facets of the original formulation.

Polyhedral studies of the TSP have been very productive, both theoretically and because of their algorithmic implications. Results for the STSP are surveyed in [9] and for the ATSP in [6]. Formulations for the TDTSP have been proposed or studied in [1,10,11,12,13]. Exact algorithms for the TDTSP are presented in [10,14,15] and, for the special case of the TDP, in [16,17,18]. Different heuristic methods for the TDTSP have been proposed in [13,19]. The study of the TDP polytope was initiated in [18]. As far as we know, ours is the first investigation of the TDTSP polytope.

This paper is organized as follows. The TDTSP polytope is defined in Section 2, where its dimension is also established. Section 3 presents Admissible Flow Constraints, a family of strong inequalities, including an important subfamily of inequalities proven to define facets of the TDTSP polytope and with nice theoretical properties related to flow decomposition. Section 4 introduces Lifted Subtour Elimination Constraints, which are a new family of facet-defining inequalities. Section 5 deals with Triangle Clique Constraints. Those inequalities were already introduced in the VRP context [3], but now we show that some, and perhaps all, define facets of the TDTSP polytope. Finally, Section 6 presents a branch-cut-and-price algorithm for the TDTSP, separating the newly proposed inequalities. Due to space limitations, proofs of the mathematical results are not in this paper.

2 Preliminaries

Let $N = \{1, 2, \ldots n\}$ and let $N_0 = N \cup \{0\}$. For a set of nodes S, $K(S)$ shall denote the complete (loopless) digraph over S. It is known that there is a one-to-one correspondence between Hamiltonian tours of $K(N_0)$ and Hamiltonian paths (with free ends) of $K(N)$.

The TDTSP on a complete graph $K(N_0)$ can be modeled as an optimization problem over a layered graph (V, A), where V consists of a source node 0, a terminal node T, and intermediate nodes (i, t) for $i, t \in N$. The first index of an intermediate node (i, t) identifies vertex i of the graph $K(N)$ and the second index will represent the position of vertex i in a path between nodes 0 and T. The arc set A is composed of three types of arcs. For $i \in N$, $(0, i, 0)$ denotes an arc from node 0 to node $(i, 1)$ and (i, T, n) denotes an arc from node (i, n) to node T. Given $i, j \in N$ such that $i \neq j$ and $1 \leq t \leq n-1$, (i, j, t) will denote an (intermediate) arc from node (i, t) to node $(j, t+1)$. The third index of an arc is its *layer*. Likewise, the second index of an intermediate node identifies its node layer.

It is convenient to define $G(n)$ to be the subgraph of (V, A) induced by $V \setminus \{0, T\}$. Thus, $G(n)$ has n^2 nodes $\{(i, t) : i, t \in N\}$ and all the $n(n-1)^2$ intermediate arcs of A. A path with n vertices in $G(n)$ is of the form $\{(v_t, t) : v_t \in N, 1 \leq t \leq n\}$. Since consecutive nodes in the path are in consecutive layers, we can describe such paths by an ordered array $(v_t : t \in N)$. Such a path can be extended to a $0-T$ path of (V, A) by appending nodes 0 and T as first and last nodes, respectively. A path in $G(n)$ with node sequence $(v_t : t \in N, v_i \neq v_j \text{ for } i \neq j)$, corresponds to a permutation of the elements of N, will be called an *s-path* . A $0-T$ path of (V, A) will be also be called an s-path if it contains an s-path of $G(n)$. Clearly, there is a one-to-one correspondence between s-paths of $G(n)$ and Hamiltonian paths of $K(N)$. Similarly, an s-path of (V, A) corresponds to a Hamiltonian tour of $K(N_0)$, where nodes 0 and T both represent node 0 of $K(N_0)$.

Picard and Queyranne [1] formulated the TDTSP over (V, A) as a linear integer program with the following set of constraints, where variable $x_{i,j}^t$ indicates if arc (i, j, t) is used and N_i denotes $N \setminus \{i\}$.

$$\sum_{j \in N} x_{0,j}^0 = 1 \tag{1a}$$

$$x_{0,j}^0 = \sum_{k \in N_j} x_{j,k}^1, \ j = 1 \ldots n \tag{1b}$$

$$\sum_{i \in N_j} x_{i,j}^t = \sum_{k \in N_j} x_{j,k}^{t+1}, \ j = 1 \ldots n, t = 1 \ldots n-2 \tag{1c}$$

$$\sum_{i \in N_j} x_{i,j}^{n-1} = x_{j,T}^n, \ j = 1 \ldots n \tag{1d}$$

$$x_{0,j}^0 + \sum_{t=1}^{n-1} \sum_{i \in N_j} x_{i,j}^t = 1, \ j = 1 \ldots n \tag{1e}$$

$$x \geq 0 \text{ and integer} \tag{1f}$$

Lemma 1. *The system of equations (1a, 1b, 1c, 1d, 1e) has rank $n^2 + n$.*

We can use equations (1a) and (1d) to eliminate the $2n$ variables corresponding to arcs incident to nodes 0 and T, obtaining the following equivalent system of constraints whose solutions correspond to s-paths in $G(n)$.

$$\sum_{i\in N}\sum_{j\in N_i} x^1_{i,j} = 1 \tag{2a}$$

$$\sum_{i\in N_j} x^t_{i,j} = \sum_{k\in N_j} x^{t+1}_{j,k},\ j = 1\dots n, t = 1\dots n-2 \tag{2b}$$

$$\sum_{k\in N_j} x^1_{j,k} + \sum_{t=1}^{n-1}\sum_{i\in N_j} x^t_{i,j} = 1,\ j = 1\dots n \tag{2c}$$

$$x \geq 0 \text{ and integer} \tag{2d}$$

Lemma 2 follows from Lemma 1. Also, the removal of any single equation from (2), such as (2a), yields a full rank system of equations.

Lemma 2. *The system of equations (2a, 2b, 2c) has rank* $n^2 - n$.

Definition 1. *Let $P(n)$ be the convex hull of the incidence vectors of s-paths of $G(n)$ and refer to it as the TDTSP polytope.*

Clearly, $P(n)$ and the convex hull of s-paths of (V, A) are equivalent polytopes with the same dimension. By enumerating all integer vectors in $P(n)$, one can determine computationally that $\dim P(1) = 0$, $\dim P(2) = 1$, $\dim P(3) = 5$, $\dim P(4) = 22$, and $\dim P(5) = 60$. We establish the dimension of $P(n)$ below.

Theorem 1. *If $n \geq 5$, then dimension of $P(n) = n(n-1)(n-2)$.*

3 Admissible Flow Constraints

Let $p = (0, v_1, v_2, \dots, v_n, T)$ be a $0-T$ path in (V, A). We define a r-cycle in p as a subpath $(v_i, \dots, v_{i+r})$ such that $v_i = v_{i+r}$. Note that no path p contains 1-cycles, since A does not have arcs of type (j, j, t). Also note that integral solutions of (1) are s-paths and do not contain r-cycles. A network flow in an acyclic digraph can be decomposed as a sum of flows along paths [20]. In particular, a fractional solution satisfying equalities (1a, 1b, 1c, 1d) can be decomposed into a set of $0-T$ paths. However, these paths may contain r-cycles, for some $r \geq 2$. The Admissible Flow Constraints are devised to improve the formulation by restricting the occurrence of r-cycles.

Consider t such that $1 \leq t \leq n-2$. The flow on arc (i, j, t) should exit node $(j, t+1)$ using arcs other than $(j, i, t+1)$ to avoid creating a 2-cycle. Constraints below model this observation.

$$x^t_{i,j} \leq \sum_{k\in N\setminus\{i,j\}} x^{t+1}_{j,k}, \qquad (i, j, t) \in A, 1 \leq t \leq n-2. \tag{3}$$

Theorem 2. *If $n \geq 6$, then each constraint of (3) defines a facet of $P(n)$.*

The following lemma relies on a characterization of feasible network flow problems obtained by Gale [21] and Hoffman [22] which, if applied to balanced transportation problems on incomplete bipartite graphs, yields a generalization of Hall's marriage theorem [23].

Lemma 3. *Let S and D be the set of supply and demand nodes of a balanced transportation problem and suppose $N(R) = D$ for every subset $R \subset S$ such that $|R| = 2$. Then the transportation problem is feasible if and only if $b(v) \leq b(N(v))$ for each supply node $v \in S$.*

Theorem 3. *Let $x \in R_+^A$ satisfy constraints (1a, 1b, 1c, 1d) and (3). Then x can be decomposed into flows along $0-T$ paths such that none of them contains a 2-cycle.*

Inequalities (3) may be aptly called *2-cycle elimination constraints*. The elimination of larger r-cycles by means of inequalities appears to be much more difficult, even for $r = 3$. Nevertheless, the following generalization of those inequalities proved to be a rich source of strong cuts.

Definition 2. *Let X be a connected set of vertices of $G = (V, A)$ not containing vertices in $\{0, T\}$. If $e \in \delta^-(X)$, define $C(X, e) \subseteq \delta^+(X)$ as the set of leaving arcs that are* admissible *for e with respect to X: those arcs f that belong to an s-path entering X at e and leaving X for the first time at f. For a set $E \subseteq \delta^-(X)$, define $C(X, E) \subseteq \delta^+(X)$ as $\cup_{e \in E} C(X, e)$. For a given X and E, the following valid inequality is called an Admissible Flow Constraint (AFC):*

$$\sum_{e \in E} x_e \leq \sum_{f \in C(X,E)} x_f \tag{4}$$

Definition 3. *Let $((i, t), (u_1, t+1), \ldots, (u_{r-1}, t+r-1), (i, t+r))$ be a minimal r-cycle in G. The AFCs where $X = \{(u_1, t+1), \ldots, (u_{r-1}, t+r-1)\}$ and $E = \{(i, u_1, t)\}$ are called* r-cycle elimination constraints.

Computational experiments and partial results not stated here support the conjecture that all r-cycle elimination constraints are facet-defining. The more general AFCs are usually not facet-defining, but are still interesting because: (i) there are AFCs that are not dominated by r-cycle elimination constraints; (ii) for a fixed set X they can be separated (finding the best set E) in polynomial time as a min-cut problem; and (iii) they proved to be very useful in practice.

4 Lifted Subtour Elimination Constraints

The classical Subtour Elimination Constraints (SECs) [24] are known to define facets of the STSP polytope [25] and also of the ATSP polytope [9]. SEC inequalities can be expressed in terms of the TDTSP variables as follows:

$$\sum_{j \in S} x_{0,j}^0 + \sum_{t=1}^{n-1} \sum_{i \notin S} \sum_{j \in S} x_{i,j}^t \geq 1, \qquad S \subset N, |S| > 1. \tag{5}$$

Eliminating the variables of arcs not in $G(n)$, we obtain the equivalent inequalities:

$$\sum_{i \in S} \sum_{j \in N_j} x_{i,j}^1 + \sum_{t=1}^{n-1} \sum_{i \notin S} \sum_{j \in S} x_{i,j}^t \geq 1, \qquad S \subset N, |S| > 1. \tag{6}$$

SECs may define quite low-dimensional faces of the TDTSP polytope. Lifting them provides a much stronger family of valid TDTSP inequalities that we call *Lifted Subtour Elimination Constraints* (LSECs):

$$\sum_{i\in S}\sum_{j\in N_j} x_{i,j}^1 + \sum_{t=1}^{n-|S|}\sum_{i\notin S}\sum_{j\in S} x_{i,j}^t \geq 1, \qquad S\subset N, |S|>1. \tag{7}$$

The above inequality states that an s-path $\{v_t : t\in N\}$ must satisfy $v_1 \in S$ or $\{v_k, v_{k+1} : v_k \notin S, v_{k+1}\in S, 1\leq k\leq n-|S|\}$. That is, an s-path either starts at a vertex in S or it must enter S no later than layer $n-|S|$. This constraint is valid because an s-path entering S for the first time after layer $n-|S|$ will not be able to cover all elements of the set S. Similarly, the inequality below states that an s-path either ends at a vertex in S or leaves S at layers greater or equal to $|S|$. This is a valid constraint because an s-path that exits the set S before arc layer $|S|$ will not have covered the set S completely and, thus, must return to it.

$$\sum_{t=|S|}^{n-1}\sum_{i\in S}\sum_{j\notin S} x_{i,j}^t + \sum_{j\in S}\sum_{i\in N_j} x_{i,j}^{n-1} \geq 1, \qquad S\subset N, |S|>1. \tag{8}$$

Let $\bar{G}(n)$ be the graph obtained from $G(n)$ by reversing all its arcs and the order of the node layers. Clearly, each s-path in $G(n)$ corresponds to a unique s-path in $\bar{G}(n)$. Note that constraint (8) can be viewed as a constraint of type (7) for the s-paths of the graph $\bar{G}(n)$, using the same set S in both inequalities. We can conclude that inequality (7) for a fixed set S defines a facet of $P(n)$ if and only if inequality (8), for the same set S, defines a facet of $P(n)$.

Lemma 4. *Inequality* (7) *defines a facet of $P(n)$ if and only if inequality* (8) *also does.*

Our main result for this section establishes that lifted subtour elimination constraints define facets. Its proof relies on a double induction.

Theorem 4. *If $n\geq 6$ and $3\leq |S|\leq n-3$, then constraint* (7) *defines a facet of $P(n)$.*

5 Triangle Clique Constraints

A well-known way of deriving strong cuts for binary integer programs is by analyzing their variable incompatibility graph. This graph has a vertex for each binary variable and an edge for each pair of variables that are incompatible, i.e., they can not have both value 1 in any solution. As each solution must induce an independent set in this graph, known facets of the independent set polytope, like clique and odd-hole inequalities [26], yield potentially strong cuts. This approach can not be used on the STSP, since any pair of edge variables can appear in some tour. However, the arc variables in the ATSP define an interesting incompatibility graph. While no clique cuts exist, in fact they are dominated by degree constraints or SECs, the facet-defining Odd Closed Alternating Trail Constraints correspond to odd-holes in the incompatibility graph. The arc-time variables in the TDTSP provide an even richer incompatibility graph, where even simple cliques provide new families of facet-defining cuts.

Let $S \subset N$ satisfy $|S| = 3$ and consider two arcs (i,j,t), (i',j',t') of $G(n)$ such that $(i,j),(i',j') \in A(S)$. Note that these two arcs are compatible if and only if they are adjacent and do not form a 2-cycle. Since few such pairs of arcs are compatible, it is more convenient to work over the compatibility graph, the complement of the incompatibility graph. Given $S = \{i,j,k\} \subset N$, let $\mathcal{G}(\mathrm{S}) = (\mathcal{V},\mathcal{E})$ be the compatibility graph associated to S, where each vertex of $\mathcal{V}$ is an arc (i,j,t) of $G(n)$ with $(i,j) \in A(S)$ and each edge of $\mathcal{E}$ is a compatible pair $\{(i,j,t),(j,k,t+1)\}$. An independent set $\mathcal{I} \subset \mathcal{V}$ is a maximal set of vertices in $\mathcal{G}$ which are all pairwise incompatible. It is clear that the following inequality is valid:

$$\sum_{(i,j,t)\in\mathcal{I}} x^t_{i,j} \leq 1 \tag{9}$$

These constraints were proposed in [3] for a more general setting and were named *Triangle Clique Constraints*. In particular, [3] describes an efficient pseudo-polynomial separation procedure (which is polynomial when restricted to the TDTSP) and demonstrates the usefulness of these constraints for solving heterogeneous vehicle routing problems with a branch-cut-and-price algorithm. We prove here that constraints (9) define facets of the TDTSP polytope when $\mathcal{I}$ has a certain regular structure, and conjecture that this result remains true for all triangle clique inequalities.

The independence sets we consider here induce bipartite subgraphs on alternating layers of $G(n)$, where each subgraph is isomorphic to $(S, A(S))$. Let $A(S,t) = \{(i,j,t) : (i,j) \in A(S)\}$, we call the following four cases of independence sets *alternating*.

1. For n even, $\mathcal{I} = \bigcup_{k=1}^{n/2} A(S, 2k-1)$.
2. For n even, $\mathcal{I} = \bigcup_{k=1}^{(n/2)-1} A(S, 2k)$.
3. For n odd, $\mathcal{I} = \bigcup_{k=1}^{(n-1)/2} A(S, 2k-1)$.
4. For n odd, $\mathcal{I} = \bigcup_{k=1}^{(n-1)/2} A(S, 2k)$.

Lemma 5. *Let $n \geq 7$. Let $S \subset N$ and $|S| = 3$. Let $\mathcal{I}$ be an alternating independence set corresponding to S. Let $a = (i,j,s)$ and $b = (k,l,t)$ be two compatible arcs such that $k,l \notin S$ and either $|t-s| = 1$ or $\{s,t\} = \{1,n-1\}$. Then there exists an s-path containing $a = (i,j,s)$ and $b = (k,l,t)$ which also contains exactly one arc in $\mathcal{I}$.*

Theorem 5. *If $n \geq 7$ and $\mathcal{I} \subset \mathcal{V}$ is an alternating independence set, then (9) defines a facet of $P(n)$.*

6 Branch-Cut-and-Price Algorithm

The main drawback of working directly with the PQ formulation is its large size, $O(n^3)$ variables and $O(n^2)$ constraints. However, an equivalent reformulation in terms of $O-T$ paths in (V, A) can be handled in an effective way. Number all possible $O-T$ paths from 1 to p. Define $q_{i,j}^{t,l}$ as a binary coefficient indicating whether arc (i, j, t) appears in l-th $O-T$ path, and λ_l as the positive variable associated to that path.

$$\text{Minimize } \sum_{l=1}^{p}\Big(\sum_{(i,j,t)\in A} q_{i,j}^{t,l} c_{i,j}^{t}\Big)\lambda_l \tag{10a}$$

$$\text{S.t.}$$

$$\sum_{l=1}^{p}\Big(\sum_{(i,j,t)\in A} q_{i,j}^{t,l}\Big)\lambda_l = 1 \; j = 1, \ldots, n \tag{10b}$$

$$\lambda \geq 0 \text{ and integer} \tag{10c}$$

The linear relaxation of this formulation can be efficiently solved by column generation, since the pricing subproblem consists in finding shortest $O-T$ paths in (V, A). This can be done in $O(n^3)$ time by dynamic programming. Stronger linear relaxations can be obtained by only pricing paths without r-cycles, for small values of r. Changing the dynamic programming procedure in order to avoid paths with 2-cycles is simple and only adds a small factor to the pricing time. On the other hand, pricing paths without larger r-cycles is much more complex. The best known algorithm has a complexity of $O(r!r^2n^3)$ [27]. Usually, this is only practical for $r \leq 4$.

A fractional solution of (10) can be translated into a fractional solution of (1). Cuts, like those presented in the previous sections, can then be separated, translated back to the space of the λ variables and added to the linear relaxation of (10) (as explained, for example, in [3]). Embedding this column and cut generation scheme within a branch-and-bound method yields a Branch-Cut-and-Price (BCP) algorithm.

Bigras, Gamache and Savard [14] recently implemented a BCP algorithm for the TDTSP that also uses formulation (10), pricing paths without 4-cycles. The main difference between their BCP and the one presented here lies in the cutting part. They separate families of TSP cuts (using procedures from Concorde [28]) and also non-structured clique cuts obtained by explicitly building the incompatibility graph and looking for maximum weighted cliques in it (using the CLIQUER package [29]). In contrast, our BCP separates only the specific TDTSP cuts presented in the previous section as follows:

– The proposed AFC separation is based on the flow decomposition of a fractional solution into $O-T$ paths. In a BCP context this decomposition comes directly from the fractional solution of (10). For each path in the decomposition, all minimal r-cycles are identified. For an r-cycle $((i, t), (u_1, t + 1), \ldots, (u_{r-1}, t + r - 1), (i, t + r))$, we try to separate AFCs in three different ways:

1. We check if the r-cycle elimination AFC for $X = \{(u_1, t+1), \ldots, (u_{r-1}, t+r-1)\}$ and $E = \{(i, u_1, t)\}$ is violated.
2. As mentioned in Section 3, for a fixed X we can find the set $E \subseteq \delta^-(X)$ leading to the most violated AFC or show that no AFC is violated by solving a max-flow min-cut problem. This is done by setting a bipartite network where one side has one vertex for each arc in $\delta^-(X)$ and the other side has one vertex for each arc in $\delta^+(X)$. There is an arc joining each $e \in \delta^-(X)$ to each arc in $C(X, E)$. All those arcs receive infinity capacity. An additional source vertex s is linked to vertices $e \in \delta^-(X)$ by arcs with capacity equal to the fractional value of e. In a similar way, arcs $f \in \delta^+(X)$ are linked to a target vertex t by arcs with capacity equal to the fractional value of f. It can checked that a violated AFC over X exists only and only if the max $s-t$ flow in that network has value strictly lesser than one. The second AFC separation applies this procedure to set $X = \{(u_1, t+1), \ldots, (u_{r-1}, t+r-1)\}$.
3. The third AFC separation applies the above procedure to set $X = \{(v, t) : v \in \{u_1, \ldots, u_{r-1}\}, t = 1, \ldots, n\}$.

– We do not know if LSECs can be separated in polynomial time. Our current separation is based on a Mixed-Integer Program model that is reasonably effective in practice.
– Triangle cliques are separated in $O(n^3)$ time by the dynamic programming procedure proposed in [3].

An additional element of the proposed BCP is the use of reduced cost fixing to eliminate arcs from formulation (1). The default parameter is pricing paths without 4-cycles. The code is written in C++ and was implemented over the Coin-Bcp framework, version 1.2.2, and used the Coin-LP solver, version 1.10.0 [30]. The experiments were conducted on a machine with an Intel Core 2 Duo 3.06Ghz processor.

Even tough the proposed algorithm is devised for general TDTSPs, all our tests were performed in TDP instances taken from the TSPLIB [31]. In those instances, the cost of an arc (i, j, t) is defined as $(n-t) \cdot d(i, j)$, where $d(i, j)$ is taken from a distance matrix. This allows direct comparisons with a larger literature, as there are relatively few articles providing computational results for non-TDP instances. Those TDP instances are much harder than their TSP counterparts - the only algorithm able to obtain optimal solutions for instances with $n > 50$ is the combinatorial branch-and-bound proposed in 1993 by Fischetti, Laporte and Martello [16]. Comparisons with the results published in that paper would be meaningless, due to the disparity between machines after almost two decades of computer hardware development. Fortunately, the authors kindly provided us with their code, so we could compare its performance with that of the proposed BCP on the same machine and on the same instances.

Table 1 reports the results of those comparisons. Columns Root LB, Nodes and Time represent the lower bound at the root node (bold values are optimal), the number of nodes and the total time to solve the instance for both our code and Fischetti et al.[16]'s (column FLM93). The fast bound computations of Fischetti

Table 1. Comparison with the branch-and-bound from [16]

Instance	OPT	Our BCP			FLM93		
		Root LB	Nodes	Time	Root LB	Nodes	Time
bayg29	22230	**22230**	1	85	20374	5535	0.5
bays29	26862	**26862**	1	3	24268	5194	0.5
berlin52	143721	**143721**	1	85	130292	585347	327.7
eil51	10178	**10178**	1	15	9658	526111	75.4
eil76	17976	**17976**	1	145	16894	≈ 4M	3515.7
brazil58*	512361	**512361**	1	15655	435016	-	-

Table 2. Comparison with the results in [14]

Instance	OPT	BGS08 BCP		Our BCP	
		Time	Nodes	Time	Nodes
gr17	12994	3	1	0.5	1
gr21	24345	10	1	1	1
gr24	13795	15	1	1.5	1
bays29	22230	76	16	4	1
bayg29	26862	191	51	85	1

Table 3. LB obtained using cuts versus eliminating more cycles in the column generation

Instance	$r = 4$, no cuts		$r = 4$, all cuts		$r = 5$, no cuts		$r = 6$, no cuts	
	Time	LB	Time	LB	Time	LB	Time	LB
bayg29	6	21824	85	**22230**	34	**22230**	1916	**22230**
bays29	3	**26862**	3	**26862**	35	**26862**	2215	**26862**
berlin52	48	141257	85	**143721**	353	142192	*	*
eil51	15	**10178**	15	**10178**	138	**10178**	5989	**10178**
eil76	93	17949	145	**17976**	452	17956.62	*	*
brazil58	178	468513	*	*	3197	490153	*	*

et al. are advantageous in smaller instances. However, as instances get larger, our stronger lower bounds give our code an advantage (in fact all instances were solved at the root). We include in our table the results of our code for instance brazil58, for which FLM93 was unable to finish solving within a time limit of 21,000 seconds. Solving that harder instance within that time limit required a special parameter setting (that is why this instance is marked with a star), pricing routes without 5-cycles. We also compare our proposed BCP with the best LP based method in the literature, the BCP described in [14].

Table 2 reports the results for all TSPLIB instances for which [14] ran their experiments. Their times were obtained in an Intel Pentium 4 3.4 GHz machine. It is worth noting that [14] also eliminate 4-cycles in the pricing, we are basically comparing cut efficacy. We also performed experiments to analyze the effect of using the proposed TDTSP cuts versus the effect of forbidding cycles of higher cardinality in the column generation phase. Notice that, as mentioned before, total elimination of r-cycles by means of cuts seems hard to do, even for $r = 3$. In contrast, if such a restriction is done in the pricing phase, r-cycles are eliminated in advance. In spite of this, our experiments show that, at least in terms of bounds, our cuts can achieve the same (or better) bounds than r-cycle elimination within a more reasonable time for larger values of r. Table 3 illustrates this effect (a star in the table represents that the run did not finish running after 2h). In those instances, it is clear that the best times are achieved by using 4-cycle elimination in the pricing phase combined with cuts.

The TDTSP families of cuts proposed in this article, strong from a polyhedral point of view, appear to be also strong in practice. They were able to completely close the integrality gap for all the TDP instances tested, with n up to 76. However, making an effective use of those cuts to solve larger instances is still a challenge.

References

1. Picard, J., Queyranne, M.: The time-dependent traveling salesman problem and its application to the tardiness problem in one-machine scheduling. Operations Research 26, 86–110 (1978)
2. Vajda, S.: Mathematical Programming. Addison-Wesley, Reading (1961)
3. Pessoa, A., Poggi de Aragão, M., Uchoa, E.: Robust Branch-Cut-and-Price Algorithms for Vehicle Routing Problems. In: Golden, B., Raghavan, S., Wasil, E. (eds.) The Vehicle Routing Problem: Latest Advances and New Challenges, pp. 297–326. Springer, New York (2008)
4. Pessoa, A., Uchoa, E., Poggi de Aragão, M.: A robust branch-cut-and-price algorithm for the heterogeneous fleet vehicle routing problem. Networks 54, 167–177 (2009)
5. Pessoa, A., Uchoa, E., Poggi de Aragão, M., Freitas, R.: Algorithms over Arc-time Indexed Formulations for Single and Parallel Machine Scheduling Problems. Technical report RPEP 8(8), Universidade Federal Fluminense, Engenharia de Produção, Niterói, Brazil (2008)
6. Balas, E., Fischetti, M.: Polyhedral theory for the ATSP. In: Gutin, G., Punnen, A. (eds.) The Traveling Salesman Problem and Its Variations, pp. 117–168. Kluwer, Dordrecht (2002)
7. Balas, E., Carr, R., Fischetti, M., Simonetti, N.: New Facets of the STS Polytope Generated from Known Facets of the ATS Polytope. Discrete Optimization 3, 3–19 (2006)
8. Gouveia, L., Simonetti, L., Uchoa, E.: Modeling hop-constrained and diameter-constrained minimum spanning tree problems as Steiner tree problems over layered graphs. Mathematical Programming (2009) (Online first)
9. Groetschel, M., Padberg, M.W.: Polyhedral theory. In: Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B. (eds.) The Traveling Salesman Problem, pp. 251–305. Wiley, Chichester (1985)
10. Miranda Bront, J.J., Méndez-Díaz, I., Zabala, P.: An Integer Programming Approach for the Time Dependent Traveling Saleman Problem. In: 23rd European Conference on Operational Research (2009)

11. Fox, K., Gavish, B., Graves, S.: An N-Constraint Formulation of the (Time Dependent) Traveling Salesman Problem. Operations Research 28, 1018–1021 (1980)
12. Gouveia, L., Voss, S.: A Classification of formulations for the (time-dependent) traveling salesman problem. European Journal of Operations Research 83, 69–82 (1995)
13. Vander Wiel, R.J., Sahinidis, N.V.: Heuristics Bounds and Test Problem Generation for the time-dependent traveling salesman problem. Transportation Science 29, 167–183 (1995)
14. Bigras, L.-P., Gamache, M., Savard, G.: The Time-Dependent Traveling Salesman Problem and Single Machine Scheduling Problems with Sequence Dependent Setup Time. Discrete Optimization 5, 685–699 (2008)
15. Vander Wiel, R.J., Sahinidis, N.V.: An Exact Solution Approach for the time-dependent traveling salesman problem. Naval Research Logistics 43, 797–820 (1996)
16. Fischetti, M., Laporte, G., Martello, S.: The Delivery Man Problem and Cumulative Matroids. Operations Research 41, 1055–1064 (1993)
17. Lucena, A.: Time-Dependent Traveling Salesman Problem - The Deliveryman Case. Networks 20, 753–763 (1990)
18. Méndez-Díaz, I., Zabala, P., Lucena, A.: A New Formulation for the Traveling Deliveryman Problem. Discrete Applied Mathematics 156, 3233–3237 (2008)
19. Bentner, J., Bauer, G., Obermair, G.M., Morgensten, I., Schneider, J.: Optimization of the time-dependent traveling salesman problem with Monte Carlo methods. Physical Review E 64, 36701-1–36701-8 (2001)
20. Ford, L.R., Fulkerson, D.R.: Flows in Networks. Princeton Univerisity Press, Princeton (1962)
21. Gale, D.: A theorem of flows in networks. Pacific Journal of Mathematics 7, 1073–1082 (1957)
22. Hoffman, A.: Some recent applications of the theory of linear inequalities to extremal combinatorial analysis. In: Proceedings of Symposium in Applied Mathematics, vol. 10, pp. 113–128 (1960)
23. Hall, P.: On representatives of subsets. Journal of London Mathematical Society 10, 26–30 (1935)
24. Dantzig, G.B., Fulkerson, D.R., Johnson, S.M.: Solution of a large-scale Traveling Salesman Problem. Operations Research 2, 393–410 (1954)
25. Grotschel, M., Padberg, M.: On the Symmetric Traveling Salesman Problem II: Lifing Theorems and Facets. Mathematical Programming 16, 281–302 (1979)
26. Padberg, M.: On the facial structure of set packing polyhedra. Mathematical Programming 5, 199–215 (1973)
27. Irnich, S., Villeneuve, D.: The shortest path problem with resource constraints and k-cycle elimination for $k \geq 3$. INFORMS Journal on Computing 18, 391–406 (2006)
28. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: On The Solution Of Traveling Salesman Problems. Documenta Mathematica (Extra vol. ICM 3), 645–646 (1998)
29. Niskanen, S., Ostergard, P.R.J.: Cliquer user's guide. Helsinki University of Technology, Communications Laboratory, Technical report 48 (2003)
30. Ralphs, T.K., Ladányi, L.: COIN/BCP User's Manual (2001), `http://www.coin-or.org/Presentations/bcp-man.pdf`
31. Reinelt, G.: TSPLIB - A traveling salesman problem library. ORSA Journal on Computing 3(4), 376–384 (1991)

An Approximate ϵ-Constraint Method for the Multi-objective Undirected Capacitated Arc Routing Problem

Lucio Grandinetti, Francesca Guerriero, Demetrio Laganà, and Ornella Pisacane

Dipartimento di Elettronica, Informatica e Sistemistica
Università della Calabria, Italy
lucio@unical.it, {guerriero,dlagana,opisacane}@deis.unical.it

Abstract. The Undirected Capacitated Arc Routing Problem is a classical arc routing problem arising in practical situations (road maintenance, garbage collection, mail delivery, school bus routing, etc.) with the aim of minimizing the total transportation cost of a set of routes that service a set of required edges under capacity constraints. Most of logistic companies are interested in minimizing not only the total transportation cost, they also are focused in managing the deliveries on the edges, in such a way that the duration of the longest trip does not exceed an upper time limit, to take into account the working day duration of the drivers. Moreover, all the demands of the required edges are satisfied by considering a limited number of vehicles at the depot. In this paper, the Multi-objective Undirected Capacitated Arc Routing Problem where different and competitive objectives are taken into account simultaneously, is defined and studied. Three objectives are considered in order to: minimize the total transportation cost, the longest route (*makespan*) and the number of vehicle used to service all the required edges (i.e., the total number of routes). To find a set of solutions belonging to the optimal pareto front, an optimization-based heuristic procedure is proposed and its performance is evaluated on a set of benchmark instances.

Keywords: Multiobjective Optimization, Capacitated Arc Routing Problem, ϵ-constraint method.

1 Introduction

The Undirected Capacitated Arc Routing Problem (UCARP) is defined on a undirected graph $G = (V, E)$, where $V = \{1, \ldots, n\}$ is the set of vertices and $E = \{(i, j) : i \in V, j \in V, i < j\}$ is the set of edges with a non-negative cost c_{ij} associated to each edge, such that triangle inequalities are satisfied. Let $R \subseteq E$ be the set of required edges with a non-negative demand d_{ij}. A fleet of homogeneous vehicles, with capacity Q and located at the depot (vertex 1 of V), is used to service all the required edges by minimizing the total cost and such that: *a*) each edge is serviced only by a vehicle; *b*) the total demand of edges serviced by a route does not exceed Q. A route is defined by a set of vertices such that the first and the last vertex correspond to the depot and each pairs of consecutive vertices is linked only by one edge: $r = \{0, \dots, i, j, \dots, 0\}$, where $(i, j) \in E$, $\forall\, i,\ j \in r$, and $i \neq j$. Observe that each edge may be traversed many times in a feasible UCARP solution, while in an optimal UCARP solution the so-called *deadhead* edges (edges traversed without being serviced) are traversed not more than twice ([14]). The number of vehicles is a decisional variable or a datum of the problem. We assume to know the number of vehicles m to service all required edges. In general, a strongly lower bound $\underline{m}$ on m may be computed by solving a one-dimensional bin packing problem, where the bins are the vehicles with capacity Q, and the items correspond to the required edges. A lower bound on m is represented by $\underline{m} = \left\lceil \sum_{(i,j)\in R} d_{ij}/Q \right\rceil$. In many benchmark UCARP instances $m = \underline{m}$. The UCARP was introduced by Golden and Wong ([15]). It is NP-hard since it includes as a special case the *Undirected Rural Postman Problem* (URPP), shown to be NP-hard by Lenstra and Rinnooy Kan ([26]). In addition, Golden and Wong ([15]) have shown that approximating the optimal UCARP solution value by a factor of 1.5 is strongly NP-hard. Applications of the UCARP arise in garbage collection, snow removal, street sweeping and gritting, mail delivery, meter reading, school bus routing, etc. We refer the reader to [12], [2], and [10], for a detailed description of early lower bounds. Polyhedral studies of the UCARP and other arc routing problems are reviewed in [11] and in [4]. For surveys of recent research, see [1] and [30]. To address real size problems many heuristic and metaheuristic approaches have been recently proposed in the scientific literature. Efficient upper bounds for the UCARP have been obtained through path scanning heuristics, based on ellipse rule criterion as in [29], tabu search algorithms proposed by [18], [16] and [7], genetic and memetic algorithms introduced by [21], [22], guided local search algorithm proposed by [6], variable neighborhood search algorithm described in [17], ant colony optimization algorithm proposed by [23], improved memetic algorithms presented by [20], [31] and [32]. The Multi-objective Undirected Capacitated Arc Routing Problem (MUCARP) derives from practical applications, where companies manage a fleet of vehicles used to offer a service on a logistic network. Companies negotiate with the trade unions the hourly duration of drivers working day; therefore, they are also interested in routing vehicles in order to minimize the longest route and satisfy the requests by using the minimum number of vehicles. In general, the goal of minimizing the total transportation cost with the minimum number of vehicles is achieved by assuming $m = \underline{m}$ whereas $\underline{m}$ is feasible, while the introduction of a second objective aiming to minimize the cost of the longest route may generate a competition with the first objective, and consequentially it may affect the number of vehicles to be used to service all required edges. A bi-objective UCARP has been studied by Lacomme at al. ([24]). They proposed a Non-dominated sorted genetic algorithm framework to face the UCARP with the aim of minimizing the total cost of the trips and the cost of the longest trip (BUCARP). The authors used good constructive heuristics to seed the initial population and improved the results by embedding a local search procedure inside the algorithm. A memetic algorithm for a bi-objective and stochastic CARP has been proposed in [13]. They studied an efficient way to modify a memetic algorithm for the CARP to face two other problems: a CARP with stochastic demands and a bi-objective CARP, that aims at minimizing the total cost and the makespan. The two new algorithms are finally combined for the bi-objective case with stochastic demands. [33] proposed to solve a Multi-Objective CARP application by performing a memetic algorithm with extended neighborhood search to escape from local optimum solutions. Due to the complexity of the UCARP and its multi-objective extension, our research activity is focused on designing an efficient heuristic procedure to efficiently solve the benchmark UCARP instances in which we want to minimize the three aforementioned objectives. Our heuristic is based on a MUCARP integer programming model defined by route variables and explained in the following. The multi-objective methodology adopted for solving the MUCAP is the so-called ϵ-constraint method, already introduced in [8]. The following definitions are needed to describe the MUCARP model:

$$\begin{aligned}
\Omega &= \text{the set of feasible routes;}\\
\delta &= \text{the longest route cost;}\\
r &= \text{a feasible route of } \Omega;\\
a^r_{ij} &= \begin{cases} 1 & \text{if the required edge } (i,j) \in R \text{ is serviced by route } r \in \Omega;\\ 0 & \text{otherwise.}\end{cases}\\
c_r &= \text{the cost of route } r \in \Omega
\end{aligned}$$

Therefore, the MUCARP can be formulated as follows:
MUCARP

$$\text{Minimize } \delta,\ \sum_{r\in\Omega} c_r x_r,\ \sum_{r\in\Omega} x_r \tag{1a}$$

subject to

$$\sum_{r\in\Omega} a^r_{ij} x_r = 1, \qquad \forall\ (i,j) \in R \tag{1b}$$

$$\delta \geq c_r x_r, \qquad \forall\ r \in \Omega \tag{1c}$$

$$x_r \in \{0,1\}, \qquad \forall\ r \in \Omega \tag{1d}$$

$$\delta \in \Re_+, \tag{1e}$$

where constraints (1b) ensure that each required edge (i,j) is serviced only by a route; constraints (1c) impose that, for each selected route r, the *makespan* is greater than or equal to its cost. Finally, constraints (1d) and (1e) define the domain of the decision variables. It is important to point out that a crucial issue in the definition of model (1) is related to the strategy used to populate the set Ω. This point will be detailed in the following section. The remainder of the paper is organized as follows. In Section 2, the ϵ-constraint method to approach the MUCARP is introduced. Section 3 illustrates some preliminary computational experiments. A summary and a perspective on some implications of the computational results are given in Section 4.

2 The Solution Approach

The MUCARP studied in this paper is focused on minimizing the total transportation cost (first objective), the cost of the longest route (second objective) and finally the number of vehicle m. Note that an upper bound on m may be easily obtained by $\overline{m} = |R|$ whenever a vehicle services only a required edge. Therefore, we can assume that $\underline{m} \leq m \leq \overline{m}$, or equivalently $\left\lceil \sum_{(i,j)\in R} d_{ij}/Q \right\rceil \leq m \leq |R|$. Consequently, a possible strategy to use different numbers of vehicles is to range m from $\underline{m}$ to $\overline{m}$, so that one BUCARP problem is defined for each value of m (i.e., the number of routes is kept equal to m), with the aim of minimizing both total transportation cost and makespan.

The solution strategy is described in the following:

- For each $m = \underline{m}, \ldots, \overline{m}$ execute the following operations:
  - Solve the problem: *BUCARP*

$$\text{Minimize } \delta, \sum_{r\in\Omega} c_r x_r \tag{2a}$$

subject to

$$\sum_{r\in\Omega} x_r = m, \tag{2b}$$

$$\sum_{r\in\Omega} a_{ij}^r x_r = 1, \qquad \forall\, (i,j) \in R \tag{2c}$$

$$\delta \geq c_r x_r, \qquad \forall\, r \in \Omega \tag{2d}$$

$$x_r \in \{0,1\}, \qquad \forall\, r \in \Omega \tag{2e}$$

$$\delta \in \Re_+, \tag{2f}$$

  - Let F^m denote the set of non-dominated BUCARP solutions representing the Pareto front obtained with m vehicles.
- Merge sets F^m, where $m = \underline{m}, \ldots, \overline{m}$, by discarding all the dominated solutions and so obtaining the MUCARP Pareto front.

In the following a brief description of the ϵ-constraint method is presented. The aim of the method is to approximate the Pareto set by solving a sequence of constrained single-objective problems. In the general case, a bj-objective optimization problem (BOP) can be formulated as follows:

$$minimize f(x) = (f_1(x), f_2(x))$$
$$subject\ \ to:\ x \in X$$

where $X \subseteq \Re^n$ is the set of feasible solutions (i.e., solution space). The main idea of the ϵ-constraint method is to solve a sequence of ϵ-constraint problems $P_k(\epsilon)$, that are defined by transforming one of the objectives into a constraint. In particular, the problems to be solved (i.e., $P_1(\epsilon_2)$ and $P_2(\epsilon_1)$) can be mathematically represented as follows:

$$\begin{array}{lll} & minimize & f_1(x) \\ (P_1(\epsilon_2)) \quad s.t. & & f_2(x) \leq \epsilon_2 \\ & & x \in X \end{array}$$

$$\begin{array}{lll} & minimize & f_2(x) \\ (P_2(\epsilon_1)) \quad s.t. & & f_1(x) \leq \epsilon_1 \\ & & x \in X \end{array}$$

It is possible to show (see, [5]) that the exact Pareto front can be determined by solving ϵ-constraint problems, as long as one knows how to modify the value of ϵ. This issue in the general case of the multiobjective optimization problems has been addressed in [25] and [28]. The updating strategy of ϵ adopted in this paper has been inspired by the application of the ϵ-constraint method to the bi-objective traveling salesman problem with profits proposed by [5], and the bi-objective covering tour problem as illustrated in [19]. More formally, let OS denote the *objective space*, with $OS = \{f = (f_1, f_2) : f_i = f_i(x), \forall x \in X, i = 1, 2\}$, let $f^I = (f_1^I, f_2^I)$ be the *ideal* point, where $f_1^I = min_{f \in OS} f_1$ and $f_2^I = min_{f \in OS} f_2$, and $f^N = (f_1^N, f_2^N)$ be the *nadir* point, where $f_1^N = min_{f \in OS}\{f_1 : f_2 = f_2^I\}$ and $f_2^N = min_{f \in OS}\{f_2 : f_1 = f_1^I\}$. Let F denote the Pareto front and assuming that the value of ϵ is decreased by a constant value Δ, then the scheme of the procedure aimed at defining a sequence of ϵ-constraint problems based on a progressive reduction of the parameters ϵ_1 and ϵ_2, can be outlined as follows.

```
The ϵ−constraint Method
Step 1. Set $i = 1$, $j = 2$ or $i = 2$, $j = 1$;
Step 2. Compute the ideal point $f^I = (f_1^I, f_2^I)$ and the nadir point $f^N = (f_1^N, f_2^N)$;
Step 3. Set $F = \{(f_i^I, f_j^N)\}$ and $\epsilon_j = f_j^N - \Delta$ with $\Delta = 1$;
Step 4. While $\epsilon_j \geq f_j^I$ perform the following steps:
    1. Solve $P_i(\epsilon_j)$ and add the optimal solution value $f^* = (f_i^* = f_i(x^*), f_j^* = f_j(x^*))$ to $F$.
    2. Set $\epsilon_j = f_j^* - \Delta$
Step 5. Remove from set $F$ all the dominated solutions.
```

In the context of the BUCARP, problems $P_1(\epsilon_2)$ and $P_2(\epsilon_1)$ assume the following form:

$P_1(\epsilon_2)$

$$\text{Minimize} \sum_{r \in \Omega} c_r x_r \tag{3a}$$

subject to

$$\sum_{r\in\Omega} x_r = m, \tag{3b}$$

$$\sum_{r\in\Omega} a_{ij}^r x_r = 1, \qquad \forall\,(i,j)\in R \tag{3c}$$

$$\delta \geq c_r x_r, \qquad \forall\, r\in\Omega \tag{3d}$$

$$\delta \leq \epsilon_2, \tag{3e}$$

$$x_r \in \{0,1\}, \qquad \forall\, r\in\Omega \tag{3f}$$

$$\delta \in \Re_+, \tag{3g}$$

$P_2(\epsilon_1)$

$$\text{Minimize } \delta \tag{4a}$$

subject to

$$\sum_{r\in\Omega} x_r = m, \tag{4b}$$

$$\sum_{r\in\Omega} a_{ij}^r x_r = 1, \qquad \forall\,(i,j)\in R \tag{4c}$$

$$\delta \geq c_r x_r, \qquad \forall\, r\in\Omega \tag{4d}$$

$$\sum_{r\in\Omega} c_r x_r \leq \epsilon_1, \tag{4e}$$

$$x_r \in \{0,1\}, \qquad \forall\, r\in\Omega \tag{4f}$$

$$\delta \in \Re_+, \tag{4g}$$

As mentioned above, an important issue to be addressed concerns the definition of the set Ω. Set Ω contains all the feasible solutions generated by the *path scanning* (PS) approach based on the ellipse rule, and proposed by Santos et al. ([29]). Moreover, for all the instances not optimally solved, in terms of total transportation cost, the solutions returned from the deterministic *Tabu Search Algorithm* (DTSA) presented in [7] and provided by the authors are inserted into Ω.

3 Preliminary Computational Results

The proposed approach has been coded in JAVA (version 1.5). The $\epsilon-$constraint problems have been solved by using ILOG CPLEX library, release 10.1. The computational experiments have been carried on an Pentium, supported by Intel Xeon processor technology (X3220) and clocked at 2.4 GHz with four Gbyte

Table 1. Computational Results obtained on the First group

Problem	E_1			E_2			$E_1^{Lacomme}$		$E_2^{Lacomme}$		η_l	η
	C	δ	up_{routes}	C	δ	up_{routes}	C	δ	C	δ		
First Class												
dea7	325	69	5	389	59	7	325	68	381	61	3	4
dea17	58	17	4	64	11	6	58	15	60	13	2	4
dea19	91	19	5	94	14	6	91	15	91	15	1	2
Third Class												
dea2	339	69	6	395	59	7	339	69	395	59	3	4
dea4	287	78	4	350	64	6	287	74	350	64	3	5
dea12	275	71	4	352	59	7	275	71	297	54	4	5
dea14	458	104	7	602	93	8	458	97	547	93	4	6
dea15	542	224	7	544	128	7	544	128	544	128	1	2
dea18	127	27	5	137	18	8	127	27	135	19	4	4
Fourth Class												
dea21	55	21	3	65	18	4	55	21	63	17	2	2
Fifth Class												
dea1	316	74	5	337	63	6	316	74	337	63	3	3
dea3	275	65	5	339	59	6	275	65	339	59	4	4
dea5	377	78	6	447	64	8	377	78	447	64	6	6
dea6	298	80	5	351	64	6	298	75	351	64	4	4
dea16	100	21	5	112	17	7	100	21	112	17	3	3
dea22	121	36	5	131	20	7	121	36	131	20	5	3

RAM. In literature, three sets of standard UCARP instances exist for generally assessing the behavior of solution approaches for the single-objective version of the problem. The first set consists of 23 problems introduced in [9], the second set corresponds to 34 test problems introduced in [3], and the last set is composed by 24 benchmark instances presented in [7]. The latter instances were extracted from [27]. These instances are referred to as the DeArmon (Dea), Benavent (Val) and Eglese (Egl) instances, respectively. All these benchmarks are downloaded from the internet, at address *http : //www.uv.es/belengue/carp.html*. Preliminary results are showed below. Three subsets of benchmark instances have been used to test the efficiency of the proposed approach. Each group is composed by test problems selected on the basis of cardinality of E:

– First Group. It collects very simple benchmark instances defined by graphs with a small number of edges (Dea1, ..., Dea7, Dea12, Dea14, ..., Dea19, Dea21 and Dea22), more precisely $11 \leq |E| \leq 28$.
– Second Group. It collects various DeArmon and Benavent instances with a large number of edges (Dea20, Dea23, Val2c, Val3c); more precisely $33 \leq |E| \leq 36$.
– Third Group. It collects DeArmon and Benavent benchmark instances with a quite large number of edges (Dea10, Dea11, Dea13, Dea24, Dea25, Val1a, Val1b, Val1c, Val4d, Val6c, Val7c); more precisely $|E| > 36$.

Computational results obtained for each group are compared to the solutions presented in [24]. The number of vehicles is ranged in the interval $[\underline{m}, \lfloor \overline{m}/2 \rfloor]$.

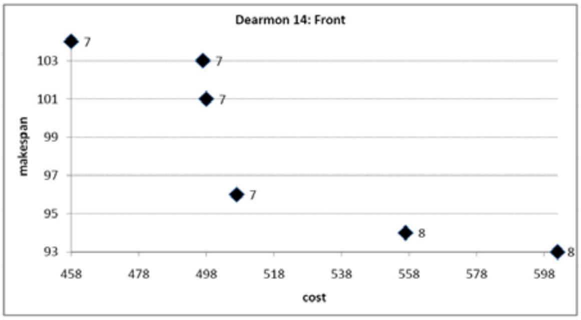


Fig. 1. Pareto Front for the instance Dearmon 14

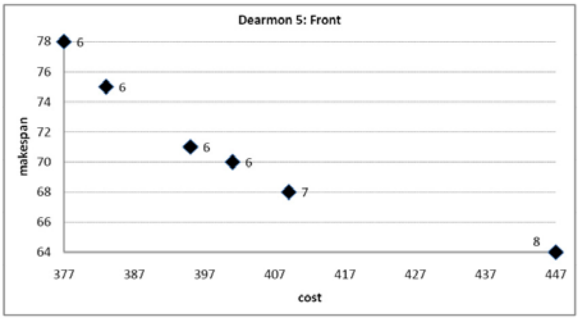


Fig. 2. Pareto Front for the instance Dearmon 5

Set Ω is populated by inserting feasible solutions detected by PS and DTSA (kindly provided by the authors). Results are presented below according to the following classification:

- the first class is defined by the BUCARP solutions showing that the proposed approach outperforms the heuristic algorithm presented in [24] in terms of makespan and number of Pareto Front solutions;
- the second class groups the BUCARP solutions in which the makespan and the total cost are better than in the solutions presented in [24], while the number of Pareto Front solutions is equal to those in the solutions found in [24];
- the third class is defined only by solutions where the number of Pareto Front points is better than in the solutions presented in [24];
- the fourth class contains solutions that are not dominated by solutions proposed in [24];
- the last class collects solutions where the makespan and the number of Pareto Front solutions are equal to those in the solutions presented in [24].

Table 2. Computational Results obtained on the Second Group

Problem	E_1 C	E_1 δ	E_1 up_{routes}	E_2 C	E_2 δ	E_2 up_{routes}	$E_1^{Lacomme}$ C	$E_1^{Lacomme}$ δ	$E_2^{Lacomme}$ C	$E_2^{Lacomme}$ δ	η_l	η
First Class												
dea23	156	28	7	174	18	10	156	30	160	22	4	5
Second Class												
val2c	457	71	8	457	71	8	463	71	463	71	1	1
Third Class												
val3c	138	28	7	141	27	8	138	27	138	27	1	3
Fourth Class												
dea20	164	33	5	172	29	6	164	33	178	27	3	2

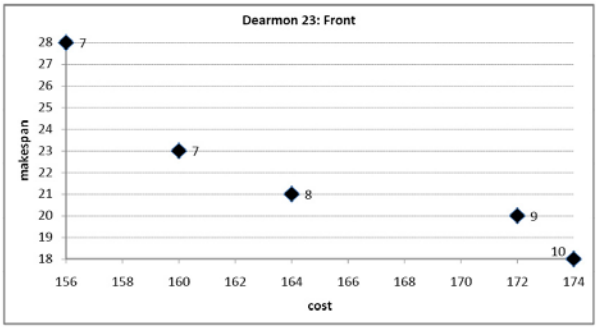


Fig. 3. Pareto Front for the instance Dearmon 23

Table 3. Computational Results obtained on the Third Group

Problem	E_1 C	E_1 δ	E_1 up_{routes}	E_2 C	E_2 δ	E_2 up_{routes}	$E_1^{Lacomme}$ C	$E_1^{Lacomme}$ δ	$E_2^{Lacomme}$ C	$E_2^{Lacomme}$ δ	η_l	η
First Class												
val1b	173	65	3	410	40	16	173	61	204	42	6	7
Second Class												
val7c	334	50	9	410	39	11	335	50	352	40	5	5
Third Class												
val1a	173	87	2	173	81	3	173	58	173	58	1	2
val1c	245	49	9	269	40	9	245	41	248	40	2	4
val4d	530	90	9	624	80	9	539	80	539	80	1	3
Fourth Class												
dea13	395	121	5	414	85	5	395	81	421	64	5	5
dea24	200	25	8	200	25	8	200	26	207	20	4	1
dea25	237	28	11	244	25	11	235	23	241	20	4	4
Fifth Class												
dea10	350	49	10	390	38	11	350	44	390	38	4	4
dea11	303	51	10	328	37	10	309	43	333	37	3	3
val6c	317	56	10	379	45	10	317	55	329	45	5	5

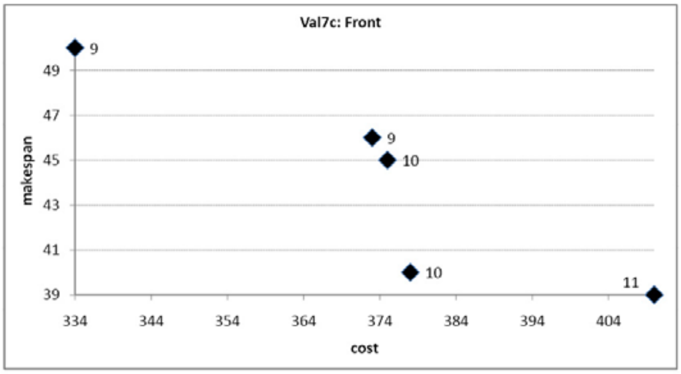


Fig. 4. Pareto Front for the instance Val7c

In the following tables, we report: the total transportation cost (first objective): C; the number of Pareto Front solutions found by [24], named by η_l, and the number of Pareto Front solutions found by the ϵ-constraint method and represented by η; the two extremes points of the Pareto domain found by [24], and referred to as $E_1^{Lacomme}$, $E_2^{Lacomme}$; the two extremes points of our Pareto domain, referred to as E_1, E_2.

Computational results show the potentiality of the proposed approach. The $\epsilon-$constraint Method developed for the MUCARP is able to reproduce the Pareto front solutions found by [24] for a quite large number of benchmark instances. For a small number of instances it improves the Pareto front obtained by [24], and shows a promising behavior for the remaining instances. We are confident we achieve good quality solutions, by improving the set Ω of feasible routes selected by the $\epsilon-$constraint model. Figures 1-4 represent the two-objective Pareto frontier for some instances. Each point represents the makespan and the total cost obtained by using a number of vehicles depicted near the point. They show a good distribution of non-dominated solutions along the Pareto front, by confirming the goodness of the proposed methodology.

4 Conclusions

In this work the multiobjective undirected capacitated arc routing problem has been addressed. In particular, a procedure to determine an approximation of the Pareto front, when the total cost, the makespan and the fleet size are minimized, is proposed and evaluated on a set of benchmarking instances. The computational results have shown that the proposed approach outperforms the genetic algorithm presented in [24], that represents the state of art approach to address the problem under investigation. Future efforts will be focused on developing an optimization-based heuristic to generate feasible routes diversifying and enlarging the search space of the MUCARP, in order to address the model to select the best solutions for each of the three objectives.

References

1. Ahr, D.: Contributions to multiple postmen problems. Ph.D. Thesis, University of Heidelberg (2004)
2. Assad, A.A., Golden, B.L.: Arc routing methods and applications. In: Ball, M.O., Magnanti, T.L., Monma, C.L., Nemhauser, G.L. (eds.) Network Routing, Handbooks in Operations Research and Management Science, pp. 375–483. North-Holland, Amsterdam (1995)
3. Benavent, E., Campos, V., Corberán, A., Mota, E.: The capacitated arc routing problem: Lower bounds. Networks 22, 669–690 (1992)
4. Benavent, E., Corberán, A., Sanchis, J.M.: Linear programming based methods for solving arc routing problems. In: Dror, M. (ed.) Arc routing. Theory, Solutions and Applications, pp. 231–275. Kluwer, Boston (2000)
5. Berube, J.F., Gendreau, M., Potvin, J.Y.: An exact ϵ-constraint method for bi-objective combinatorial optimization problems: Application to the Traveling Salesman Problem with Profits. European Journal of Operational Research 194, 39–50 (2009)
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(4), 1112–1126 (2008)
8. Chankong, V., Haimes, Y.Y.: Short-Haul Routing. In: Multiobjective Decision Making: Theory and Methodology. North-Holland, Amsterdam (1983)
9. DeArmon, J.S.: A Comparison of Heuristics for the Capacitated Chinese Postman Problem. University of Maryland, Dissertation (1981)
10. Dror, M. (ed.): Arc Routing: Theory, Solutions and Applications. Kluwer, Boston (2000)
11. Eglese, R.W., Letchford, A.: Polyhedral theory for arc routing problems. In: Dror, M. (ed.) Arc routing. Theory, Solutions and Applications, pp. 199–230. Kluwer, Boston (2000)
12. Eiselt, H.A., Gendreau, M., Laporte, G.: Arc routing problems, part II: The rural postman problem. Operations Research 43, 399–414 (1995)
13. Fleury, G., Lacomme, P., Prins, C., Sevaux, M.: A memetic algorithm for a bi-objective and stochastic CARP. In: Proceeding of the MIC 2005: The Sixth Metaheuristics International Conference, Vienna, Austria, August 22-26 (2005)
14. Ghiani, G., Laporte, G.: A branch-and-cut algorithm for the undirected rural postman problem. Mathematical Programming 87, 467–481 (2000)
15. Golden, B.L., Wong, R.T.: Capacitated arc routing problems. Networks 11(3), 305–315 (1981)
16. Greistorfer, P.: A tabu scatter search metaheuristic for the arc routing problem. Computers & Industrial Engineering 44(2), 249–266 (2003)
17. Hertz, A., Mittaz, M.: A variable neighborhood descent algorithm for the undirected capacitated arc routing problem. Transportation Science 35(4), 425–434 (2001)
18. Hertz, A., Laporte, G., Mittaz, M.: A tabu search heuristic for the capacitated arc routing problem. Operation Research 48(1), 129–135 (2000)
19. Jozefowiez, N., Semet, F., Talbi, E.-G.: The bi-objective covering tour problem. Computers and Operations Research 34, 1929–1942 (2007)

20. Ke, T., Mei, Y., Xin, Y.: Memetic Algorithm with Extended Neighborhood Search for Capacitated Arc Routing Problems. IEEE Transactions on Evolutionary Computation 13(5), 1151–1166 (2009)
21. Lacomme, P., Prins, C., Ramdane-Cherif, W.: A genetic algorithm for the capacitated arc routing problem and its extensions. In: Proceedings European WorkShops on Applications of Evolutionary Computation, Como, Italy, pp. 473–483 (2001)
22. Lacomme, P., Prins, C., Ramdane-Cherif, W.: Competitive memetic algorithms for arc routing problem. Annals of Operations Research 131(14), 159–185 (2004)
23. Lacomme, P., Prins, C., Tanguy, A.: First competitive ant colony scheme for the CARP. Research Report LIMOS/RR-04-21;2004
24. Lacomme, P., Prins, C., Sevaux, M.: A genetic algorithm for a bi-objective capacitated arc routing problem. Computers & Operations Research 33(12), 3473–3493 (2006), Part Special Issue: Recent Algorithmic Advances for Arc Routing Problems
25. Laumanns, M., Thiele, L., Zitzler, E.: An efficient, adaptive parameter variation scheme for metaheuristics based on the epsilon-constraint method. European Journal of Operational Research 169, 932–942 (2006)
26. Lenstra, J.K., Rinnooy Kan, A.H.G.: On general routing problems. Networks 6, 273–280 (1976)
27. Li, L., Eglese, R.W.: An interactive algorithm for vehicle routing for winter-gritting. Journal of the Operational Research Society, 217–228 (1996)
28. Pardalos, P.M., Siskos, Y., Zopoundis, C. (eds.): Advances in Multicriteria Analysis. Series: Non Convex Optimization and its Applications 5. Kluwer Academic Publishers, Dordrecht (1995)
29. Santos, L., Rodrigues, J.C., Current, J.R.: An improved heuristic for the capacitated arc routing problem. Computers & Operations Research 36(9), 2632–2637 (2009)
30. Wohlk, S.: A decade of capacitated arc routing. In: Golden, B.L., Raghavan, S., Wasil, E.A. (eds.) The Vehicle Routing Problem Latest Advances and New Challenges, pp. 29–48. Springer, Boston (2008)
31. Mei, Y., Tang, K., Xin, Y.: Improved Memetic Algorithm for Capacitated Arc Routing Problem. In: Proceedings of the 2009 IEEE Congress on Evolutionary Computation (CEC 2009), Trondheim, Norway, May 18-21, pp. 1699–1706. IEEE Press, Los Alamitos (2009)
32. Mei, Y., Tang, K., Xin, Y.: A Memetic Algorithm for Periodic Capacitated Arc Routing Problem. IEEE Transactions on Intelligent Transportation Systems (July 2009) (submitted)
33. Mei, Y., Tang, K., Xin, Y.: Decomposition-Based Memetic Algorithm for Multi-Objective Capacitated Arc Routing Problem. IEEE Transactions on Evolutionary Computation (October 2009) (submitted)

A Branch-and-Price Algorithm for Multi-mode Resource Leveling

Eamonn T. Coughlan[1], Marco E. Lübbecke[2], and Jens Schulz[1]

[1] Technische Universität Berlin, Institut für Mathematik,
Straße d. 17. Juni 136, 10623 Berlin, Germany
{coughlan,jschulz}@math.tu-berlin.de

[2] Technische Universität Darmstadt, Fachbereich Mathematik,
Dolivostr. 15, 64293 Darmstadt, Germany
luebbecke@mathematik.tu-darmstadt.de

Abstract. Resource leveling is a variant of resource-constrained project scheduling in which a non-regular objective function, the resource availability cost, is to be minimized. We present a branch-and-price approach together with a new heuristic to solve the more general turnaround scheduling problem. Besides precedence and resource constraints, also availability periods and multiple modes per job have to be taken into account. Time-indexed mixed integer programming formulations for similar problems quite often fail already on instances with only 30 jobs, depending on the network complexity and the total freedom of arranging jobs. A reason is the typically very weak linear programming relaxation. In particular for larger instances, our approach gives tighter bounds, enabling us to optimally solve instances with 50 multi-mode jobs.

1 Introduction

Motivated by an industrial application from chemical engineering, we study a resource leveling problem, which was recently introduced as *turnaround scheduling problem* [MMS10]. In turnaround scheduling, for inspection and renewal of parts, plants are shut down, disassembled, and rebuilt, so there is a partial ordering of jobs to be done. The time horizon and the number of workers hired for each job determine production downtime and working cost, the two of which are conflicting in a time-cost tradeoff manner. Once a time horizon is fixed, the problem turns into a resource leveling problem, on which we focus in this paper.

Different types of renewable resources are given, each associated with *availability periods* which can be thought of as working shifts. Besides the actual scheduling of jobs, the task is to decide how many workers need to be assigned to each job such that working costs are minimized, that is, we must determine a minimum amount of resources needed. We break down the granularity of planning, so that each job needs exactly one resource, possibly several units of which.

Heuristics, rather than exact methods, are prominent for solving such complex scheduling problems. This is also due to the fact that mixed integer programming formulations for scheduling problems in general, and for ours in particular, often yield very weak bounds from the linear programming relaxation.

Our Contribution. We approach the turnaround scheduling from both ends. Besides presenting a heuristic which improves on the results reported in [MMS10], we formulate a mixed integer program which is based on working shifts, and thus has an exponential number of variables. A branch-and-price algorithm to solve this model computes optimal schedules for instances with up to 50 jobs, which is a large number in this area of scheduling. In particular the derived lower bounds demonstrate that our heuristic solutions are mostly near the optimum or at least near the best solution found by exact methods within half an hour.

2 Formal Problem Description

For a recent survey on resource-constrained project scheduling (RCPSP) we refer to [HB09]. We are given a set $\mathcal{J}$ of non-preemptable jobs and a set $\mathcal{R}$ of renewable resources. Precedence constraints between jobs are given as an acyclic digraph $G = (\mathcal{J}, E)$ with $ij \in E$ iff job i has to be finished before job j starts. Each job j may be run in exactly one out of a set $\mathcal{M}_j$ of modes. Processing job j in mode $m \in \mathcal{M}_j$ takes p_{jm} time units and requires r_{jmk} units of resource $k \in \mathcal{R}$. Due to a fine granularity of planning, in our setting each job needs exactly one resource for execution, so we write r_{jm} if the resource is clear from the context.

All jobs have to finish before T, the time horizon. Each resource $k \in \mathcal{R}$ has a set $\mathcal{I}_k := \{[a_1, b_1], \ldots, [a_{k_i}, b_{k_i}]\}$ of $k_i \in \mathbb{N}$ availability periods, also called *shifts*, where $a_1 < b_1 < \ldots < a_{k_i} < b_{k_i}$. A job requiring resource k can only be executed during a time interval $I \in \mathcal{I}_k$, see Fig. 1. We use a parameter δ_{kt} which is one if resource k is available at time t, i.e., $t \in I$ for some $I \in \mathcal{I}_k$, and zero otherwise. Each resource $k \in \mathcal{R}$ is associated with a per unit cost c_k. For each resource k we have to determine the available capacity R_k such that at any time the total resource requirement of all the jobs does not exceed R_k.

We denote by $S = (S_1, \ldots, S_n)$ the vector of start times of jobs, and by $M = (m_1, \ldots, m_n)$ the vector of modes in which jobs are executed. For a given *schedule* (S, M), denote by $A(S, M, t) := \left\{ j \in \mathcal{J} : S_j \leq t < S_j + p_{jm_j} \right\}$ the set of jobs *active* at time t. The amount $r_k(S, M, t) := \sum_{j \in A(S,M,t)} r_{jm_jk}$ of resource k used at time t must never exceed the provided capacity. Thus, we obtain resource constraints with calendars: $r_k(S, M, t) \leq R_k \cdot \delta_{kt}$, $\forall k \in \mathcal{R}$, $\forall t$. Besides this *resource feasibility* a feasible schedule must obey *precedence feasibility*, i.e., $S_i + p_{im_i} \leq S_j$ for all $ij \in E$.

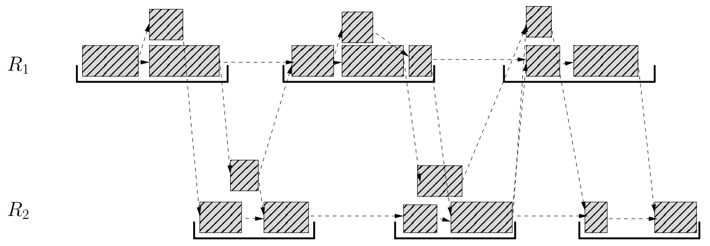


Fig. 1. Schematic representation of turnaround scheduling with two resources

Following the extended $\alpha|\beta|\gamma$-classification scheme [BDM+99], we consider $\mathit{MPSm}, \infty|\mathit{prec, shifts}|\sum c_k \cdot \max r_k(S, M, t)$ for multi-mode project scheduling with m renewable resources of unbounded capacity, with precedence constraints and working shifts, with the objective to minimize the total *resource availability cost*, i.e., minimizing $\sum_{k \in \mathcal{R}} c_k \cdot R_k$.

Related Work. Turnaround scheduling comprises project scheduling with calendars, multi-mode scheduling, and resource leveling; see [MMS10] for an industrial application. The zoo of scheduling problems is large, and we give an idea only of the most related problems. Makespan minimization is a classical scheduling goal. Lower bound schemes for this objective are presented in [BK00], where column generation is employed to solve a relaxed problem, allowing preemption and precedence constraints formulated as disjunctions. A variable represents a set of jobs selected to run at a certain point in time. For the case of generalized precedence constraints, [BC09] derive lower bounds by relaxing resource constraints for jobs which are not precedence related. This allows a dynamic programming approach on a modified activity-on-nodes network. In contrast to this regular objective function, i.e., being non-decreasing in the completion time of the jobs, a measure of the variation of resource utilization, e.g., $f(r_k(S, t))$ is not regular; see e.g., [NSZ03].

The resource leveling problem with single-modes per job, which is denoted by $\mathit{PS}|\mathit{temp}|\sum c_k \max r_k(S, t)$ with general temporal constraints has been considered earlier under the name *resource investment problem*. The special case without generalized precedence constraints $\mathit{PS}|\mathit{prec}|\sum c_k \max r_k(S, t)$ has been considered e.g., by [Dem95] and [Möh84]. They competed on the same instance set which contained about 16 jobs and four resources, with a time horizon between 47 and 70. Further computational studies were done containing 15 to 20 jobs and four resources. In the same setting, [DK01] propose lower bound computations, one based on Lagrangian relaxation, and one based on a column generation procedure, where variables represent schedules as in our approach. Small job sizes with 20 jobs can be handled; but for 30 jobs the Lagrangian relaxation wins against the column generation approach.

Multi-mode jobs are a key feature of turnaround scheduling. Such problems of the form $\mathit{MPS}|\mathit{prec}|C_{\max}$ have been investigated with renewable and non-renewable resources, with limited capacity, and makespan minimization, known as *multi-mode RCPSP*, see e.g., [DH02, Har01].

Previous algorithms have also taken calendars into account. Scheduling problems with fixed processing times and calendars, but without resource capacities were considered by [Zha92] who provides an exact pseudo-polynomial time algorithm (turned into a polynomial one by [FNS01]) for computing earliest and latest start times for preemptable as well as non-preemptable jobs.

For benchmarking, different problem sets are available in the PSPLib [PSP], where several variants of the RCPSP and of resource investment problems can be found. For the RCPSP single-mode case, test sets containing 60 jobs could not be solved in total by a vast number of researchers. In the multi-mode case instances with 30 jobs are not solved yet. For the resource investment problem, test sets containing 10, 20, or 30 jobs are available, but they do not contain working shifts, are in single-mode or include time-lags. On the other hand a job may need more than one resource. Even though none of these problems is suited for a direct comparison, they are similar to ours, and the mentioned instances inspired us when generating our own test set (see Sect. 5).

3 Integer Programming Formulations

For solving large-scale scheduling problems, mixed integer programming (MIP) is not considered as primary choice since the linear programming (LP) relaxations may be weak. Huge numbers of variables and constraints may result in high computation times and memory failures for solving only the LP relaxation. For the remainder of the paper, we need to assume the reader be familiar with MIP solving [Ach09] and branch-and-price [DL05].

3.1 Obstacles of Integer Programming for RCPSP

One of the most prominent models for the RCPSP was introduced by [PWW69]. Their formulation adapted to resource leveling looks as follows:

$$\min \sum_k c_k \cdot \bar{R}_k \tag{1}$$

$$\text{s.t.} \quad \sum_t x_{jt} = 1 \qquad \forall\, j \in \mathcal{J} \tag{2}$$

$$\sum_t t \cdot x_{jt} = S_j \qquad \forall\, j \in \mathcal{J} \tag{3}$$

$$S_i + p_i \leq S_j \qquad \forall\, ij \in E \tag{4}$$

$$\sum_{j \in \mathcal{J}} \sum_{\substack{\tau = t - p_j + 1 \\ \tau \geq 0}}^{t} r_{jk} \cdot x_{j\tau} \leq \bar{R}_k \qquad \forall\, k\, \forall\, t \tag{5}$$

$$x_{jt} \in \{0, 1\} \qquad \forall\, j\, \forall\, t \tag{6}$$

Binary variables x_{jt} model whether job j starts at time t or not. Each job j must start exactly once (2). The start times S_j are linked to the binary variables x_{jt} in (3). Also precedence constraints (4), and resource capacity constraints (5) are linear. The integer program decides on the resource capacities $\bar{R}_k$ for each resource k, such that the total resource availability cost is minimized.

Depending on several factors, such as network complexity (the density of G) or the time discretization considered, this formulation may yield good or poor lower bounds. We show an example LP solution, where even an optimal assignment of the start time variables S_j does not yield an optimum solution value.

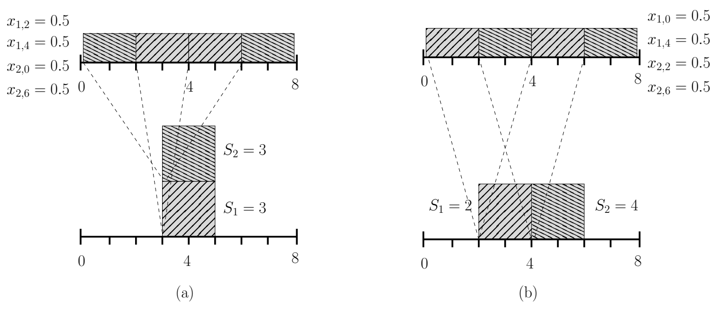


Fig. 2. Two schedules where primal and dual bounds do not match, even though in (b) the start times are optimal

Example. In Fig. 2 two jobs are given, each with processing time 2 and resource demand 2. The integrality gap may be large depending on the jobs' parameters. In Fig. 2(a) the two jobs are running in parallel using as many resource units as possible, according to their start times $S_1 = S_2 = 3$, but a corresponding LP solution may only yield a lower bound of 1 since binary variables are maximally fractional. Even for an optimal start time solution $S_1 = 2$ and $S_2 = 4$, as in Fig. 2(b), the dual bound may not be tight. The convex combination (3) of start times of a job which are far apart from each other to form S_j gives us irrelevant information about the schedule and we lose all structure in the model.

Furthermore, branching on the binary variables leads to a confusing result. In Fig. 2(b), when branching on $x_{1,0}$, job 1 which starts at time $S_1 = 2$ now would be scheduled at $t = 0$ or not. Thus, a more sophisticated branching rule that is aware of the linking of continuous variables S_j and binary variables x_{jt} and that prefers branching on the start time variables S_j is desirable. Therefore, natural branching candidates are start time variables the corresponding binary variables of which are fractional. This conforms branch-and-price theory.

3.2 Master Problem: A Model Based on Shift Configurations

In order to reduce the effects of "losing the timing information" just described, we propose a model which exploits the problem structure by decomposing the time horizon. Based on the calendar for each resource type, every working shift represents a smaller subproblem for which sub-schedules are generated independently. These sub-schedules are linked by constraints ensuring that exactly one is chosen for each working shift. For each such sub-schedule, or *configuration*, we introduce a binary variable x_ξ which indicates whether configuration ξ is chosen. We abbreviate $j \in \xi$ to express that job j is executed in the shift corresponding to ξ. Every configuration ξ has an associated resource capacity R_ξ and start times $S_{j\xi}$ and completion times $C_{j\xi}$ for each $j \in \xi$. Note that the mode of each job is determined by the start and completion times. The model reads:

$$\min \sum_k c_k \cdot \bar{R}_k \tag{7}$$

$$\text{s.t.} \quad C_i \leq S_j \qquad \forall\, ij \in E \tag{8}$$

$$S_j = \sum_{\xi : j \in \xi} S_{j\xi} x_\xi \qquad \forall j \in \mathcal{J} \tag{9}$$

$$C_j = \sum_{\xi : j \in \xi} C_{j\xi} x_\xi \qquad \forall j \in \mathcal{J} \tag{10}$$

$$\sum_{\xi : \xi \in I} R_\xi x_\xi \leq \bar{R}_k \qquad \forall k\, \forall\, I \in \mathcal{I}_k \tag{11}$$

$$\sum_{\xi : j \in \xi} x_\xi = 1 \qquad \forall j \in \mathcal{J} \tag{12}$$

$$x_\xi \in \{0, 1\} \qquad \forall \xi \tag{13}$$

Each job is executed in exactly one configuration by (12). The start and completion times for each job are computed from the chosen configurations via the linking constraints (9) and (10). Constraints (8) model the precedence relations between jobs. These could be directly expressed by substituting S_j and C_j from the linking constraints, but (9) and (10) are helpful in the upcoming pricing problem where they penalize or encourage certain start or completion times of jobs. Constraints (11) link resource consumptions to the capacities.

3.3 Column Generation: Pricing Problem

Since the number of feasible configurations is exponential in the number of jobs, we solve the relaxation by column generation embedded into a branch-and-bound scheme [DL05]. By defining dual variables s_j, c_j, ρ, π_j for constraints (9), (10), (11), and (12), respectively, we obtain a pricing problem for each shift I.

$$\max \quad \sum_j \pi_j X_j - \sum_j c_j C_j + \sum_j s_j S_j - \rho R$$

$$\text{s.t.} \quad X_j = \sum_{m,t} x_{jmt} \qquad \forall j \in J \tag{14}$$

$$S_j = \sum_{m,t} t x_{jmt} \qquad \forall j \in J \tag{15}$$

$$C_j = \sum_{m,t} (t + p_{jm}) x_{jmt} \qquad \forall j \in J \tag{16}$$

$$\sum_{j \in \mathcal{J}} \sum_m \sum_{\substack{\tau = t - p_{jm} + 1 \\ t \geq 0}}^{t} r_{jm} x_{jm\tau} \leq R \qquad \forall t \in I \tag{17}$$

$$x_{jmt} \in \{0, 1\} \qquad \forall j \in J, m, t \tag{18}$$

$$X_j \in \{0, 1\} \qquad \forall j \in J \tag{19}$$

This is a scheduling problem with non-regular objective function where a new configuration ξ for a specific shift I is generated. It must be decided, see constraint (14), whether a job corresponding to the binary decision variable X_j is running in this shift or not, and if so, which mode $m \in \mathcal{M}_j$ is used. Constraints (15) and (16) fix the start and completion times of jobs according to the chosen mode assignment. Resource capacity constraints (17) have to be satisfied such that the total profit is maximized. The objective value is increased by π_j if a job is taken into the configuration and by multiples of s_j and c_j if it has late start times and early completion times. With each unit increase of resource capacity the objective value decreases by a factor of ρ.

The pricing problem is NP-hard as it contains a leveling problem. This can be seen when all π_j are set to a value large enough to ensure that each job must be scheduled, and by setting s_j and c_j to zero for all j.

4 Branch-and-Price Algorithm

A solution to the original problem is given by the resource capacities R_k, and an assignment of start times S_j and completion times C_j for each job j. The mode is given by the closest resource allocation, such that $p_{jm_j} \leq C_j - S_j$. We refer to R_k, S_j, C_j as *original variables* since these correspond to original decisions.

The variables of the master problem x_ξ that symbolize configurations of different shifts are generated by the pricing problem and whenever a heuristic finds a feasible solution. We refer to these variables as *master variables*.

4.1 Branching Scheme

Experiments revealed as branching order R_k, S_j, and then C_j. Start and completion time variables are considered as branching candidates, only if any corresponding binary configuration variable is fractional. After the resource capacities are fixed in the search tree, a start time variable S_j with LP solution value $S_j^\star$ is selected. Completion times are handled accordingly. The node is split into two subnodes with $S_j \leq \lfloor S_j^\star \rfloor$, and $S_j \geq \lceil S_j^\star \rceil$, respectively. This scheme is used together with some propagation rules to overcome the smeared LP solutions and to create a more balanced branching tree.

4.2 Propagation

For scheduling problems a large variety of propagation algorithms is known. *Edge-finding* is a constraint programming technique concerned with deriving better bounds for earliest start and latest end times of jobs using energy arguments. The first correct algorithm, proposed in [MVH08] can be adapted to the multi-mode case, by using the minimum energy of all modes for each job, which naturally seems to give weaker bounds. This is balanced by the fact that jobs are not preemptive, may not cross shift-bounds and obey precedence constraints which enables further propagation of start and completion times.

Furthermore, propagation serves the technical purpose of communicating the branching decisions to the pricing problem.

4.3 Primal Bounds

For the master problem rounding heuristics for LP solutions are not promising, since values of binary variables may be smeared over the time horizon as in Fig. 2. To improve upper bounds we extended a leveling heuristic from [MMS10] and implemented a generic list scheduling algorithm which is used as a stand alone heuristic during the branching process as well as in the leveling procedure.

Ready scheduling heuristic. The general idea of *ready scheduling* is similar to that of list scheduling with jobs sorted by earliest start times. Jobs are divided according to the resource they need, and scheduled as soon as their predecessors are completed, if possible, thus increasing the chance to meet a given time horizon. We say a job is "ready" if all its predecessors are scheduled.

For each resource k we maintain a set of jobs $J_k = (j_{k_1}, .., j_{k_{|J_k|}})$ that use this resource and have no unscheduled predecessors, together with a lower bound t_k on the next feasible start FS_j of any job $j \in J_k$. If J_k is empty, we set t_k to ∞.

Algorithm 1. Ready Scheduling

```
   Input: Set of jobs $\mathcal{J}$ to be scheduled and max. total duration $T$
   Output: Job start times and modes, or that no solution was found.
1  for $k \in \mathcal{R}$ do
2      Let $J_k \subset \mathcal{J}$ be the set of jobs that use resource $k$, and are ready.
3      Set $t_k$ to $\min_{j \in J_k} FS_j$ .
4  Set $t := 0$
5  while $t < T$ do
6      $\ell := \operatorname{argmin}_k t_k$, and $t := t_l$ .
7      $m := \min(s, |J_\ell|)$ with $s$ a small constant and $I := \{j_{\ell_1}, .., j_{\ell_m}\}$.
8      Schedule jobs in $I$ such that $\max_{i \in I} C_i$ is minimal. .
9      Add all successors of jobs in $I$ that become ready to their respective $J_k$.
10     $J_\ell := J_\ell \setminus I$
11     Update $t_k$ for all changed $J_k$ as in Step 1.
```

The heuristic loops over t, which increases to the minimal t_k in each iteration. A subset $I \subset J_k$ of constant size s (we chose $s = 6$ in our computational studies) is scheduled so that the overall completion time is kept small (line 1). This is accomplished by trying all mode combinations for I recursively, and bounding recursion using the currently shortest feasible solution found in this way. If s is small, this can be done quickly. Finally, for each scheduled job in I, those successors which become ready, are added to the corresponding set J_k, each t_k is updated, and the next iteration begins. This process continues until all jobs are scheduled, or a makespan violation occurs.

Resource Leveling heuristic. We now describe the resource leveling heuristic by [MMS10], and how the ready scheduler ties into the framework of the leveling procedure. This heuristic uses a binary search on the capacity bounds of the resources, while greedily selecting the resource whose upper bound is to be improved in each iteration. This selection is based on a parameter μ_k, measuring how badly a resource k is leveled.

One iteration of the binary search consists of trying to find a feasible schedule for the current bounds. These bounds for the selected resource $k^\star$ are set to $(\text{UB}_{k^\star} + \text{LB}_{k^\star})/2$, $\text{UB}_{k^\star}$ and $\text{LB}_{k^\star}$ being the upper and lower bounds on the capacity of resource $k^\star$, while all other resource bounds remain fixed. We try list scheduling, and on failure fall back to ready scheduling to prove the bounds feasible. If neither ready scheduling nor list scheduling yield a feasible schedule, we consider the current upper bound for the selected resource as a new lower bound, and the next iteration begins.

Algorithm 2. Resource Leveling

```
  Input: Set $\mathcal{R}$ of resource types to be leveled, project duration $T$.
  Output: Leveled resource utilization $R_k$ for each resource type $k \in \mathcal{R}$.
1 Set $\text{LB}_k$ and $\text{UB}_k$ to initial values for each resource type $k \in \mathcal{R}$.
2 while $\exists\, k \in \mathcal{R} : \text{LB}_k < \text{UB}_k$ do
3     Choose resource type $k^\star \in \mathcal{R}$ with $\text{LB}_{k^\star} < \text{UB}_{k^\star}$ and $\mu_{k^\star}$ maximum.
4     Perform binary search using list scheduling and ready scheduling in
      order to decrease the capacity bounds of $k^\star$.
```

LP solution and ready scheduling heuristics. In each node of the branch-and-bound tree these heuristics set the maximal capacity of each resource to the LP solution value rounded up, and fix the earliest and latest start and completion times for each job to the global bounds of the corresponding variables. We perform list scheduling using the LP solution with jobs sorted by earliest completion times, and each job's mode is chosen as the one matching $C_i - S_i$ best. If no feasible solution is obtained we try ready scheduling. Both of these heuristics produce solutions that are not necessarily feasible w.r.t. the current primal bound, since resource capacities are rounded up. Regardless of this, if a feasible schedule is found new columns representing that schedule are added to the master problem, in order to reduce the total number of pricing steps.

5 Computational Study

5.1 Experimental Setup

As there is no publicly available set of instances reflecting the precise setup of our problem, we needed to compile our own. The PSPLib [PSP] guided our design. Our set is composed of three sets of job scenarios, with 10 instances each. Each job can run in 3 different modes, using 1 to 3 units of its resource, with durations ranging from 5 to 12. In the first scenario instances have 30 jobs and 60 edges,

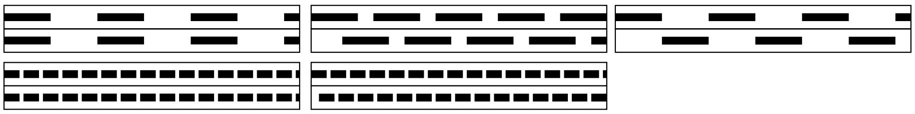


Fig. 3. Calendar configurations C1–C3 (top) and C4 and C5 (bottom) used in our test set. Black bars symbolize the temporal location of shifts.

and in the second (third) scenario instances contain 50 jobs with 70 (100) edges, respectively. The maximal width W of the precedence graph is 5 for scenario one, and 6 for the other two. These width bounds are achieved by constructing W chains of length $|\mathcal{J}|/W$, and randomly choosing the remaining edges.

There are two different resources which come in 5 calendar configurations, called C1 to C5. These calendars are described schematically in Fig. 3. In the top row, calendars C1 to C3 are shown. In each of these, the length of the shifts is 60. In C1 and C3 shift breaks are 60 units long, in C2 only 20. Both resources are available at the same time in C1, while in C3 availability periods are complementary. In C2 the second resource is offset at 40 units. Calendars C4 and C5 show shifts with length 20 and breaks having length 5. In C5 one of the resources is offset by 10. All scenarios are tested with each of the 5 different calendars. Time horizons were chosen by computing a minimal and maximal makespan heuristically using simple list scheduling, and averaging these.

Our algorithm was tested on an Intel 2.66 GHz processor, and each test run had a time-limit of 30 minutes. The implementation uses SCIP 1.2.0.6 [SCI], to perform the branch-and-bound process, with custom plug-ins for the heuristics, branching rules, and the pricer. For the standard MIP (1)–(6) we used CPLEX 12 with quad-core parallelization on a stronger machine.

Our empirical results can be seen in Tabs. 1 and 2. These tables show four columns per algorithm. The first column contains the average time in seconds to solve the instance with a limit of 1800. The second and third columns represent the number of times the algorithm reached the best known lower and upper bounds over all algorithms, denoted by "LB" and "UB." The fourth column is the number of timeouts, marked by "†." An additional last column "RL" per table shows the number of times the resource leveling heuristic reached the best known upper bound for each set of instances. The rows in each table represent the calendar configurations used.

Table 1. Comparison of the BP approach with the standard MIP ($|\mathcal{J}| = 30$)

	Branch-and-Price				B&P w/o heur.				CPLEX				RL
Calendar	time	LB	UB	†	time	LB	UB	†	time	LB	UB	†	UB
C1	446	9	9	1	435	10	10	0	1350	6	3	7	6
C2	326	9	10	1	360	9	9	1	435	9	9	2	5
C3	9	10	10	0	11	10	10	0	1	10	10	0	9
C4	81	10	10	0	81	10	10	0	494	9	10	1	8
C5	72	10	10	0	146	10	10	0	1631	3	4	7	6
Total	187	48	49	2	207	49	49	1	782	37	36	17	34

Table 2. Comparison of the BP approach with the standard MIP ($|\mathcal{J}| = 50$)

	$\lvert\mathcal{J}\rvert = 50, \lvert E\rvert = 70$								
	Branch-and-Price				CPLEX				RL
Calendar	time	LB	UB	†	time	LB	UB	†	UB
C1	1660	10	9	7	1800	6	1	10	7
C2	1662	4	7	8	1361	7	6	7	5
C3	27	10	10	0	4	10	10	0	10
C4	719	10	10	2	1800	6	0	10	5
C5	1293	8	8	5	1335	7	6	7	5
Total	1072	42	44	22	1260	36	23	34	32
	$\lvert\mathcal{J}\rvert = 50, \lvert E\rvert = 100$								
Calendar	time	LB	UB	†	time	LB	UB	†	UB
C1	1137	9	10	3	1800	3	0	10	7
C2	1224	6	9	5	1622	5	4	9	6
C3	9	10	10	0	2	10	10	0	9
C4	254	10	10	0	1800	1	0	10	8
C5	365	10	10	0	1451	5	5	6	6
Total	598	45	49	8	1335	24	19	35	36

In the first table, three different algorithms are compared on the first scenario. "Branch-and-price" refers to the general scheme presented here, while "B&P w/o heur." does not use the LP solution or ready scheduling heuristics during the branch-and-bound process. However, it does start with the solution found by the resource leveling heuristic. The "CPLEX" columns correspond to the results of the standard MIP (1)–(6). The "Branch-and-price" algorithm, as well as CPLEX and the resource leveling heuristic, are tested on the second and third scenarios in Tab. 2.

Impact of calendars. Obviously, the calendar choice has a big influence on running time. For 30 jobs we observe that with the one hour shifts the makespan generally increases as the resources overlap less. Interestingly, for the short shifts the makespan increases with more overlap (C4 instances generally have larger makespan than those in C5), yet these instances can nevertheless be solved faster. The "hardness" of a problem does not solely depend on the makespan though, as CPLEX shows the worst behavior on instances with 50 jobs and calendars C1 and C4, where the shifts are overlapping completely. This is not the case for the branch-and-price algorithm, which actually performs much better on C4 and C1, see also Tab. 2. All approaches easily solve instances using C3, because many jobs have a successor of a different resource, introducing long waiting times.

Usefulness of the heuristics. Our heuristics exploit problem specific knowledge during the branch-and-bound process, however, this does not appear to have a big impact on the number of problems solved, as can be seen in Tab. 1. Still, the average solving time decreases noticeably across all instances with the heuristics enabled. The running times of all our heuristics are negligible, and in contrast to the exact methods, the resource leveling heuristic is able to handle instances of practically relevant size, while still achieving the best found upper bound in 68% of all instances.

Influence of the network complexity. When the number $|E|$ of edges in G is increased, the makespan typically increases as well, so that the time-indexed MIP formulation grows fairly large and CPLEX generally fares better on the instances with less edges. For 50 jobs and 70 edges CPLEX finds the best lower bound 36 times and the best upper bound 23 times, in contrast to 24 best lower and 19 best upper bounds for 100 edges (Tab. 2). In contrast the branch-and-price algorithm does not suffer from this, and the computation times actually decrease on the instances with more edges.

As expected, CPLEX has considerable problems as the instances get larger. One of the main features of the turnaround scheduling problem is the presence of availability periods, which motivated this branch-and-price approach. It achieves the best results across all shift configurations, in the worst case 88% of the best upper bounds are found. In most cases lower bounds computed by CPLEX are of poor quality. Proving optimality succeeded in 80% of cases for branch-and-price , whereas CPLEX only manages 43%. Summarizing, our branch-and-price algorithm significantly outperforms the standard time-indexed MIP formulation solved by the state-of-the-art solver CPLEX on large instances.

6 Summary

Our own experiments and instance sizes as reported e.g., in [DK01] or [PSP] for scheduling problems of comparable complexity give good reasons to believe that even today instances with only 30 jobs are hard to solve to optimality. We are encouraged to further investigate branch-and-price algorithms for this type of problem, as we do not only report better results on similarly sized instances, but are also able to optimally solve some instances with up to 50 jobs.

References

[Ach09] Achterberg, T.: SCIP: solving constraint integer programs. Math. Programming Computation 1(1), 1–41 (2009)

[BC09] Bianco, L., Caramia, M.: A new lower bound for the resource-constrained project scheduling problem with generalized precedence relations. Computers and Operations Research (in press, 2009)

[BDM$^+$99] Brucker, P., Drexl, A., Möhring, R., Neumann, K., Pesch, E.: Resource-constrained project scheduling: Notation, classification, models, and methods. European J. Oper. Res. 112, 3–41 (1999)

[BK00] Brucker, P., Knust, S.: A linear programming and constraint propagation-based lower bound for the RCPSP. European J. Oper. Res. 127(2), 355–362 (2000)

[Dem95] Demeulemeester, E.: Minimizing resource availability costs in time-limited project networks. Management Sci. 41, 1590–1598 (1995)

[DH02] Demeulemeester, E.L., Herroelen, W.S.: Project Scheduling: A Research Handbook. Kluwer, Dordrecht (2002)

[DK01] Drexl, A., Kimms, A.: Optimization guided lower and upper bounds for the resource investment problem. The Journal of the Operational Research Society 52(3), 340–351 (2001)

[DL05] Desrosiers, J., Lübbecke, M.E.: A primer in column generation. In: Desaulniers, G., Desrosiers, J., Solomon, M.M. (eds.) Column Generation, pp. 1–32. Springer, Berlin (2005)

[FNS01] Franck, B., Neumann, K., Schwindt, C.: Project scheduling with calendars. OR Spektrum 23, 325–334 (2001)

[Har01] Hartmann, S.: Project scheduling with multiple modes: A genetic algorithm. Ann. Oper. Res. 102(1-4), 111–135 (2001)

[HB09] Hartmann, S., Briskorn, D.: A survey of variants and extensions of the resource-constrained project scheduling problem. European J. Oper. Res. (in press, 2009)

[MMS10] Megow, N., Möhring, R.H., Schulz, J.: Decision support and optimization in shutdown and turnaround scheduling. INFORMS J. Computing (2010) (forthcoming)

[Möh84] Möhring, R.H.: Minimizing costs of resource requirements in project networks subject to a fixed completion time. Oper. Res. 32(1), 89–120 (1984)

[MVH08] Mercier, L., Van Hentenryck, P.: Edge finding for cumulative scheduling. INFORMS J. Computing 20(1), 143–153 (2008)

[NSZ03] Neumann, K., Schwindt, C., Zimmermann, J.: Project scheduling with time windows and scarce resources. Springer, Heidelberg (2003)

[PSP] Project Scheduling Problem LIBrary, `http://129.187.106.231/psplib/` (last accessed 2010/02/01)

[PWW69] Pritsker, A.A.B., Watters, L.J., Wolfe, P.M.: Multi project scheduling with limited resources: A zero-one programming approach. Management Sci. 16, 93–108 (1969)

[SCI] Solving Constraint Integer Programs, `http://scip.zib.de/`

[Zha92] Zhan, J.: Calendarization of time planning in MPM networks. ZOR – Methods and Models for Oper. Res. 36(5), 423–438 (1992)

Experiments with a Generic Dantzig-Wolfe Decomposition for Integer Programs

Gerald Gamrath[1] and Marco E. Lübbecke[2]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
`gamrath@zib.de`
[2] Technische Universität Darmstadt, Fachbereich Mathematik,
Dolivostr. 15, 64293 Darmstadt, Germany
`luebbecke@mathematik.tu-darmstadt.de`

Abstract. We report on experiments with turning the branch-price-and-cut *framework* `SCIP` into a generic branch-price-and-cut *solver*. That is, given a mixed integer program (MIP), our code performs a Dantzig-Wolfe decomposition according to the user's specification, and solves the resulting re-formulation via branch-and-price. We take care of the column generation subproblems which are solved as MIPs themselves, branch and cut on the original variables (when this is appropriate), aggregate identical subproblems, etc. The charm of building on a well-maintained framework lies in avoiding to re-implement state-of-the-art MIP solving features like pseudo-cost branching, preprocessing, domain propagation, primal heuristics, cutting plane separation etc.

1 Situation

Over the last 25 years, branch-and-price algorithms developed into a very powerful tool to optimally solve huge and extremely difficult combinatorial optimization problems. Their success relies on exploiting problem structures in an integer program (via a decomposition or re-formulation) to which standard branch-and-cut algorithms are essentially "blind." While both, commercial and open-source solvers feature very effective generic implementations of branch-and-cut, almost every application of branch-and-price is *ad hoc*, that is, problem specific. Even though the situation improved considerably due to the availability of open-source branch-price-and-cut frameworks, implementations are often started from scratch or from previous projects. In addition, even though all concepts are reasonably easy to understand, experience and expert knowledge is still indispensable to get most out of the approach. In order to easily test new ideas, while using the current state-of-the-art, it would be much more satisfactory to have a generic implementation. This—ideally—performs a decomposition if this is likely to be promising, and takes care of branch-and-price (not to forget: -and-cut), without the user's notice and interaction—just as it is the case for generic branch-and-cut today. A future solver could terminate with the message

```
Integer optimal solution found (1209.71 sec., 2 threads)
```

```
Mixed integer rounding cuts applied: 65
Dantzig-Wolfe decomposition performed (3 subproblems)
```

While one may be sceptical about such a complete automatism (it requires at least detecting decomposable structures, and deciding how to best exploit them), a publicly available generic implementation which requires only a little user interaction is rather a matter of months than years from now. Our work is a contribution to this aim.

Related Work. There are several frameworks which support the implementation of branch-and-price algorithms like ABACUS [7], BCP [13], and MINTO [9], to name only a few. We restrict attention to (non-commercial) codes which perform a Dantzig-Wolfe decomposition of a general (mixed) integer program, and handle the resulting column generation subproblems in a generic way. François Vanderbeck has been developing important features [16,18,19,20] for his own implementation called BaPCod [17] which is a "prototype code that solves mixed integer programs (MIPs) by application of a Dantzig-Wolfe reformulation technique." Also the COIN-OR initiative (www.coin-or.org) hosts a generic decomposition code, called DIP [12] (formerly known as DECOMP), which is a "framework for implementing a variety of decomposition-based branch-and-bound algorithms for solving mixed integer linear programs" as described in [11]. The constraint programming G12 project develops "user-controlled mappings from a high-level model to different solving methods," one of which is branch-and-price [10]. As of this writing, among these projects (including ours), only DIP is open to the public (as trunk development, there is no release yet).

Our Approach. We witness a development towards generic, re-usable implementations, however, in a sense, *again* started from scratch each time, at least in terms of standard MIP techniques like preprocessing and tree management etc. This is why, in this paper, we follow the different approach in complementing an *existing*, well-accepted, and well-maintained MIP solver, namely SCIP [1]. The rationale behind this is, of course, to have the full range of MIP tools available in one solver one future day, including a generic and automatic decomposition; so we found it reasonable to start from the state-of-the-art in non-commercial MIP solving [8]. The implementation is in such a way that it benefits from improvements of the solver itself, e.g., when better branching or node selection rules become available, preprocessing or propagation are getting more effective.

2 Decomposition of Integer Programs

We wish to solve an MIP, which is called the *original* (or *compact*) problem,

$$\min\{c^t x \mid Ax \geq b,\ Dx \geq d,\ x \in \mathbb{Z}_+^n \times \mathbb{Q}_+^q\} \ . \tag{OP}$$

It exposes a *structure* in the sense that $X = \{x \in \mathbb{Z}_+^n \times \mathbb{Q}_+^q \mid Dx \geq d\}$ is a mixed integer set, optimization over which is considerably easier (computationally) than solving (OP) itself. For many problems, D can be brought into a (bordered) block-diagonal form, so that (OP) can be written as

$$\min\{\sum_k c_k^t x^k \mid \sum_k A^k x^k \geq b,\ D^k x^k \geq d_k\ \forall k,\ x^k \in \mathbb{Z}_+^{n_k} \times \mathbb{Q}_+^{q_k}\ \forall k\}\ . \quad (\text{OP}k)$$

In this case $X = X_1 \times \cdots \times X_K$ (possibly permuting variables), that is, X *decomposes* into $X_k = \{x^k \in \mathbb{Z}_+^{n_k} \times \mathbb{Q}_+^{q_k} \mid D^k x \geq d_k\}$, $k = 1, \ldots, K$, with all matrices and vectors of compatible dimensions and $\sum_k n_k = n$, $\sum_k q_k = q$. We discuss two ways of exploiting this structure when solving (OP). A very thorough exposition of advantages and disadvantages, possibilities, extensions, examples, and much more context can be found in the most recent survey [22].

Convexification. By the Minkowski-Weyl theorem we express each $x^k \in \text{conv}(X_k)$ as a convex combination of extreme points P_k of $\text{conv}(X_k)$ plus a non-negative combination of extreme rays R_k of $\text{conv}(X_k)$, with P_k and R_k finite. For ease of presentation, we assume X_k bounded, i.e., $R_k = \emptyset$. In analogy to a Dantzig-Wolfe decomposition of linear programs, we introduce a variable λ_p^k for each $p \in P_k$ and require $\sum_{p \in P_k} \lambda_p^k = 1$ (*convexity constraints*). Substituting $x^k = \sum_{p \in P_k} p\lambda_p^k$, we obtain the *extended* formulation

$$\begin{aligned}\min\{\sum_k \sum_{p \in P_k} c_p^k \lambda_p^k \mid \sum_k \sum_{p \in P_k} a_p^k \lambda_p^k \geq b,\ \sum_{p \in P_k} \lambda_p^k = 1,\ x^k = \sum_{p \in P_k} p\lambda_p^k\ \forall k,\\ x^k \in \mathbb{Z}_+^{n_k} \times \mathbb{Q}_+^{q_k}\ \forall k,\ \lambda^k \in \mathbb{Q}_+^{|P_k|}\ \forall k\}\end{aligned} \quad (\text{EPC})$$

where $c_p^k = c_k p$ and $a_p^k = A_k p$. Integrality is required on the original variables.

Discretization. For pure integer programs, i.e., $q_k = 0$ for all k, one can implicitly express x^k as an *integer* convex combination of the integer points in X_k, i.e., $x^k = \sum_{p \in X_k} p\lambda_p^k$, $\lambda_p^k \in \{0, 1\}\ \forall k$ [16]. Unifying the notation with the convexification approach, we denote the set X_k of points by P_k and obtain

$$\min\{\sum_k \sum_{p \in P_k} c_p^k \lambda_p^k \mid \sum_k \sum_{p \in P_k} a_p^k \lambda_p^k \geq b,\ \sum_{p \in P_k} \lambda_p^k = 1,\ \lambda^k \in \mathbb{Z}_+^{|P_k|}\ \forall k\}\ . \quad (\text{EPD})$$

This can be generalized to MIPs, when continuous variables are convexified [21]. Often, some or all X_k are identical, e.g., for bin packing, vertex coloring, or some vehicle routing problems. This introduces a symmetry which is avoided by aggregating (summing up) the λ_p^k variables. We choose a representative $P := P_1$, substitute $\lambda_p := \sum_k \lambda_p^k$, and add up the convexity constraints. This leads to the *aggregated extended formulation*

$$\min\{\sum_{p \in P} c_p \lambda_p \mid \sum_{p \in P} a_p \lambda_p \geq b,\ \sum_{p \in P} \lambda_p = K,\ \lambda \in \mathbb{Z}_+^{|P|}\}\ . \quad (\text{EPDa})$$

Column Generation and Branch-and-Price. For the LP relaxation of the extended problem, we drop the integrality constraints, and also omit the original variables in the convexification approach. We obtain the *master problem*

$$\min\{\sum_k \sum_{p\in P_k} c_p^k \lambda_p^k \mid \sum_k \sum_{p\in P_k} a_p^k \lambda_p^k \geq b,\ \sum_{p\in P_k} \lambda_p^k = 1,\ \lambda^k \in \mathbb{Q}_+^{|P_k|}\ \forall k\}\ . \tag{MP}$$

Since (MP) typically has an exponential number of variables, it is solved via column generation. That is, we work with a *restricted master problem (RMP)* that contains only a subset of the variables. In each node of the branch-and-bound tree, the RMP is solved to optimality, and variables with negative reduced cost are added. One iterates until no more variables are found. As the reduced cost of a variable λ_p^k is given by $\bar{c}_p^k = c_k^t p - (\pi^t A^k p + \gamma_k)$ with $(\pi^t, \gamma^t)^t$ being the optimal dual solution to the RMP, we solve, for each block $k \in [K]$, the *pricing problem* $\bar{c}_k^\star = \min\left\{\left(c_k^t - \pi^t A^k\right) x - \gamma_k \mid x \in X_k\right\}$. The LP relaxation can be strengthened by additional valid inequalities (in different ways). If the solution is still fractional, branching takes places, for convexification typically on the original variables, but see e.g., [20] for a different generic rule which does not interfere with the pricing problem and avoids symmetry.

3 Some Details on the Implementation in SCIP

Our implementation GCG (generic column generation) extends SCIP [1] which was well received in the computational mathematical programming community. While the flexible plugin-based architecture enables the user to easily implement column generation in every node of the search tree, it neither provides methods for decomposition nor does it work with original and extended problem formulations simultaneously. Our work aims at complementing SCIP in this respect, turning the branch-price-and-cut *framework* into a branch-price-and-cut *solver*.

3.1 Overview: Synchronizing Two Trees

We maintain two SCIP *instances*, one for the original, one for the extended problem (called original and extended instance, respectively). The original instance is the primary one which coordinates the solving process, the extended instance is controlled by a relaxation handler that is included into the original instance. At the moment, information about the structure of the problem has to be provided by an additional input file, that defines the relation between variables and blocks and may explicitly force constraints as linking constraints, i.e., constraints that will be transferred to the extended (master) problem.

After the original instance is presolved, the relaxator performes the Dantzig-Wolfe decomposion and initializes the extended SCIP instance as well as the SCIP instances representing the pricing problems. The extended instance initially contains no variables. Original variables that are labeled to be part of a block, and constraints containing variables of just one block are copied into the corresponding pricing problem unless explicitly forced otherwise. All remaining constraints are transferred to the extended problem.

During the solving process, the extended instance builds the branch-and-bound tree in the same way as the original instance. There is a bijection between nodes of the original instance and nodes of the extended instance; two corresponding nodes are solved at the same time. When solving a node of the original instance, the node lower bound is not computed by solving the node's LP relaxation, but the special relaxator is used for this purpose which instructs the extended `SCIP` instance to solve the next node. A special node selector in the extended instance chooses as the next node to be processed the node corresponding to the current node in the original instance. Branching restrictions are imposed by a branching rule included in the original instance, so they have to be transferred to the node of the extended instance when it is activated. The solving process of the node starts with domain propagation, i.e., tightening the domains of the variables for the local problem, a concept that is also used for the enforcement of branching decisions. The LP relaxation of the problem corresponding to the node—the master problem—is then solved by column generation.

After the master problem is solved, bounding is performed and branching is performed if needed, creating two children without further problem restrictions. The solving process of the extended problem is then halted and the relaxator in the original instance transfers the local dual bound and the master problem's current solution as well as new primal solutions to the original instance. The current node in the original instance is pruned if and only if the corresponding node in the extended instance was pruned, since both nodes have the same dual bound and both instances have the same primal bound—each solution of one instance corresponds to a solution of the other instance with the same objective function value. However, it is possible that the master solution is fractional but leads to an integral solution to the original problem. In this case, the current subproblem is solved to optimality, otherwise, branching is performed. Two children are created and branching restrictions are imposed in the original instance, that will be transferred to the corresponding nodes in the extended instance on activation. After the branching, the original instance selects a next node and the process is iterated. Fig. 1 shows `GCG`'s solving process.

We decided to work with both formulations simultaneously rather than transforming the original problem into an extended problem and solving this problem with a branch-price-and-cut algorithm, since it fits better into the `SCIP` framework and makes better use of the functionalities already provided. The original problem can be read in a variety of formats by the original instance using `SCIP`'s default file reader plugins. If we do not read a file defining the structure of the problem, the problem is solved by `SCIP` with a branch-and-cut algorithm. Otherwise, the special relaxator is activated, creates the extended `SCIP` instance, performs the Dantzig-Wolfe decomposition, and substitutes the LP relaxation in the branch-and-bound process. Both problems are solved in parallel, so that techniques that speed up the solving process, like presolving, domain propagation, and heuristics, can be used in both instances. For details see [6].

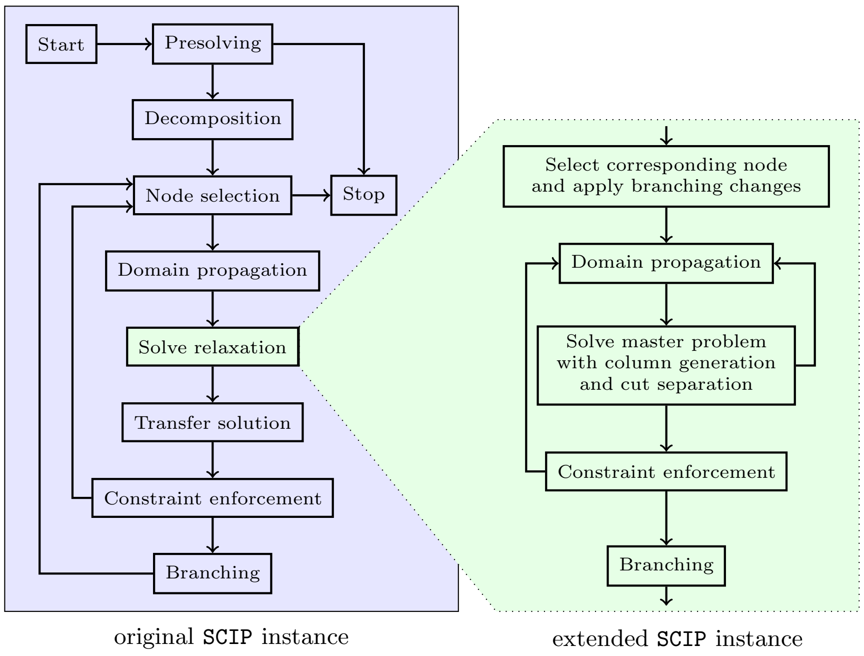


Fig. 1. Solving process of `GCG`

3.2 Pricing

We structure the pricing implementation in *variable pricer* and a set of *pricing solvers*. The former coordinates the pricing process, while the latter are called by the variable pricer to solve a specific pricing problem. A variable pricer plugin is added to the extended instance. These plugins have two essential callbacks that are called during the pricing process, one for *Farkas pricing*, which is called by `SCIP` whenever the RMP is infeasible, the other for the reduced cost pricing, which is called in case the RMP is feasible.

We introduced the concept of pricing solvers, which are used by the pricer in a black box fashion: Whenever a specific pricing problem should be solved, it is given to the set of solvers, solved by one of the solvers, and a set of solutions is returned. We chose this concept, which is similar to the way the LP solver is handled in `SCIP`, in order to provide a possiblity to add further problem specific solvers as external plugins without the need to modify the variable pricer.

We compute the intermediate Lagrangean dual bounds every time all pricing problems were solved to optimality in a pricing round and update the dual bound of the current node each time this leads to an improvement. We make use of *early termination*, i.e., if all solutions have integral values—this is detected in the presolving process of the original problem—we abort the pricing process at a node whenever $\lceil LB \rceil \geq z_{RMP}$ for the current local dual bound LB and the optimal objective value z_{RMP} of the RMP in this pricing iteration.

When using the discretization approach, we identify identical subproblems and automatically aggregate them. That is, if $X_i = X_j$ for all $i, j \in \{1, \ldots, k\}$, we define aggregate variables $\lambda_p = \sum_k \lambda_p^k$ in the master problem.

3.3 Branching Rules

We provide two branching rules, one that branches on the variables of the original problem and Ryan and Foster's branching scheme [14] for problems with a set partitioning master problem. When branching on original variables, the master solution is transferred into a solution of the original problem, an integer variable x_i^k with fractional value v is identified and the domain of the variable is split by adding constraints $x_i^k \geq \lceil v \rceil$ and $x_i^k \leq \lfloor v \rfloor$, respectively, to the two child nodes. These constraints can be enforced in the pricing problems as well as in the extended problem. The latter leads to an additional constraint in the master problem, its dual variable is respected in the objective function of the pricing problem like it is done for the other master constraints. Enforcing the branching decisions in the pricing problems can lead to better dual bounds at the expense that all variables in the master problem have to be checked for their feasibility w. r. t. the current pricing problems. This is done via domain propagation in the extended instance; variables that are not compatible with branching decisions are locally fixed to zero. It is well-known that the choice of the variable to branch on has a big impact on the performance of the branch-and-bound process [2]. Hence, apart from most infeasibility branching, we provide the possibility to make use of pseudocosts of the variables in the original problem. This leads to a considerable decrease of computational time for the class of capacitated p-median problems (Sect. 4.1). The Ryan and Foster branching scheme is used for problems with a set partitioning master structure and for problems with identical blocks.

3.4 Presolve and Propagation

We perform standard SCIP presolving on the original problem. Furthermore, at each node, we perform domain propagation in the original instance as well as the extended instance. In the extended instance, we use it primarily to remove variables from the problem, that are not valid for the current pricing problem. In the original instance, standard domain propagation methods are used that lead to domain reductions especially when branching on the original variables. These reductions can then be imposed on the variables of the pricing problems, too. Variables that fulfill these bounds are called *proper* [21].

3.5 Cutting Plane Separators

A by now "standard" way to strengthen the dual bound it to derive valid inequalities from the original variables. An original fractional solution can be separated by SCIP's default cut separation plugins. Note that $x^k = \sum_{p \in P_k} p\lambda_p^k$ is not a basic solution in (OPk), thus we cannot use *Gomory mixed integer cuts* or *strong Chvátal-Gomory cuts*. Nevertheless, we can use all kinds of cutting planes that do not need any further information besides the problem and the current solution in their separation routine. This applies, for example, to *knapsack cover cuts*, *mixed integer rounding cuts*, and *flow cover cuts*. An alternative is to derive cuts from the extended formulation. This may lead to stronger cuts but has the drawback that the dual variable of a cut has to be respected in the pricing problems, which typically results in additional variables and constraints added to the pricing problems, see [5] for a very recent unified view.

3.6 Primal Heuristics

The default primal heuristics of SCIP (including sophisticated ones) are applied to the extended instance. Simple rounding heuristics are performed in each iteration of the column generation process which often find feasible primal solutions. Currently, most heuristics make use of the LP relaxation, so we cannot run them on the original instance. This is to be changed in the future.

3.7 Customization and Extendibility

It is to be expected that a tailor-made approach will outperform a generic one, so we keep the possibility to extend and customize the framework, in addition to adding pricing solvers. SCIP's interface for branching rules is extended in order to give the possibility to define branching rules operating on both the original as well as the extended formulation. This includes ways to enforce branching decisions in the pricing problems and to store pseudocosts for branching on constraints. Finally, problem specific plugins for presolving, node selection, domain propagation, separation and primal heuristics can be added to either one of the two SCIP instances as usual.

4 Computational Study

We tested our solver GCG 0.7 on MIPs which expose various different structures using SCIP 1.2.0.5 with CPLEX 12.1 as embedded LP solver. The computations presented in Section 4.1 and 4.2 were performed on a 2.66 Ghz Core2 Quad with 4MB Cache and 4GB RAM, those of Section 4.3 on a 2.83 GHz Core2 Quad with 6MB cache and 16GB RAM. We compute averages using the shifted geometric mean, i.e., for non-negative numbers $a_1, \dots, a_k \in \mathbb{R}_+$, e.g., the number of nodes, the solving time, or the final gap of the individual instances of a test set, and a shift $s \in \mathbb{R}_+$, the average is defined by

$$\gamma_s(a_1, \dots, a_k) = \left(\prod_{i=1}^{k} (a_i + s) \right)^{\frac{1}{k}} - s.$$

We use a shift of 10 for the runtime and the number of branch-and-bound nodes and 100 for the final gap in percent. The average value of multiple test sets is computed in the same way, using the shifted geometric means of the individual test sets.

Table 1. Comparison of GCG and SCIP for the capacitated p-median test sets. We list the shifted geometric mean of the number of branch-and-bound nodes (top), and the runtime (bottom) for SCIP (first column) and GCG (second column) with default settings. The next columns illustrate the performance effect of disabling the pseudocost branching rule and using the most fractional rule instead (third column) and of using a specialized knapsack solver to solve the pricing problems (last column). Following the runtime, in brackets, we list the absolute number of timeouts.

	test set	SCIP	GCG	no pseudocost	knapsack
nodes	CPMP50S	896.2	44.9	82.5	42.1
	CPMP100S	14234.2	587.1	1962.6	491.6
	CPMP150S	15128.7	847.6	2211.7	1228.6
	CPMP200S	26263.9	1753.3	5577.7	2338.0
	sh. geom. mean	8454.9	461.8	1216.6	515.1
time (outs)	CPMP50S	14.5 (0)	12.8 (0)	20.7 (0)	1.8 (0)
	CPMP100S	234.8 (3)	184.7 (1)	469.5 (6)	37.8 (0)
	CPMP150S	714.5 (9)	493.5 (5)	920.3 (10)	253.8 (1)
	CPMP200S	1950.7 (7)	1243.9 (3)	2978.0 (10)	519.3 (0)
	sh. geom. mean	294.0 (19)	220.1 (9)	439.7 (26)	84.3 (1)

4.1 Different Subproblems: The Capacitated *p*-Median Problem

In the *capacitated p-median problem* we are given a set N of nodes, each with a demand $q_n \in \mathbb{Z}_+$, $n \in N$. In each node $n \in N$, a facility with capacity C can be opened; p of which have to opened in total. The distance between a node $n \in N$ and a facility placed at node $m \in N$ is given as $d_{n,m} \in \mathbb{Z}$. Nodes are assigned to opened facilities so that the total sum of connection distances is minimized and the capacity constraints are respected. To solve large instances by branch-and-price, so far an *ad hoc* implementation was necessary [4].

The problem can be decomposed by defining one block for each possible facility location which contains the capacity constraint corresponding to this location. The blocks are not identical since the objective function coefficients of the variables depend on the location represented by this block. Therefore, we branch on the original variables. We used a subset of the instances used in [4] and defined four test sets CPMPNS, each of which contains instances with N nodes, $N \in \{50, 100, 150, 200\}$. In order to reduce the computational effort, we missed out every second instance in the test sets with up to 150 nodes and selected only 12 instances for test set CPMP200S, three for each number of facilities.

Tab. 1 shows that the generic branch-and-price approach performs better than plain SCIP. It particularly pays off to have a state-of-the-art branching rule at hand: using most fractional branching instead of pseudocost branching doubles the shifted geometric mean of the solution time. Furthermore, by using a dynamic programming knapsack solver to solve the pricing problems and customizing the code in this way, we are able to decrease the shifted geometric mean of the solving time by more than 60% compared to GCG with default settings. More detailed computational experiments are reported in [6].

Table 2. Computational results for the 180 instances of each size in the bin packing test set

	nodes		time	
number of items	total	sh. geom. mean	total	sh. geom. mean
50	713	3.3	54.7	0.3
100	1617	6.7	254.4	1.4
200	5229	17.6	2186.8	11.7

4.2 Identical Subproblems: Bin Packing

Bin packing instances have identical blocks and a set partitioning master problem, so the variables were aggregated and Ryan and Foster's branching scheme was used. It is well-known that the Dantzig-Wolfe decomposition of the bin packing problem leads to strong dual bounds, so we were able to solve all 540 instances (all 180 instances with 50, 100, and 200 items, respectively, of data set 1 of [15]) in less than 90 minutes altogether, cf. Tab. 2. For each number of items, GCG solves the whole test set faster than SCIP solves the first instance of the set.

4.3 No Block Structure: A Resource Allocation Problem

The following generalized knapsack problem [3] does not have a block structure (but staircase structure). Given a number of periods $n \in N$ and items $i \in I$, each item has a profit p_i, a weight w_i, and a starting and ending period. In each period, the knapsack has capacity C and items consume capacity only during their *life time*. The problem can be modelled in the following way:

$$\max\{\sum_{j\in I} p_i x_i \mid \sum_{i\in I(n)} w_i x_i \leq C \; \forall n \in N,\; x_i \in \{0,1\} \; \forall i \in I\} \;, \tag{RAP}$$

where $I(n)$ is the set of items that are alive in period $n \in N$. The matrix can be transformed into block structure by splitting the capacity constraints into groups of size M [3]. For each variable that appears in more than one group, we create a copy of this variable for each group and link the values of these copies to each other by additional constraints. These additional constraints will be part of the extended formulation, the M capacity constraints corresponding to a block are transferred to this block's pricing problem.

We performed computational experiments (see Tab. 3) for the instances described in [3]. We used SCIP 1.2.0.5 for solving formulation RAP explicitly, and GCG 0.7 to solve the reformulation of the problem grouping 32 and 64 constraints to form one block, respectively. The same grouping was used in [3]. SCIP was able to solve five instances within the timelimit of one hour, the remaining 65 instances remained unsolved with a final gap between 0.1 and 3.0 percent. For both numbers of constraints grouped per block, GCG was able to solve 56 instances, the final gap of the remaining instances was typically lower than 0.3

Table 3. Computational results for the test set of RAP instances. We list the final gap, the number of branch-and-bound nodes and the runtime for `SCIP`, `GCG` with 32 constraints assigned to a block, and `GCG` with 64 constraints assigned to a block.

	`SCIP`			`GCG` (32)			`GCG` (64)		
instance	gap	nodes	time	gap	nodes	time	gap	nodes	time
new1_1	1.3	>757186	>3600.0	0.0	29	201.2	0.0	3	111.4
new1_2	1.7	>589833	>3600.0	0.0	119	979.6	0.0	13	350.5
new1_3	1.6	>408635	>3600.0	0.0	23	359.7	0.0	9	529.5
new1_4	1.7	>296254	>3600.0	0.0	187	1888.9	0.0	7	473.3
new1_5	2.0	>262849	>3600.0	0.0	41	828.6	0.0	11	562.3
new1_6	1.4	>194120	>3600.0	0.0	17	468.5	0.0	39	1143.4
new1_7	1.5	>163145	>3600.0	0.0	27	460.6	0.0	1	478.7
new1_8	1.7	>161800	>3600.0	0.0	13	583.6	0.0	3	350.6
new1_9	1.8	>108307	>3600.0	0.0	87	1687.5	0.0	2	453.4
new1_10	2.2	>84190	>3600.0	0.0	35	1085.1	0.0	7	896.1
new2_1	2.0	>750775	>3600.0	0.0	7	109.5	0.0	3	140.0
new2_2	1.4	>580746	>3600.0	0.0	136	898.0	0.0	44	744.7
new2_3	1.1	>415357	>3600.0	0.0	23	299.5	0.0	2	135.9
new2_4	1.2	>428950	>3600.0	0.0	1	118.9	0.0	1	367.1
new2_5	1.9	>207493	>3600.0	0.0	161	1622.6	0.1	>154	>3600.0
new2_6	2.0	>198288	>3600.0	0.0	11	425.6	0.0	4	326.9
new2_7	1.6	>188555	>3600.0	0.0	36	642.2	0.0	13	740.3
new2_8	1.7	>142970	>3600.0	0.0	13	399.6	0.0	14	782.2
new2_9	1.8	>75900	>3600.0	0.0	31	702.4	0.0	1	293.5
new2_10	1.9	>131284	>3600.0	0.0	151	1489.0	0.0	41	1274.4
new3_1	1.3	>233708	>3600.0	0.0	31	1205.1	0.0	1	1205.8
new3_2	1.2	>89037	>3600.0	0.0	>119	>3600.0	0.0	>49	>3600.0
new3_3	1.3	>75770	>3600.0	0.0	13	1227.5	0.0	>7	>3600
new3_4	1.4	>84830	>3600.0	0.0	39	2596.2	0.0	9	1688.5
new3_5	1.7	>19150	>3600.0	0.0	33	1937.4	0.0	9	3039.5
new3_6	2.2	>8632	>3600.0	0.1	>55	>3600.0	0.0	>39	>3600.0
new3_7	1.8	>35455	>3600.0	0.0	>88	>3600.0	0.0	>62	>3600.0
new3_8	1.8	>30130	>3600.0	0.0	53	3535.1	0.0	5	3172.4
new3_9	2.4	>25500	>3600.0	0.1	>40	>3600.0	73.8	>1	>3600.0
new3_10	2.6	>864	>3600.0	0.0	>45	>3600.0	0.2	>27	>3600.0
new4_1	0.0	4907	16.1	0.0	1	26.1	0.0	1	15.6
new4_2	0.0	187	4.8	0.0	1	17.9	0.0	1	17.1
new4_3	0.0	132	6.0	0.0	1	24.6	0.0	1	28.4
new4_4	0.1	>1571331	>3600.0	0.0	1	73.0	0.0	1	44.5
new4_5	0.0	70140	342.6	0.0	13	86.4	0.0	13	106.3
new4_6	0.2	>792713	>3600.0	0.0	1	58.8	0.0	1	66.1
new4_7	0.0	670545	3406.9	0.0	1	57.0	0.0	1	54.1
new4_8	0.2	>535592	>3600.0	0.0	4	99.3	0.0	3	59.0
new4_9	0.2	>546140	>3600.0	0.0	3	92.6	0.0	3	93.6
new4_10	0.2	>507737	>3600.0	0.0	3	61.0	0.0	5	154.4

Table 3. (*continued*)

instance	SCIP gap	SCIP nodes	SCIP time	GCG (32) gap	GCG (32) nodes	GCG (32) time	GCG (64) gap	GCG (64) nodes	GCG (64) time
new5_1	0.7	>640409	>3600.0	0.0	23	254.4	0.0	1	130.1
new5_2	1.3	>317663	>3600.0	0.0	11	248.6	0.0	1	230.7
new5_3	0.9	>318430	>3600.0	0.0	3	177.1	0.0	3	282.4
new5_4	1.4	>247945	>3600.0	0.0	75	1445.6	0.0	3	265.3
new5_5	1.4	>120474	>3600.0	0.0	42	871.3	0.0	17	887.4
new5_6	1.3	>147990	>3600.0	0.0	161	3570.1	0.0	>129	>3600.0
new5_7	1.0	>135050	>3600.0	0.0	133	2008.2	0.0	13	730.5
new5_8	1.3	>114213	>3600.0	0.0	15	554.7	0.0	1	564.4
new5_9	1.2	>78548	>3600.0	0.0	>124	>3600.0	0.0	>87	>3600.0
new5_10	1.7	>22990	>3600.0	0.0	85	2109.2	0.0	25	1799.1
new6_1	1.0	>215663	>3600.0	0.0	25	692.2	0.0	7	484.2
new6_2	1.2	>159970	>3600.0	0.0	7	518.3	0.0	3	686.3
new6_3	1.0	>92896	>3600.0	0.0	27	975.0	0.0	11	559.0
new6_4	1.0	>93850	>3600.0	0.0	17	1091.2	0.0	1	628.5
new6_5	1.1	>69570	>3600.0	0.0	29	1546.7	0.0	21	1826.8
new6_6	1.8	>14540	>3600.0	0.0	13	1094.0	0.0	7	1714.0
new6_7	2.3	>10384	>3600.0	0.0	>51	>3600.0	0.0	3	1368.9
new6_8	1.6	>6209	>3600.0	0.1	>77	>3600.0	0.0	>38	>3600.0
new6_9	1.7	>38634	>3600.0	0.0	>37	>3600.0	0.0	3	2030.4
new6_10	3.0	>772	>3600.0	0.0	>55	>3600.0	0.0	>35	>3600.0
new7_1	1.4	>537230	>3600.0	0.0	58	614.3	0.0	7	276.9
new7_2	1.5	>373513	>3600.0	0.0	31	624.5	0.0	41	1263.9
new7_3	1.4	>230756	>3600.0	0.0	46	823.1	0.0	11	545.2
new7_4	1.8	>164797	>3600.0	0.0	25	665.2	0.0	>37	>3600.0
new7_5	1.3	>147376	>3600.0	0.0	9	486.2	0.0	9	791.8
new7_6	1.7	>158523	>3600.0	0.0	>186	>3600.0	0.0	34	3032.1
new7_7	1.7	>95711	>3600.0	0.0	>140	>3600.0	0.1	>74	>3600.0
new7_8	2.2	>96525	>3600.0	0.1	>148	>3600.0	0.0	19	3205.4
new7_9	1.7	>80942	>3600.0	0.0	31	1051.1	0.0	11	1800.4
new7_10	2.4	>78398	>3600.0	0.0	>107	>3600.0	0.1	>42	>3600.0
total	97.1	16259k	237776.4	0.4	3484.0	98169.6	74.3	1305.0	95403.1
timeouts			65/70			14/70			14/70
sh. geom. mean	1.4	97564.0	2772.4	0.0	32.2	727.4	0.8	11.5	670.7

percent. For both sizes of blocks, GCG was about four times faster than SCIP in the shifted geometric mean.

The relaxation given by the master problem is tighter the more constraints are assigned to a block, so with 64 constraints per block, we need less nodes to solve the problems. The shifted geometric mean of the number of nodes accounts 11.5 for the former variant, compared to 21 nodes when assigning 32 constraints to each block. In return, more time is needed to solve the master problem, but this pays off for this test set since the total time is reduced by 8%. The average gap is higher when grouping 64 constraints, however, this is caused by one single instance for which the master problem at the root node could not be solved within the time limit of one hour so that just a trivial dual bound is obtained, leading to a gap of more than 70 percent.

5 Summary and Discussion

We report first computational experiments with a basic generic implementation of a branch-price-and-cut algorithm within the non-commercial framework `SCIP`. Given an MIP and information about which rows belong to which subproblem (or the master problem), either a convexification or discretization style decomposition is performed. For structured problems, the approach is very promising.

The modular design of `SCIP` allowed us to include the described functionality in the form of *plugins*. A true integration would require a few extensions, some of which have been incorporated into `SCIP` during this project already, but some are still missing. Examples include per-row dual variable stabilization, column pool, primal heuristics on original variables, LP basis of original variables for cutting planes like Gomory cuts, etc. It is planned that our implementation becomes part of a future release of `SCIP`. We hope that this enables researchers to play with and quickly test ideas in decomposing mixed integer programs.

It remains to be demonstrated that there really is a significant share of problems on which decomposition methods are more effective than (or a reasonable complement to) standard branch-and-cut, even when one does not know about a possibly contained structure. This requires detecting whether it may pay to decompose any given MIP, and if so, how this should be done. This is, of course, a much more challenging question which is the subject of our current research.

Acknowledgment. We thank Alberto Ceselli and Enrico Malaguti for providing us with the p-median and RAP instances from Sections 4.1 and 4.3, respectively.

References

1. Achterberg, T.: SCIP: Solving constraint integer programs. Math. Programming Computation 1, 1–41 (2009)
2. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Operations Research Letters 33, 42–54 (2005)
3. Caprara, A., Furini, F., Malaguti, E.: Exact algorithms for the temporal knapsack problem. Technical report OR-10-7, DEIS, University of Bologna (2010)
4. Ceselli, A., Righini, G.: A branch-and-price algorithm for the capacitated p-median problem. Networks 45, 125–142 (2005)
5. Desaulniers, G., Desrosiers, J., Spoorendonk, S.: Cutting planes for branch-and-price algorithms, Les Cahiers du GERAD G-2009-52, HEC Montréal (2009)
6. Gamrath, G.: Generic branch-cut-and-price. Master's thesis, Institut für Mathematik, Technische Universität Berlin (2010)
7. Jünger, M., Thienel, S.: The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. Softw. Pract. Exper. 30, 1325–1352 (2000)

8. Mittelmann, H.: Benchmarks for optimization software (2010), http://plato.asu.edu/bench.html
9. Nemhauser, G., Savelsbergh, M., Sigismondi, G.: MINTO, a Mixed INTeger Optimizer. Oper. Res. Lett. 15, 47–58 (1994)
10. Puchinger, J., Stuckey, P., Wallace, M., Brand, S.: Dantzig-Wolfe decomposition and branch-and-price solving in G12. Constraints (to appear, 2010)
11. Ralphs, T., Galati, M.: Decomposition and dynamic cut generation in integer linear programming. Math. Programming 106, 261–285 (2006)
12. Ralphs, T., Galati, M.: DIP – decomposition for integer programming (2009), https://projects.coin-or.org/Dip
13. Ralphs, T., Ladányi, L.: COIN/BCP User's Manual (2001), http://www.coin-or.org/Presentations/bcp-man.pdf
14. Ryan, D., Foster, B.A.: An integer programming approach to scheduling. In: Wren, A. (ed.) Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling, pp. 269–280. North Holland, Amsterdam (1981)
15. Scholl, A., Klein, R.: Bin packing instances: Data set 1, http://www.wiwi.uni-jena.de/Entscheidung/binpp/
16. Vanderbeck, F.: On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. Oper. Res. 48, 111–128 (2000)
17. Vanderbeck, F.: BaPCod – a generic branch-and-price code (2005), https://wiki.bordeaux.inria.fr/realopt/pmwiki.php/Project/BaPCod
18. Vanderbeck, F.: Implementing mixed integer column generation. In: Desaulniers, G., Desrosiers, J., Solomon, M. (eds.) Column Generation, pp. 331–358. Springer, Heidelberg (2005)
19. Vanderbeck, F.: A generic view of Dantzig-Wolfe decomposition in mixed integer programming. Oper. Res. Lett. 34, 296–306 (2006)
20. Vanderbeck, F.: Branching in branch-and-price: A generic scheme. Math. Programming (to appear, 2010)
21. Vanderbeck, F., Savelsbergh, M.: A generic view of Dantzig-Wolfe decomposition in mixed integer programming. Oper. Res. Lett. 34, 296–306 (2006)
22. Vanderbeck, F., Wolsey, L.: Reformulation and decomposition of integer programs. In: Jünger, M., Liebling, T., Naddef, D., Nemhauser, G., Pulleyblank, W., Reinelt, G., Rinaldi, G., Wolsey, L. (eds.) 50 Years of Integer Programming 1958–2008. Springer, Berlin (2010)

Using Bound Sets in Multiobjective Optimization: Application to the Biobjective Binary Knapsack Problem

Charles Delort and Olivier Spanjaard

LIP6-CNRS, Université Pierre et Marie Curie (UPMC),
4 Place Jussieu F-75005 Paris, France
{charles.delort,olivier.spanjaard}@lip6.fr

Abstract. This paper is devoted to a study of the impact of using bound sets in biobjective optimization. This notion, introduced by Villareal and Karwan [19], has been independently revisited by Ehrgott and Gandibleux [9], as well as by Sourd and Spanjaard [17]. The idea behind it is very general, and can therefore be adapted to a wide range of biobjective combinatorial problem. We focus here on the biobjective binary knapsack problem. We show that using bound sets in a two-phases approach [18] based on biobjective dynamic programming yields numerical results that outperform previous ones, both in execution times and memory requirements.

Keywords: Multiobjective combinatorial optimization, bound sets, biobjective binary knapsack problem.

1 Introduction

Multiobjective combinatorial optimization (MOCO) deals with combinatorial problems where every solution is evaluated according to several objectives. Interest in this area has tremendously grown over the last two decades. A thorough presentation of the field can be found for instance in a book by Ehrgott [7]. The standard approach aims at generating the whole set of Pareto optimal solutions, i.e. solutions that cannot be improved on one objective without being depreciated on another one. Most of the classical exact and approximate methods for finding an optimal solution in single objective discrete optimization have been revisited for finding the Pareto set under multiple objectives, e.g. dynamic programming [6,12], branch and bound [4,11,14], greedy algorithm [16], as well as many heuristic and metaheuristic methods [8].

In order to perform implicit enumeration in multiobjective optimization problems, the formal notion of *bound set* needs to be introduced. This has been done several times in the literature. Roughly speaking, bound sets are *sets* of bounds. Indeed, due to the partial nature of the ordering relation between solutions, the use of a set of bounds instead of a single bound makes it possible to more tightly approximate the image set of the solutions in the objective space. To our knowledge, one of the first work mentioning that notion was performed by Villareal and Karwan [19], and deals with branch and bounds for multiobjective integer linear programming problems. However, in this work and subsequent ones, no operational way to compute bound sets has been devised where the bound set does not reduce to a singleton. Very recently, based on the convex hull of the image of the solutions in the objective space, new bound sets have been proposed [9,17]. The use of these new bound sets has proved very efficient in the biobjective spanning tree problem [17]. The purpose of the present paper is to show how these bound sets can be used to design efficient algorithms for the biobjective binary knapsack problem. Our contribution is twofold: we first explain how to hybridize multiobjective dynamic programming with the fathoming criterion provided by the bound sets, and then detail how multiobjective dynamic programming can be embedded in a two-phases approach to further improve the method. The hybridization we propose is in the spirit of the dominance relations between states used in a work by Bazgan *et al.* [2,3], but enables huge savings in memory requirements as well as improvements in execution times. The two-phases version of the algorithm provides even better results thanks to a *shaving* procedure [13] that makes use of the bound sets.

2 Preliminaries

2.1 Preliminary Definitions

We first recall some preliminary definitions concerning MOCO problems. They differ from the standard single objective ones mainly in their cost structure, as solutions are valued by m-vectors instead of scalars. Let us denote by $\mathcal{X}$ the set of feasible solutions, and by $\mathcal{Y}$ its image in the objective space. The image of solution $x \in \mathcal{X}$ is $f(x) = (f_1(x), \ldots, f_m(x))$. Comparing solutions in $\mathcal{X}$ amounts then to comparing m-vectors in $\mathcal{Y}$. In this framework, the following notions prove useful (in a maximisation setting):

Definition 1. *The* weak dominance *relation on m-vectors of $\mathbb{Z}_+^m$ is defined, for all $y, y' \in \mathbb{Z}_+^m$, by $y \succsim y' \iff [\forall i \in \{1, \ldots, m\}, y_i \geq y'_i)]$. The* dominance *relation is defined as the asymmetric part of $\succsim$: $y \succ y' \iff [y \succsim y'$ and $y' \not\succsim y]$.*

Definition 2. *Within a set $Y \subseteq \mathcal{Y}$, an element y is said to be* dominated *(resp.* weakly dominated*) when $y' \succ y$ (resp. $y' \succsim y$) for some y' in Y, and* non-dominated *when there is no y' in Y such that $y' \succ y$. The set of non-dominated elements in Y is denoted by $Y^\star$.*

By abuse of language, when $f(x) \succ f(x')$, we say that solution x *dominates* solution x'. Similarly, we use the term of *non-dominated* solutions. The set of non-dominated solution of $X \subseteq \mathcal{X}$ is denoted by $X^\star$. Following Bazgan *et al.* [2,3], we say that a set of non-dominated solutions is *reduced* if it contains one and only one solution for each non-dominated objective vector in $Y = f(X) = \{f(x) : x \in X\}$. The aim of a multiobjective combinatorial problem is to determine a reduced set of non-dominated solutions.

2.2 Multiobjective Binary Knapsack Problem

An instance of the multiobjective binary knapsack problem (0-1 MOKP) consists of a knapsack of integer capacity c, and a set of items $N = \{1, \dots, n\}$. Each item j has a weight w^j and a m-vector profit $p^j = (p_1^j, \dots, p_m^j)$, variables w^j, p_k^j ($k \in \{1, \dots, m\}$) being integers. A solution is characterized by a binary n-vector x, where $x_j = 1$ if item j is selected. Furthermore, a solution x is feasible if it satisfies the constraint $\sum_{j=1}^{n} w^j x_j \leq c$. The goal of the problem is to find a reduced set of non-dominated solutions, which can be formally stated as follows:

$$\begin{aligned} \text{maximize} \; & \sum_{j=1}^{n} p_k^j x_j \qquad k \in \{1, \dots, m\} \\ \text{subject to} \; & \sum_{j=1}^{n} w^j x_j \leq c \\ & x_j \in \{0,1\} \quad j \in \{1, \dots, n\} \end{aligned}$$

The special case when $k = 2$ is named *biobjective binary knapsack problem (0-1 BOKP).*

Example 1. Consider the following problem:

$$\begin{aligned} \text{maximize} & \begin{cases} 10x_1 + 2x_2 + 6x_3 + 9x_4 + 12x_5 + x_6 \\ 2x_1 + 7x_2 + 6x_3 + 4x_4 + x_5 + 3x_6 \end{cases} \\ \text{subject to} \; & 4x_1 + 4x_2 + 5x_3 + 4x_4 + 3x_5 + 2x_6 \leq 6 \\ & x_j \in \{0,1\} \;\; j \in \{1, \dots, 6\} \end{aligned}$$

The non-dominated solutions are: $\mathcal{X}^\star = \{(0,0,0,0,1,1), (1,0,0,0,0,1), (0,0,0,1,0,1), (0,1,0,0,0,1)\}$, and their image set in the objective space is $\mathcal{Y}^\star = \{(13,4), (11,5), (10,7), (3,10)\}$ (see Figure 1). Note that all solutions in $\mathcal{X}^\star$ have distinct images in the objective space, therefore $\mathcal{X}^\star$ is a reduced set of non-dominated solutions.

Problem 0-1 MOKP can be solved by using a dynamic programming (DP) procedure. For the ease of presentation, we only detail here the way the non-dominated points in the objective space are computed. Note that the non-dominated solutions themselves can of course be recovered, by using standard bookkeeping techniques that do not impact on the computational complexity of the algorithm. Let subproblem $P(i, w)$ denote an instance of 0-1 MOKP consisting of item set $\{1, \dots, i\}$, and capacity w. Let $Y(i, w)$ be the image set of the feasible solutions in $P(i, w)$. If all sets $Y^\star(i-1, w)$ are known, for $w \in \{0, \dots, c\}$, then $Y^\star(i, w)$ can be computed by the recursive formula:

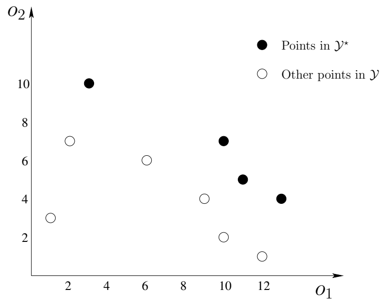


Fig. 1. Objective space

$$Y^\star(i,w) = \begin{cases} Y^\star(i-1,w) & \text{if } w < w^i \\ \mathrm{ND}\left(Y^\star(i-1,w) \cup \{y + p^i : y \in Y^\star(i-1, w-w^i)\}\right) & \text{if } w \geq w^i \end{cases}$$

Notation ND$(\cdot)$ stands for a set function returning the subset of non-dominated points in a set of m-vectors. The complexity in time and space of the DP procedure crucially depends on the cardinality of sets $Y^\star(i,w)$. Any result enabling to discard elements in these sets is therefore worth investigating. Obviously, an element $y \in Y^\star(i,w)$ can be discarded if there exists an element $y' \in Y^\star(i,w')$ such that $w' < w$ and $y' \succsim y$. With the same goal in mind (discarding elements in the dynamic programming procedure), Villareal and Karwan presented a hybrid DP approach to solve multicriteria integer linear programming problems [19]. They hybridize DP with fathoming criteria and relaxations, so as to discard some elements that would not lead to non-dominated solutions. Since we use a similar technique (by providing a more powerful fathoming criterion), we are going to present and define the bound sets used to discard most of the unwanted elements.

3 Bound Sets in MOCO Problems

3.1 Definition of Upper and Lower Bound Sets

Having good upper and lower bounds is very important in many implicit enumeration methods. It is well known that the tightness of these bounds is a key parameter for the efficiency of the methods. In a multiobjective optimization setting, since one handles sets of m-vectors, the very notion of upper and lower bound has to be revisited. This work has been undertaken by Villareal and Karwan [19], by introducing the notion of *bound sets* (in the terminology of Ehrgott and Gandibleux [9]). Since the formalism used here slightly differs from the one presented in these works, we give below our own definitions of upper and lower bound sets.

Upper bound set. The simplest idea that comes to mind to upper bound a set Y of vectors is to define a single vector y^I such that $y^I_i = \max_{y \in Y} y_i$ for $i = 1, \ldots, m$. This point is called the *ideal point* of Y. However, this ideal point is usually very "far" from the points in Y. For this reason, it is useful to define an upper bound from a *set* of vectors instead of a singleton. Such a set is then called an *upper bound set* [9].

Definition 3 (upper bound set). *A set* **UB** *is an upper bound set of* Y *if* $\forall y \in Y$, $\exists u \in \mathbf{UB}$ $: u \succsim y$.

This is compatible with the definition of an upper bound in the single objective case (**UB** reduces then to a singleton). As previously indicated, the upper bound set defined by $\mathbf{UB} = \{y^I\}$ is poor. In practice, a general family of good upper bound sets of Y can be defined as $\mathbf{UB}_\Lambda = \bigcap_{\lambda \in \Lambda} \{u \in \mathbb{R}^m : \langle \lambda, u \rangle \leq \mathbf{UB}_\lambda\}$, where the $\lambda \in \Lambda$ are weight vectors of the form $(\lambda_1, \ldots, \lambda_m) \geq 0$, $\langle ., . \rangle$ denotes the scalar product, and $\mathbf{UB}_\lambda \in \mathbb{R}$ is an upper bound for $\{\langle \lambda, y \rangle : y \in Y\}$. Of course, the larger $|\Lambda|$ is, the better the upper bound set becomes. Clearly, the best upper bound set in this family is obtained for $\Lambda = \Lambda_c(Y)$ where $\Lambda_c(Y)$ characterizes the facets of the non-dominated boundary of the convex hull of Y (see Example 2). Interestingly, we will see in the next subsection that this boundary can be efficiently computed in the biobjective case, provided $\mathbf{UB}_\lambda$ can be determined within polynomial or pseudo-polynomial time.

Lower bound set. Similarly to the upper bound set, the simplest idea that comes to mind to lower bound a set Y of vectors is to define a single vector y^A such that $y^A_i = \min_{y \in Y} y_i$ for $i = 1, \ldots, m$. This point is called the *anti-ideal point* of Y. Here again, taking several points simultaneously into account in the lower bound enables to bound more tightly set Y. Such a set is then called a *lower bound set* [9].

Definition 4 (lower bound set). *A set* $\mathbf{LB}$ *is a lower bound set of* Y *if* $\forall y \in Y, \ \exists l \in \mathbf{LB} \ : y \succcurlyeq l$.

As above, the compatibility with the single objective case holds. In the biobjective case, when Y only includes mutually non-dominated points, we will show in the next subsection a way to refine the lower bound set defined by $\mathbf{LB} = \{y^A\}$.

Comparing bound sets. Implicit enumeration is about eliminating entire subsets of solutions by using simple rules. In order to perform the elimination, we need to evaluate if a subset $X \subseteq \mathcal{X}$ of feasible solutions potentially includes non-dominated solutions in $\mathcal{X}$. To do this, one compares an upper bound set $\mathbf{UB}$ of $f(X)$ and a lower bound set $\mathbf{LB}$ of $f(\mathcal{X}^\star) = \mathcal{Y}^\star$. Unlike the single objective case, the comparison is not trivial since one handles sets instead of scalars. We introduce here two notions that make it possible to simply define this operation in a multiobjective setting.

Definition 5 (upper and lower relaxations). *Given an upper bound set* $\mathbf{UB}$*, the upper relaxation* $\mathbf{UB}^{\preccurlyeq}$ *is defined as:* $\mathbf{UB}^{\preccurlyeq} = \{x \in \mathbb{R}^m_+, \exists u \in \mathbf{UB}, \ u \succcurlyeq x\}$. *Similarly, given a lower bound set* $\mathbf{LB}$*, the lower relaxation* $\mathbf{LB}^{\succcurlyeq}$ *is defined as:* $\mathbf{LB}^{\succcurlyeq} = \{x \in \mathbb{R}^m_+, \exists l \in \mathbf{LB}, \ x \succcurlyeq l\}$.

Coming back to the comparison of $\mathbf{UB}$ and $\mathbf{LB}$, it is clear that $\mathbf{UB}^{\preccurlyeq} \supseteq f(X)$ and $\mathbf{LB}^{\succcurlyeq} \supseteq \mathcal{Y}^\star$. Consequently, $\mathbf{UB}^{\preccurlyeq} \cap \mathbf{LB}^{\succcurlyeq} = \emptyset$ implies that $f(X) \cap \mathcal{Y}^\star = \emptyset$. In this case, subset X can of course be safely pruned. Note that this pruning condition can be refined by using the fact that one only looks for a reduced set of non-dominated solutions as well as the fact that valuations are integers. Due to space constraints, this refinement is not detailed here. The main point is now to be able to efficiently compute good lower and upper bound sets. In the following subsection, this issue will be answered for the 0-1 BOKP.

3.2 Computation of Bound Sets in 0-1 BOKP

We now detail the algorithms used in 0-1 BOKP to compute the bound sets and perform their comparison.

Computation of an upper bound set. Given a subset $X \in \mathcal{X}$ of feasible solutions, upper bound set $\mathbf{UB}_{\Lambda_c(f(X))}$ can be compactly represented by storing the *extreme points* of $Y = f(X)$, i.e. the vertices of the non-dominated boundary of the convex hull of Y (points y^1, y^2, y^3, y^4 in the left part of Figure 2). Aneja and Nair's method [1] enables to efficiently compute these vertices in biobjective combinatorial problems whose single objective version is solvable within polynomial or pseudo-polynomial time. It proceeds by launching a single objective version of the problem for determining each extreme points. The number of times the single objective solution method is launched is therefore linear in the number of extreme points.

Example 2. Let us come back to Example 1. Assume that one wants to upper bound the set $X_{\bar{6}}$ of feasible solutions where item 6 is not selected. Aneja and Nair's method yields the following list L of extreme points, characterizing $\mathbf{UB}_{\Lambda_c(f(X_{\bar{6}}))}$: $L = ((12,1),(9,4),(6,6),(2,7))$. The corresponding upper relaxation $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c(f(X_{\bar{6}}))}$ is represented in Figure 2.

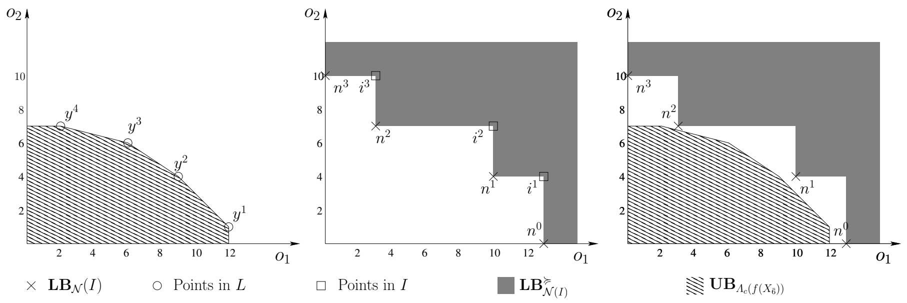


Fig. 2. Upper and lower bound sets in a biobjective setting

Computation of a lower bound set. Given a subset $I \subseteq \mathcal{Y}$, a tight lower bound set $\mathbf{LB}$ of $I^\star$ can be computed as follows. When there are two objectives and $\{(i_1^j, i_2^j) : 1 \leq j \leq k\}$ are the points of $I^\star$ maintained in lexicographical order (i.e., in decreasing order of the first objective, and increasing order of the second one), one can set $\mathbf{LB}_{\mathcal{N}(I)} = \{n^j = (i_1^{(j+1)}, i_2^j) : 0 \leq j \leq k\}$, where $i_2^0 = 0$ and $i_1^{(k+1)} = 0$. The set $\mathbf{LB}_{\mathcal{N}(I)}$ can here be viewed as a generalization of the nadir point of I (whose components are the worst possible value among the points of $I^\star$). The points in $\mathbf{LB}_{\mathcal{N}(I)}$ are therefore sometimes called *local nadir points* [9]. One can note that $\mathbf{LB}_{\mathcal{N}(I)}$ is also a lower bound set for $\mathcal{Y}^\star$.

Example 3. Let us come back to Example 1 once again, and consider the following subset of points in $\mathcal{Y}$: $I = \{(13,4),(10,7),(3,10)\}$. The lower bound set is then: $\mathcal{N}(I) = \{(13,0),(10,4),(3,7),(0,10)\}$. This lower bound set is represented in the middle part of Figure 2, as well as its lower relaxation $\mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)}$.

As described in the previous subsection, in order to know if one can prune a subset X of solutions, one must compute the intersection of the relaxations of a lower bound set of $\mathcal{Y}^\star$ and an upper bound set of $Y = f(X)$. Testing if $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c(f(X))} \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$ amounts to check whether one element of $\mathbf{LB}_{\mathcal{N}(I)}$ is included in $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c(f(X))}$. It can be formally expressed by:

$$\forall n \in \mathbf{LB}_{\mathcal{N}(I)}, \exists \lambda \in \Lambda_c(f(X)) : \lambda_1 n_1 + \lambda_2 n_2 > \max_{y \in f(X)} (\lambda_1 y_1 + \lambda_2 y_2)$$

Example 4. Continuing Example 2 and Example 3, we shall compare the two obtained relaxations. Both sets are represented in the right part of Figure 2. Their intersection is empty, meaning that subset $X_{\bar{6}}$ can be safely discarded.

4 A New Solution Algorithm for 0-1 BOKP

Unlike the single objective case, in an implicit enumeration procedure for biobjective optimization, there is not a single incumbent but a *set* of incumbents: the set of non-dominated solutions among the solutions explored so far. For simplicity, we only refer to its image set $I \subseteq \mathcal{Y}$ in the following. The idea is then of course to discard subsets X of solutions such that $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c(f(X))} \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$. We now detail the various parts of our solution method for 0-1 BOKP.

4.1 Shaving Procedure

The term "shaving" was introduced by Martin and Shmoys [13] for the job-shop scheduling problem. It enables to reduce the size of a problem by making some components forbidden or mandatory before starting the solution procedure. In knapsack problems, it amounts to consider subsets of solutions of the following form: for each item j, a subset X_j where item j is made mandatory, and a subset $X_{\bar{j}}$ where item j is made forbidden. For 0-1 BOKP, after initializing I with the extreme points of $\mathcal{Y}$ (by Aneja and Nair's method), the shaving procedure we propose consists in checking whether $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c(f(X_j))} \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$ or $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c(f(X_{\bar{j}}))} \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$. If X_j or $X_{\bar{j}}$ grants no non-dominated solution in $\mathcal{X}$, item j can be excluded from the problem by permanently setting $x_j = 0$ or $x_j = 1$. Note that the computation of the upper bound sets yields feasible solutions, possibly non-dominated. Consequently, during the running of the shaving procedure, set I is updated by inserting these possible new non-dominated elements. The shaving procedure is therefore launched twice in order to exclude some additional items during the second round of the procedure. Example 4 above shows that it is possible to shave item 6 in Example 1, by setting $x_6 = 1$.

4.2 Hybrid Dynamic Programming

During the dynamic programming (DP) procedure, the use of bound sets as a fathoming criterion, makes it possible to considerably reduce the number of stored elements in each set $Y^\star(i, w)$. This is called *hybridization*. Given an element $y \in Y^\star(i, w)$, by abuse of notation, we denote by $f^{-1}(y)$ a feasible solution in $P(i, w)$ such that $f(f^{-1}(y)) = y$ (if there are several solutions with the same image in the objective space, $f^{-1}(y)$ is any of them), and we denote by $X_y \subseteq \mathcal{X}$ the subset of feasible solutions in $P(n, c)$ whose projection on $P(i, w)$ is $f^{-1}(y)$. When computing $Y^\star(i, w)$ by DP, the fathoming criterion consists in discarding any element y such that $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c(f(X_y))} \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$. Finding $\mathbf{UB}_{\Lambda_c(f(X_y))}$ can be done by applying Aneja and Nair's method to find the extreme points of the subproblem on $\{i + 1, \ldots, n\}$ with capacity $c - w$, that is denoted by $\bar{P}_{(i+1,w)}$:

$$\begin{array}{llr} \text{maximize} & \displaystyle\sum_{j=i+1}^{n} p_k^j x_j & k \in \{1, 2\} \\ \text{subject to} & \displaystyle\sum_{j=i+1}^{n} w^j x_j \leq c - w & x_j \in \{0, 1\} \end{array}$$

One can then obtain the vertices of $\mathbf{UB}_{\Lambda_c(f(X_y))}$ by simply translating the extreme points of $\bar{P}_{(i+1,w)}$ by y.

4.3 Two-Phases Method

Visée *et al.* [20] introduced a *two-phases method* to solve the biobjective binary knapsack problem. They first calculate the set of *extreme solutions* (i.e., whose images in the objective space are extreme points of $\mathcal{Y}$), and second, by launching several branch-and-bound procedures, they compute the set of non-extreme non-dominated solutions located in the triangles generated in the objective space by two successive extreme solutions. Since the work of Visée *et al.*, other approaches have been proposed that outperform the two-phases method: a labeling approach developed by Captivo *et al.* [5], and the already mentioned DP approach by Bazgan *et al.* [2,3]. We propose here a two-phases version of our DP procedure. This technique is called *two-phasification* in the sequel. Instead of applying one single DP procedure directly on the 0-1 BOKP instance, one first computes the extreme solutions, and then applies one DP procedure for each triangle T in the objective space. Let us denote by $Y_T \subseteq \mathbb{R}^m$ the subset of the objective space corresponding to triangle T. When applying the DP procedure for finding feasible solutions within T, one checks whether $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c(f(X_j))} \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} \cap Y_T = \emptyset$ during the local shaving procedure, and one checks whether $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c(f(X_y))} \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} \cap Y_T = \emptyset$ for the fathoming criterion. Clearly, these conditions will hold much more frequently than if the problem is considered in its whole. Moreover, one

Example 5. In Figure 3 are represented the triangles that would be obtained in the problem described in Example 1. In a two-phases method, the feasible solutions corresponding to the extreme points (in black) would be found during the first phase, and the other non-dominated solutions (grey points) would be found during the second phase.

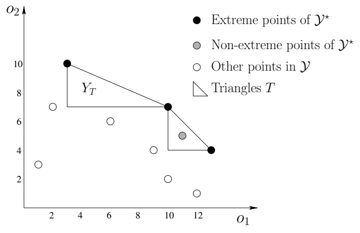


Fig. 3. Two-phases method

can limit the computation of the upper bound sets to the area of the triangle under consideration. By subdividing the problem in this way, both the shaving procedure and the fathoming criterion are more efficient, since one focuses on a restricted area of the objective space. This is confirmed by the numerical experiments.

5 Numerical Experiments

All experiments presented here were performed on an Intel® Core$^{\text{TM}}$ 2 Duo CPU E8400 @ 3.00GHz personal computer, endowed with 3.2GB of RAM memory. All algorithms were written in C++. To solve the single objective knapsack problems, we used the minknap algorithm [15] which proved to be one of the quickest in the literature (see the book by Kellerer *et al.* [10]).

5.1 Instances

The types of instances considered here are the same as in [2,3], where the parameters are uniformly randomly generated and $c = \lceil 0.5 \sum_{j=1}^{n} w^j \rceil$.
Type A: random instances, where p_1^j, p_2^j and $w^j \in \{1, \dots, 1000\}$;
Type B: unconflicting instances, where $p_1^j \in \{101, \dots, 1000\}$, $p_2^j \in \{p_1^j - 100, \dots, p_1^j + 100\}$ and $w^j \in \{1, \dots, 1000\}$;
Type C: conflicting instances, where $p_1^j \in \{1, \dots, 1000\}$, $p_2^j \in \{\max\{900 - p_1^j, 1\}, \dots, \min\{1100 - p_1^j, 1000\}\}$ and $w^j \in \{1, \dots, 1000\}$;
Type D: conflicting instances with correlated weights, where $p_1^j \in \{1, \dots, 1000\}$, $p_2^j \in \{\max\{900 - p_1^j, 1\}, \dots, \min\{1100 - p_1^j, 1000\}\}$ and $w^j \in \{p_1^j + p_2^j - 200, \dots, p_1^j + p_2^j + 200\}$.

5.2 Results

We compared our method (named S2H for Shaving, 2-phases, and Hybrid DP) and the one of Bazgan *et al.* [2,3] (named BHV: initials of the authors) by running both methods on the same instances[1]. Table 1 shows the time and memory spent to solve different types and sizes of instances. The first two columns indicate the size and type of the instances solved. For each size and type, 30 randomly generated instances have been solved using different methods, and the average and maximum times and memory requirements are indicated. Numbers in bold represent the best value for a given type and size. Shaving, hybridization and two-phasification are the three main parts of the algorithm presented in this paper. We evaluated some variations of our method in order to measure the importance of each part: 2H is a two-phases method using a hybridized DP procedure, SH is a hybridized DP procedure applied to a shaved problem, and finally S2 is a two-phases method using simple DP on problems reduced by shaving. A time limit was set to 10000 seconds. Symbol "-" in the table denotes that

[1] We wish to thank Hadrien Hugot who kindly sent us the code of the BHV method.

Table 1. Computation times, and memory requirements of different methods for the 0-1 BOKP

Type	Size	Method	Time (sec.)		RAM (MB)		Type	Size	Method	Time (sec.)		RAM (MB)	
			Avg.	Max.	Avg.	Max.				Avg.	Max.	Avg.	Max.
A	300	S2H	**17**	**30**	**2.6**	**3.1**	B	1000	S2H	7.0	11	**2.0**	**2.3**
		BHV	51	103	80	113			BHV	**4.1**	**10**	11	15
		2H	73	134	5.8	6.3			2H	17	40	15.2	16
		SH	60	88	4.0	4.4			SH	10	16	2.5	2.8
		S2	113	208	13.4	16			S2	10	15	2.3	3.6
	500	S2H	**73**	**109**	**3.1**	**3.7**		2000	S2H	**50**	**68**	**2.6**	**3.0**
		BHV	564	1031	401	449			BHV	132	272	132	272
		2H	459	626	8.8	9.3			2H	195	334	30.1	31
		SH	448	679	6.2	6.6			SH	109	181	3.8	4.3
		S2	671	1045	38	56			S2	91	123	14	21
	700	S2H	**193**	**254**	**3.6**	**3.8**		3000	S2H	**160**	**211**	**3.1**	**3.4**
		BHV	2740	4184	1308	1800			BHV	874	1292	449	449
		2H	1566	2038	11.1	12			2H	830	1227	44.6	45
		SH	2209	3353	8.7	9.4			SH	517	699	4.9	5.2
		S2	2820	3624	116	159			S2	344	468	45	74
	1000	S2H	**558**	**705**	**4.2**	**4.7**		4000	S2H	**358**	**435**	**3.7**	**3.9**
		BHV	*	*	*	*			BHV	3017	4184	1307	1800
		2H	5588	6981	15.7	16			2H	2292	3032	59.1	60
		SH	-	-	-	-			SH	1648	2097	6.1	6.4
		S2	-	-	-	-			S2	970	1308	84	130
C	200	S2H	73	121	4.3	5.0	D	100	S2H	84	136	**5.1**	**6.0**
		BHV	**32**	**47**	63	113			BHV	**35**	57	80	113
		2H	112	172	4.8	5.3			2H	108	169	5.1	6.0
		SH	147	239	**3.1**	**3.4**			SH	125	165	6.0	6.7
		S2	1835	2307	107	163			S2	2138	3252	124	168
	300	S2H	319	497	**5.9**	**6.9**		150	S2H	389	723	**7.5**	**8.8**
		BHV	**206**	**288**	257	449			BHV	**154**	**228**	311	449
		2H	539	832	6.7	7.3			2H	517	879	7.5	8.8
		SH	788	1159	9.4	10			SH	698	1123	9.2	10
		S2	-	-	-	-			S2	-	-	-	-
	400	S2H	946	1479	**7.7**	**9.0**		200	S2H	1143	2015	9.7	**12**
		BHV	**748**	**1006**	782	897			BHV	**770**	897	897	897
		2H	1756	2647	8.9	9.9			2H	1596	2796	**9.5**	12
		SH	2806	3956	14.8	18			SH	2689	3747	13.1	16
		S2	-	-	-	-			S2	-	-	-	-
	500	S2H	2138	3046	**9.6**	**10**		250	S2H	2555	3540	**11.7**	**17**
		BHV	**2014**	**2651**	1458	1800			BHV	**1989**	1100	1730	1800
		2H	4165	5952	10.4	11			2H	3585	4668	11.7	17
		SH	-	-	-	-			SH	6984	8516	18.1	21
		S2	-	-	-	-			S2	-	-	-	-

2H: method S2H without shaving   SH: method S2H without two-phases
S2: method S2H without hybridization

at least one instance of this type and size reached this limit. Symbol "*" indicates that at least one instance couldn't be solved due to insufficient memory.

Shaving. The shaving procedure is particularly effective on instances of type A or B: there is indeed a lot of items that are not interesting, such as items with low profits on both objectives, and high weights, and conversely items that have high profits and low weights, which will be taken in all non-dominated solutions. On the other hand, since all items in types C and D have conflicting profits, it is more difficult to shave items.

Two-phases. Using a two-phases method makes it possible to divide the problem into several smaller problems, but there can be a lot of them (for problems of type A and size 1000, there are on average 155 subproblems to solve). The combination with the shaving procedure is interesting, because it further reduces the sizes of the subproblems. The memory space spared this way is not as important as that of the shaving for type A and B; the opposite is observed for types C and D.

Hybridization. Hybridizing the DP is the main part of our algorithm. Not only does it tremendously reduces the memory requirements, it also saves a lot of computation time for larger, or more difficult instances. This can be seen by looking at the results of method S2 on instances of types C and D, both in terms of time and memory requirements.

We will now compare the S2H method to the BHV method. First, from a memory consumption point of view, our method largely outperforms the BHV method for all sizes and types of instances. From the computation time perspective, results depend on the sizes and types of instances. For types A and B the S2H method is much faster than the BHV method, while for types C and D it is slower, although when the size grows our method seems to become more and more competitive (it is as good as method BHV for type C and size 500, and within a factor two for type D and size 250). The reason for this behaviour is that the fathoming criterion is rather time consuming, but this is compensated for bigger instances by the fact that a lot of computation time is saved thanks to the important number of elements that have been fathomed.

6 Conclusion

In this paper, we have presented a new solution algorithm for 0-1 BOKP, based on the use of bound sets. It outperforms previous dynamic programming approaches from the viewpoint of memory requirements. Concerning the resolution times, the performances are better than the best known algorithm for this problem on random and unconflicting instances, and slower on conflicting instances (but within the same order of magnitude). A natural extension of this work would be to investigate the impact of the use of bound sets on other MOCO problems. Another extension would be to study how to improve the resolution times on conflicting instances of 0-1 BOKP. For this purpose, an incremental resolution of the single objective problems is worth investigating. Finally, note that our fathoming criterion has only been implemented in the biojective case up to now. The study of its practical implementation in problems involving more than two objectives is an interesting and potentially fruitful task in our opinion.

Acknowledgements. This work was supported by ANR project GUEPARD.

References

1. Aneja, Y.R., Nair, K.P.K.: Bicriteria transportation problem. Management Science 25, 73–78 (1979)
2. Bazgan, C., Hugot, H., Vanderpooten, D.: An efficient implementation for the 0-1 multi-objective knapsack problem. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 406–419. Springer, Heidelberg (2007)
3. Bazgan, C., Hugot, H., Vanderpooten, D.: Solving efficiently the 0-1 multi-objective knapsack problem. Computers & Operations Research 36(1), 260–279 (2009)
4. Bitran, G., Rivera, J.M.: A combined approach to solve binary multicriteria problems. Naval Research Logistics Quarterly 29, 181–201 (1982)
5. Captivo, M.E., Clìmaco, J., Figueira, J., Martins, E., Santos, J.L.: Solving bicriteria 0-1 knapsack problems using a labeling algorithm. Computers & Operations Research 30(12), 1865–1886 (2003)
6. Daellenbach, H.G., De Kluyver, C.A.: Note on multiple objective dynamic programming. Journal of the Operational Research Society 31, 591–594 (1980)
7. Ehrgott, M.: Multicriteria Optimization, 2nd edn. Springer, Heidelberg (2005)
8. Ehrgott, M., Gandibleux, X.: Approximative solution methods for multiobjective combinatorial optimization. Journal of the Spanish Statistical and Operations Research Society 12(1), 1–88 (2004)
9. Ehrgott, M., Gandibleux, X.: Bound sets for biobjective combinatorial optimization problems. Computers & Operations Research 34(9), 2674–2694 (2007)
10. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Berlin (2004)
11. Kiziltan, G., Yucaoglu, E.: An algorithm for multiobjective zero-one linear programming. Management Science 29(12), 1444–1453 (1983)
12. Klamroth, K., Wiecek, M.M.: Dynamic programming approaches to the multiple criteria knapsack problem. Naval Research Logistics 47, 57–76 (2000)
13. Martin, P.D., Shmoys, D.B.: A new approach to computing optimal schedules for the job shop scheduling problem. In: Cunningham, W.H., Queyranne, M., McCormick, S.T. (eds.) IPCO 1996. LNCS, vol. 1084, pp. 389–403. Springer, Heidelberg (1996)
14. Mavrotas, G., Diakoulaki, D.: A branch and bound algorithm for mixed zero-one multiple objective linear programming. European Journal of Operational Research 107, 530–541 (1998)
15. Pisinger, D.: A minimal algorithm for the 0-1 knapsack problem. Operations Research 45, 758–767 (1997)
16. Serafini, P.: Some considerations about computational complexity for multiobjective combinatorial problems. In: Recent advances and historical development of vector optimization. LNEMS, vol. 294 (1986)
17. Sourd, F., Spanjaard, O.: A multi-objective branch-and-bound framework. Application to the bi-objective spanning tree problem. INFORMS Journal of Computing 20(3), 472–484 (2008)

18. Ulungu, B., Teghem, J.: The two-phase method: An efficient procedure to solve bi-objective combinatorial optimization problems. Foundations of Computing and Decision Sciences 20(2), 149–165 (1995)
19. Villareal, B., Karwan, M.H.: Multicriteria integer programming: A (hybrid) dynamic programming recursive approach. Mathematical Programming 21, 204–223 (1981)
20. Visée, M., Teghem, J., Pirlot, M., Ulungu, B.: Two-phases method and branch and bound procedures to solve the bi–objective knapsack problem. J. of Global Optimization 12(2), 139–155 (1998)

Improving Cutting Plane Generation with 0-1 Inequalities by Bi-criteria Separation

Edoardo Amaldi, Stefano Coniglio, and Stefano Gualandi

Politecnico di Milano, Dipartimento di Elettronica e Informazione, Italy
{amaldi,coniglio,gualandi}@elet.polimi.it

Abstract. In cutting plane-based methods, the question of how to generate the "best possible" cuts is a central and critical issue. We propose a bi-criteria separation problem for generating valid inequalities that simultaneously maximizes the cut violation and a measure of the diversity between the new cut and the previously generated cut(s). We focus on problems with cuts having 0-1 coefficients, and use the 1-norm as diversity measure. In this context, the bi-criteria separation amounts to solving the standard single-criterion separation problem (maximizing violation) with different coefficients in the objective function. We assess the impact of this general approach on two challenging combinatorial optimization problems, namely the Min Steiner Tree problem and the Max Clique problem. Computational experiments show that the cuts generated by the bi-criteria separation are much stronger than those obtained by just maximizing the cut violation, and allow to close a larger fraction of the gap in a smaller amount of time.

1 Introduction

Cutting planes are a central component of modern methods for solving Integer or Mixed-Integer Linear Programs (IPs or MILPs). In theory, proofs of convergence in a finite number of iterations are known for many important cases (e.g., for Integer Programs when separating w.r.t. Fractional Gomory cuts). In practice, however, convergence is difficult to achieve, often due to numerical issues. The relaxation of the problem, which is iteratively enriched with newly found violated inequalities and re-optimized, often becomes numerically ill-conditioned [5]. Cuts which are not valid can also be found [11]. In practice, cutting planes are typically used within a Branch-and-Cut framework.

A recurrent and important issue in cutting plane-based methods is the generation, at each iteration, of the "best possible" cut to be added to the current relaxation. The fundamental underlying question is to define a quantitative measure that favors (possibly strong) valid inequalities that are "better" than others [13]. The usual criterion for cutting plane generation is the maximization of the cut violation (or depth), i.e., the amount by which the cut is violated by the optimal solution of the current relaxation. The advantage of this criterion is that it leads to optimizing a linear objective function. Alternative criteria have been proposed, e.g., the maximization of the Euclidean distance from the optimal solution of the relaxation to the cut.

In several computational studies [3,2], a large number of cuts is first generated, and then a subset is selected to be added to the current relaxation. Usually, cuts are first ranked with respect to the Euclidean distance between the optimal solution and the cut and then with respect to their pairwise angles. As pointed out in [4], the purpose is to select, on the one hand, valid inequalities that cut away as much as possible the feasible region of the relaxation and, on the other hand, cuts that are as orthogonal as possible, typically to avoid similar cuts to be added. In the same paper, it is observed that, for Lift-and-Project cuts, when inequalities that cut the polyhedron of the relaxation from different angles are introduced, new tighter cuts are likely to be found in the next cutting plane iterations. In [6], when optimizing over the rank 1 Chvátal-Gomory closure, the presence of multiple maximally violated cuts is exploited by solving a modified separation problem, where an additional penalty term favors the generation of undominated cuts. The resulting solution, though, is no longer optimal w.r.t. the violation of the cut. In the same work, it is also observed that cuts which are as *diverse* as possible turn out to be computationally beneficial. These ides are systematically applied, for several families of cuts, in the solver SCIP [1].

In this work, we propose a bi-criteria separation problem for generating valid inequalities with 0-1 coefficients. This problem amounts to finding, among all maximally violated cuts, one that also maximizes a diversity measure between this cut and the previously generated one(s). As diversity measure, we use the 1-norm of the difference between two successive cuts. This clearly differs from what has been previously done in the literature, where a (fast) generation of a large number of valid inequalities is followed by a cut selection phase. We show that, whenever the cuts have 0-1 coefficients, the bi-criteria separation problem amounts to solving the standard single-criterion one (where the violation is maximized) with different coefficients in the objective function. We assess the impact of this general approach on two challenging combinatorial optimization problems, namely the Min Steiner Tree problem and the Max Clique problem.

2 Preliminaries

Consider the general Mixed-Integer Linear Program

$$\begin{aligned} (P) \quad & \min cx : Ax \le b \\ & s.t.\ x_i \in \mathbb{R},\ \text{for } i = 1, \dots, p \\ & \qquad x_i \in \mathbb{Z},\ \text{for } i = p+1, \dots, n \\ & \qquad \alpha x \le \alpha_0,\ \text{for } (\alpha, \alpha_0) \in \mathcal{C}, \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and $\mathcal{C}$ is the set of valid inequalities w.r.t. we want to separate. We suppose that $\mathcal{C}$ is known only implicitly, i.e., as the set of feasible solutions to some combinatorial problem, possibly described as another Mixed-Integer Linear Program.

In this work, we assume that $\mathcal{C}$ only contains inequalities with 0-1 coefficients, i.e. $\alpha \in \{0,1\}^n$. This encompasses a number of families of inequalities that are valid for many combinatoral optimization problems, e.g., cut, clique, stable set, rank, cycle, and cover inequalities [10]. For the sake of simplicity, here we assume $\alpha_0 = 1$, but the proposed approach is valid for any α_0.

Let t be the index of the cutting plane generation iteration. Let

$$(P^t) \qquad \min\{cx : Ax \leq b, x \in \mathbb{R}^n, \alpha x \leq 1 \text{ for } (\alpha, 1) \in \mathcal{C}^t\}$$

be the current relaxation of *(P)*, where only a subset $\mathcal{C}^t \subseteq \mathcal{C}$ of the inequalities is considered, and the integrality constraints are neglected. Let x^* be the corresponding optimal solution. The standard single-criterion separation problem, where a maximally violated cut is found, is

$$(\mathit{SEP}) \qquad \max\{\alpha x^* - 1 : (\alpha, 1) \in \mathcal{C}\}.$$

3 Bi-criteria Separation Problems

Let t be the iteration index of the current iteration, and let $\alpha^t x \leq 1$ be the cut generated at the previous iteration. Motivated by the considerations in Sec. 1, we would like to generate a cut $\alpha x \leq 1$ that not only maximizes the violation w.r.t. x^* but that also, among all maximally violated cuts, is as diverse as possible from $\alpha^t x \leq 1$.

As diversity measure, we propose the 1-norm between successive cuts, namely the function $||\alpha - \alpha^t||_1$, where $||v||_1 = \sum_{i=1}^n |v_i|$ for any $v \in \mathbb{R}^n$. Since all α_i and α_i^t take 0-1 values, we clearly have

$$\sum_{i=1}^n | \alpha_i - \alpha_i^t | = \sum_{i=1}^n \alpha_i (1 - \alpha_i^t) + \sum_{i=1}^n (1 - \alpha_i) \alpha_i^t = \sum_{i=1}^n \alpha_i - 2 \sum_{i=1}^n \alpha_i \alpha_i^t + \sum_{i=1}^n \alpha_i^t. \quad (1)$$

By letting e denote the all ones vector, and neglecting the last term, which is constant, this diversity measure can be expressed as $e\alpha - 2\alpha\alpha^t$. This is of practical importance, since we aim at a separation problem which is, computationally, not too hard to solve. From an empirical point of view, $||\alpha - \alpha^t||_1$ also nicely captures the differences between the angles of 0-1 vectors.

The cut violation $\alpha x^* - 1$ and the diversity measure $||\alpha - \alpha^t||_1$ can be combined into the following bi-criteria separation problem, where they are optimized with priority:

$$(\mathit{BC\text{-}SEP}) \qquad \max \left\{ \alpha x^* - 1 + \epsilon (e\alpha - 2\alpha\alpha^t) : (\alpha, 1) \in \mathcal{C} \right\}.$$

A small enough value of the proportionality parameter $\epsilon > 0$ can always be found so as to guarantee that the priority between $\alpha x^* - 1$ and $e\alpha - 2\alpha\alpha^t$ is respected. It can be derived as follows. Let $M := n$ be the maximum of $e\alpha - 2\alpha\alpha^t$, and $m := e\alpha^t$ its minimum. Let also $\delta := \min_{i=1,\dots,n}\{x_i^*\}$ be the smallest variation in $\alpha x^* - 1$, corresponding to a variation in the variables α. Then, it suffices to select ϵ such that $\epsilon(M - m) < \delta$.

We also propose the variant of *(BC-SEP)* in which the diversity w.r.t. the whole set of previously generated cuts is taken into account. This is achieved by defining the multi-cut bi-criteria separation problem as

$$(\textit{MC-BC-SEP}) \qquad \max\left\{\alpha x^* - 1 + \epsilon(e\alpha - 2\alpha\bar{\alpha}^t) : (\alpha, 1) \in \mathcal{C}\right\},$$

where $\bar{\alpha}^t := \frac{1}{t}\sum_{l=1}^{t} \alpha^l$ is the average of all the previously generated cuts. Since (1) also holds when $\bar{\alpha}^t \in [0,1]^n$, the objective function has the same structure of that of *(BC-SEP)*.

4 Solving the Bi-criteria Separation Problems

Since the objective functions of *(BC-SEP)* and *(MC-BC-SEP)* are linear functions of the variable α, any of the two proposed separation problems is obtained by just taking *(SEP)*, and changing its objective function vector x^* into a new vector $\hat{x}^*$. For *(BC-SEP)*, $\hat{x}^*_i := x^*_i + \epsilon(2\alpha^t_i - 1)$, for all $i = 1, \ldots, n$. For *(MC-BC-SEP)*, $\hat{x}^*_i := x^*_i + \epsilon(2\bar{\alpha}^t_i - 1)$, for all $i = 1, \ldots, n$. In this sense, the bi-criteria separation problem just amounts to solving the standard *(SEP)* to separate $\hat{x}^*$ instead of x^*. Note, however, that even if $x^* \geq 0$, $\hat{x}^*$ can be negative.

The bi-criteria separation problem is here adapted to two challenging combinatorial problems: Min Steiner Tree and Max Clique. For both of them, we consider a single family $\mathcal{C}$ of facet defining valid inequalities.

Min Steiner Tree. Given a graph $G = (V, E)$, a set $T \subset V$ of terminals and a cost function $c : E \to \mathbb{R}^+$, find a Steiner tree of G, i.e., a tree that spans all the nodes in T, of minimum total cost. Let $G' = (V, A)$ be the directed version of G, with a pair of arcs $(i, j), (j, i)$ for each edge $\{i, j\} \in E$. Let $r \in T$ be an arbitrary root node. As in [8], we consider the following directed formulation of the problem, where the integrality constraints are relaxed:

$$\min \quad \sum_{(ij)\in A} c_{ij}x_{ij} \tag{2}$$

$$\text{s.t.} \quad \sum_{(ij)\in\nu^+(S)} x_{ij} \geq 1, \ \text{ for } S \subset V : r \in S, V \setminus S \cap T \neq \emptyset, \tag{3}$$

$$0 \leq x_{ij} \leq 1, \qquad \text{for } (i,j) \in A. \tag{4}$$

In this case, we take as $\mathcal{C}$ the set of all cut inequalities (3). The separation problem amounts to finding, for each terminal $t \in T \setminus \{r\}$, a Min $s - t$ Cut[1] on G' with $s = r$, where the values of the solution x^*_{ij} of the current relaxation are used as arc capacities. This separation problem can be formulated as the following Linear Program

[1] To comply with the standard notation, in this subsection, the index t denotes a terminal node in T, instead of the index of the cutting plane algorithm iteration.

$$\min \sum_{(ij)\in A} x^*_{ij}\alpha_{ij} \tag{5}$$

$$\text{s.t. } \alpha_{ij} \geq \pi_i - \pi_j, \text{ for } (ij) \in A, \tag{6}$$

$$\pi_r - \pi_\tau \geq 1, \tag{7}$$

$$\alpha_{ij} \geq 0, \quad \text{for } (i,j) \in A, \tag{8}$$

where α is the incidence vector of an $s-t$ cut and π is the vector of node potentials. Since the cut inequalities can be separated in polynomial time, due to the equivalence between separation and optimization [12], (2)–(4) can be also solved in polynomial time.

Max Clique. Given an undirected graph $G = (V, E)$, find a maximum clique of G, i.e., a largest set of nodes that are pairwise connected. We consider the following formulation of the problem:

$$\max \sum_{i\in V} x_i \tag{9}$$

$$\text{s.t. } \sum_{i\in S} x_i \leq 1, \quad \text{for } S \in \mathcal{S}, \tag{10}$$

$$0 \leq x_{ij} \leq 1, \text{ for } \{i,j\} \in E, \tag{11}$$

where the integrality constraints are relaxed. In this case, we take as $\mathcal{C}$ the set $\mathcal{S}$ of all maximal Stable Sets of G. The separation problem, which amounts to finding a maximum weighted stable set in G, where the weights are given by the components of x^*, and is formulated as the following 0-1 Integer Program

$$\max\left\{\alpha x^* - 1 : \alpha_i + \alpha_j \leq 1, \text{ for } \{i,j\} \in E, \alpha \in \{0,1\}^n\right\},$$

where $\alpha \in \{0,1\}^n$ is the incidence vector of a stable set. Note that, since this separation problem is $\mathcal{NP}$-hard again because of the equivalence between separation and optimization, solving (9)-(11) to optimality is also $\mathcal{NP}$-hard.

For the separation of cut inequalities, the following result holds. Remember that, because of the Linear Programming duality, an $s-t$ cut of minimum total capacity can be found by solving the corresponding Max Flow problem.

Proposition 1. *Solving* (BC-SEP) *or* (MC-BC-SEP) *for cut inequalities amounts to solving a Max Flow problem where* $\alpha_{ij} = 0$ *for all* $(ij) : \hat{x}^*_{ij} < 0$.

Proof. Consider the Linear Programming dual of (5)–(8), with the additional upper bounds $\mu_{ij} \leq 1$ for $(ij) \in A$. *(BC-SEP)* or *(MC-BC-SEP)* amounts to

$$\max \phi - \sum_{(ij)\in A} h_{ij} \tag{12}$$

$$\text{s.t. } \sum_{ij} x_{ij} - \sum_{ji} x_{ij} = \begin{cases} \phi & \text{if } i = r \\ -\phi & \text{if } i = t \\ 0 & \text{else} \end{cases} \tag{13}$$

$$0 \leq x_{ij} \leq \hat{x}^*_{ij} + h_{ij}, \quad \text{for } (i,j) \in A, \tag{14}$$

$$h_{ij} \geq 0, \quad \text{for } (i,j) \in A, \tag{15}$$

which is a Max Flow problem with the extra set of variables h_{ij}, where ϕ is the value of the flow. If $\hat{x}^*_{ij} \geq 0$ for all $(i,j) \in A$, by letting any $h_{ij} = q$, the resulting capacity on the arc (i,j) is increased by q, and q units of cost are paid in the objective function, while ϕ is increased by q only if (i,j) belongs to the unique optimal cut of the network. If it is the case, the objective function value is unaltered and hence this solution is equivalent to the corresponding one in which $h_{ij} = 0$. Otherwise, if the cut containing (i,j) is not unique or not optimal, in any optimal solution, h_{ij} will be 0 for all $(i,j) \in A$. It follows that an optimal solution can always be achieved by letting $h_{ij} = 0$ for all $(i,j) \in A$. Whenever $\hat{x}^*_{ij} < 0$ for any (i,j), $h_{ij} \geq -\hat{x}^*_{ij}$ is implied, and (12)–(15) can be reformulated by adding the term $-\sum_{(i,j)\in A:\hat{x}^*_{ij}<0} \hat{x}^*_{ij}$ to (12) and substituting 0 for every $\hat{x}^*_{ij} < 0$, thus obtaining a problem with capacities $\hat{x}^*_{ij} \geq 0$, where the h_{ij} variables can be set to 0. □

For the separation of stable set inequalities, a similar results holds.

Fact 1. *Solving* (BC-SEP) *or* (MC-BC-SEP) *for stable set inequalities amounts to solving a Max Weighted Stable Set problem where* $\alpha_i = 0$ *for all* $i \in V : \hat{x}^*_i < 0$.

5 Computational Results

In this section, we assess the impact of the proposed bi-criteria separation problems *(BC-SEP)* and *(MC-BC-SEP)*, in the context of a pure cutting plane algorithm, when compared to *(SEP)*. The experiments are carried out for the Min Steiner Tree and Max Clique problems.

The algorithms are implemented in C++, using the gnu-g++-4.3 compiler. For both problems, the relaxations are solved with CPLEX 11 (with default parameters). The Boost Graph Library is used for the graph algorithms and data structures. The experiments are carried out on a standard desktop computer with an Intel Core2Duo processor and 2.0 GB of RAM.

5.1 Min Steiner Tree

We compare *(SEP)* and *(BC-SEP)* for the Min Steiner Tree problem on two set of instances taken from the SteinLib [9]: I640 and PUC. Those sets are among the hardest in the library. We consider the following setting. At each iteration, we generate a round of cuts containing as many violated inequalities as they are found, by solving a Max Flow problem for each pair (r,t) with $t \in T \setminus \{r\}$. When solving *(BC-SEP)* for terminal t, we consider, as previous cut, the last cut generated when solving *(BC-SEP)* w.r.t. the same terminal. The separation problems are solved using the Edmonds-Karp's algorithm that has complexity $O(|V|\,|E|^2)$. The root node $r \in T$ is chosen as the terminal with the largest node degree, i.e. the number of neighbors in G. Experimentally, we observed that this choice allows to close a much larger fraction of the gap when using both *(SEP)* and *(BC-SEP)*. For each instance, we derive an initial pool $\mathcal{C}^0$ of cuts by solving, for each pair of source $s = r$ and sink $t \in T \setminus \{r\}$, a Min $s-t$ Cut problem with unit capacity on every arc.

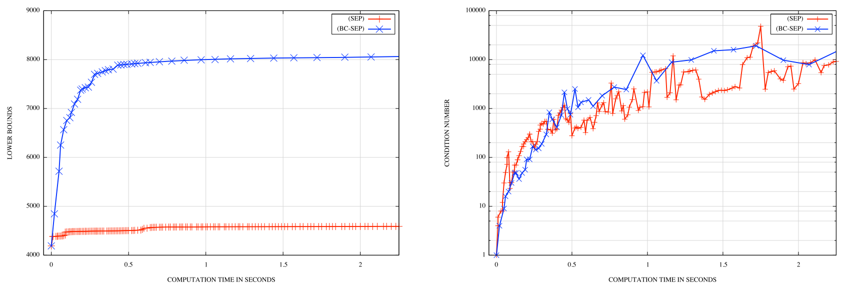
Table 1. *(SEP)* vs. *(BC-SEP)*, on the I640 SteinLib instances

				(SEP)				*(BC-SEP)*			
Instance	$\|V\|$	$\|E\|$	$\|T\|$	Gap	Time	Rnds	Avgκ	Gap	Time	Rnds	Avgκ
i640-001	640	1920	9	47.4	1000.0	1024	2.9E+5	**100.0**	**0.5**	**75**	**5.1E+3**
i640-011	640	8270	9	81.4	1000.0	1005	2.9E+6	**100.0**	**5.7**	**146**	**1.2E+4**
i640-031	640	2560	9	47.9	1000.0	943	6.9E+5	**100.0**	**0.9**	**79**	**8.5E+3**
i640-041	640	40896	9	**78.7**	1000.0	2228	9.0E+0	**78.7**	1000.0	**1985**	**6.0E+0**
i640-101	640	1920	25	48.4	1000.0	779	3.9E+5	**100.0**	**4.0**	**57**	**1.1E+5**
i640-131	640	2560	25	58.6	1000.0	729	3.9E+5	**100.0**	**5.7**	**55**	**1.1E+5**
i640-141	640	40896	25	91.1	1000.0	333	1.5E+5	**97.5**	1000.0	**138**	**4.4E+3**
i640-201	640	1920	50	48.2	1000.0	600	2.6E+6	**100.0**	**2.6**	**48**	**1.5E+5**
i640-211	640	4135	50	80.4	1000.0	515	**2.5E+5**	**97.6**	1000.0	**114**	4.7E+5
i640-231	640	2560	50	56.2	1000.0	588	2.0E+6	**99.7**	**243.9**	**119**	**6.0E+5**
i640-301	640	1920	160	55.7	1000.0	394	**7.0E+4**	**100.0**	**10.1**	**28**	7.8E+4
i640-311	640	8270	160	84.5	1000.0	**1335**	**9.0E+0**	**86.4**	1000.0	1376	1.2E+1
i640-331	640	2560	160	61.5	1000.0	**2338**	**9.0E+0**	**62.6**	1000.0	2482	**9.0E+0**
Aggregate				64.6				94.0	0.04	0.2	0.2

Table 2. *(SEP)* vs. *(BC-SEP)*, on the PUC SteinLib instances

				(SEP)				*(BC-SEP)*			
Instance	$\|V\|$	$\|E\|$	$\|T\|$	Gap	Time	Rnds	Avgκ	Gap	Time	Rnds	Avgκ
cc3-4p	64	288	8	**91.9**	334.5	871	1.6E+5	91.2	**10.6**	**148**	**4.7E+4**
cc3-4u	64	288	8	79.5	1000.0	903	5.5E+4	**90.5**	**31.7**	**240**	**5.1E+4**
cc3-5p	125	750	13	30.5	1000.0	872	**3.5E+5**	**91.8**	**374.6**	**352**	6.2E+5
cc3-5u	125	750	13	29.9	1000.0	843	3.4E+5	**90.3**	1000.0	**305**	**1.3E+5**
cc5-3p	243	1215	27	8.5	1000.0	521	**1.2E+5**	**97.9**	1000.0	**163**	4.1E+5
cc5-3u	243	1215	27	8.2	1000.0	510	**1.2E+5**	**97.3**	1000.0	**157**	3.5E+5
cc6-2p	64	192	12	**94.1**	96.6	531	9.4E+4	93.7	**1.5**	**61**	**2.3E+4**
cc6-2u	64	192	12	66.6	1000.0	866	1.4E+5	**92.7**	**2.5**	**74**	**5.3E+4**
hc10p	1024	5120	512	0.6	1000.0	307	**1.1E+5**	**97.4**	1000.0	**24**	1.2E+6
hc10u	1024	5120	512	0.5	1000.0	409	**2.5E+3**	**97.3**	1000.0	**19**	6.1E+4
hc6p	64	192	32	**96.6**	188.2	431	7.7E+4	96.5	**0.1**	**15**	**7.5E+2**
hc6u	64	192	32	41.9	1000.0	497	4.2E+4	**95.2**	**8.2**	**55**	**1.5E+4**
hc7p	128	448	64	17.4	1000.0	346	9.6E+4	**96.6**	**1.2**	**21**	**1.0E+4**
hc7u	128	448	64	14.2	1000.0	399	**1.2E+5**	**95.2**	**406.9**	**90**	1.3E+5
hc8p	256	1024	128	3.7	1000.0	228	**2.1E+5**	**98.6**	**30.9**	**33**	2.6E+5
hc8u	256	1024	128	3.2	1000.0	459	**1.1E+5**	**98.0**	1000.0	**47**	1.5E+5
hc9p	512	2304	256	1.4	1000.0	261	**2.2E+5**	**98.6**	**153.3**	**35**	4.6E+5
hc9u	512	2304	256	1.0	1000.0	571	**4.9E+4**	**98.2**	1000.0	**25**	9.6E+4
Aggregate				32.8				95.4	0.1	0.1	0.9

For all the experiments, illustrated in Tab. 1 and Tab. 2, the time limit was set to 1000 seconds. We report the fraction of gap closed (Gap), the CPU time in seconds (Time), the number of rounds of cuts generated (Rnds), and the exact condition number of the scaled basis matrix (see `ExactKappa` in CPLEX User Guide), averaged of the over the last 20 iterations (Avgκ). The gap is evaluated w.r.t. the value of an optimal solution, if known, or w.r.t. the best known upper bound. The condition number is averaged to mitigate the oscillations that we observed along the runs. The best values for *(SEP)* and *(BC-SEP)* are reported in bold. For the columns Time, Cuts, and Avgκ of *(BC-SEP)*, we computed the ratios, over all instances, of the corresponding value and that obtained with *(SEP)*. The inverse of their geometric mean is reported in the last line (Aggregate). In the same line, for the Gap column, we report the arithmetic mean for both *(SEP)* and *(BC-SEP)*.



(a) Lower Bound (b) Condition Number κ

Fig. 1. Growth rate of the lower bound and the condition number as a function of the CPU time for the instance `i640-131`. The vertical axis in (b) is in logarithmic scale.

Table 1 reports the results for the I640 instances. With *(BC-SEP)*, we managed to solve to optimality 7 instances out of 13. None is solved, within the time limit, with *(SEP)*. On average, with *(BC-SEP)* 94% of the gap was closed, as opposed to the 64% closed with *(SEP)*, generating only 20% of the cuts and in just 4% of the CPU time. The average condition number is also 20% smaller. Figure 1.a and 1.b show the relation between the improvement of the bounds and the increase in the condition numbers for *(SEP)* and *(BC-SEP)*, on two I640 instances. Note that, although the growth rates of the condition number are similar, the improvement in the bound for *(BC-SEP)*, with respect to that for *(SEP)*, is substantial. Table 2 shows the computational results obtained for the PUC instances. With *(BC-SEP)*, 11 instances are solved to optimality out of 18, while with *(SEP)* only 3 are solved. On average, with *(BC-SEP)* 95% of the gap is closed, as opposed the 32.8% closed with *(SEP)*, in 10% of the time, and also generating 20% of the rounds of cuts.

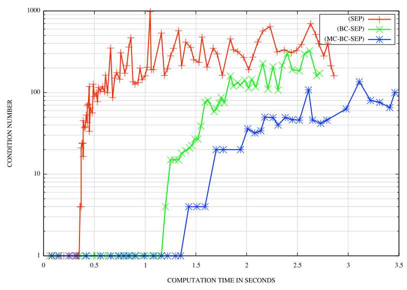
5.2 Max Clique

We compare *(SEP)*, *(BC-SEP)*, and *(MC-BC-SEP)* for the Max Clique problem on a set of instances taken from the Second DIMACS Implementation Challenge [7]. We consider a setting where the pool of cuts $\mathcal{C}^0$ is initialized as the empty set. Therefore, the initial relaxation *(P^0)* only contains the box constraints (11). Since we consider a single family of inequalities, and solve each separation problem to optimality, a single cut is added at each round. The separation problems are solved with CPLEX.

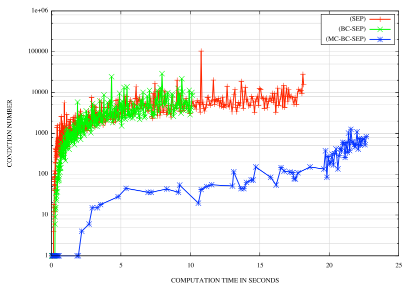
Table 3 reports, for each of *(SEP)*, *(BC-SEP)*, and *(MC-BC-SEP)* and for each instance, the CPU time in seconds (Time), the number of cuts generated (Cuts) and the condition (κ) of the basis matrix of the solution of the last relaxation, as computed by CPLEX. The time limit is set to 1000 seconds. When *(BC-SEP)* or *(MC-BC-SEP)* outperform *(SEP)*, or *(SEP)* outperforms both, w.r.t. either of the quantities that we report, the corresponding value is highlighted in bold. The last line (Aggregate) reports, for each column of *(BC-SEP)* and

Table 3. *(SEP)* vs. *(BC-SEP)* and *(MC-BC-SEP)*, on DIMACS Max Clique instances

			(SEP)			*(BC-SEP)*			*(MC-BC-SEP)*		
Instance	$\|V\|$	$\|E\|$	Time	Cuts	κ	Time	Cuts	κ	Time	Cuts	κ
c-fat200-1	200	1534	2.0	222	1314	**1.6**	**183**	**1079**	**1.2**	**162**	**785**
c-fat200-2	200	3235	0.5	79	103	0.8	**77**	**73**	**0.5**	**45**	**2**
c-fat200-5	200	8473	6.6	260	299	6.8	249	290	**5.4**	**206**	**192**
c-fat500-10	500	4459	50.2	410	194	59.1	**409**	330	**34.9**	**249**	**2**
c-fat500-1	500	46627	22.3	450	3976	**11.9**	**329**	**2851**	0.4	**27**	**12**
c-fat500-2	500	9139	59.4	544	4397	64.7	551	4626	**7.2**	**182**	**1494**
c-fat500-5	500	23191	12.8	247	401	18.3	258	418	**8.7**	**127**	**69**
hamming6-2	64	1824	**0.6**	89	35	1.2	101	**30**	0.7	**62**	**12**
hamming6-4	64	704	22.6	232	483	**4.3**	**70**	**234**	**3.5**	**60**	**156**
hamming8-2	256	31616	> *1000*	*57*	*233*	189.5	497	**75**	73.2	236	**40**
johnson16-2-4	120	5460	97.4	515	993	**31.6**	**49**	**38**	**12.8**	**22**	**11**
johnson8-2-4	28	210	0.1	47	43	**0.0**	**12**	**6**	**0.0**	**12**	**6**
johnson8-4-4	70	1855	**3.2**	110	181	3.3	**59**	**76**	3.5	**53**	**52**
MANN_a9	45	918	**0.3**	**49**	51	0.9	**49**	**25**	0.9	**49**	**24**
myciel3	11	20	**0.0**	23	28	0.1	**16**	**19**	0.1	**16**	**19**
myciel4	23	71	1.1	67	136	**0.7**	**39**	**83**	**0.6**	**31**	**59**
myciel5	47	236	21.9	228	782	**8.2**	**90**	**564**	**4.9**	**71**	**298**
queen10_10	100	2940	41.4	391	1524	**41.1**	**339**	**1330**	43.4	**338**	**1365**
queen11_11	121	3960	40.5	433	2194	**32.9**	**351**	3058	**35.1**	**353**	6804
queen12_12	144	5192	54.2	488	1999	**46.3**	**407**	**1056**	**42.1**	**372**	2974
queen13_13	169	6656	71.0	555	2651	**62.0**	**456**	**2631**	**65.4**	**453**	**2527**
queen14_14	196	8372	121.0	648	3553	**120.4**	**538**	**3235**	137.0	**542**	**3106**
queen15_15	225	10360	234.7	713	4211	**217.1**	**608**	**4179**	**193.8**	**600**	**4080**
queen16_16	256	12640	351.7	809	4910	**335.6**	**673**	**4905**	**350.6**	**689**	**4887**
Aggregate						0.9	0.7	0.7	0.6	0.5	0.3



(a) johnson8-4-4



(b) c-fat-500-1

Fig. 2. Growth rate of the condition number as a function of the CPU time for the instances `johnson8-4-4` and `c-fat-500-1`. The vertical axis is in logarithmic scale.

(MC-BC-SEP), the inverse of the geometric means of the ratios, over all instances, of the corresponding column value and that obtained with *(SEP)*. The instance *hamming8-2*, reported in italics, is neglected, since with *(SEP)* the cutting plane algorithm did not terminate within the time limit. Since, within the time limit, all problems were solved to optimality, we omit the column Gap. With *(BC-SEP)*, when compared to *(SEP)*, we solve all the instances in, on average, 90% of the time, generating 70% of the cuts and yielding a final relaxation with 70% the condition number. *(MC-BC-SEP)* yields even better results. When compared to *(SEP)*, we manage, on average, to solve all the problem in 60% of the time, generating 50% of the cuts and yielding a final condition number 30% smaller.

Figure 2 shows the growth rate of the condition number for *(SEP)*, *(BC-SEP)*, and *(MC-BC-SEP)*, plotted as a function of the CPU time, for two instances.

6 Concluding Remarks

We have proposed a bi-criteria separation problem for the generation of cutting panes. This problem amounts to generating, by solving a single optimization problem, a maximally violated cut which also maximizes, among all the maximally violated ones, a measure of the diversity between the new cut and the previously generated one(s). For cuts with 0-1 coefficients, the bi-criteria separation problem reduces to a standard single-criterion separation problem with a different objective function vector.

Computational results for the Min Steiner Tree and Max Clique problems show that, by solving the bi-criteria separation problem, not only tighter bounds are obtained in a shorter time, but also significantly less cuts are generated.

References

1. Achterberg, T.: Constrained Integer Programming. PhD thesis, Technische Universitat at Berlin (2007)
2. Andreello, G., Caprara, A., Fischetti, M.: Embedding 0,1/2-cuts in a branch-and-cut framework: a computational study. J. on Computing 19(2), 229–238 (2007)
3. Balas, E., Ceria, S., Cornuéjols, G.: A lift-and-project cutting plane algorithm for mixed 0-1 programs. Math. Program. 58(1-3), 295–324 (1993)
4. Balas, E., Ceria, S., Cornuéjols, G.: Mixed 0–1 programming by lift-and-project in a branch-and-cut framework. Manag. Sci. 42(9), 1229–1246 (1996)
5. Balas, E., Fischetti, M., Zanette, A.: Can pure cutting plane algorithms work? In: Lodi, A., Panconesi, A., Rinaldi, G. (eds.) IPCO 2008. LNCS, vol. 5035, pp. 416–434. Springer, Heidelberg (2008)
6. Fischetti, M., Lodi, A.: Optimizing over the first Chvatal closure. Math. Program. 110(1), 3–20 (2006)
7. Johnson, D.S., Trick, M.A.: Cliques, Coloring, and Satisfiability. In: Second DIMACS Implementation Challenge. DIMACS Series in Disc. Math. and Theo. Comp. Scie. Amer. Math. Soc., vol. 26 (1996)
8. Koch, T., Martin, A.: Solving Steiner Tree Problems in Graphs to Optimality. Networks 32(3), 207–232 (1998)
9. Koch, T., Martin, A., Voß, S.: SteinLib: An updated library on steiner tree problems in graphs. Technical Report ZIB-Report 00-37, ZIB Berlin, Takustr. 7, Berlin (2000)
10. Marchand, H., Martin, A., Weismantel, R., Wolsey, L.: Cutting planes in integer and mixed integer programming. Discrete Applied Mathematics 123, 397–446 (2002)
11. Margot, F.: Testing Cut Generators for mixed-integer linear programming. Math. Prog. Comp. 1, 69–95 (2009)
12. Nemhauser, G., Wolsey, L.: Integer and Combinatorial Optimization. Wiley, Chichester (1988)
13. Padberg, M., Rinaldi, G.: A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Reviews 33, 60–100 (1991)

New Lower Bounds for the Vehicle Routing Problem with Simultaneous Pickup and Delivery

Anand Subramanian[1], Eduardo Uchoa[2], and Luiz Satoru Ochi[1]

[1] Universidade Federal Fluminense, Instituto de Computação, Rua Passo da Pátria 156 - Bloco E - 3° andar, Niterói-RJ 24210-240, Brazil
[2] Universidade Federal Fluminense, Departamento de Engenharia de Produção, Rua Passo da Pátria 156 - Bloco E - 4° andar, Niterói-RJ 24210-240, Brazil

Abstract. This work deals with the Vehicle Routing Problem with Simultaneous Pickup and Delivery. We propose undirected and directed two-commodity flow formulations, which are based on the one developed by Baldacci, Hadjiconstantinou and Mingozzi for the Capacitated Vehicle Routing Problem. These new formulations are theoretically compared with the one-commodity flow formulation proposed by Dell'Amico, Righini and Salani. The three formulations were tested within a branch-and-cut scheme and their practical performance was measured in well-known benchmark problems. The undirected two-commodity flow formulation obtained consistently better results. Several optimal solutions to open problems with up to 100 customers and new improved lower bounds for instances with up to 200 customers were found.

Keywords: Vehicle Routing, Simultaneous Pickup and Delivery, Commodity Flow Formulations.

1 Introduction

The Vehicle Routing Problem with Simultaneous Pickup and Delivery (VRPSPD) is a variant of the Capacitated Vehicle Routing Problem (CVRP), in which clients require both pickup and delivery services. This problem was first proposed two decades ago by Min [1]. The VRPSPD is clearly $\mathcal{NP}$-hard since it can be reduced to the CVRP when all the pickup demands are equal to zero. Practical applications arise especially in the Reverse Logistics context. Companies are increasingly faced with the task of managing the reverse flow of finished goods or raw-materials. Thus, one should consider not only the Distribution Logistics, but also the management of the reverse flow.

The VRPSPD can be defined as follows. Let $G = (V, E)$ be a complete graph with a set of vertices $V = \{0, ..., n\}$, where the vertex 0 represents the depot and the remaining ones the customers. Each edge $\{i, j\} \in E$ has a non-negative cost c_{ij} and each client $i \in V' = V - \{0\}$ has non-negative demands d_i for delivery and p_i for pickup. Let $C = \{1, ..., m\}$ be a set of homogeneous vehicles with capacity Q. The VRPSPD consists in constructing a set up to m routes in such a way that: (i) every route starts and ends at the depot; (ii) all the pickup and delivery demands are accomplished; (iii) the vehicle's capacity is not exceeded in any part of a route; (iv) a customer is visited by only a single vehicle; (v) the sum of costs is minimized.

Although heuristic strategies are by far the most employed to solve the VRP-SPD, some exact algorithms were also explored in the literature. A branch-and-price algorithm was developed by Dell'Amico et al. [2], in which two different strategies were used to solve the subpricing problem: (i) exact dynamic programming and (ii) state space relaxation. The authors managed to find optimal solutions for instances with up to 40 clients. Angelelli and Manisini [3] also developed a branch-and-price approach based on the set covering formulation, but for the VRPSPD with time-windows constraints. The subproblem was formulated as a shortest-path with resource constraints but without the elementary condition and it was solved by applying a permanent labeling algorithm. The authors were able find optimal solutions for instances with up to 20 clients. Three-index formulations for the VRPSPD were proposed by Dethloff [4] and Montané and Galvão [5], however only the last authors had tested it in practice. They ran their formulation in CPLEX 9.0 within a time limit of 2 hours and had reported the lower bounds produced for benchmark instances with 50-400 customers.

In this work we propose an undirected and a directed two-commodity flow formulations for the VRPSPD. These formulations extend the one developed by Baldacci et al. [6] for the CVRP. They were compared with the one-commodity flow formulation presented by Dell'Amico et al. [2]. In addition, the three formulations were implemented within a branch-and-cut algorithm, including cuts from the CVRPSEP library [7], and they were tested in well-known benchmark problems with up to 200 customers.

The remainder of this paper is organized as follows. Section 2 describes the one-commodity flow formulation [2]. In Section 3 we present the undirected and the directed two-commodity flow formulations for the VRPSPD and we compare these formulations with the one developed in [2]. Section 4 contains the experimental results obtained by means of a branch-and-cut algorithm. Section 5 presents the concluding remarks.

2 One-Commodity Flow Formulation

Reasonably simple and effective formulations for the CVRP can be defined only over the natural edge variables (arc variables in the asymmetric case), see [8]. Similar formulations are not available for the VRPSPD. This difference between these two problems can be explained as follows. In the CVRP, the feasibility of a route can be determined by only checking whether the sum of its client demands does not exceed the vehicle's capacities. In contrast, the feasibility of a VRPSPD route depends crucially on the sequence of visitation of the clients.

The following directed one-commodity flow formulation for the VRPSPD was proposed by Dell'Amico et al. [2]. Define A as the set of arcs consisting of a pair of opposite arcs (i, j) and (j, i) for each edge $\{i, j\} \in E$ and let D_{ij} and P_{ij} be the flow variables which indicate, respectively, the delivery and pickup loads carried along the arc $(i, j) \in A$. Let x_{ij} be 1 if the arc $(i, j) \in A$ is in the solution and 0 otherwise. The formulation F1C is described next.

$$\min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} \tag{1}$$

subject to

$$\sum_{j \in V} x_{ij} = 1 \qquad \forall i \in V' \tag{2}$$

$$\sum_{j \in V} x_{ji} = 1 \qquad \forall i \in V' \tag{3}$$

$$\sum_{j \in V'} x_{0j} \leq m \tag{4}$$

$$\sum_{j \in V} D_{ji} - \sum_{j \in V} D_{ij} = d_i \qquad \forall i \in V' \tag{5}$$

$$\sum_{j \in V} P_{ij} - \sum_{j \in V} P_{ji} = p_i \qquad \forall i \in V' \tag{6}$$

$$D_{ij} + P_{ij} \leq Q x_{ij} \qquad \forall (i, j) \in A \tag{7}$$

$$D_{ij} \geq 0 \qquad \forall (i, j) \in A \tag{8}$$

$$P_{ij} \geq 0 \qquad \forall (i, j) \in A \tag{9}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in A \tag{10}$$

The objective function (1) minimizes the sum of the travel costs. Constraints (2)-(3) impose that each client should be visited exactly once. Constraints (4) refer to the number of vehicles available. Constraints (5)-(7) are the flow conservation equalities. Constraints (8)-(10) define the domain of the decision variables.

Dell'Amico et al. [2] basically extended the one-commodity flow formulation proposed by Gavish and Graves [9] for the CVRP by adding constraints (6) and (9), and the term P_{ij} in (7). Gouveia [10] showed that it is possible to obtain stronger inequalities for D_{ij} by using the tighter bounds (11) instead of (8) in the Gavish and Graves formulation. Accordingly, we can apply the same idea to develop stronger inequalities for P_{ij} by replacing (9) with (12) and for $D_{ij} + P_{ij}$ by replacing (7) with (13).

$$d_j x_{ij} \leq D_{ij} \leq (Q - d_i) x_{ij} \qquad \forall (i, j) \in A \tag{11}$$

$$p_i x_{ij} \leq P_{ij} \leq (Q - p_j) x_{ij} \qquad \forall (i, j) \in A \tag{12}$$

$$D_{ij} + P_{ij} \leq (Q - \max\{0, p_j - d_j, d_i - p_i\}) x_{ij} \qquad \forall (i, j) \in A \tag{13}$$

It should be noticed that a lower bound for (13) is implicit in (11) and (12), i.e, $D_{ij} + P_{ij} \geq d_j x_{ij} + p_i x_{ij}$. Another valid inequality for F1C, given by (14), is due to the fact that each edge not adjacent to the depot is traversed at most once, i.e.

$$x_{ij} + x_{ji} \leq 1 \qquad \forall i, j, i < j, \in V' \tag{14}$$

3 Two-Commodity Flow Formulations

In this section we present both an undirected and a directed two-commodity flow formulations for the VRPSPD which are based on the one proposed by Baldacci et al. [6] for the CVRP.

3.1 Undirected Two-Commodity Flow Formulation

For the sake of convenience let vertex $n+1$ be a copy of the depot, $\bar{V} = V \cup \{n+1\}$ and $\bar{E}$ be the complete set of edges $\bar{E}$, excepting $\{0, n+1\}$. Let x'_{ij} be 1 if the edge $\{i,j\} \in \bar{E}$ is in the solution and 0 otherwise. Let the variables D'_{ij}, P'_{ij} and SPD_{ij} denote, respectively, the delivery, pickup and simultaneous pickup and delivery flows when a vehicle goes from $i \in \bar{V}$ to $j \in \bar{V}$ and let the same variables denote, respectively, the associated residual capacities when a vehicle goes from $j \in \bar{V}$ to $i \in \bar{V}$, in such a way that $D'_{ij} + D'_{ji} = Qx'_{ij}$, $P'_{ij} + P'_{ji} = Qx'_{ij}$ and $SPD_{ij} + SPD_{ji} = Qx'_{ij}$. Also, an integer variable v, which denotes the number of vehicles utilized, is included with an upper bound m.

The undirected formulation F2C-U is as follows.

$$\min \sum_{\{i,j\} \in \bar{E}} c_{ij} x'_{ij} \tag{15}$$

subject to

$$\sum_{i \in \bar{V}, i<k} x'_{ik} + \sum_{j \in \bar{V}, j>k} x'_{kj} = 2 \qquad \forall k \in V' \tag{16}$$

$$\sum_{j \in \bar{V}} (D'_{ji} - D'_{ij}) = 2d_i \qquad \forall i \in V' \tag{17}$$

$$\sum_{j \in V'} D'_{0j} = \sum_{i \in V'} d_i \tag{18}$$

$$\sum_{j \in V'} D'_{j0} = vQ - \sum_{i \in V'} d_i \tag{19}$$

$$\sum_{j \in \bar{V}} (P'_{ij} - P'_{ji}) = 2p_i \qquad \forall i \in V' \tag{20}$$

$$\sum_{j \in V'} P'_{jn+1} = \sum_{i \in V'} p_i \tag{21}$$

$$\sum_{j \in V'} P'_{n+1j} = vQ - \sum_{i \in V'} p_i \tag{22}$$

$$\sum_{j \in \bar{V}} (SPD_{ji} - SPD_{ij}) = 2(d_i - p_i) \qquad \forall i \in V' \tag{23}$$

$$SPD_{0j} = D'_{0j} \qquad \forall j \in V' \tag{24}$$

$$SPD_{j0} = D'_{j0} \qquad \forall j \in V' \tag{25}$$

$$SPD_{jn+1} = P'_{jn+1} \qquad \forall j \in V' \tag{26}$$

$$SPD_{n+1j} = D'_{n+1j} \quad \forall j \in V' \tag{27}$$
$$D'_{ij} + D'_{ji} = Qx'_{ij} \quad \forall \{i,j\} \in \bar{E} \tag{28}$$
$$P'_{ij} + P'_{ji} = Qx'_{ij} \quad \forall \{i,j\} \in \bar{E} \tag{29}$$
$$SPD_{ij} + SPD_{ji} = Qx'_{ij} \quad \forall \{i,j\} \in \bar{E} \tag{30}$$
$$D'_{jn+1} = P'_{0j} = 0 \quad \forall j \in V' \tag{31}$$
$$\sum_{j \in V'} D'_{n+1j} = \sum_{j \in V'} P'_{j0} = vQ \tag{32}$$
$$\sum_{j \in V'} x'_{0j} = \sum_{j \in V'} x'_{jn+1} = v \tag{33}$$
$$0 \leq v \leq m \tag{34}$$
$$D'_{ij} \geq 0, D'_{ji} \geq 0 \quad \forall \{i,j\} \in \bar{E} \tag{35}$$
$$P'_{ij} \geq 0, P'_{ji} \geq 0 \quad \forall \{i,j\} \in \bar{E} \tag{36}$$
$$SPD_{ij} \geq 0, SPD_{ji} \geq 0 \quad \forall \{i,j\} \in \bar{E} \tag{37}$$
$$x'_{ij} \in \{0,1\} \quad \forall (i,j) \in \bar{E} \tag{38}$$

The objective function (15) minimizes the sum of the travel costs. Constraints (16) are the degree equations. Constraints (17) ensure that the delivery demands are satisfied. Constraints (18) state that the sum of the vehicle loads leaving the vertex 0 must be equal to the sum of the demand of all costumers. Constraints (19) enforce that the sum of the vehicle loads arriving at the vertex 0 must be equal to the sum of the residual capacity of all vehicles. Constraints (20)-(22) are related to the pickup flow and their meaning are, respectively, analogous to (17)-(19). Constraints (23) guarantee that the pickup and delivery demands are simultaneously satisfied. Constraints (24)-(27) are self-explanatory. Constraints (28)-(30) state, respectively, that the sum of the delivery, pickup and combined loads arriving and leaving each customer must be equal to the vehicle capacity. Constraints (31)-(32) are self-explanatory. Constraint (33) is related to the number of vehicles. Constraints (34)-(38) define the domain of the decision variables.

The formulation F2C-U was obtained by simply adding constraints (20)-(27), (29)-(34) and (36)-(37) to the formulation presented in [6]. As in F1C, stronger inequalities can be developed by tightening the bounds of the flow variables, i.e, replacing (35)-(36) with (39)-(40) and (37) with (41).

$$D'_{ij} \geq d_j x'_{ij} \quad \forall (i,j) \in \bar{E} \tag{39}$$
$$P'_{ij} \geq p_i x'_{ij} \quad \forall (i,j) \in \bar{E} \tag{40}$$
$$SPD_{ij} \geq \max\{0, d_j - p_j, p_i - d_i\} x'_{ij} \quad \forall (i,j) \in \bar{E} \tag{41}$$

Although the lower bounds of the flow variables are not explicit in (39)-(41) it can be easily verified that they become inherent to the formulation when these upper bound inequalities are combined with (28)-(30), resulting in $D'_{ij} \leq (Q - d_i)x'_{ij}$, $P'_{ij} \leq (Q - d_j)x'_{ij}$ and $SPD_{ij} \leq (Q - \max\{0, d_i - p_i, p_j - d_j\})x'_{ij}$.

3.2 Directed Two-Commodity Flow Formulation

Let $\bar{A}$ be the set of arcs (i,j), $\forall i,j \in \bar{V}$ and $\bar{x}_{ij}$ be 1 if the arc $(i,j) \in \bar{A}$ is in the solution and 0 otherwise. A directed version of the two-commodity flow formulation (F2C-D) is as follows.

$$\min \sum_{i \in \bar{V}} \sum_{j \in \bar{V}} c_{ij} \bar{x}_{ij} \tag{42}$$

subject to

$$\sum_{j \in \bar{V}} \bar{x}_{ij} = 1 \qquad \forall i \in V' \tag{43}$$

$$\sum_{j \in \bar{V}} \bar{x}_{ji} = 1 \qquad \forall i \in V' \tag{44}$$

$$\bar{x}_{j0} = \bar{x}_{n+1j} = 0 \qquad \forall j \in V' \tag{45}$$

$$D'_{ij} + D'_{ji} = Q(\bar{x}_{ij} + \bar{x}_{ji}) \qquad \forall (i,j), i < j, \in A \tag{46}$$

$$P'_{ij} + P'_{ji} = Q(\bar{x}_{ij} + \bar{x}_{ji}) \qquad \forall (i,j), i < j, i \neq 0 \in \bar{A} \tag{47}$$

$$SPD_{ij} + SPD_{ji} = Q(\bar{x}_{ij} + \bar{x}_{ji}) \qquad \forall i,j, i < j, \in V' \tag{48}$$

$$\sum_{j \in V'} \bar{x}_{0j} = \sum_{j \in V'} \bar{x}_{jn+1} = v \tag{49}$$

$$\bar{x}_{ij} \in \{0,1\} \qquad \forall (i,j) \in \bar{A} \tag{50}$$

$$(17)\text{-}(32) \text{ and } (34)\text{-}(37)$$

Constraints (43)-(44) are the degree equations. Constraint (45) is self-explanatory. Constraints (46)-(48) are the capacity equalities. Constraints (49)-(50) have already been defined.

The stronger flow inequalities defined for the F2C-U also hold for the F2C-D as can be observed in (51)-(53). Also, the arc inequalities (14) used in F1C can be directly converted to F2C-D as shown in (54).

$$D'_{ij} \geq d_j(\bar{x}_{ij} + \bar{x}_{ji}) \qquad \forall (i,j) \in \bar{A} \tag{51}$$

$$P'_{ij} \geq p_i(\bar{x}_{ij} + \bar{x}_{ji}) \qquad \forall (i,j) \in \bar{A} \tag{52}$$

$$SPD_{ij} \geq (\max\{0, d_j - p_j, p_i - d_i\})(\bar{x}_{ij} + \bar{x}_{ji}) \qquad \forall (i,j) \in \bar{A} \tag{53}$$

$$\bar{x}_{ij} + \bar{x}_{ji} \leq 1 \qquad \forall i,j, i < j, \in V' \tag{54}$$

Proposition 1. *The linear relaxation of F1C with (11)-(14) is strictly stronger than the one obtained by F2C-D with (51)-(54), which in turn is strictly stronger than the the linear relaxation of F2C-U with (39)-(41).*

4 Computational Experiments with a Branch-and-Cut Algorithm

We evaluated the practical performance of the formulations presented in this work when used in a branch-and-cut (BC) algorithm. Traditional CVRP inequalities were used, namely the rounded capacity, multistar and comb inequalities. The cuts were separated using the CVRPSEP package [7]. The reader is referred to [11] for details concerning the separation routines. At first, we try to separate the cuts using the delivery demands. When no valid inequalities are found we then use the pickup demands. All of the three kinds of cuts are generated at the root node, but just the rounded capacity cuts are used throughout the tree up to the 5th level. Preliminary tests have shown that the overhead of separating

Table 1. Results obtained by the F1C on Dethloff's instances

Instance/ Customers	$\#v$	LP	Root LB	Root Time (s)	Tree size	Total Time (s)	Prev. LB	New LB	F-LB	UB	Gap (%)
SCA3-0/50	4	551.14	613.38	41	73473	7200	583.77	627.66	622.73	635.62	2.03
SCA3-1/50	4	645.95	682.40	107	1712	1230	655.63	697.84	**697.84**	**697.84**	0.00
SCA3-2/50	4	592.56	658.35	19	1	19	627.12	659.34	659.34	659.34	0.00
SCA3-3/50	4	586.30	667.37	70	1083	415	633.56	680.04	**680.04**	**680.04**	0.00
SCA3-4/50	4	627.29	672.92	80	8718	1599	642.89	690.50	**690.50**	**690.50**	0.00
SCA3-5/50	4	604.31	646.14	83	15560	1901	603.06	659.90	**659.90**	**659.90**	0.00
SCA3-6/50	4	587.97	624.92	47	37655	7200	607.53	645.56	639.97	651.09	1.71
SCA3-7/50	4	584.69	654.30	82	26	103	616.40	659.17	**659.17**	**659.17**	0.00
SCA3-8/50	4	638.75	688.77	94	72785	7200	668.04	719.48	703.12	719.48	2.27
SCA3-9/50	4	597.02	668.09	82	1674	417	619.03	681.00	**681.00**	**681.00**	0.00
SCA8-0/50	9	849.35	922.36	96	8854	7200	877.55	936.89	933.12	961.50	2.95
SCA8-1/50	9	937.71	998.04	75	9948	7200	954.29	1020.28	1015.05	1049.65	3.30
SCA8-2/50	9	931.93	1008.83	78	10334	7200	950.74	1024.24	1019.99	1039.64	1.89
SCA8-3/50	9	874.31	954.55	74	11375	7200	905.29	975.87	970.88	983.34	1.27
SCA8-4/50	9	958.58	1022.44	72	12054	7200	972.62	1041.65	1036.45	1065.49	2.73
SCA8-5/50	9	923.50	996.01	79	9207	7200	940.60	1015.19	1011.57	1027.08	1.51
SCA8-6/50	9	870.58	933.57	133	6219	7200	885.34	959.91	944.53	971.82	2.81
SCA8-7/50	9	937.30	1013.86	65	12533	7200	955.86	1031.56	1029.97	1051.28	2.03
SCA8-8/50	9	962.50	1023.86	102	8510	7200	986.52	1048.93	1036.90	1071.18	3.20
SCA8-9/50	9	953.36	1012.73	89	9031	7200	978.90	1034.28	1031.51	1060.50	2.73
CON3-0/50	4	577.74	606.00	91	40753	4836	592.38	616.52	616.46	616.52	0.01
CON3-1/50	4	506.41	543.71	73	52033	6498	532.55	554.47	**554.47**	**554.47**	0.00
CON3-2/50	4	468.40	503.14	61	13874	7200	491.04	517.26	514.11	518.00	0.75
CON3-3/50	4	541.46	581.45	55	20044	1941	557.99	591.19	**591.19**	**591.19**	0.00
CON3-4/50	4	537.90	577.61	63	78398	7200	558.26	588.79	588.47	588.79	0.06
CON3-5/50	4	511.88	553.87	107	32652	5975	531.33	563.70	**563.70**	**563.70**	0.00
CON3-6/50	4	468.90	486.59	128	14248	7200	475.33	499.05	493.01	499.05	1.21
CON3-7/50	4	533.86	562.10	38	53629	5522	550.73	576.48	**576.48**	**576.48**	0.00
CON3-8/50	4	477.81	513.90	87	15317	1923	492.69	523.05	**523.05**	**523.05**	0.00
CON3-9/50	4	528.34	564.87	63	15461	5602	547.31	578.25	**578.25**	**578.25**	0.00
CON8-0/50	9	774.69	829.80	47	16498	7200	795.45	845.19	842.62	857.17	1.70
CON8-1/50	9	680.24	719.03	80	7552	7200	693.22	734.71	732.44	740.85	1.14
CON8-2/50	9	636.18	682.76	128	9856	7200	650.81	695.70	693.07	712.89	2.78
CON8-3/50	10	732.55	784.93	71	7536	7200	754.41	797.57	796.31	811.07	1.82
CON8-4/50	9	710.36	749.83	122	6374	7200	729.09	767.63	759.11	772.25	1.70
CON8-5/50	9	696.85	728.10	78	7901	7200	709.76	741.51	736.79	754.88	2.40
CON8-6/50	9	611.16	647.04	61	10400	7200	631.41	662.14	<u>662.14</u>	678.92	2.47
CON8-7/50	9	729.28	787.89	64	11861	7200	762.03	810.08	800.22	811.96	1.44
CON8-8/50	9	689.23	741.02	74	10324	7200	705.08	757.45	753.42	767.53	1.84
CON8-9/50	9	716.21	770.66	101	5435	7200	729.10	786.40	778.65	809.00	3.75
									Avg. Gap (%)		1.34

Table 2. Results obtained by the F2C-U on Dethloff's instances

Instance/ Customers	$\#v$	LP	Root LB	Root Time (s)	Tree size	Total Time (s)	Prev. LB	New LB	F-LB	UB	Gap (%)
SCA3-0/50	4	550.85	613.36	28	211456	7200	583.77	627.66	625.00	635.62	1.67
SCA3-1/50	4	645.59	682.33	42	1467	142	655.63	697.84	**697.84**	**697.84**	0.00
SCA3-2/50	4	592.44	658.89	12	1	12	627.12	659.34	**659.34**	**659.34**	0.00
SCA3-3/50	4	586.02	667.37	25	847	81	633.56	680.04	**680.04**	**680.04**	0.00
SCA3-4/50	4	626.93	673.28	50	2866	252	642.89	690.50	**690.50**	**690.50**	0.00
SCA3-5/50	4	603.95	646.29	38	19724	731	603.06	659.90	**659.90**	**659.90**	0.00
SCA3-6/50	4	587.85	624.89	27	176561	7200	607.53	645.56	644.37	651.09	1.03
SCA3-7/50	4	584.59	653.76	34	30	43	616.40	659.17	**659.17**	**659.17**	0.00
SCA3-8/50	4	638.41	693.71	42	108017	4899	668.04	719.48	**719.48**	**719.48**	0.00
SCA3-9/50	4	596.79	668.09	36	1452	96	619.03	681.00	**681.00**	**681.00**	0.00
SCA8-0/50	9	847.39	922.58	79	17695	7200	877.55	936.89	936.89	961.50	2.56
SCA8-1/50	9	933.44	997.07	56	18086	7200	954.29	1020.28	1020.28	1049.65	2.80
SCA8-2/50	9	931.34	1008.27	75	12789	7200	950.74	1024.24	1024.24	1039.64	1.48
SCA8-3/50	9	872.37	953.67	49	18487	7200	905.29	975.87	975.87	983.34	0.76
SCA8-4/50	9	955.74	1021.35	44	25853	7200	972.62	1041.65	1041.65	1065.49	2.24
SCA8-5/50	9	922.25	995.93	58	19464	7200	940.60	1015.19	1013.87	1027.08	1.29
SCA8-6/50	9	868.00	933.76	74	10467	7200	885.34	959.91	959.91	971.82	1.23
SCA8-7/50	9	935.55	1015.11	69	15193	7200	955.86	1031.56	1031.56	1051.28	1.88
SCA8-8/50	9	960.17	1023.60	87	8262	7200	986.52	1048.93	1048.93	1071.18	2.08
SCA8-9/50	9	952.34	1014.89	64	16262	7200	978.90	1034.28	1034.28	1060.50	2.47
CON3-0/50	4	577.52	606.82	46	3048	247	592.38	616.52	**616.52**	**616.52**	0.00
CON3-1/50	4	506.23	545.53	54	16039	823	532.55	554.47	**554.47**	**554.47**	0.00
CON3-2/50	4	468.22	504.44	59	22107	7200	491.04	517.26	516.23	518.00	0.34
CON3-3/50	4	541.40	582.83	35	6608	330	557.99	591.19	**591.19**	**591.19**	0.00
CON3-4/50	4	537.73	577.57	42	50663	3198	558.26	588.79	**588.79**	**588.79**	0.00
CON3-5/50	4	511.59	554.35	65	10191	729	531.33	563.70	**563.70**	**563.70**	0.00
CON3-6/50	4	468.75	486.61	100	48466	5230	475.33	499.05	**499.05**	**499.05**	0.00
CON3-7/50	4	533.73	561.87	37	9822	1141	550.73	576.48	**576.48**	**576.48**	0.00
CON3-8/50	4	477.45	514.13	71	5541	450	492.69	523.05	**523.05**	**523.05**	0.00
CON3-9/50	4	527.94	564.78	53	6372	790	547.31	578.25	**578.25**	**578.25**	0.00
CON8-0/50	9	773.46	827.14	74	13038	7200	795.45	845.19	845.19	857.17	1.40
CON8-1/50	9	678.95	719.09	67	13302	7200	693.22	734.71	734.71	740.85	0.83
CON8-2/50	9	635.23	682.37	127	9409	7200	650.81	695.70	695.70	712.89	2.41
CON8-3/50	10	731.55	785.00	71	18680	7200	754.41	797.57	797.57	811.07	1.66
CON8-4/50	9	708.64	751.32	60	15700	7200	729.09	767.63	767.63	772.25	0.60
CON8-5/50	9	696.08	727.26	66	9765	7200	709.76	741.51	741.51	754.88	1.77
CON8-6/50	9	610.20	646.78	94	11947	7200	631.41	662.14	661.36	678.92	2.59
CON8-7/50	9	726.55	788.64	74	5520	7200	762.03	810.08	810.08	811.96	0.23
CON8-8/50	9	688.25	741.76	81	13325	7200	705.08	757.45	757.45	767.53	1.31
CON8-9/50	9	713.85	770.85	109	12833	7200	729.10	786.40	786.40	809.00	2.79
									Avg. Gap (%)		0.94

comb and multistar inequalities outside the root node was not worthwhile. For each separation routine of the CVRPSEP package we have established a limit of 50 violated cuts per iteration.

The BC procedures were implemented using the CPLEX 11.2 callable library and executed in an Intel Core 2 Quad with 2.4 GHz and 4 GB of RAM running under Linux 64 bits (kernel 2.6.27-16). Only a single thread was used in our experiments. Each BC is respectively associated with the formulations F1C, F2C-U and F2C-D and they were tested on the set of instances proposed by Dethloff [4], Salhi and Nagy [12] and Montané and Galvão [5]. The first group contains 40 instances with 50 customers, the second contains 14 instances with 50-199 customers, while the third contains 12 instances with 100-200 customers. The number of vehicles is not explicitly specified in these 66 instances. The barrier algorithm was used to solve the initial linear relaxation of the last two

Table 3. Results obtained by the F2C-D on Dethloff's instances

Instance/ Customers	$\#v$	LP	Root LB	Root Time (s)	Tree size	Total Time (s)	Prev. LB	New LB	F-LB	UB	Gap (%)
SCA3-0/50	4	550.93	613.35	25	132649	7200	583.77	627.66	627.66	635.62	1.25
SCA3-1/50	4	645.60	682.23	29	1262	114	655.63	697.84	**697.84**	**697.84**	0.00
SCA3-2/50	4	592.47	659.11	11	1	11	627.12	659.34	**659.34**	**659.34**	0.00
SCA3-3/50	4	586.02	667.35	21	374	50	633.56	680.04	**680.04**	**680.04**	0.00
SCA3-4/50	4	626.93	673.22	31	6156	334	642.89	690.50	**690.50**	**690.50**	0.00
SCA3-5/50	4	603.96	646.41	31	12594	330	603.06	659.90	**659.90**	**659.90**	0.00
SCA3-6/50	4	587.85	624.92	26	167625	7200	607.53	645.56	645.56	651.09	0.85
SCA3-7/50	4	584.59	654.30	33	23	40	616.40	659.17	**659.17**	**659.17**	0.00
SCA3-8/50	4	638.41	694.13	42	196322	7200	668.04	719.48	714.19	719.48	0.73
SCA3-9/50	4	596.79	668.09	38	1567	116	619.03	681.00	**681.00**	**681.00**	0.00
SCA8-0/50	9	847.73	922.85	107	3758	7200	877.55	936.89	933.89	961.50	2.87
SCA8-1/50	9	933.47	997.57	103	3661	7200	954.29	1020.28	1013.38	1049.65	3.46
SCA8-2/50	9	931.42	1008.87	147	2435	7200	950.74	1024.24	1019.31	1039.64	1.96
SCA8-3/50	9	872.45	953.21	98	3914	7200	905.29	975.87	968.55	983.34	1.50
SCA8-4/50	9	955.96	1022.13	134	4773	7200	972.62	1041.65	1032.49	1065.49	3.10
SCA8-5/50	9	922.32	996.33	184	14246	7200	940.60	1015.19	1015.19	1027.08	1.16
SCA8-6/50	9	868.05	933.74	143	1593	7200	885.34	959.91	943.47	971.82	2.92
SCA8-7/50	9	935.82	1013.12	102	3334	7200	955.86	1031.56	1028.04	1051.28	2.21
SCA8-8/50	9	960.27	1023.53	159	1559	7200	986.52	1048.93	1036.29	1071.18	3.26
SCA8-9/50	9	952.41	1013.82	111	7476	7200	978.90	1034.28	1031.54	1060.50	2.73
CON3-0/50	4	577.52	605.97	34	21249	1141	592.38	616.52	**616.52**	**616.52**	0.00
CON3-1/50	4	506.24	543.70	41	19638	1199	532.55	554.47	**554.47**	**554.47**	0.00
CON3-2/50	4	468.22	504.31	71	97732	7200	491.04	517.26	517.26	518.00	0.14
CON3-3/50	4	541.40	582.89	37	4116	213	557.99	591.19	**591.19**	**591.19**	0.00
CON3-4/50	4	537.73	577.59	28	78652	2932	558.26	588.79	**588.79**	**588.79**	0.00
CON3-5/50	4	511.60	554.43	50	16215	772	531.33	563.70	**563.70**	**563.70**	0.00
CON3-6/50	4	468.75	486.76	96	65792	6979	475.33	499.05	**499.05**	**499.05**	0.00
CON3-7/50	4	533.75	561.89	31	15895	1134	550.73	576.48	**576.48**	**576.48**	0.00
CON3-8/50	4	477.45	513.99	51	3833	269	492.69	523.05	**523.05**	**523.05**	0.00
CON3-9/50	4	527.95	564.77	48	4637	585	547.31	578.25	**578.25**	**578.25**	0.00
CON8-0/50	9	773.51	826.63	122	2526	7200	795.45	845.19	840.60	857.17	1.93
CON8-1/50	9	679.00	719.00	132	2885	7200	693.22	734.71	729.26	740.85	1.56
CON8-2/50	9	635.25	682.12	200	2416	7200	650.81	695.70	692.34	712.89	2.88
CON8-3/50	10	731.55	785.01	151	3406	7200	754.41	797.57	794.94	811.07	1.99
CON8-4/50	9	708.64	751.40	121	5468	7200	729.09	767.63	766.37	772.25	0.76
CON8-5/50	9	696.08	726.88	133	3688	7200	709.76	741.51	734.84	754.88	2.66
CON8-6/50	9	610.20	646.22	125	2458	7200	631.41	662.14	658.43	678.92	3.02
CON8-7/50	9	726.57	787.53	142	1995	7200	762.03	810.08	801.59	811.96	1.28
CON8-8/50	9	688.33	741.06	166	4021	7200	705.08	757.45	749.66	767.53	2.33
CON8-9/50	9	713.94	770.74	234	2307	7200	729.10	786.40	778.72	809.00	3.74
									Avg. Gap (%)		1.26

group of instances. A time limit of 2 hours of execution was imposed for the BC algorithms. In some very particular cases, the CPLEX run have slightly exceeded this time limit, namely on few instances involving more than 100 customers. The values of the best known solutions found in the literature were given as initial primal bound for the BC, namely those reported in [13].

In the tables presented hereafter, $\#v$ represents the number of vehicles in the best known solution, **LP** is the linear relaxation, **Root LB** indicates the root lower bound, after CVRPSEP cuts are added, **Root Time** is the CPU time in seconds spent at the root node, **Tree size** corresponds to the the number of nodes opened, **Total time** is the total CPU time in seconds of the BC procedure, **Prev. LB** is the lower bound obtained in [5], **New LB** is the best lower bound determined among the three flow formulations, **F-LB** is the lower bound found by the respective formulation, **UB** is the upper bound reported in [13], and **Gap**

Table 4. Results obtained by the F2C-U on Salhi and Nagy's instances

Instance/ Customers	#v	LP	Root LB	Root Time (s)	Tree size	Total Time (s)	New LB	F-LB	UB	Gap (%)
CMT1X/50	3	449.00	459.98	102	2282	300	466.77	**466.77**	**466.77**	0.00
CMT1Y/50	3	449.00	460.02	70	3205	213	466.77	**466.77**	**466.77**	0.00
CMT2X/75	6	632.11	652.85	346	2073	7200	655.88	655.21	684.21	4.24
CMT2Y/75	6	632.11	653.13	449	2610	7200	655.41	655.41	684.21	4.21
CMT3X/100	5	682.18	701.10	504	13820	7200	705.54	704.35	721.27	2.35
CMT3Y/100	5	682.18	701.12	612	19865	7200	705.62	705.28	721.27	2.22
CMT12X/100	5	564.08	628.59	813	991	7201	629.39	629.39	662.22	4.96
CMT12Y/100	5	564.08	628.58	923	118	7201	629.18	629.09	662.22	5.00
CMT11X/120	4	687.42	775.51	4835	42	7201	776.35	776.35	833.92	6.90
CMT11Y/120	4	687.42	775.40	6138	22	7200	775.74	775.74	833.92	6.98
CMT4X/150	7	796.48	817.11	7288	1	7292	817.11	817.11	852.46	4.15
CMT4Y/150	7	796.48	816.99	5747	1	7201	816.99	816.99	852.46	4.16
CMT5X/200	10	933.21	954.87	6939	1	7201	954.87	954.87	1029.25	7.23
CMT5Y/200	10	933.21	953.56	6600	1	7202	953.56	953.56	1029.25	7.35
								Avg. Gap (%)		4.27

corresponds to the gap between the LB and the UB. Proven optimal solutions are highlighted in boldface. If the F-LB is the one associated with the New LB (F-LB = New LB), then its value is underlined only if New LB is not an optimal solution.

Tables 1, 2 and 3 contain, respectively, the results obtained by F1C, F2C-U and F2C-D on the set of instances of Dethloff. It can be seen that the three formulations were able to prove the optimality of almost all instances of 4 vehicles. F2C-U appears to be the most effective under this aspect, being capable of proving the optimality of 17 instances. The performance of the three formulations on the instances of 9 vehicles were inferior in terms of optimality proof, but their LBs are significantly better than the previous values reported in [5]. F2C-U also seems to be the most effective in terms of LBs, with an average gap of 0.94%, against 1.34% and 1.26% of F1C and F2C-D, respectively.

In order to check if the values of the UB of the instances SCA3-0, SCA3-6, SCA8-3, SCA8-6, CON3-2, CON8-1, CON8-4 and CON8-7 are optimal we ran F2C-U with a time limit of 48 hours. The formulation was successful to prove the optimality of each of these instances within up to 36 hours of execution.

As for the Salhi and Nagy and Montané and Galvão instances, we will present only the results obtained by F2C-U, not only because it produced the best results on average, but also due to lack of space. From Table 4 it can be observed that optimality of the instances CMT1X and CMT1Y has been proven. In addition, to our knowledge these are the first LBs presented for this set of instances. Montané and Galvão [5] had reported LBs for the case where the demands were rounded to the nearest integer. From Table 5 it can be noticed that optimality of the instances r201, c201 and rc201 was proven. The main characteristic of these three instances is the fact of having relatively very few vehicles.

Table 6 shows a summary of the results obtained by the three formulations in all set of instances. In this table, **G1** is the average gap between the linear relaxation and the UB, **G2** is the average gap with respect to the root LB, including the CVRPSEP cuts, and **G3** is average gap for the LB, possibly after branching, found within the time limit established. Those results can be explained as

Table 5. Results obtained by the F2C-U on Montané and Galvão's instances

Instance/ Customers	#v	LP	Root LB	Root Time (s)	Tree size	Total Time (s)	Prev. LB	New LB	F-LB	UB	Gap (%)
r101/100	12	939.19	972.88	2910	122	7201	934.97	973.91	973.10	1009.95	3.65
r201/100	3	643.07	664.80	292	21	307	643.65	666.20	**666.20**	**666.20**	0.00
c101/100	16	1070.40	1195.53	1396	302	7201	1066.19	1196.70	1195.89	1220.18	1.99
c201/100	5	598.47	657.97	197	17	241	596.85	662.07	**662.07**	**662.07**	0.00
rc101/100	10	944.21	1028.15	2940	138	7201	937.41	1029.38	1029.38	1059.32	2.83
rc201/100	3	600.24	671.84	134	4	134	602.70	672.92	**672.92**	**672.92**	0.00
r1_2_1/200	23	3013.16	3084.97	6971	1	7200	2951.12	3084.97	3084.97	3360.02	8.19
r2_2_1/200	5	1549.60	1618.76	7869	1	7874	1501.82	1618.76	1618.76	1665.58	2.81
c1_2_1/200	28	3325.20	3475.03	7041	1	7202	3299.07	3475.03	3475.03	3629.89	4.27
c2_2_1/200	9	1560.22	1647.83	7370	1	7374	1542.96	1647.83	1647.83	1726.59	4.56
rc1_2_1/200	23	3015.44	3093.30	7064	1	7201	2939.98	3093.30	3093.30	3306.00	6.43
rc2_2_1/200	5	1438.91	1551.07	7301	1	7308	1396.95	1551.07	1551.07	1560.00	0.57
									Avg. Gap (%)		2.94

Table 6. Summary of the results obtained by the three formulations

Formulation	Dethloff			Salhi and Nagy			Montané and Galvão		
	G1 (%)	G2 (%)	G3 (%)	G1 (%)	G2 (%)	G3 (%)	G1 (%)	G2 (%)	G3 (%)
F1C	9.74	2.96	1.34	9.21	4.85	4.57	8.75	3.66	3.57
F2C-U	9.85	2.92	0.94	9.30	4.62	4.27	8.82	3.04	2.94
F2C-D	9.85	2.94	1.26	9.30	4.66	4.31	8.82	3.57	3.62

follows. The linear relaxation of is F1C is indeed a little better than the linear relaxations of F2C-D and F2C-U. However, after the cuts, there is no significant difference in the LB quality. This can be clearly seen in the column **G2** under Dethloff instances. For those smaller instances, the cut separation in the root node could always be completed within the time limit. In those cases, the small gap differences (2.96%, 2.92% and 2.94%) are not significant and can be attributed to the heuristic nature of the routines in the CVRPSEP library. The consistent advantage of formulation F2C-U shown in columns **G3** is explained by the fact that CPLEX has a significantly better performance when reoptimizing its LPs. This means that more cuts can be separated and more nodes can be explored within the same time limit.

5 Concluding Remarks

This work dealt with Mixed Integer Programming formulations for the the Vehicle Routing Problem with Simultaneous Pickup and Delivery. An undirected and a directed two-commodity flow formulations were proposed. They were tested within a branch-and-cut scheme and their results were compared with the one-commodity flow formulation of Dell'Amico et al. [2]. The optimal solutions of 30 open problems were proved, and new lower bounds were obtained for instances with up to 200 customers. In addition, although we have shown that the one-commodity flow formulation produces a stronger linear relaxation, the two-commodity flow formulations have found, on average, better lower bounds after 2 hours of execution time.

References

1. Min, H.: The multiple vehicle routing problem with simultaneous delivery and pick-up points. Transportation Research 23(5), 377–386 (1989)
2. Dell'Amico, M., Righini, G., Salani, M.: A branch-and-price approach to the vehicle routing problem with simultaneous distribution and collection. Transportation Science 40(2), 235–247 (2006)
3. Angelelli, E., Mansini, R.: A branch-and-price algorithm for a simultaneous pick-up and delivery problem. In: Quantitative Approaches to Distribution Logistics and Supply Chain Management, pp. 249–267. Springer, Heidelberg (2002)
4. Dethloff, J.: Vehicle routing and reverse logistics: the vehicle routing problem with simultaneous delivery and pick-up. OR Spektrum 23, 79–96 (2001)
5. Montané, F.A.T., Galvão, R.D.: A tabu search algorithm for the vehicle routing problem with simultaneous pick-up and delivery service. Computers & Operations Research 33(3), 595–619 (2006)
6. Baldacci, R., Hadjiconstantinou, E., Mingozzi, A.: An exact algorithm for the capacitated vehicle routing problem based on a two-commodity network flow formulation. Operations Research 52(5), 723–738 (2004)
7. Lysgaard, J.: A package of separation routines for the capacited vehicle routing problem. Technical report (2003), http://www.asdb.dk/~lys
8. Toth, P., Vigo, D.: Models, relaxations and exact approaches for the capacitated vehicle routing problem. Discrete Applied Mathematics 123, 487–512 (2002)
9. Gavish, B., Graves, S.: The traveling salesman problem and related problems. Working Paper (1979)
10. Gouveia, L.: A result on projection for the vehicle routing problem. European Journal of Operational Research 85, 610–624 (1995)
11. Lysgaard, J., Letchford, A.N., Eglese, R.W.: A new branch-and-cut algorithm for the capacitated vehicle routing problem. Mathematical Programming 100, 423–445 (2004)
12. Salhi, S., Nagy, G.: A cluster insertion heuristic for single and multiple depot vehicle routing problems with backhauling. Journal of the Operational Research Society 50, 1034–1042 (1999)
13. Subramanian, A., Drummond, L.M.A., Bentes, C., Ochi, L.S., Farias, R.: A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery. Computers & Operations Research (to appear, 2010)

A Metaheuristic for a Two Echelon Location-Routing Problem

Maurizio Boccia[1], Teodor G. Crainic[2], Antonio Sforza[3], and Claudio Sterle[3]

[1] Dept. of Engineering, Universitá del Sannio
boccia@unisannio.it
[2] Dept. Management et Technologie, Université du Québec á Montréal and Centre for Research on Transportation, Montreal
TeodorGabriel.Crainic@cirrelt.ca
[3] Dept. of Computer Science and Systems, Universitá di Napoli "Federico II"
{sforza,sterle}@unina.it

Abstract. In this paper we consider the design problem of a two-echelon freight distribution system. The aim is to define the structure of a system optimizing the location and the number of two different kinds of facilities, the size of two different vehicle fleets and the related routes on each echelon. The problem has been modeled as a two-echelon location-routing problem (2E-LRP). A tabu-search heuristic efficiently combining the composing subproblems is presented. Results on small, medium and large size instances are reported.

Keywords: tabu search, location-routing.

1 Introduction

Freight transportation to and from a city is generally performed through platforms, called *City Distribution Centers* (*CDC*), located on the outskirt of urban areas. In the last years two-echelon freight distribution systems have been proposed, based on the utilization of intermediate facilities between platforms and customers (Crainic et al. [5]). In these facilities, referred as *satellites*, freights coming from the *CDC*s are transferred and consolidated into vehicles of smaller sizes. Satellites perform basically trans-dock operations and therefore already existing infrastructures can be exploited (i.e. underground parking slots, bus depots, etc.). This system contemplates the usage of two kinds of vehicles of decreasing dimensions on the two echelons, referred respectively as *urban-trucks* and *city-freighters*.

The design of a two-echelon freight distribution system is a strategical and tactical decisional problem. Indeed strategical decisions concern the choice of location and number of facilities and the assignment among consecutive levels, whereas tactical decisions concern the size of two vehicle fleets and related routing. The problem has been modeled as a two-echelon location-routing problem (2E-LRP). To the best of our knowledge, multi-level location-routing problems have been addressed, from the modeling point of view, only in the paper of Ambrosino and Scutellá (2005).

The paper is structured as follows: a synthetic overview of location-routing problems (Section 2); basic assumptions of the 2E-LRP (Section 3); tabu search heuristic description (Section 4); results of the heuristic on several small, medium and large instances (Section 5).

2 Location-Routing Problems: Literature Review

Papers on location-routing problems (LRP) appeared just from the '80s. LRP surveys have been proposed by Laporte [9] and Min et al. [15] and recently by Nagy and Salhi [17]. The hardness of LRPs has been shown in Karp [8] and it directed the research mainly towards heuristic approaches.

Laporte [9] introduced an expression to represent LRPs: $\lambda/M_1/.../M_{\lambda-1}$, where λ is the number of layers and $M_1/.../M_{\lambda-1}$ are the kind of routes among consecutive layers (R for direct routes and T for tours). The most treated case is the $2/1/T$ problem. It has been approached with decomposition based heuristics in Or and Pierskalla [18], Perl and Daskin [19], Hansen et al. [6], Srivastava [20], Chien [4], Nagy and Salhi [16], Tuzun and Burke [22], Wu et al. [23], Albareda-Sambola et al. [2], Melechovsky et al. [14], Barreto et al. [3]. Only few papers propose exact approaches for this problem, i.e. Laporte et al. [10], [11], [12]. The $3/1/T/T$ problem has been addressed with decomposition approaches in Jacobsen and Madsen [7] and Madsen [13]. Ambrosino and Scutellá [1] study a $4/2/R/T/T$ problem. They propose different formulations and solve several instances with CPLEX.

3 The Echelon Location-Routing Problem

The following basic assumptions are considered in the definition of our 2E-LRP:

- All freight starts from platforms. One representative freight is considered.
- Platforms and satellites are characterized by limited capacity.
- Customers are freight destinations. A demand is associated to each customer.
- Customers and satellites are single sourced by respectively satellites and platforms. A single demand cannot be split among different vehicles, but more demands can be loaded on the same vehicle.
- Direct shipping of freight from platforms to customers is not allowed, i.e. freight must be distributed first to satellites and then to customers.
- Vehicles belonging to the same echelon have the same capacity. The capacity of first echelon vehicles is much higher than capacity of second echelon vehicles and of satellites. The capacity of second echelon vehicles is much higher than the demand of customers.
- 1^{st} echelon routes start from a platform, serve one or more satellites and ends to the same platform; 2^{nd} echelon routes start from a satellite, serve one or more customers and ends to the same satellite.

The problem consists in the following decisions: *location decisions* (define number and locations of platforms and satellites); *allocation decisions* (assignment of customers to open satellites and open satellites to open platforms, satisfying capacity constraints); *routing decisions* (number of vehicles and related routes). A three-index mixed-integer model derived by Ambrosino and Scutellá [1] has been formulated in Sterle [21].

4 Tabu Search Heuristic

The proposed TS heuristic is based on the integration of the nested approach of Nagy and Salhy [16] and the two-phase iterative approach of Tuzun and Burke [22]. Hence it can be defined as an "*iterative-nested approach*".

The problem is decomposed in its two main components, i.e. two location-routing problems. Each component, in turn, is decomposed in a capacitated facility location problem (CFLP) and a multi-depot vehicle routing problem (MDVRP). A bottom-up approach is used, i.e. first echelon solution is built and optimized on second echelon solution.

TS operates on each echelon in two coordinate and integrated phases (location and routing). In the following the main issues of the proposed tabu search are described: initialization and evaluation criteria; location and routing moves, tabu attributes and stopping criteria; combination of subproblems solutions.

4.1 First Feasible Solution

The TS starts with a fast heuristic for the construction of a first feasible solution of the two-echelon capacitated facility location problem (2E-FLP).

The aim is to open the minimum number of facilities on both echelons. At first satellites are sorted in function of their capacity. Then we open the minimum number of satellites, S^*, whose capacities are able to satisfy the total demand of the customers. In particular, in order to have higher probability of determining a feasible assignment, we impose that the total capacity of the open satellites, decreased of a given percentage α ($\alpha \in [90\% \div 95\%]$), has to exceed the total customer demand. Then customers are assigned to the nearest satellite with enough residual capacity, in decreasing order of their demand. Once determined the demand assigned to each satellite, the procedure is repeated to find the minimum number (P^*) of platforms to open.

The application of this simple heuristic returns a solution to the 2E-LRP, where dedicated routes are defined on the two echelons.

4.2 Evaluation Criteria

Two ways to evaluate a solution during the tabu search are considered: *"estimated cost"* and *"actual cost"*. Both are given by the sum of two components, location and routing (including also vehicle costs) on the two echelons. The difference is in the way routing costs are computed. In estimated costs, the routing component is approximated with the double of direct distances among the nodes, whereas, in the actual cost, the routing component is given by the sum of transportation costs of each route. The estimated cost is used to evaluate the first feasible solution and the goodness of a location move, whereas the actual cost is computed each time a routing move is performed.

4.3 Solution Neighborhood Definition

In the following, the TS moves will be presented. For sake of clarity location and routing moves and the related parameters will be presented separately and then we will focus on the combination of sub-problem solutions. Moreover for sake of brevity the moves will be presented referring just to the second echelon, since the extension to first echelon is straightforward. In the following we will use the notation P and S for the set of all possible platform and satellite locations and respectively with S^* and P^* for the set of open satellites and platforms.

Location phase. The location phase of the heuristic affects the solution in terms of number and location of facilities. Two simple moves are performed: *swap* and *add* moves. The two moves are applied sequentially and iteratively on each configuration. More precisely, for a given number of open facilities, we try to change the configuration of the solution with swap moves, then, when no improvements are obtained, we increase the number of facilities with an add move and repeat swap moves for the increased number of open facilities.

- Swap moves. With this move the status of two facilities is exchanged, i.e. an open facility is closed and a closed facility is opened. Hence the number of the open facilities is kept constant. The key element of these moves is the selection of the facilities to be swapped. With reference to second echelon, the satellite to be removed from solution set S^* is chosen with one of the following criteria:

1. *Rand-sel-out*: random selection of a node belonging to the solution set S^*;
2. *Max-loc*: node of S^* associated with the highest location cost.

The set of possible satellites to be opened is defined considering just the nodes able to satisfy the total demand of the customers. Once determined the set of candidate satellites, the entering node is the one associated with the minimum estimated total cost.

A move is performed only if it is not tabu. Then the two facilities are declared *tabu* for a number of iteration depending on the number of open facilities. More precisely tabu tenure is *tabu-swap-loc-s* $= \alpha\,|S^*|$, $\alpha \in [\alpha_{min} \div \alpha_{max}]$. Swap moves are performed until a max number of not-improving iterations, *max-swap-loc-s* is met.

For platforms *tabu-swap-loc-p* $= \alpha\,|P^*|$, $\alpha \in [\alpha_{min} \div \alpha_{max}]$ and *max-swap-loc-p* is the fixed number of allowed iterations without improvement.

- Add move. Once the maximum number of swap moves without improvement is met, we perform an add move, i.e. we increase the number of open facilities. The node to be added to solution set is the one associated with the minimum estimated total cost. A move is performed just if it is not tabu. The added node is declared tabu for a number of iterations depending on the overall number of available locations of satellites and platforms ($|P|$ and $|S|$). In particular tabu tenures are *tabu-add-loc-s* $= \alpha\,|S|$ for satellites and *tabu-add-loc-p* $= \alpha\,|P|$ for platforms, $\alpha \in [\alpha_{min} \div \alpha_{max}]$. Add moves are performed until max number of not-improving iterations, $max - add - loc - s$ and $max - add - loc - p$, respectively for satellites and platforms, are met.

Routing phase. At this step, starting from the first feasible solution, we perform sequentially several operations to improve the routing cost component, acting locally on each route and then on multiple routes. These operations can be classified in three phases:

1. Define and improve multi-stop routes: sequential application of Clarke and Wright (C&W) algorithm and 2-opt/3-opt algorithms.
2. Optimize *multiple* routes assigned to a *single* facility: insert and swap moves.
3. Optimize *multiple* routes assigned to *multiple* facilities: insert and swap moves.

The three phases have different effects on the global solution. Indeed first and second phases do not affect the assignment of customer to satellites, i.e. they do not affect the demand assigned to a satellite and consequently first echelon routing cost does not change. On the contrary, third phase can provide significant changes of demand assigned to satellites and therefore the assignment of satellites to platforms can change, affecting also first echelon routing cost. The used approach to face this issue will be explained in the following.

We will first focus on the explanation of second and third phases. The main issue for the used moves is the neighborhood definition. Two selection criteria are used to restrict the sizes of neighborhoods:

1. *One-select*: select one node and evaluate the related neighborhood;
2. *Perc-sel*: select a percentage of all the nodes composing an echelon and evaluate the related neighborhoods.

For second phase, the nodes to be selected are the ones assigned to a single facility. Instead, for third phase, the nodes to be selected are the ones composing an echelon. For both phases a simple aspiration criterion is used, i.e. a move is performed, even if tabu, but it provides an improvement of the best solution.

Intra-routes improvements for a single facility. These moves are feasible only if vehicle capacity constraints are satisfied.

- Insert move. a customer is deleted from one route and is assigned to another route belonging to the same satellite. The neighborhood is defined evaluating the insertions of selected customer (customers) in all paths assigned to the satellite under investigation. If a neighbor solution provides an improvement, then the move is performed and added node is declared tabu, otherwise we choose the best not-tabu deteriorating move and added node is declared tabu.

Tabu tenure *tabu-r-ins-single-s* is variable and depends on number of customers assigned to a satellite. Being Z_s the number of customers assigned to a satellite, then *tabu-r-ins-single-s* $= \lceil \alpha\ Z_s \rceil$, $\alpha \in [\alpha_{min} \div \alpha_{max}]$. These moves are performed until max number of not-improving moves, *max-r-ins-single*, is met.

The extension to platforms is straightforward. Being S_p the number of satellites assigned to a platform , *tabu-r-ins-single-p* $= \lceil \alpha\ S_p \rceil$, $\alpha \in [\alpha_{min} \div \alpha_{max}]$.

- Swap moves. The positions of two customers belonging to two routes assigned to the same satellite are exchanged. The neighborhood is defined evaluating the exchanges of the selected customer (customers) with all the customers assigned to the satellite under investigation. If the move is not-tabu and it provides a saving on the routing cost, it is performed. Otherwise we perform the best not-tabu deteriorating move. Then customers are both declared tabu. Tabu tenure values are *tabu-r-swap-single-s* $= \lceil \alpha\ Z_s \rceil$ and *tabu-r-swap-single-p* $= \lceil \alpha\ S_p \rceil$ respectively for first and second echelon. These moves are performed until the max number, *max-r-swap-single*, of not-improving moves is met.

Intra-routes improvements for multiple facilities. The moves presented for a single facility are extended to multiple facilities. In this case, a move to be feasible has to satisfy capacity constraints for both facilities and vehicles. Once performed an insert or swap move, a local search is run to re-optimise locally the routes of the involved facilities, performing 2-opt and 3-opt moves and insert and swap moves for a single facility.

- Insert move. A customer is deleted from its route and inserted in another route belonging to another open satellite. The neighborhood is defined considering the insertions of the selected customer (customers) in all the routes belonging to the "closest" open satellites. The *closest satellites*, *near-ins-s*, are a percentage of the number of open satellites and is computed as follows: *near-ins-s* $= \lceil \beta\ S^* \rceil$, $\beta \in [0 \div 1]$. If a move is not-tabu and it provides an improvement, it is performed, otherwise, the best deteriorating not-tabu one is performed.

Tabu tenure value for this move depends on the total number of customers Z. Hence *tabu-r-ins-multi-s* $= \lceil \alpha\ Z \rceil$, $\alpha \in [\alpha_{min} \div \alpha_{max}]$.

The same relations can be extended to the first echelon for which we have *near-ins-p* $= \lceil \beta\ P^* \rceil$, $\beta \in [0, 1]$ and, being S the total number of satellite locations, *tabu-r-ins-multi-p* $= \lceil \alpha\ S \rceil$, $\alpha \in [\alpha_{min} \div \alpha_{max}]$. This move is performed until the max number, $max - r - ins - multi$ of not-improving moves is reached.

- Swap move. The position of two customers belonging to routes assigned to different satellites are exchanged. The neighborhood is defined considering the exchanges of the selected customer (customers) with all the "closest" customers. The *closest customers*, *near-swap-s*, are a percentage of all customers and is computed as follows: *near-swap-s* $= \lceil \beta\ Z \rceil$, $\beta \in [0 \div 1]$. If a move is not tabu and it provides an improvement, then it is performed, otherwise the best deteriorating not-tabu move is performed. Tabu tenure value for this move depends on the total number of customers, i.e. *tabu-r-swap-multi-s* $= \lceil \alpha\ Z \rceil$, $\alpha \in [\alpha_{min} \div \alpha_{max}]$.

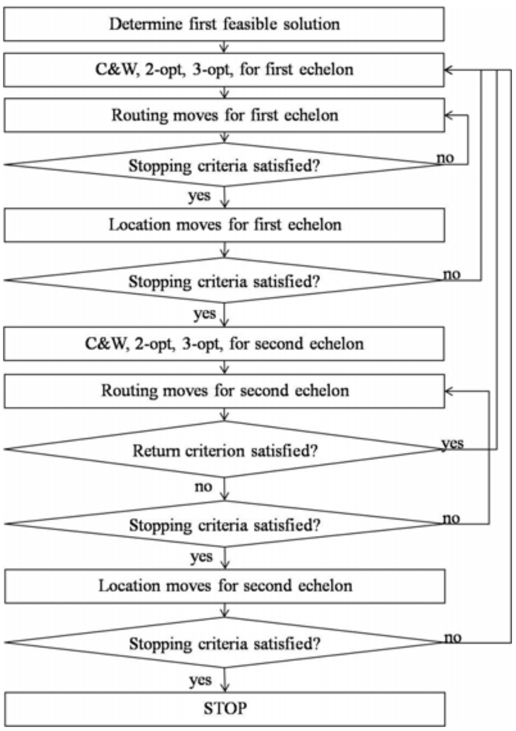
The same relations are extended to the first echelon, for which we have *near-swap-p* $= \lceil \beta\ S^* \rceil$, ,$\beta \in [0 \div 1]$ and *tabu-r-swap-multi-p* $= \lceil \alpha\ S^* \rceil$, $\alpha \in [\alpha_{min} \div \alpha_{max}]$. This move is performed until the max number, *max-r-swap-multi* of not-improving moves is met.

4.4 Combining Sub-problems

The application of the previous location and routing moves on each echelon locally optimise the four sub-components of the 2E-LRP. At this point the key element is the way we combine location and routing solutions on each echelon and location-routing solutions of the two echelons.

The four sub-problems are solved separately, but not in a pure sequential way. Indeed our approach foresees their resolution several times, in order to explore different location-routing solution combinations of first and second echelon. In particular, concerning a single echelon, the idea proposed in Tuzun and Burke [22] is adopted, i.e. each time a move is performed in the location phase, then the routing phase is run for the new location configuration. Concerning instead the two echelons, each time a change of the demand assigned to a set of open satellites occurs, i.e. each time a routing move for multiple satellites is performed, then the location-routing problem of the first echelon could be re-solved in order to find the best location and routing solution to serve the new demand configuration. Two criteria have been defined to control the return on the first echelon:

1. *Imp-CR2*: each time a better solution for the second echelon routing problem has been determined.



Determine first feasible solution
C&W, 2-opt, 3-opt, for first echelon
Routing moves for first echelon
Stopping criteria satisfied?
no
yes
Location moves for first echelon
Stopping criteria satisfied?
no
yes
C&W, 2-opt, 3-opt, for second echelon
Routing moves for second echelon
Return criterion satisfied?
yes
no
Stopping criteria satisfied?
no
yes
Location moves for second echelon
Stopping criteria satisfied?
no
yes
STOP

Fig. 1. TS scheme

2. *Violated-cap*: each time an improvement of second echelon routing cost is obtained and capacity constraints of the best determined first echelon routing solution are violated by the new demand configuration.

The main steps of the Tabu Search can be summarized as follows (figure 1):

- *Step 0*: determine the first feasible solution.
- *Step 1*: define and optimize multi-stop routes on first echelon with C&W, 2-opt and 3-opt algorithms and go to *Step 2*.
- *Step 2*: perform sequentially insert and swap moves for a single platform. If max number of not improving moves is met, then update solution with the best determined one and go to *Step 4*. Otherwise repeat *Step 2*.
- *Step 3*: perform sequentially insert and swap moves for multiple platforms. If max number of not improving moves is met, then update solution with the best determined one and go to *Step 4*. Otherwise repeat *Step 3*.
- *Step 4*: perform sequentially swap and add location moves for first echelon and return to *Step 2*. If max number of not improving moves is met, then

Table 1. Tabu Search settings TS1 and TS2

Setting TS1		Setting TS2	
Rand-sel-out	true	*Rand-sel-out*	true
Perc-sel	0.10	*Perc-sel*	0.50
Violated-cap	true	*Violated-cap*	true
tabu-swap-loc-s	$[25\% \div 50\%]$	*tabu-swap-loc-s*	$[30\% \div 80\%]$
max-swap-loc-s	4	*max-swap-loc-s*	7
tabu-swap-loc-p	$[25\% \div 50\%]$	*tabu-swap-loc-p*	$[30\% \div 50\%]$
max-swap-loc-p	2	*max-swap-loc-p*	5
tabu-add-loc-s	$[15\% \div 30\%]$	*tabu-add-loc-s*	$[10\% \div 30\%]$
max-add-loc-s	3	*max-add-loc-s*	5
tabu-add-loc-p	$[15\% \div 30\%]$	*tabu-add-loc-p*	$[10\% \div 30\%]$
max-add-loc-p	3	*max-add-loc-p*	5
tabu-r-ins-single-s	$[30\% \div 80\%]$	*tabu-r-ins-single-s*	$[20\% \div 50\%]$
tabu-r-ins-single-p	$[30\% \div 80\%]$	*tabu-r-ins-single-p*	$[20\% \div 50\%]$
max-r-ins-single	3	*max-r-ins-single*	5
tabu-r-swap-single-s	$[30\% \div 80\%]$	*tabu-r-swap-single-s*	$[30\% \div 80\%]$
tabu-r-swap-single-p	$[30\% \div 80\%]$	*tabu-r-swap-single-p*	$[30\% \div 80\%]$
max-r-swap-single	3	*max-r-swap-single*	5
near-ins-s	0.10	*near-ins-s*	0.50
tabu-r-ins-multi-s	$[10\% \div 15\%]$	*tabu-r-ins-multi-s*	$[5\% \div 25\%]$
near-ins-p	0.10	*near-ins-p*	0.50
tabu-r-ins-multi-p	$[10\% \div 15\%]$	*tabu-r-ins-multi-p*	$[5\% \div 25\%]$
max-r-ins-multi	5	*max-r-ins-multi*	7
near-swap-s	0.10	*near-swap-s*	0.25
tabu-r-swap-multi-s	$[10\% \div 15\%]$	*tabu-r-swap-multi-s*	$[5\% \div 25\%]$
near-swap-p	0.10	*near-swap-p*	0.50
tabu-r-swap-multi-p	$[10\% \div 15\%]$	*tabu-r-swap-multi-p*	$[5\% \div 25\%]$
max-r-swap-multi	3	*max-r-swap-multi*	7

Table 2. Tabu Search vs. models on small instances I1

Instance	MS	CPU	TS1	CPU-1	GAP-1	TS2	CPU-2	GAP-2
I1-238	591.83*	10.23	591.83	0.39	0.000	591.83	0.50	0.000
I1-239	878.69*	9.87	902.45	0.59	-0.027	878.69	1.08	0.000
I1-248	625.96*	175.60	625.96	0.85	0.000	625.96	1.67	0.000
I1-2410	862.91*	582.90	862.91	0.85	0.000	862.91	4.07	0.000
I1-2415	1105.67	1469.90	1121.50	1.92	-0.014	1105.67	4.15	0.000
I1-3510	829.25*	2194.70	952.86	1.25	-0.149	829.25	5.29	0.000
I1-3515	1019.57	3893.50	1068.00	2.21	-0.048	1019.57	6.16	0.000
I1-2820	1055.65	7200.00	1114.41	5.28	-0.056	1055.20	48.22	0.000
I1-2825	992.08	7200.00	1021.69	4.00	-0.030	979.85	35.91	0.012
I1-21015	732.48	7200.00	754.63	1.53	-0.030	732.48	10.81	0.000
I1-21020	951.01	7200.00	1008.17	3.40	-0.060	947.65	51.94	0.004
I1-21025	1170.72	7200.00	1085.67	6.81	0.073	1084.26	86.17	0.074
I1-3810	604.37	4982.30	604.37	1.24	0.000	604.37	11.26	0.000
I1-3815	730.36	7200.00	730.36	1.61	0.000	730.36	11.12	0.000
I1-3820	898.75	7200.00	968.59	5.54	-0.078	898.08	154.62	0.001
I1-3825	1141.26	7200.00	943.25	7.65	0.173	896.99	171.55	0.214
I1-31015	699.11	7200.00	744.57	2.35	-0.065	731.77	28.49	-0.047
I1-31020	810.26	7200.00	979.07	4.55	-0.208	851.18	189.97	-0.051
I1-31025	1291.68	7200.00	1131.59	3.94	0.124	1105.91	113.48	0.144
I1-41020	1208.72	7200.00	1287.14	9.49	-0.065	1158.92	243.36	0.041
I1-41025	1615.33	7200.00	1588.95	45.01	0.016	1582.01	308.68	0.021

update solution with the best determined one and go to *Step 5*. Otherwise repeat *Step 4*.

– *Step 5*: define and optimize multi-stop routes on the second echelon with C&W, 2-opt and 3-opt algorithms and go to *Step 6*.
– *Step 6*: perform sequentially insert and swap moves for a single open satellite. If max number of not improving moves is met, then update solution with the best determined one and go to *Step 7*. Otherwise repeat *Step 6*.
– *Step 7*: perform sequentially insert and swap moves for multiple satellites. If one of the criteria *Imp-CR2* or *Violated-cap* is satisfied, return to *Step 1*, otherwise repeat *Step 7*. If max number of not improving moves is met, then update the solution with the best determined one and go to *Step 8*.
– *Step 8*: perform sequentially swap and add location moves for the second echelon and return to *Step 1*. If max number of not improving moves is met, then update solution with the best determined one and *STOP*. Otherwise return to *Step 1*.

5 Computational Results

TS heuristics require an important tuning phase to be effective. In the following, for sake of brevity, we will not report the results obtained with all the experienced

Table 3. Tabu Search vs. models on small instances I2

Instance	MS	CPU	TS1	CPU-1	GAP-1	TS2	CPU-2	GAP-2
I2-238	589.38*	6.45	589.38	0.42	0.000	589.38	0.66	0.000
I2-239	413.54*	8.31	413.54	0.54	0.000	413.54	1.01	0.000
I2-248	605.40*	182.50	605.40	0.56	0.000	605.40	1.61	0.000
I2-2410	629.38*	834.30	629.38	0.91	0.000	629.38	2.34	0.000
I2-2415	912.73*	1525.30	943.35	1.76	-0.034	912.73	3.71	0.000
I2-3510	551.45*	2281.50	551.45	1.13	0.000	551.45	5.87	0.000
I2-3515	1170.83*	4365.50	1214.31	6.05	-0.037	1170.83	32.12	0.000
I2-2820	822.85	7200.00	867.41	2.67	-0.054	822.85	37.72	0.000
I2-2825	947.84	7200.00	959.13	6.07	-0.012	956.34	37.48	-0.009
I2-21015	727.77	7200.00	749.17	3.97	-0.029	727.77	39.36	0.000
I2-21020	801.28	7200.00	856.57	4.17	-0.069	790.57	54.55	0.013
I2-21025	1263.54	7200.00	1017.53	4.79	0.195	961.74	61.29	0.239
I2-3810	504.20	6412.23	583.73	1.06	-0.158	504.20	13.04	0.000
I2-3815	685.48	7200.00	688.68	1.74	-0.005	685.48	20.06	0.000
I2-3820	805.38	7200.00	769.04	4.57	0.045	765.01	82.03	0.050
I2-3825	1026.36	7200.00	1055.80	4.73	-0.029	1026.36	38.65	0.000
I2-31015	812.13	7200.00	813.52	2.15	-0.002	777.49	82.22	0.043
I2-31020	806.67	7200.00	843.23	5.39	-0.045	794.58	153.01	0.015
I2-31025	1254.62	7200.00	1015.10	6.56	0.191	1010.51	152.30	0.195
I2-41020	1093.34	7200.00	868.03	20.29	0.206	802.60	433.90	0.266
I2-41025	1380.86	7200.00	1193.23	20.35	0.136	1185.31	320.03	0.142

parameter settings, but we will concentrate on two of them, referred as *TS1* and *TS2*. The values used in the two settings are reported in Table 1. These settings differ for the size of the explored neighborhood.

For small instances, TS results have been compared with those obtained by the formulation proposed in Sterle [21] within 2 hours computation time.

For medium and large instances, the comparison has been done with the results obtained with a decomposition approach, sequentially solving one 2E-FLP and two MDVRP (one for each echelon). The evaluation of the gap $\Delta(z)$ between TS and bounds, for a generic instance I, is computed as $\Delta(z) = [1 - z(TS_I)/z(BS_I)]$, where $z(TS_I)$ and $z(BS_I)$ are respectively the solution value obtained by the TS heuristic and the bound value. A positive gap value indicates that TS solution improves available bound.

Exact models have been solved by Xpress-MP 7.0 solver. TS and models were run on an Intel(R) Pentium(R) 4(2.40 GHz, RAM 4.00 GB) on three set of instances. Instances have been generated through an *instance generator* developed in *C++*. We just point out that the three sets of instances differ for the spatial distribution of satellites. The notation, used to describe instances, refers to instance set and the number of platforms, satellites and customers. Therefore *I1-51050* refers to an instance of set *I1* with *5* platforms, 10 *satellites* and *50* customers.

Table 4. Tabu Search vs. models on small instances I3

Instance	MS	CPU	TS1	CPU-1	GAP-1	TS2	CPU-2	GAP-2
I3-238	589.80*	8.13	589.80	0.39	0.000	589.78	1.00	0.000
I3-239	454.63*	7.10	466.01	0.41	-0.025	454.63	1.01	0.000
I3-248	451.62*	164.70	451.62	1.61	0.000	451.62	1.61	0.000
I3-2410	546.36*	416.80	546.36	0.92	0.000	546.36	2.29	0.000
I3-2415	718.16*	1225.30	805.46	1.25	-0.122	718.16	4.73	0.000
I3-3510	745.85*	2674.20	747.37	1.76	-0.002	745.85	5.49	0.000
I3-3515	1033.79*	3065.50	1071.98	2.48	-0.037	1033.79	7.08	0.000
I3-2820	829.20	7200.00	893.36	2.59	-0.077	829.20	32.24	0.000
I3-2825	1100.31	7200.00	1004.86	4.89	0.087	959.97	41.69	0.128
I3-21015	620.86	7200.00	620.86	1.98	0.000	620.86	15.05	0.000
I3-21020	790.99	7200.00	757.21	2.54	0.043	756.51	38.06	0.044
I3-21025	944.84	7200.00	879.83	5.74	0.069	867.60	49.18	0.082
I3-3810	412.91*	3376.23	490.78	1.00	-0.189	412.91	14.88	0.000
I3-3815	624.55	7200.00	626.84	1.39	-0.004	624.55	22.35	0.000
I3-3820	707.57	7200.00	732.83	3.22	-0.036	707.57	187.18	0.000
I3-3825	977.10	7200.00	860.26	4.10	0.120	806.71	133.70	0.174
I3-31015	574.26	7200.00	624.73	1.70	-0.088	574.26	40.61	0.000
I3-31020	789.49	7200.00	781.39	3.69	0.010	745.85	256.91	0.055
I3-31025	1038.58	7200.00	913.31	2.70	0.121	860.805	91.11	0.171
I3-41020	1287.23	7200.00	1301.56	10.49	-0.011	1204.57	274.56	0.064
I3-41025	1089.40	7200.00	1141.80	20.28	-0.048	1089.40	467.69	0.000

Results on small instances are shown in Tables 2, 3, 4, reporting, for each instance, the model solution value (*MS*) and TS solution value (*TS1* and *TS2*) with related CPU time and gap. From these tables we can observe that in all cases, where the optimal solution for an instance was known (marked with *), TS was able to determine it at least with one setting. More precisely, concerning *setting TS1*, the gap varies between +0.206 and −0.208. In the worst cases it is equal to −0.208 for set *I1*, −0.158 for set *I2* and −0.189 for set *I3*. On the other side computation time is always lower than *45* seconds. Concerning instead *setting TS2*, the gap is, in the most of the cases, positive and it varies between +0.256 and −0.051. Computation times increase with respect to*setting 1*, but they are significantly lower than the ones of the solver (less than *360* seconds).

Results on medium and large size instances are shown in Tables 5, 6 and 7. TS solution values are compared with those of the decomposition approach (*DA*). Concerning *setting TS1* we can observe that TS results are very close to the ones of the decomposition approach, but the saving in terms of computation time is meaningful. The gap varies between +0.283 and −0.094 and computation times are always lower than *600* seconds with the only exception of instance *I2-41025*. Concerning instead *setting TS2*, it outperforms decomposition approach in most of the instances, but the saving in terms of computation time is not so large as for *setting TS1*. In particular the gap varies between +0.295 and −0.008 and computation time varies between 390.62 and 7850.52 seconds.

Table 5. Tabu Search vs. decomposition approach on medium-large instances I1

Instance	DA	CPU	TS1	CPU-1	GAP-1	TS2	CPU-2	GAP-2
I1-5850	1226.24	4421.30	1236.65	15.57	-0.008	1210.27	521.72	0.013
I1-51050	1783.60	6134.90	1279.02	30.26	0.283	1256.59	853.57	0.295
I1-51075	1591.60	7512.60	1669.67	61.06	-0.049	1591.60	1026.12	0.000
I1-51575	1783.60	6134.90	1780.32	32.82	0.002	1708.79	2614.13	0.042
I1-510100	2247.32	8033.80	2458.50	121.66	-0.094	2257.35	1906.17	-0.004
I1-520100	2055.88	10218.10	2124.69	249.70	-0.033	2071.76	3780.61	-0.008
I1-510150	2177.77	8407.10	2220.47	345.53	-0.020	2097.81	3740.38	0.037
I1-520150	1933.82	7786.60	2098.87	538.99	-0.085	1919.35	3271.92	0.007
I1-510200	2625.11	10119.50	2761.73	440.27	-0.052	2601.33	2239.09	0.009
I1-520200	3140.17	12750.30	2546.74	473.41	0.189	2407.33	6037.38	0.233

Table 6. Tabu Search vs. decomposition approach on medium-large instances I2

Instance	DA	CPU	TS1	CPU-1	GAP-1	TS2	CPU-2	GAP-2
I2-5850	1185.75	2023.34	1207.39	21.02	-0.018	1185.75	665.22	0.000
I2-51050	1325.61	5039.50	1350.55	18.08	-0.019	1335.81	390.624	-0.008
I2-51075	1768.88	7061.00	1813.01	68.34	-0.025	1756.13	1252.878	0.007
I2-51575	1644.79	9499.40	1710.38	53.43	-0.040	1644.79	944.352	0.000
I2-510100	2391.17	10379.60	2411.03	60.60	-0.008	2290.64	769.242	0.042
I2-520100	2051.39	12405.60	2051.39	257.63	0.000	2041.13	2608.404	0.005
I2-510150	2111.97	14060.90	2018.49	302.78	0.044	1907.71	4852.92	0.097
I2-520150	1800.89	10134.50	1772.90	631.48	0.016	1707.73	4540.74	0.052
I2-510200	2430.93	8871.80	2435.05	101.01	-0.002	2407.88	1078.866	0.009
I2-520200	2274.29	15602.10	2260.65	1237.99	0.006	2223.72	7850.52	0.022

Table 7. Tabu Search vs. decomposition approach on medium-large instances I3

Instance	DA	CPU	TS1	CPU-1	GAP-1	TS2	CPU-2	GAP-2
I3-5850	1298.89	7741.90	1351.27	16.67	-0.040	1240.80	474.94	0.045
I3-51050	1256.68	4929.60	1297.51	24.68	-0.032	1243.87	919.53	0.010
I3-51075	1879.56	13720.00	1937.27	45.20	-0.031	1839.38	806.94	0.021
I3-51575	1704.65	12903.90	1602.72	42.19	0.060	1590.00	1910.94	0.067
I3-510100	2601.44	20599.60	2420.47	37.79	0.070	2294.44	546.61	0.118
I3-520100	2261.36	15724.50	2278.57	75.72	-0.008	2170.45	696.22	0.040
I3-510150	1470.77	243.90	1398.69	182.22	0.049	1342.18	2635.06	0.087
I3-520150	1508.07	21240.50	1454.31	232.34	0.036	1343.72	3379.30	0.109
I3-510200	2193.32	41145.10	2030.30	351.31	0.074	1893.68	2633.48	0.137
I3-520200	2784.47	23319.40	2737.23	343.97	0.017	2692.31	2765.42	0.033

6 Conclusions

2E-LRP has been scarcely treated in literature with both exact and heuristic methods. A Tabu Search heuristic has been proposed. It decomposes the problem in four subproblems, one CFLP and one MDVRP for each echelon. The four sub-problems are sequentially and iteratively solved and their solutions are opportunely combined in order to determine a good global solution. Tabu Search has been experienced on three set of small, medium and large instances and the obtained results have been compared with bounds derived from exact models. Experimental results prove that the proposed TS is effective in terms of quality of solutions and computation times in the most of the solved instances. The proposed Tabu Search could be easily integrated with additional constraints and adapted to the asymmetric case.

References

1. Ambrosino, D., Scutellá, M.G.: Distribution network design: New problems and related models. Eur. Jour. of Oper. Res. 165, 610–624 (2005)
2. Albareda, S.M., Diaz, J., Fernández, E.: A compact model and tight bounds for a combined location-routing problem. Comp. and Oper. Res. 32, 407–428 (2005)
3. Barreto, S., Ferreira, C., Paixao, J., Santon, B.S.: Using clustering analysis in a capacitated location-routing problem. Eur. Jour. of Oper. Res. 179(3), 968–977 (2007)
4. Chien, T.W.: Heuristic Procedures for Practical-Sized Uncapacitated Location-Capacitated Routing Problems. Decision Sciences 24(5), 993–1021 (1993)
5. Crainic, T.G., Ricciardi, N., Storchi, G.: Advanced freight transportation systems for congested urban areas. Transportation Research Part C: Emerging Technologies 12(2), 119–137 (2004)
6. Hansen, P.H., Hegedahl, B., Hjortkjaer, S., Obel, B.: A heuristic solution to the warehouse location-routing problem. Eur. Jour. of Oper. Res. 76, 111–127 (1994)
7. Jacobsen, S.K., Madsen, O.B.G.: A comparative study of heuristics for a two-level location routing problem. Eur. Jour. of Oper. Res. 5, 378–387 (1980)
8. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J. (eds.) Complexity of Computer Computations, pp. 85–104. Plenum Press, New York (1972)
9. Laporte, G.: Location-routing problems. In: Golden, B.L., Assad, A.A. (eds.) Vehicle routing: Methods and studies. North-Holland, Amsterdam (1988)
10. Laporte, G., Nobert, Y.: An exact algorithm for minimizing routing and operating costs in depot location. European Journal of Oper. Res. 6, 224–226 (1981)
11. Laporte, G., Nobert, Y., Arpin, D.: An exact algorithm for solving a capacitated location-routing problem. Annals of Operations Research 6, 293–310 (1986)
12. Laporte, G., Nobert, Y., Taillefer, S.: Solving a family of multi-depot vehicle routing and location-routing problems. Transportation Science 22, 161–172 (1988)
13. Madsen, O.B.G.: Methods for solving combined two level location-routing problems of realistic dimensions. European Journal of Oper. Res. 12(3), 295–301 (1983)
14. Melechovský, J., Prins, C., Calvo, R.W.: A metaheuristic to solve a location-routing problem with non-linear costs. Journal of Heuristics 11, 375–391 (2005)

15. Min, H., Jayaraman, V., Srivastava, R.: Combined location-routing problems: A synthesis and future research directions. Eur. Jour. of Oper. Res. 108, 1–15 (1998)
16. Nagy, G., Salhi, S.: Nested Heuristic Methods for the Location-Routeing Problem. The Journal of the Operational Research Society 47(9), 1166–1174 (1996)
17. Nagy, G., Salhi, S.: Location-routing: Issues, models and methods. European Journal of Operational Research 177, 649–672 (2007)
18. Or, I., Pierskalla, W.P.: A transportation location-allocation model for regional blood banking. AIIE Transactions 11, 86–95 (1979)
19. Perl, J., Daskin, M.S.: A warehouse location-routing problem. Transportation Research Part B 19(5), 381–396 (1985)
20. Srivastava, R.: Alternate solution procedures for the location-routing problem. Omega International Journal of Management Science 21(4), 497–506 (1993)
21. Sterle, C.: Location-routing models and methods for Freight Distribution and Infomobility in City Logistics. PhD thesis, University of Naples, Italy (2009)
22. Tuzun, D., Burke, L.I.: A two-phase tabu search approach to the location routing problem. European Journal of Operational Research 116, 87–99 (1999)
23. Wu, T.H., Chinyao, L., Bai, J.W.: Heuristic solutions to multi-depot location-routing problems. Computers & Operations Research 29, 1393–1415 (2002)

New Fast Heuristics for the 2D Strip Packing Problem with Guillotine Constraint

Minh Hoang Ha[1,2], François Clautiaux[3],
Saïd Hanafi[4], and Christophe Wilbaut[4]

[1] Ecole des Mines de Nantes, IRCCyN UMR CNRS 6597
[2] Ecole Polytechnique de Montréal, CIRRELT
[3] Université des Sciences et Technologies de Lille, LIFL UMR CNRS 8022
[4] Université de Valenciennes et du Hainaut-Cambrésis, LAMIH FRE CNRS 3304

Abstract. In this paper, we propose new and fast level-packing algorithms to solve the two-dimensional strip rectangular packing problem with guillotine constraints. Our methods are based on constructive and destructive strategies. The computational results on many different instances show that our method leads to the best results in many cases among fast heuristics.

Keywords: strip packing, fast heuristic, level algorithms, construction and destruction.

1 Introduction

Given a set $I = \{1, \ldots, n\}$ of rectangular items i, and a bin of fixed width and infinitive height, the two-dimensional strip packing problem (2D-SPP) consists in orthogonally packing all the items into the bin, without overlapping, with the objective of minimizing the total height of the packing within the strip. The size of the bin is denoted by W, and the width (resp. the height) of each item i is denoted by w_i (resp. h_i).

In the typology of [20], 2D-SPP belongs to the class of *two-dimensional open-dimension packing problems*. In many industrial applications, two constraints can be added to the problem: (C1) Orientation constraint. In general, the rotating of the items by *90* degrees is permitted. However, in some real applications, the orientation of all items is fixed. (C2) Guillotine constraint. This constraint requires that the layout can be obtained by a serie of guillotine cuts, *i.e.* edge-to-edge cuts parallel to the edges of the bin.

There are several exact algorithms [8,13] and metaheuristic-based algorithms [4,10,12] proposed for this problem. However, generally speaking, these algorithms are more time consuming and are hence less practical for problems having a large number of items. Moreover, in the nowadays shipping industry, the customer demands must be satisfied austerely and rapidly. In this paper, we focus on fast algorithms, whose running time is no more than a few seconds for the large instances available (about 500 items).

Our methods belong to the class of so-called *level* heuristics (see [17]). In this class of methods, items are packed side-by-side by horizontal *levels*. The height of a level is determined by the tallest item it contains. We use classical strategies to build levels into a construction/destruction scheme. At each step of the algorithm, several different levels are built and only the "best" is kept in the solution. Several strategies for level construction and selection were tested. Computational experiments show that our approach leads to high quality results compared to those of the literature.

The remainder of the paper is organized as follows. Section 2 reintroduces known level algorithms in the literature. In Section 3, we present an adaptation of an existing heuristic for the case with rotation. Section 4 describes our *Construction and Destruction Heuristic algorithm* denoted by CDH. Computational results are reported and analyzed in Section 5. Finally, Section 6 summarizes our conclusions.

2 Efficient Level Algorithms

Many level heuristics have been developed for strip packing without rotations. The most basic ones are based on classical strategies: *first-fit decreasing height* (FFDH) [9], *next-fit decreasing height* (NFDH) [9], and *best-fit decreasing height* (BFDH) [16]. For these heuristics, the items are ordered according to non-increasing height, and considered one by one following this initial ordering. The current item i can be packed in an existing level, or in a newly created level. The three variants only differ in the choice of the level. In NFDH, i is packed in the highest level (if possible). In FFDH, i is packed in the first level in which it fits. Finally, in BFDH, i is packed in the level with the minimum residual horizontal space. Among these heuristics, NFDH is often the worst and BFDH has shown to be useful for guillotineable problems.

Based on these basic level heuristics and exploiting the non-used spaces, some other heuristics were introduced such as *Split Fit* (SF) [9], *Size Alternating Stack* (SAS) [17], or *Floor-Ceiling* (FC) [15,14]. According to computational results reported in [17], FCN^R (that corresponds to the FC algorithm used for the oriented case with guillotineable constraint) is the best one in terms of the distance between the obtained height and an optimal solution, and in the number of times the algorithm obtains the smallest strip height. According to this paper, FCN^R is followed by SAS. The reader is refered to [17] for more details.

In the case where rotations are allowed, Bortfeldt [4] introduced BFDH* improved upon the BFDH algorithm by two modifications. The first modification deals with the selection of an existing layer for the current item. As in BFDH the layer with minimal remaining free width is sought to accommodate the item. However during the search the orientation of the item is no longer fixed: for each layer, both orientations of the item are tested. The second modification consists in trying to utilize the remaining free space within levels above the items at the level bottom by including the item with the largest surface in this space, which is on the leftmost side of the available area.

In 2006, Zhang et al. [21] proposed the so-called *heuristic recursive* (HR) algorithm. After packing an item, the unpacked space is divided into two subspaces according to two cases (See Figure 1 (a)). Then these subspaces are packed recursively by the next item with the largest area. Local search is used when all orderings of items are checked. HR obtains quickly good results on the benchmarks of Hopper and Turton [10]. The computational results have shown that the HR algorithm outperforms the best metaheuristic proposed so far [10] in relative distance of best solution to optimum height and in running time.

3 Adaptation of BFDH to the Case with Rotation

In this section we introduce IBFDH^R, a variant of BFDH [16] where the rotations are allowed. Firstly, we reintroduce BFDH^R that was adapted to use for non-oriented case. This heuristic works as follows: first, the items are oriented such that the width is greater than or equal to the height, and then they are ordered by non-increasing height. After this, the items are processed one by one. Each item is packed into a rectangular level, at the level bottom and left justified. The width of a level is given by the bin width while its height is determined by the height of the first item packed into the level. If at least one existing level can contain the current item, the level with minimum residual horizontal space is chosen. Otherwise, a new level is created above the existing levels and initialized with the current item. The first modification that was introduced by Bortfeldt [4] is reused (see Section 2).

As this traditional version of BFDH does not work when there exists an item whose longer edge is larger than the width of the bin (i.e., $l_i = \max(w_i, h_i) > W$) (see ngcut04, ngcut05, ngcut06 [1,2], for example), a modification is added as follows: the items are divided into two groups. Group 1 includes the items whose longer edges are larger than the width of bin; all remaining items are put in group 2. First the items of group 1 are oriented vertically whereas the items of group 2 are oriented horizontally. Then items of group 1 are packed using the BFDH^R algorithm. Finally items of group 2 are packed using a slight modification of the BFDH^R algorithm: the existing levels that were created in the previous phase (packing of group 1) are considered first. In this process, the current item is oriented vertically and normally packed. The process is repeated until all items are packed.

To improve this algorithm, especially for the non-zero waste instances, another simple algorithm is used. In this algorithm, the items are first ordered by non-increasing height. Then they are packed with BFDH^R. The algorithm IBFDH^R is obtained by applying the previous two sub-algorithms, and by keeping the best solution obtained.

4 Constructive and Destructive Heuristics(CDH)

In this section we propose a new algorithm based on the use of two phases. The main idea of this algorithm is to construct a solution by generating iteratively several solutions according to classical rules, and by keeping at every stage one level of the best solution (the strategies used to choose this level are discussed in Section 5). At the end of each iteration, the rest of the solution is destructed. After each destructive phase, the items of the best strip are removed from the instance, and the method is rerun until all items are packed. Note that this construction / destruction methodology is an approach that has been successfully applied to several other problems (under the name of iterated greedy [18] or ruin-and-recreate [19]).

The overall method is described in Algorithm 1. The creation and the selection of one strip are detailed in Algorithm 2.

In order to simplify the notations in the algorithms, for a given strip z, we will respectively denote $h(z)$ and $I(z)$ the height of z and the set of items packed in z.

Algorithm 1. General method used in CDH

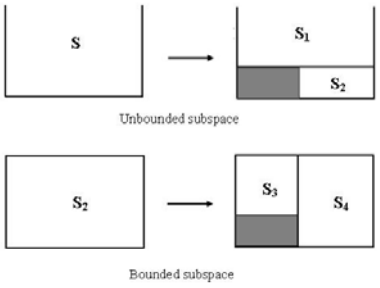
```
Input
$I$ : the inital set of items;
$W$ : the width of the strip;
$h \leftarrow 0$;
while $I \neq \emptyset$ do
    $z \leftarrow createOneStrip(W, I)$ ;
    $I \leftarrow I \setminus I(z)$;
    $h \leftarrow h + h(z)$;
return $h$;
```

Algorithm 2. CreateOneStrip

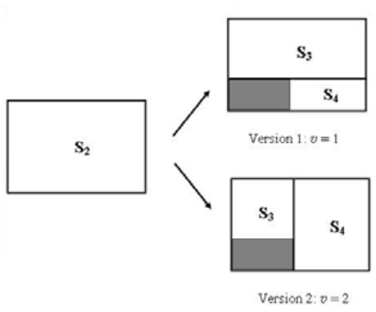
```
Input
$I$ : the current set of items;
$W$ : the width of the strip;
$cpt \leftarrow 1$;
//Let $\mathbf{Z}$ be a list of set of strips
forall combinations of ordering $\sigma$, versions $v$ and parameters $r$ do
    $\mathbf{Z}[cpt] \leftarrow$ PackUnBounded($I$, $W$, $\sigma$, $v$, $r$);
    $cpt \leftarrow cpt + 1$;
Select a value of $cpt$ such that $\mathbf{Z}[cpt]$ leads to the best height of strip;
Select the best strip $z^*$ in $\mathbf{Z}[cpt]$ following a given criterion;
return $z^*$;
```

After packing an item i into the original space S to initialize a new level, we have an unbounded space S_1 and a bounded space S_2 as in HR (see Figure 1 (a)). The unbounded space is a vertical strip of infinite height, whereas the bounded space is a strip of height h_i. Here S_1 is similar to S, so we can use the same packing procedure to treat both.

To pack the remaining items in I in the two subspaces S_1 and S_2, we use two algorithms: `PackingUnBounded`(I, S_1, σ, v, r) and `PackingBounded`(I, S_2,



(a) Dividing the subspaces in HR



(b) Dividing the bounded subspace in CDH

Fig. 1. Two versions for dividing the subspaces

Algorithm 3. PackUnBounded

```
Input
$I$: the set of items to be packed;
$W$: the width of the bin;
$\sigma$: an ordering of $I$;
$v$: the version used for dividing the space;
$r$: the rule used for choosing the next item;
Let $Z$ be an empty list of strips;
Sort rectangles following order $\sigma$;
while $I \neq \emptyset$ do
    Pack the first item $i$;
    $I \leftarrow I \setminus \{i\}$;
    $z \leftarrow$ PackBounded($I$, $(W - w_i, h_i)$, $v$, $r$);
    append item $i$ to the strip $z$;
    $Z \leftarrow Z \cup \{z\}$;
    $I \leftarrow I \setminus I(z)$;
return $Z$;
```

v, r) respectively (see Algorithms 3 and 4), where σ is a given ordering of the items, and parameters v and r stand for *version* (for dividing the space into two subspaces when an item is packed) and *rule* (for choosing the next item to be packed) respectively. Please note that S_2 in the call of `PackingBounded` is defined by a width and a height (see Algorithm 4).

In `PackingUnBounded`(I, S, σ, v, r), we use four different orderings σ of the items: (i) decreasing height, then decreasing width; (ii) decreasing width, then decreasing height; (iii) decreasing area; and (iv) decreasing perimeter.

In `PackingBounded`(I, S_2, v, r) we used two rules to select an item: the tallest item ($r = 1$) and the largest item (i.e. an item with the maximum area) ($r = 2$). One can observe that when dividing S_2 into S_3 and S_4 we use two versions: $v = 1$

Algorithm 4. PackBounded

```
Input
$I$: the set of remaining items;
$(\bar{w}, \bar{h})$: the width and the height of the bounded subspace;
$v$: the version used for dividing the space;
$r$: the rule used for choosing the next item;
if no item of I can be packed in the current subspace then
    return a strip of height 0 and with an empty item set;
Select an item $i$ to be packed using the rule $r$;
Pack $i$ and create the two corresponding bounded subspaces $(w^1, h^1)$ and
$(w^2, h^2)$ following rule $v$;
$Z_1 \leftarrow$ PackBounded$(I, (w^1, h^1), v, r)$;
$I \leftarrow I \setminus I(Z_1)$;
$Z_2 \leftarrow$ PackBounded$(I, (w^2, h^2), v, r)$;
$I \leftarrow I \setminus I(Z_2)$;
Assemble $Z_1$, $Z_2$ and item $i$ to form one strip $z$;
return $z$
```

and $v = 2$ (see Figure 1 (a) for the version related to HR and Figure 1 (b) for the version related to CDH).

Now we consider the time complexity of our method. For `PackingUnBounded`(I, S, σ, v, r), we have to take into account the ordering of items, which is in $O(n \log n)$. Then `PackingBounded` is called until all items are packed. PackingBounded always takes the next item following rule r ($O(1)$ if the items are stored in a suitable data structure) and never consider an item twice. Thus the time complexity of `PackingBounded` is $O(n \log n)$. Since the maximum levels created is n, the procedure `createOneStrip` cannot be called more than n times. This gives the overall time complexity of the CDH algorithm: $T(n) = O(n^2 \log n)$.

We consider the cases with and without rotation, which respectively lead to two versions: CDH non-oriented (CDH^R) and CDH oriented (CDH^O). To deal with rotations, in the CDH^R algorithm, n new items are added. Each item in the new instance considered corresponds with an item in one of its two possible orientations, similarly to [7]. Hence, there are $2n$ items and after selecting an item, its *double* has to be deleted. The computational results showed that this additional strategy is more effective than treating n items. For both cases, by combining with two rules to select the item and two versions to divide bounded spaces, we used 16 different sets of parameters for `PackingUnbounded`(I, S, σ, v, r).

5 Computational Experiments

In this section, we first study the practical effectiveness of several strategies for choosing levels in our constructive-destructive heuristics. Then we compare our new heuristics to those of the literature on a large set of classical instances.

All new algorithms were implemented on a 1.6 GHz CPU with 1GB RAM. The results of the HR^R algorithm [21] were carried out on a 2.4 GHz CPU. The results of all other algorithms in the literature are taken from [17]. We tested our algorithms on 141 instances: ngcut01-12 and gcut01-13 [1,2], cgcut01-03 [6], beng01-10 [3], N1-N12 [5], and C1-C7, H1-H7, T1-T7 [11,10]. Note that the results of some algorithms are not available for some instances. We only report the results for which all results are known. We respectively use 122 and 129 instances in Tables 3 and 4.

In the following tables, $IBFDH^R$ corresponds to our adaptation of the algorithm in [16] and CDH^O (resp. CDH^R) corresponds to the constructive-desctructive heuristics dedicated to the oriented (resp. non-oriented) case. Finally HR^O denotes algorithm obtained from HR [21] where the orientation of each item is determined.

5.1 Comparisons of Various Strategies

In our new heuristic CDH, the choice of the level to keep at each step is crucial. First we report our experiments on this issue.

We tested the following four strategies: (i) first level (Strategy 1), (ii) minimum wasted area (Strategy 2), (iii) minimum ratio between wasted area and height of level (Strategy 3); and (iv) maximum number of items in level (Strategy 4). We compare these strategies on the 141 instances listed above. The results are reported in Tables 1 and 2. We used four criteria for the comparison:

– % improvement: this criterion is equal to (number of improved instances compared to the original constructive heuristics) / (total number of instances). The higher this ratio is, the more effective our strategy is;
– Average: the average value of solution;
– Number of times the best known solution is obtained;
– Number of times the algorithm is strictly better than the others.

Tables 1 and 2 underline that Strategy 3 is slightly better than the others for the benchmarks we used, in particular for criteria 1 and 2. However, note that Strategies 1 and 2 lead to more unique best solutions for the case with rotations. We will use Strategy 3 for additional experiments in the remainder of the paper. Note that our method clearly outperforms simple constructive algorithms.

5.2 Evaluation of CDH

In this subsection we distinguish two cases: (i) without rotations or (ii) with rotations. In the first case, we compare HR^O and CDH^O with two algorithms from the literature: SAS and FCN^R. Table 3 provides the results obtained over 122 instances. It shows that CDH^O and HR^O outperform the other methods. To compare these two algorithms we consider two criteria (as in [17]): (i) the frequencies with which each algorithm achieved the smallest strip height and (ii) how close the strip heights obtained were to an optimal solution (if known, otherwise to the continuous lower bound). The associated results are presented in Figure 2.

Table 1. Comparing several strategies for the case without rotations (Heuristic CDH^O)

Criterion	Strategy 1	Strategy 2	Strategy 3	Strategy 4
% improvement	36.17	27.66	41.13	29.79
Average	575.64	574.47	574.10	575.66
Nb. best solutions	109	105	120	98
Nb. unique best solutions	7	5	9	4

Table 2. Comparing several strategies for the case with rotations (Heuristic CDH^R)

Criterion	Strategy 1	Strategy 2	Strategy 3	Strategy 4
% improvement	52.48	55.32	56.03	41.13
Average	544.29	541.97	540.18	547.84
Nb. best solutions	98	108	102	78
Nb. unique best solutions	13	14	8	6

Table 3. Results when rotations are not allowed

Data	n	OPT	SAS	FCN^R	HR^O	CDH^O
NGCUT01	10	23	27	25	28	25
NGCUT02	17	30	33	33	33	33
NGCUT03	21	28	31	34	30	30
NGCUT04	7	20	23	23	23	23
NGCUT05	14	36	37	46	37	37
NGCUT06	15	31	35	36	35	33
NGCUT07	8	20	20	21	20	20
NGCUT08	13	33	38	38	41	38
NGCUT09	18	-	60	64	62	62
NGCUT10	13	80	85	85	86	85
NGCUT11	15	52	75	63	64	60
NGCUT12	22	87	91	96	87	87
Mean			46.25	47	45.5	44.42
CGCUT01	16	23	25	25	35	25
CGCUT02	23	-	75	73	75	74
CGCUT03	62	-	744	744	718	721
Mean			281.33	280.67	276	273.33
GCUT01	10	1016	1016	1016	1016	1016
GCUT02	20		1499	1349	1300	1329
GCUT03	30	1803	2077	1810	1873	1812
GCUT04	50		3396	3216	3143	3163
Mean			1997	1847.75	1833	1830
C1	16/17	20	25.7	22	21	21
C2	25	15	19	17	16.3	16.7
C3	28/29	30	36.3	35.3	33.7	33.3
C4	49	60	68	69.0	65.3	66.3
C5	72/73	90	102.3	98.3	95.7	94.3
C6	97	120	135.3	128.7	126	126
C7	196/197	240	266	252.7	250.3	249.3
Mean		82.14	93.23	89	86.90	86.70
H1	17	200	259.4	242.6	247.6	237.6
H2	25	200	269.4	244.8	242.2	237.8
H3	29	200	259.6	246	246.6	236.8
H4	49	200	244.8	237.6	221.6	224.8
H5	73	200	238	227.8	226.2	222.2
H6	97	200	235.4	230	220.2	214.6
H7	197	200	229.4	221.2	210.4	210.4
Mean		200	248	235.71	230.69	226.31
T1	17	200	282.4	274.4	260.6	256
T2	25	200	268.8	263.2	243	240
T3	29	200	269.2	251.8	242.6	237.6
T4	49	200	247.2	235	223.4	224.4
T5	73	200	234.8	228.2	227	221
T6	97	200	233.6	233.2	218.8	214.8
T7	199	200	224.6	226	210.2	210
Mean		200	251.51	244.54	232.23	229.11
N1	10	40	60	46	40	40
N2	20	50	65	61	53	54
N3	30	50	63	62	57	53
N4	40	80	103	93	87	87
N5	50	100	115	111	111	108
N6	60	100	110	105	106	104
N7	70	100	120	124	116	116
N8	80	80	104	92	85	85
N9	100	150	177	162	157	158
N10	200	150	158	182	155	155
N11	300	150	159	155	153	153
N12	500	300	341	343	311	308
Mean		112.5	131.25	128	119.25	118.42

Figure 2 indicates that CDH^O is better in terms of both criteria. CDH^O is also faster: its average running time for these 122 instances is 0.046 seconds against 16.08 seconds for HR^O. In addition with $n = 500$ (instance N12) the running time of CDH^O is 1.63 seconds while the running time of HR^O is 1355.48 seconds.

Table 4. Results when rotations are allowed

Data	n	OPT	IBFDH^R	HR^R	CDH^R	Data	n	OPT	IBFDH^R	HR^R	CDH^R
NGCUT01	10	20	22	24	21	CGCUT01	16	23	27	26	23
NGCUT02	17	28	33	33	30	CGCUT02	23	-	74	67	72
NGCUT03	21	28	33	32	29	CGCUT03	62	-	695	692	682
NGCUT04	7	18	21	21	21	Mean			265.33	261.67	259
NGCUT05	14	36	37	38	37	GCUT01	10	-	795	852	791
NGCUT06	15	29	35	32	30	GCUT02	20	-	1328	1510	1266
NGCUT07	8	10	10	10	10	GCUT03	30	-	1867	1905	1765
NGCUT08	13	33	37	37	35	GCUT04	50	-	3259	3217	3100
NGCUT09	18	49	56	66	56	GCUT05	10	-	1267	1404	1267
NGCUT10	13	59	64	64	61	GCUT06	20	-	2842	3403	2773
NGCUT11	15	-	63	68	56	GCUT07	30	-	4537	4799	4356
NGCUT12	22	77	90	93	87	GCUT08	50	-	6286	6608	6058
Mean			41.75	43.17	39.42	GCUT09	10	-	2523	2459	2459
BENG01	20	30	37	32	32	GCUT10	20	-	6394	6877	5865
BENG02	40	57	64	59	59	GCUT11	30	-	8050	8646	7221
BENG03	60	84	89	85	85	GCUT12	50	-	14004	14792	13440
BENG04	80	107	113	108	108	GCUT13	32	-	5407	4950	5202
BENG05	100	134	139	134	135	Mean			4504.54	4724.77	4274.08
BENG06	40	36	39	37	36	H1	17	200	246	223.6	223
BENG07	80	67	72	68	68	H2	25	200	238.4	216.6	227
BENG08	120	101	106	102	101	H3	29	200	235.8	217.2	222.6
BENG09	160	126	130	126	126	H4	49	200	229.4	212.6	218.6
BENG10	200	156	160	156	156	H5	73	200	226.8	208.6	214.4
Mean		89.8	94.9	90.7	90.6	H6	97	200	222	207.4	210.8
C1	16/17	20	23.3	21.7	21.3	H7	197	200	216.6	202.8	205
C2	25	15	18	15.7	16.3	Mean		200	230.71	212.69	217.34
C3	28/29	30	36.3	32	32.7	T1	17	200	250.2	231	232.2
C4	49	60	68.3	61.3	62.3	T2	25	200	236.4	217.6	228.4
C5	72/73	90	99.7	91.7	92.3	T3	29	200	234.4	217.4	223.4
C6	97	120	132	123	122.7	T4	49	200	229.2	212.8	218.4
C7	196/197	240	255.3	244.7	244.7	T5	73	200	225	209.8	214.4
Mean		82.14	90.41	84.3	84.61	T6	97	200	222.4	207	210.4
						T7	199	200	218.4	204.4	205.4
						Mean		200	230.86	214.29	218.97

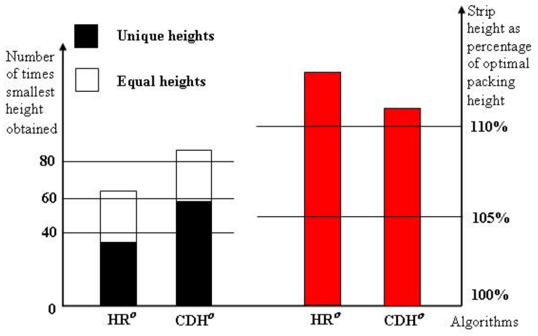


Fig. 2. Statistical results for HR^O and CDH^O over the 122 benchmark data sets in Table 3

For the case with rotations, we compare three algorithms: IBFDH^R, HR^R and CDH^R. The results reported in Table 4 show that our algorithm outperforms the HR algorithm for non-zero waste instances (*i.e.,* ngcut01-12, cgcut01-03, gcut01-13, and beng01-10). It seems that the HR algorithm is not effective for this class

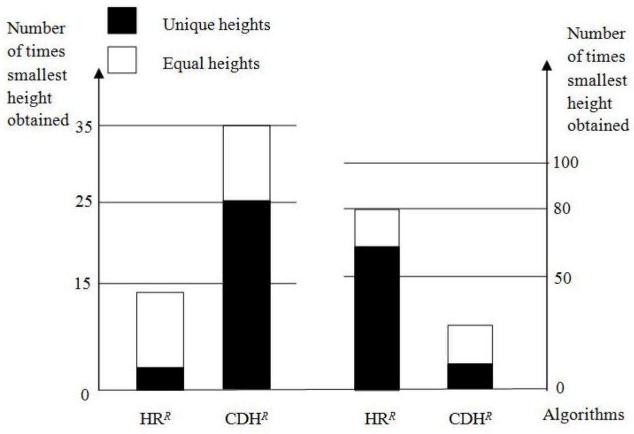


Fig. 3. Statistical results for HR^R and CDH^R over the 38 non-zero waste instances (left) and the 91 zero waste instances (right)

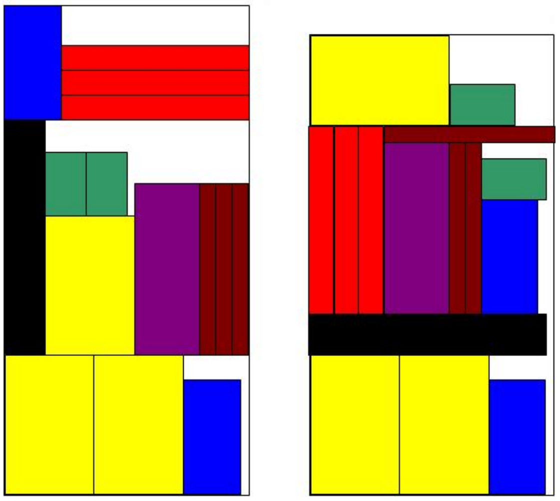


Fig. 4. Packing result of ngcut11 (n=15) for CDH

of instances since the corresponding results are even less than the results of IBFDH^R for the instances of Beasley. These results are confirmed by the Figure 3. Finally CDH^R is also faster than HR^R: its average running time for the 129 instances used in Table 4 is 0.097 seconds against 6.43 seconds for HR^R.

To illustrate the behaviour of our algorithm we provide the solution generated by CDH^O and CDH^R for the instance ngcut11 in Figure 4.

6 Final Remarks

In this paper, we introduced a new and fast algorithm for 2D strip packing problem with guillotine constraints. We tested its performance on benchmarks for the oriented and non-oriented cases. Our method outperforms all other fast algorithms except HR of Zhang *et al.* [21]. When comparing with this method, our algorithm performs better for oriented case. For the case with rotations, its results are better only on non-zero waste instances. However in both cases, our method is clearly faster.

The notion of construction/destruction can also be used in a metaheuristic framework, were levels would be iteratively removed and added. We also plan to apply this methodology to the three-dimensional case.

Acknowledgments

The present research work has been partially supported by International Campus on Safety and Intermodality in Transportation (CISIT), the Nord-Pas-de-Calais Region, the European Community, the Regional Delegation for Research and Technology, the Ministry of Higher Education and Research, and the National Center for Scientific Research (CNRS). The authors gratefully acknowledge the support of these institutions.

We would like to thank Defu Zhang who sent us his detailed computational results on different instances.

References

1. Beasley, J.E.: Algorithms for unconstrained two-dimensional guillotine cutting. Journal of the Operational Research Society 36, 297–306 (1985)
2. Beasley, J.E.: An exact two-dimensional non-guillotine cutting tree search procedure. Operations Research 33, 49–64 (1985)
3. Bengtsson, B.E.: Packing rectangular pieces - a heuristic approach. The computer journal 25, 353–357 (1982)
4. Bortfeldt, A.: A genetic algorithm for the two dimensional strip packing problem. European Journal of Operational Research 172, 814–837 (2006)
5. Burke, E.K., Kendall, G., Whitwell, G.: A new placement heuristic for the orthogonal stock-cutting problem. Operations Research 52(4), 655–671 (2004)
6. Christofides, N., Whitlock, C.: An algorithm for two-dimensional cutting problems. Operations Research 25, 30–44 (1977)
7. Clautiaux, F., Jouglet, A., El Hayek, J.: A new lower bound for the non-oriented two-dimensional bin-packing problem. Operations Research Letters 35(3), 365–373 (2007)
8. Clautiaux, F., Jouglet, A., Moukrim, A.: A new graph-theoretical model for k-dimensional guillotine-cutting problems. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 43–54. Springer, Heidelberg (2008)
9. Coffman, E., Garey, M.R., Johnson, D.S., Tarjan, R.E.: Performance bounds for level-oriented two-dimensional packing algorithms. SIAM Journal on Computing 9(4), 808–826 (1980)

10. Hopper, E., Turton, B.C.H.: An empirical investigation on metaheuristic and heuristic algorithms for a 2D packing problem. European Journal of Operational Research 128, 34–57 (2001)
11. Hopper, E., Turton, B.C.H.: Problem generators for rectangular packing problems. Studia Informatica Universalis 2(1), 123–136 (2002)
12. Iori, M., Martello, S., Monaci, M.: Metaheuristic algorithms for the strip packing problem. In: Applied Optimization, ch. 7, vol. 78. Springer, Heidelberg (2003)
13. Kenmochi, M., Imamichi, T., Nonobe, K., Yagiura, M., Nagamochi, H.: Exact algorithms for the two-dimensional strip packing problem with and without rotations. European Journal of Operational Research 198(1), 73–83 (2009)
14. Lodi, A., Martello, S., Vigo, D.: Heuristic and metaheuristic approaches for a class of two dimensional bin packing problem. INFORMS Journal on Computing 11(4), 345–357 (1999)
15. Lodi, A., Martello, S., Vigo, D.: Neighborhood search algorithm for the guillotine non-oriented two-dimensional bin packing problem. In: MIC 1997: 2nd metaheuristics international conference, pp. 125–139 (1999)
16. Mumford-Valenzuela, C., Vick, J., Wang, P.Y.: Heuristics for large strip packing problems with guillotine patterns: an empirical study, pp. 501–522 (2004)
17. Ntene, N., van Vuuren, J.H.: A survey and comparison of guillotine heuristics for the 2d oriented offline strip packing problem. Discrete Optimization 6(2), 174–188 (2009)
18. Ruiz, R., Stutzle, T.: A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. European Journal of Operational Research 177(3), 2033–2049 (2007)
19. Schrimpf, G., Schneider, J., Stamm-Wilbrandt, H., Dueck, G.: Record breaking optimization results using the ruin and recreate principle. Journal of Computational Physics 159, 139–171 (2000)
20. Waescher, G., Haussner, H., Schumann, H.: An improved typology for C&P problems. presentation, and final discussion. In: 2nd ESICUP Meeting, Southampton, UK (April 2005)
21. Zhang, D., Kang, Y., Deng, A.: A new heuristic recursive algorithm for the strip rectangular packing problem. Computers and Operations Research 33(8), 2209–2217 (2006)

An Experimental Comparison of Different Heuristics for the Master Bay Plan Problem*

Daniela Ambrosino[1], Davide Anghinolfi[2], Massimo Paolucci[2], and Anna Sciomachen[1]

[1] Department of Economics and Quantitative Methods (DIEM), University of Genoa Via Vivaldi 5, 16126 Genova, Italy
{ambrosin,sciomach}@economia.unige.it

[2] Department of Communication, Computer and Systems Sciences (DIST), University of Genoa Via Opera Pia 13, 16145 Genova, Italy
paolucci@dist.unige.it, davide.anghinolfi@unige.it

Abstract. Different heuristics for the problem of determining stowage plans for containerships, that is the so called Master Bay Plan Problem (MBPP), are compared. The first approach is a tabu search (TS) heuristic and it has been recently presented in literature. Two new solution procedures are proposed in this paper: a fast simple constructive loading heuristic (LH) and an ant colony optimization (ACO) algorithm.

An extensive computational experimentation performed on both random and real size instances is reported and conclusions on the appropriateness of the tested approaches for the MBPP are drawn.

Keywords: heuristics, metaheuristics, stowage plans.

1 Introduction and Problem Definition

The stowage of containers on a ship, that is the so called master bay plan problem (MBPP), is faced daily by each terminal management. This problem can be defined as follows: given a set C of n containers of different types to load on a ship and a set S of m available locations within the ship, we have to determine the assignment of the containers to the ship locations in order to satisfy the given structural and operational constraints related to both the ship and the containers and to minimise the total loading time.

Set C is partitioned into two subsets, namely T and F, consisting of 20 and 40 feet (20' and 40') containers, respectively. Each location is addressed by three indices, i, j and k, representing its bay (i), row (j) and tier (k) position in the ship; let I, J and K be, respectively, the corresponding set of bays, rows and tiers available for the stowage. Moreover, let $E \subset I$ be the subset of even bays, that are used for stowing 40'containers,

* This work has been developed within the research project "Container import and export flow in terminal ports: decisional problems and efficiency analysis" PRIN 2007j494p3_005, Italy.

and $O\subset I$ the one of odd bays, that are used for stowing 20'containers. Tiers in the hold and on the deck are denoted by K^H and K^D, respectively. Finally, the locations having the same pair of bay and row identify a *stack*; let hence P be the set of all the available stacks in the ship.

In this paper the MBPP involves only the loading decisions at the first port without taking into account possible loading operations at the next ports of the ship route. We assume that the container handling operations are performed by quay cranes, which are positioned on the quay side of the ship. In [12], the problem of defining stowage plans is split into a two-step process concerning first the shipping line and then the terminal management; the authors provide for each step a review of the corresponding optimization models. A relevant literature update is provided in [11].

In this work we also assume that the ship is empty and only one yard crane is available for handling the containers at the quay. In spite of such simplifying assumptions, the resulting mathematical formulation is not simple. In fact, in order to model the MBPP we must deal with the assignment constraints (i.e., each container must be assigned to at most one location and each location must receive at most one container) and the knapsack constraint (i.e. the total weight of the containers loaded on the ship cannot exceed the ship capacity); furthermore, besides these basic combinatorial constraints, some constraints related to the size, weight and destination of the containers, together with others related to the ship stability, have to be taken into account. Stability conditions involve three different types of equilibrium: *horizontal, cross* and *vertical.*

MBPP is NP-Hard [5]. Some Integer Programming models for MBPP are proposed in [6] and [10]. Unfortunately these papers deal with simplified version of problem so that the proposed models are not suitable for real life large scale applications. MBPP is described in details in [1] where a 0/1 Integer Programming (IP) model is presented. In that model the decision variables are related to the assignment of containers to locations of the ship; the model is used for solving up to optimality only very small instances. Successively, in [3] a new 0/1 IP model is proposed where variables correspond to the assignment of ship locations to groups of containers, each one characterized by range of weight (e.g., low, medium or high), type and destination; using that model the authors solve larger instances, even if not always integer feasible solutions are found within a reasonable CPU time, that is some hours. In [3] MBPP is solved by using a three step approach. First a *bay assignment* procedure is performed to assign subsets of containers with the same destination to predefined subsets of bays; successively for each partition of the ship a single destination 0/1 IP model is considered, where the ship stability constraints are relaxed. Finally, possible infeasibilities of the global solution due to the violation of either cross or horizontal stability conditions are removed by means of a tabu search procedure. The tabu search algorithm also tries to improve the global solution, i.e., to reduce the total loading time. The main features of the tabu search presented in [3] is that it is based on seven classes of moves which combine three kinds of items that can be moved, that is a single container, a stack of containers and a bay, with three kinds of position exchanges, that is anterior-posterior location exchange, left side-right side exchange, and cross exchange.

By using such heuristic method it is possible to check and force feasibility up to small- medium sized instances, while for larger instances, it is not possible to use the 0/1 IP model for obtaining an initial solution. For this reason we propose a simple constructive procedure that is described in Section 2. Moreover, in this paper we present an Ant Colony Optimization (ACO) heuristic, described in Section 3, that is successfully applied to very large instances. Section 4 reports the performed experimental analysis; finally, conclusions are drawn in Section 5.

2 Simple Constructive Heuristic

In this section we describe a new solution method, that is a simple constructive loading heuristic (LH). LH determines a solution for MBPP that satisfies both the destination and the weight constraints in the following steps.

1. Let C'={ C_1 , C_2 ,... C_h,.... C_D} be a partition of C, where C_h denotes the set of containers having as destination port h. Split C_h according to the type of containers, that is 20' and 40', thus obtaining sets C_{hT} and C_{hF}, respectively.
2. Apply the bay partitioning procedure described before (see [3] for more details) to determine the subsets $I_h \subseteq I$ of bays assigned to destination h, $h = 1, ...,D$.
3. For each destination h, starting from the last one (D) back to 1, assign first containers belonging to C_{hT} and then containers belonging to C_{hF} as follows.
   3.1 Sort C_{hX} (where $X \in \{T,F\}$) in increasing order of weights.
   3.2 Repeat - Until $C_{hX} \neq \varnothing$ or $I_{hX} = \varnothing$
      1. Select $i \in I_{hX}$
      2. For k=1 to K and j=1 to J assign container $c \in C_{hX}$ to location (i, j, k), starting from the first container in the set (i.e., the heaviest one), then set $C_{hX} = C_{hX} \setminus \{c\}$ and $I_{hX} = I_{hX} \setminus \{i\}$
   3.3 If $C_{hX} \neq \varnothing$ and $I_{hX} = \varnothing$ try to locate the remaining containers without violating the destination and weight constraints as follows
      1. If $C_{hT} \neq \varnothing$, the remaining containers are possibly located above 20' containers having destination $h' > h$ without violating the weight constraints.
      2. If $C_{hF} \neq \varnothing$, the remaining containers are possibly located above 20' containers if they have the same destination; otherwise, either above 40' or 20' containers having destination $h' > h$, provided that the weight constraints are satisfied.

3 The Proposed ACO Approach for the MBPP

In this section we describe the main aspects of an ant colony optimization (ACO) approach for MBPP. ACO is a population-based metaheuristic which tries to emulate the successful behaviour of real ants cooperating to find shortest paths to food for solving combinatorial problems ([7], [9]). Real ants have an effective indirect way to communicate each other which is the most promising trail towards food: ants produce a natural essence, called pheromone, which they leave on the followed trail to food in order to mark it. The pheromone trail evaporates with time and it disappears on the paths left by the ants; however, it can be reinforced by the passage of further ants: thus, effective (i.e., shortest) paths leading to food are finally characterized by a strong pheromone track, such that shortest paths are followed by most ants. The ACO metaheuristic has been both the subject of theoretical studies and successfully applied to many combinatorial optimization problems.

We consider a set of m artificial ants. At each iteration the ants progressively fix the destination, type and class of weight for the available ship locations; thus they construct a solution by assigning the containers to the locations, accordingly. After that, a Local Search (LS) is executed starting from the best solution found in the current iteration, a global pheromone update phase takes place and the whole process is iterated. The algorithm terminates when a maximum number of iterations is reached.

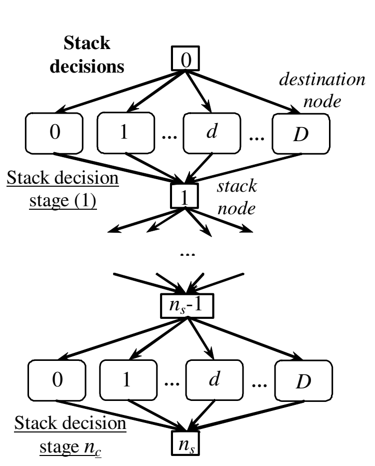
The proposed ACO is the adaptation of the algorithm introduced in [4]. Such an approach is inspired by the Ant Colony System (ACS) [8] and *Max-Min* Ant System (MMAS) [13] and it includes a new local and global pheromone update mechanism. For the sake of brevity, hereinafter we focus only on the modelling aspects highlighting how the ACO approach can be applied to MBPP. Details of the algorithm can be found in [4], while readers interested in a comprehensive presentation of the ACO metaheuristic can find a valuable and general reference in [7].

Let $CD=\{1,...,D\}$ denote the set of destinations for the containers in C ordered according to the ship route. At each iteration the ants take a sequence of decisions nested in two levels: *stack decision level* and *location decision level.* At the higher stack decision level the ants consider a stack at a time and fix the latest destination for the containers loaded in its locations (e.g., if d^{max} is the latest destination for a stack, then only containers bound for $1,...,d^{max}$ can be loaded in it). At the lower location decision level, the ants establish the type $t\in\{\mathrm{T}, \mathrm{F}\}$ and class of weight $g\in G$ for each location in the considered stack, finally assigning to it a container with a compatible destination, type and class of weight, which satisfy both destination and vertical equilibrium constraints.

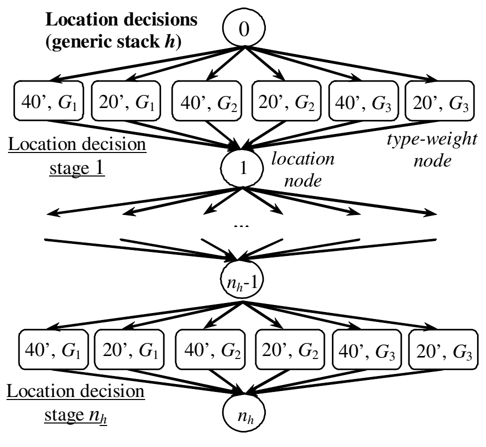
Such decisions correspond to the search of a path from a start node (the ant colony *nest*) to an end node (the *food*) in a *construction graph* structured in two nested levels, as depicted in Figure 1.

Figure 1.a shows the graph corresponding to the stack decision level; here the ants determine a path from the stack node 0 (the ant nest) to the stack node n_s (associated with the last considered stack), selecting at each stage one destination in $CD\cup\{0\}$, where 0 denotes that the stack is not assigned. The order according to which stacks are considered in the stack decision stages is heuristically fixed for both favouring the ship stability and reducing the loading time. In particular, we built a sorted list of rows in increasing order of the associated loading time t_j, $j\in J$ (computed as $t_j = \max_{k\in K} t_{jk}$, being t_{jk}, $j\in J$, $k\in K$, the loading time for row j and tier k) but alternating a left and a right row. Similarly, we build a sorted list of bays alternating a bay in the middle of the ship, one in the anterior and one in the posterior part with a procedure similar to the pre-assignment performed in [2]. Finally, the sequence of stacks defining the stack decision stages is produced by extracting the index i and the index j respectively from the sorted lists of bays and rows.

After a stack level decision the ants proceed considering the locations in the stack. Figure 1.b details the location decision level for a generic stack h. Also in this case the ants' decisions correspond to a path from the location node 0 to the location node n_h (associated with the last available location in stack h), determining at each stage the type and class of weight (here $G=\{G_1, G_2, G_3\}$, where G_1 stands for light, G_2 for medium and G_3 for heavy) for the location so that no vertical equilibrium constraint



(1.a) The stack decision level



(1.b) The location decision level

Fig. 1. The construction graph of the ACO algorithm

or type compatibility is violated. In the following, we denote by N_s, N_d, N_l and N_{tw} respectively the sets of stack, destination, locations and type-weight nodes. Note that (a) in the stack decision level of the construction graph the stage associated with a stack on deck immediately follows the corresponding stack in hold; (b) in the location decision level the location nodes are ordered from the lowest to the higher. Once both the latest destination and the type and class of weight for a location are fixed at a location decision stage, the ants actually assign to it a specific compatible container selected from the not yet loaded ones that do not violate any destination constraint (this is simply obtained sorting the available compatible containers in decreasing order of destination).

Note that n_s may be greater than the number of available stacks on the ship since the stacks with available locations may be reconsidered whenever there are not assigned containers. We must remark that the number of stages considered at the location decision level may vary since the ants' decisions at a stage force the available alternatives at the successive stages: in particular, a single type of container is allowed for a stack, and weight and destination constraints can prohibit certain assignment patterns. Therefore, assuming n_s=|P|, we have $2\cdot|I|\cdot|J|$ stack decision stages and at most $|K^H|$ and $|K^D|$ location decision stages for the locations associated with a stack in the hold and on the deck respectively, which in turn consist of $3\cdot|K^H|$ and $3\cdot|K^D|$ pairs of type-weight nodes.

The information about the ants state are (a) the sets of containers that remain to be located $C_R=\{(p, t, g, nc_{ptg})$: $p\in CD$, $t\in\{20', 40'\}$, $g\in G\}$, where nc_{pgt} is the number of containers to be loaded with destination p, type t and group of weight g; (b) the partial set of decisions for stow locations $C_A=\{(i, j, k, p, t, g)$: $i\in I$, $j\in J$, $k\in K$, $p\in D$, $t\in\{20',$ $40'\}$, $g\in G$ }. A pheromone trail is associated with a subset U of the arcs of the construction graph: U includes the arcs (s, d): $s\in N_s$, $d\in N_d$ connecting stack nodes to destination nodes at the stack decision level, and the arcs (l, z): $l \in N_l$, $z \in N_{tw}$ connecting location nodes to type-weigh nodes at the location decision level.

The ant solution construction process outlined so far is quite flexible as it allows to locate containers with different destinations in the same stack or bay. The selection of a destination node and type-weigh node at each stage of the two decision levels is performed in two steps similarly to the ACS: first an ant determines the node selection rule between *exploitation* and *exploration*, then it actually selects the node. The ant extracts a random uniform number $q \sim [0,1]$ and chooses exploitation if $q \le q_0$ (where $q_0 \in [0,1]$ is a fixed parameter), otherwise exploration. The *exploitation* rule at a generic decision node s such that $(s, h) \in U$ selects the next node h^* in a deterministic way as

$$h^* = \arg \max_{h:(s,h)\in U} \{\tau_e(s,h) \cdot [\eta(s,h)]^\beta\} \tag{1}$$

whereas the *exploration* rule according to a *selection probability* $\pi_e(s,h)$ computed as

$$\pi_e(s,h) = \frac{\tau_e(s,h) \cdot [\eta(s,h)]^\beta}{\sum\limits_{z:(s,z)\in U} \tau_e(s,z) \cdot [\eta(s,z)]^\beta} \tag{2}$$

The subset of arcs $(s, h) \in U$ identifies the solution components and $\tau_e(s,h)$ is the pheromone trail associated with the component (s, h) at iteration e. The pheromone trails represent the ACO learning device and provide a measure of the appropriateness of selecting a component during the construction of "good" solutions. Following the same approach in [4], the pheromone values assigned to $\tau_e(s,h)$ are independent of the objective function values associated with previously explored solutions including the component (s, h); pheromone values vary in an arbitrary range $[\tau_{Min}, \tau_{Max}]$, with $\tau_{Min} < \tau_{Max}$, which is fixed independently of the specific problem or instance considered (actually the algorithm behaviour does not depend on the choice of τ_{Max} and τ_{Min}). The pheromone trails are initialized as $\tau_0(s,h) = (\tau_{Max} + \tau_{Min})/2$ and their variation in the range $[\tau_{Min}, \tau_{Max}]$ during the exploration process, i.e., the ant colony learning mechanism, is controlled by the same global pheromone update rule proposed in [4] that allows a smooth variation of $\tau_e(s,h)$ within these bounds such that both extremes are asymptotically reached. The quantity $\eta(s,h)$, associated with the component (s, h), is a heuristic value that is used to direct the ants' selections during the first iterations when the pheromone trails are almost the same for all the components. We based the computation of $\eta(s,h)$ on a very simple rule which returns $\eta(s,h) = \eta_t(s,h) \cdot \eta_c(s,h)$, where $\eta_t(s,h)$ and $\eta_c(s,h)$ are the heuristic values relevant to the choice of the type of containers (i.e., the use of an even bay or the two paired odd bays) and of the class of weight for the location associated with h respectively. In order to minimize the loading time we favour the assignment of 20' containers to the locations in the rows closer to the berth side, fixing $\eta_{20}(s,h) = 2$ for the relevant nodes in the construction graph, as well as the assignment of the heaviest containers to the lowest tiers in the hold and on the deck, fixing $\eta_t(s,h) = 2$ for the relevant nodes, letting $\eta_t(s,h) = \eta_g(s,h) = 1$ in all the other cases. The quantity β in (1) and (2) is a parameter representing the importance of the heuristic value with respect to the pheromone trail in the ant selection rules.

Whenever an ant reaches the final state node n_s a tentative solution x for MBPP is build. Tentative solutions could not be feasible as some equilibrium constraints could be violated and some containers could be not loaded. Then, we compute the global cost for the solution as

$$Z(x) = M_s(\sigma_1(x) + \sigma_2(x)) + M_{nl}\mu(x) + L(x) \tag{3}$$

being $\mu(x)$ the number of not loaded containers and M_s and M_{nl} two penalties for the violation of stability constraints and for not loaded containers in x, respectively. After all ants a=1,..., m have determined a solution x_e^a at an iteration e, the best solution found in the iteration, i.e., $x_e^{best} = \arg\min_a Z(x_e^a)$, is determined and a LS step takes place. The purpose of this LS is that of perturbing x_e^{best} by means of a set of moves that change the locations of a subset of loaded containers in order to eliminate the possible violation of stability constraints and to improve the overall loading time. The LS procedure at each iteration performs the following sequence of three types of random moves, whose details are provided in [3]: Anterior-Posterior exchange of containers (APC); Left-Right side exchange of containers (LRC); Cross exchange of containers (CEC). As regards the solution feasibility, we should note that the LS is devoted only to recover violations of stability which, emerging from the solutions considered as a whole, are difficult to avoid during the ACO solution construction process. Thus, the objective function minimized by the LS disregards the $M_{nl}\mu(x)$ component since, leaving in this way the task of loading all the required containers to the ACO solution construction process. The LS adopts the first improvement move acceptance rule and it terminates after a fixed maximum number of iterations. However, whenever the overall ACO algorithm reaches a maximum number of non improving iterations, the LS maximum number of iterations is temporarily doubled until an improved solution is found.

Finally, the pheromone trails associated with the solution components included in the best solution found so far are reinforced while the other components are evaporated according to the global pheromone update rule introduced in [4], and the algorithm iterates.

4 Experimental Results

The proposed MBPP algorithms were coded in C++, using the commercial Cplex 9.0 as 0/1 IP solver, and tested on a 1.5GHz, Intel Celeron PC with 1Gb RAM. The tests were related to two containerships of different sizes. The first containership is the medium size European Senator, whose data have been provided by the SECH Terminal of Genova, Italy. The European Senator has a 2124 TEU capacity and is composed by 17 odd bays, 10 rows and 6 tiers in the hold and 21 odd bays, 12 rows and 5

tiers in the upper deck. Then we consider a larger containership with a capacity of 5632 TEUs.
The considered instances are characterized by different level of occupancy of the ship and different type of containers to load:

- the total number of containers (*TEU* and absolute number) ranges from 945 to 1800 (TEUs) and 715 to 1413 containers for instances with 2 or 3 destinations, and from 4920 to 5510 (TEUs) and 3800 to 4110 containers for instances with 4 or 5 destinations. The level of occupancy ranges from 50% to 97%;
- the percentage of 20' and 40' containers are 70% - 30% and 60% - 40%, respectively, for instances with 2/4 and 3/5 destinations;
- the percentage of containers for three groups of weight ranges from 30% - 60% - 10% to 40% - 35% - 15%;
- the partition of containers for each destination is 50% - 50% and 30 % 35% 25%, respectively, for instances with 2 or 3 destinations, whilst containers are uniformally distributed in case of 4 and 5 destinations.

The loading times depend on the row and tier and grow from left side rows to the right ones and from highest tiers on the deck to the lowest ones in the hold; times are expressed in 1/100 of minute and range from 120 to 330. We first tried to find a global optimal solution using the exact 0/1 IP model given in [3]. We fixed as termination conditions for the solver an absolute gap = $5 \cdot Nd$ minutes, where *Nd* is the number of destinations of the considered instance, and a maximum time limit of 1 hour, and we obtained the following results. Medium size ship:

- no integer solution was found in one hour of computation for all the 4 instances with 3 destinations;
- the solver stopped with a feasible solution after reaching the maximum time limit for 5 instances out of 10 with 2 destinations;
- the solver required on the average after 16m and 47s to terminate for the remaining 5 instances with 2 destinations.

Large size ship: no integer solution was found even extending the maximum computation time to two hours.

Note that finding an exact solution for the instances with more than two destinations was significantly hard; therefore, the need of an effective heuristic for medium and large containerships is apparent.

We tested the TS and the ACO approaches comparing the obtained results also with the ones produced by the exact 0/1 IP model for medium size instances and by the simple constructive LH followed by the TS (LH-TS) for large size instances.

We performed some preliminary tests to select suitable values for the TS and the ACO parameters, identifying the following configurations: for the TS we fixed tenure = 80, diversification after 30 non improving iterations, diversification length = 35 iterations, termination conditions corresponding to maximum number of iterations = 500 and maximum not improving iterations = 50. For the ACO we used 80 ants, the pheromone evaporation factor α=0.05, q_0=0.95, β=1, maximum number of non improving iterations = 8 and maximum number of iterations =1000. Since random choices are used in both TS and ACO, five independent runs were performed for each instance and the average results were considered.

Table 1 shows the results for the medium size test instances. We report in the first group of columns (*Exact 0/1 IP*) the results produced by the exact 0/1 IP model, followed by three groups of columns showing respectively the starting solutions of the TS obtained by solving the 0/1 IP model for the single destination MBPP (*SD-MBPP*), the final solutions yielded by the TS (*Tabu Search*) and the final solutions produced by the ACO. An absolute gap of 5m was fixed as the termination condition for the SD-MBPP 0/1 IP model and the loading times shown in the columns *Obj* are expressed as 1/100 of a minute. Note that *Time (a)* and *Time (b)* columns report the CPU times in seconds needed by the 0/1 IP solver respectively for the exact model and the single destination model, whereas the *Avg Time* columns for the TS and ACO show the total CPU times needed by the two heuristic approaches. The last table row finally shows the overall average CPU times for this set of instances.

First, we must remark that both the TS and the ACO approaches were able to find solutions satisfying both the cross and the horizontal stability conditions on every run for each instance. The average CPU times in Table 1 show that both heuristics were able to find feasible solutions in a fraction of the time needed by the exact 0/1 IP

Table 1. The results for the medium size ship

		Exact 0/1 IP (max time=1h, absolute gap=5×N_d m)		**SD-MBPP (absolute gap=5m)**				**Tabu Search (average over 5 runs)**		**ACO (average over 5 runs)**	
Ist.	**Nd**	**Obj (a)**	**Time (a)**	**Obj (b)**	**σ_1**	**σ_2**	**Time (b)**	**Avg obj**	**Avg Time**	**Avg obj**	**Avg Time**
1	2	139530	1300.8	150570	15	5380	68.4	142990	101.1	144162	142.0
2	2	149440	1606.4	155100	20	1065	11.6	151700	43.8	154918	158.6
3	2	161670	1876.7	168310	0	1225	7.4	164444	40.5	167780	180.1
4	2	178320	1613.3	186310	0	1435	21.8	180906	54.3	183336	174.0
5	2	219650	3698.6	225510	0	1115	17.6	222600	58.1	227486	189.2
6	2	197410	1999.8	203040	0	1555	18.0	200072	49.7	203566	191.6
7	2	213520	3698.5	219330	0	1235	12.1	216036	52.0	221572	154.9
8	2	244660	3698.4	252640	0	1495	21.3	247810	63.5	254168	141.8
9	2	248050	3295.0	253640	5	1010	212.3	250128	243.4	258578	137.9
10	2	293150	3704.7	293310	0	1960	17.0	293530	41.5	305228	115.7
11	3	-	3600.0	206590	20	5685	65.0	200140	119.6	204452	168.2
12	3	-	3600.0	220210	15	1300	7.5	217660	45.5	224822	181.2
13	3	-	3600.0	237050	0	1625	19.8	233608	65.9	239912	183.7
14	3	-	3600.0	278780	0	470	30.8	275964	66.7	285948	176.3
		Averages	2953.4				37.9		74.7		167.7

model to find the optimal; note that the CPU time consumed by the exact 0/1 model for obtaining for each instance the first suboptimal solution not worse than the average one produced by the TS is more than 10 times the CPU of the TS (in particular, an average of 846s compared to 74.8s). We compare the results of instances 1-10 in Table 1 by computing the percentage deviations of the solutions yield by the SD-MBPP model and the TS and ACO approaches from the ones found by the exact 0/1 IP model. The solutions obtained by the SD-MBPP model are worse but not very far from the ones yielded by the exact 0/1 IP model: the average percentage deviation of 3.41%, which corresponds to an average difference in the total loading time = 1h 2m 21s, was obtained with a very short average CPU time (37.9s); however, we must remark that the SD-MBPP was never able to satisfy the stability conditions (i.e., $\sigma_1, \sigma_2 > 0$), especially the horizontal one. The best results are apparently due to the TS approach that was only 1.33% worse (corresponding to an average difference in the total loading time = 24m 49s) than the exact 0/1 IP model that needed a very much longer computation. Compared to TS, the designed ACO algorithm presents worse performances with respect to average loading times and computation requirements.

For the instances with 3 destinations (11-14 of Table 1) we compared results obtained by TS and ACO, observing that the loading times produced by the ACO were worse on average of 2.94% than the ones obtained by the TS.

The results obtained for the large size instances are reported in Tables 2 and 3. As for large instances with more than 3 destinations, the ACO algorithm turned out to be the best one, and in some cases it was the only approach able to generate a solution. Note that for these instances the exact 0/1LP model never was able to determine a feasible integer solution. Since also the SD-MBPP model failed to find a feasible solution for some of the instances with a larger number of containers, we were not able to initialize the TS (in particular, is true for the instances 4 and 6 denoted with (*) in Table 2). For this reason we decided to implement the LH described in Section 2. Tables 2 and 3 show the results produced by the two steps of the TS, LH-TS and those produced by the ACO algorithm. In both tables, the first group of columns reports the characteristics of the initial solutions obtained, respectively, by the SD-MBPP model (*SD-MBPP*) in Table 2 and by the LH (*LH*) in Table 3; then, the second group of columns shows the final average results, respectively, for the TS approach (*TS*) in Table 2 and the LH-TS one (*LH-TS*) in Table 3. Note that, as the tabu search is only able to improve the stability and the loading time of a solution, the number of non loaded containers (*NLC*) in the initial solutions found by the SD-MBPP model and LH is not modified in the final TS and LH-TS solutions. The last column of Table 5 shows the non loaded containers (NLC) in the final ACO solution together with the loading time and the CPU time. Tables 2 and 3 report the stability violations (σ_1 and σ_2) only for the initial solutions as no stability violation was found in the final solutions; this is true also for the ACO solutions.The LT (i) and LT (f) columns show the loading times, respectively, for the initial and final solutions, and the LT %var column reports the percentage variation of the loading time in the final solutions with respect to the initial ones. Finally, Table 2 and 3 include three CPU times expressed in seconds: the one needed by the SD-MBPP model (CPU), the time required by the tabu search (CPU TS) and the overall (initialization plus search) cumulative time (CPU TOT). Note that the averages in the last row of Table 2 are computed without instances 4 and 6. The ACO needed less CPU time than the TS procedure, always being able to find a feasible solution. However, the ACO approach was also able to completely locate all the containers only in one case out of 11. The number of non loaded containers is lower than those obtained by TS.

Table 2. Tabu Search solutions for the large size tests

		SD-MBPP					TS (average over 5 runs)			
Ist.	**Nd**	**NLC**	**LT (i)**	**σ_1**	**σ_2**	**CPU**	**LT (f)**	**LT %var**	**CPU TS**	**CPU TOT**
1	4	5	868400	0	40	14400.0	868388	0.00	25.9	14425.9
2	4	0	852590	0	735	14400.0	853002	0.05	100.4	14500.4
3	4	100	850070	0	335	14400.0	850194	0.01	33.0	14500.4
4(*)	4	84	896760	-	-	-	-	-	-	-
5	4	92	906310	5	300	11099.3	906288	0.00	22.3	11121.7
6(*)	4	85	915329	-	-	-	-	-	-	-
7	5	112	918920	20	425	7325.2	918996	0.01	48.5	7373.7
8	5	47	901750	0	270	4181.7	901760	0.00	30.5	4212.2
9	5	194	733240	35	150	10187.8	732640	-0.08	37.4	10225.1
10	5	213	727830	20	85	14953.3	727510	-0.04	40.2	14993.5
11	5	91	842650	5	60	157.3	842616	0.00	25.3	182.6
Avg.		95	844640			10122.7	844599		40.38	10163.2

Table 3. LH-TS and ACO solutions for the large size tests

	LH					LH-TS				ACO		
ist	**NLC**	**LT (i)**	**σ_1**	**σ_2**	**CPU**	**LT (f)**	**LT %var**	**CPU TS**	**CPU TOT**	**NLC**	**LT**	**CPU**
1	240	897400	140	175	10	833640	-7.10	91.99	102.0	102.0	879162	154.8
2	188	896860	50	425	14	841420	-6.18	144.8	158.8	77.4	896676	123.4
3	424	855200	140	435	10	792606	-7.32	94.7	104.7	321.0	851946	100.9
4	106	972000	90	695	12	926130	-4.72	92.0	104.0	55.6	964970	73.1
5	44	991190	395	4080	15	969842	-2.15	80.5	95.5	0.0	986150	195.9
6	95	993200	25	4100	12	965824	-2.76	85.5	97.5	65.2	978940	130.1
7	100	996540	360	2975	14	946722	-5.00	160.8	174.8	19.6	999440	201.4
8	90	966160	230	515	10	935508	-3.17	94.7	104.7	1.4	979046	109.1
9	195	820640	80	1425	12	765732	-6.69	107.3	119.3	162.6	787590	166.7
10	219	815170	130	2315	11	758280	-6.98	107.5	118.5	152.6	799750	169.9
11	123	907340	625	660	10	882852	-2.70	95.6	105.6	40.0	918210	78.7
Avg	165.8	919245			11.8	874414.8		105.0	116.8	90.67	912898	139.4

Finally, comparing the results obtained by applying the proposed approaches to large size instances, in term of percentage number of non loaded containers (NLC) and CPU time, we noted that none of the proposed approaches was able to load on board all containers: as the large ship test instances were randomly generated with occupancy level between about 87% and 97% to stress the solution capacity of the compared algorithms, we cannot claim that a solution where all the containers are loaded exists for this benchmark. What we can observe for the large size tests is that all the proposed methods produced feasible solutions in terms of stability conditions, and the ACO approach behaved better since it required lower CPU time and produced a lower average NLC.

5 Conclusions

In this paper different solution methods for the NP-hard MBPP are presented and compared. The results obtained from an extensive computational experimentation, based on both real size instances and random ones, enable us to derive two main conclusions. First, for very large size instances the ACO approach is recommended among the proposed ones. Second, the performance of the TS based approaches is strongly depending on the quality of the initial solution. In particular, for medium size instances, where it is possible to apply the TS to a good initial solution, the resulting final solutions got by the TS are very good; this is not the case for large size instances.

Starting from the results discussed in this paper, the authors intend to extend their analysis to the multi-port MBPP. Finally, in this future development the assumption about the availability of only one quay crane will be removed.

References

1. Ambrosino, D., Sciomachen, A., Tanfani, E.: Stowing a containership: The Master Bay Plan problem. Transportation Research 38, 81–99 (2004)
2. Ambrosino, D., Sciomachen, A., Tanfani, E.: A decomposition heuristics for the container ship stowage problem. Journal of Heuristics 12, 211–233 (2006)
3. Ambrosino, D., Anghinolfi, D., Paolucci, M., Sciomachen, A.: A new three-step heuristic for the master bay plan problem. Maritime Economics & Logistics, Special issue on OR models in Maritime Transport and Freight Logistics 11(1), 98–120 (2009)
4. Anghinolfi, D., Paolucci, M.: A new ant colony optimization approach for the single machine total weighted tardiness scheduling problem. International Journal of Operations Research 5(1), 1–17 (2008)
5. Avriel, M., Penn, M., Shpirer, N.: Container ship stowage problem: complexity and connection to the colouring of circle graphs. Discrete Applied Mathematics 103, 271–279 (2000)
6. Chen, C.S., Lee, S.M., Shen, Q.S.: An analytical model for the container loading problem. European Journal of Operation Research 80(1), 68–76 (1995)
7. Dorigo, M., Blum, C.: Ant colony optimization theory: A survey. Theoretical Computer Science 344, 243–278 (2005)
8. Dorigo, M., Gambardella, L.M.: Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation 1, 53–66 (1997)
9. Dorigo, M., Stützle, T.: The ant colony optimization metaheuristics: algorithms, applications and advances. In: Glover, F., Kochenberger, G. (eds.) Handbooks of metaheuristics. Int. Series in Operations Research & Man. Science, vol. 57, pp. 252–285. Kluwer, Dordrecht (2002)
10. Imai, A., Nishimura, E., Papadimitriu, S., Sasaki, K.: The containership loading problem. International Journal of Maritime Economics 4, 126–148 (2002)
11. Stahlbock, R., Voss, S.: Operations research at container terminal: a literature update. OR Spectrum 30, 1–52 (2008)
12. Steenken, D., Voss, S., Stahlbock, R.: Container terminal operation and Operations Research - a classification and literature review. OR Spectrum 26, 3–49 (2004)
13. Stützle, T., Hoos, H.H.: Max-min ant system. Future Generation Computer System 16, 889–914 (2000)

An Analysis of Heuristics for Vertex Colouring

Marco Chiarandini[1] and Thomas Stützle[2]

[1] University of Southern Denmark, Campusvej 55, Odense, Denmark
marco@imada.sdu.dk
[2] IRIDIA, Université Libre de Bruxelles, Av. Roosevelt 50, Brussels, Belgium
stuetzle@ulb.ac.be

Abstract. Several heuristics have been presented in the literature for finding a proper colouring of the vertices of a graph using the least number of colours. These heuristics are commonly compared on a set of graphs that served two DIMACS competitions. This set does not permit the statistical study of relations between algorithm performance and structural features of graphs. We generate a new set of random graphs controlling their structural features and advance the knowledge of heuristics for graph colouring. We maintain and make all algorithms described here publically available in order to facilitate future comparisons.

Keywords: graph coloring, heuristics, experimental analysis.

1 The Graph Colouring Problem

The graph-vertex colouring problem (GCP) is a central problem in graph theory [1]. It consists in finding an assignment of colours to vertices of a graph in such a way that no adjacent vertices receive the same colour. Graph colouring problems arise in many real life applications like register allocation, air traffic flow management, frequency assignment, light wavelengths assignment in optical networks, and timetabling [4].

In the GCP, one is given an undirected graph $G = (V, E)$, with V being the set of $|V| = n$ vertices and E being the set of $|E| = m$ edges. A *k-colouring* of G is a mapping $\phi : V \mapsto \Gamma$, where $\Gamma = \{1, 2, \ldots, k\}$ is a set of $|\Gamma| = k$ integers, representing the colours. A k-colouring is *proper* if for all $[u, v] \in E$ it holds that $\varphi(u) \neq \varphi(v)$; otherwise it is *improper*. If for some $[u, v] \in E$ it is $\varphi(u) = \varphi(v)$, the vertices u and v are said to be *in conflict*. A k-colouring can also be seen as a partitioning of the set of vertices into k disjoint sets, called *colour classes*. In the decision version of the GCP, also called the *(vertex) k-colouring problem*, we are asked whether for some given k a proper k-colouring exists. In the optimisation version, we are asked for the smallest number k, called the *chromatic number* $\chi(G)$, for which a proper k-colouring exists. For general graphs, the decision version of the GCP problem is NP-complete.

In this work, we consider algorithms for solving the optimisation version of the GCP on general graphs. This version can be algorithmically approached by solving a sequence of k-colouring problems: an initial value of k is considered and each time a proper k-colouring is found, the value of k is decreased by one. The chromatic number is found when for some k the answer to the decision version is no. In this case, $\chi(G) = k + 1$. If a proper colouring cannot be found but no proof of its non-existence is given, as it is typically the case with heuristic algorithms, $k + 1$ is an upper bound on the chromatic number. For reviews of heuristic approaches to the GCP we refer to [3,4,9,15].

New heuristic and exact algorithms are commonly tested on a set of graphs maintained by M. Trick [19] and used for the two DIMCAS challenges on the GCP in 1996 and 2002 [12,13]. The results of algorithm comparisons, above all those concerning heuristics, are often hard to interpret. Apparently, in most articles on the GCP published in major journals [4], a newly presented heuristic outperforms some others on some instances. Nevertheless, generally, no insight is given on which instance features are separators of algorithm performance. Culberson et al. [7] provided an interesting analysis about what makes a graph hard to colour. The aim of our research is to exploit the insights on instance hardness from Culberson's work, to extend the analysis to some of the best known and best performing algorithms for the GCP, and to give indications on which algorithms perform best for specific classes of graphs. In this way we hope to advance both the understanding of algorithm behaviour and the dependence of algorithmic performance on structural properties of graphs. We use Culberson's generator [7] to produce new random graphs. We make the code of most of the algorithms studied available in an online compendium `http://www.imada.sdu.dk/~marco/gcp-study`. All experiments were run on a 2 GHz AMD Athlon MP 2400+ Processor with 256 KB cache and 1 GB of RAM.

2 Instance Generation

Culberson's random generator [7] allows us to create families of graphs of different structure and size. Structure may be induced by (i) imposing a graph to have a given number of independent sets (i.e., hiding a k-colouring) and by (ii) influencing the variability of the size of these independent sets. In our analysis, we control these two structural aspects of graphs together with the edge distribution. In future studies it might be possible to control also the limit of the size of an induced clique or the girth of the graph.

Hidden colour classes and variability of their size. In its *smooth k-colourable* modality, the generator controls the variability of the size of the colour classes. A graph is generated by assigning each vertex to the independent set with label $\lfloor kx(ax + 1 - a)\rfloor$, where $a \in [0, 1]$ is a parameter and x is a random number from the interval $[0, 1)$. For $a = 0$, the size of the independent sets tends to be nearly equal and the graph be quasi equi-partite, while for $a = 1$ the size tends to vary considerably. This structural feature of the graph was shown to be relevant for understanding differences in the instance hardness for algorithms [7]. We generated graphs using `variability` $\in \{0, 1, \texttt{no}\}$, where 0 and 1 are the values assigned to a and `no` indicates that no hidden colouring and biased size of colour classes is enforced.

Edge distribution. We considered the following three types. ***(i) Uniform graphs***. An edge between a pair of vertices (u, v) is included with a fixed probability p. These graphs are denoted by G_{np}, and by G_{knp} if a k-colouring is hidden. ***(ii) Geometric Graphs***. These graphs are created by disposing n points uniformly at random in a two dimensional square with coordinates $0 \leq x, y < 1$. Vertices in the graph correspond to the n points and an edge is associated to a pair of vertices (u, v) if their Euclidean distance, $d(u, v)$ is smaller or equal to a value r. We denote these graphs by U_{nr}, and by U_{knr} if a k-colouring is hidden. (The use of the letter G for uniform graphs and U for geometric graphs, instead of the viceversa, is common in the literature, see for example [11].) According to [11], geometric graphs have "inherent structure and clustering". ***(iii) Weight Biased Graphs***. These graphs are generated by first assigning vertices to independent sets. Then, a weight w is assigned to all $\binom{n}{2}$ pairs of vertices, except those in the same hidden independent set, which are assigned weight zero. Vertex pairs are selected as edges with a probability proportional to their weight. When an edge is added to the graph, weights are decreased in such a way that the formation of large cliques becomes unlikely. This procedure is controlled by two parameters, α and γ, that we set equal to 0 and 1, respectively, as recommended in [7]. The process terminates when either all weights are zero, or when $\lfloor p\binom{n}{2}\rfloor$ edges have been selected. As a side effect, the vertices may have small variability in vertex degree. These graphs are among the hardest to colour [7]. The edge density of the resulting graph, measured as the ratio between number of present edges and the number of edges in the corresponding complete graph, depends on the parameters p and w. In order to attain graphs of edge density $\{0.1, 0.5, 0.9\}$, we set p equal to $\{0.1, 0.5, 0.9\}$ and w equal to $\{2, 114, 404\}$ if $n = 500$ and equal to $\{4, 230, 804\}$ if $n = 1000$ (see [7] for details on the choice of these values). We denote these graphs by W_{np}, and by W_{knp} if a k-colouring is hidden.

We generated 1260 graphs of size 500 and 1000. We summarise the characteristics of the graphs in Table 1a, organised by five factors: size, type, edge density, variability and hidden colouring. These factors are parameters of the instances and, using a statistical terminology, stratify the experimental units in subgroups. In each subgroup, corresponding to specific combinations of the five factors, we have 5 graphs constructed with different random seeds in the generator. Table 1b gives aggregate statistics on the number of graphs considered. Note that due to random decisions in the generation of the edges, the value of k in a hidden colouring is only an upper bound to the chromatic number of the graph.

Selection of a time limit. For a comparison of heuristic algorithms, stopping criteria need to be fairly defined. We decided to use a classical local search algorithm as benchmark: our implementation of TabuCol by de Werra (1990) [20] that we refer to as TS_{N_1}. It uses the improved dynamic tabu list by [8]. In our implementation, this algorithm performs the evaluation of a neighbour in $O(1)$ by means of an auxiliary matrix that stores delta values and that can be updated in $O(n)$ by a simple scan of the vertices adjacent to the vertex that changed colour. In addition, we maintain both an adjacency matrix and an adjacency list for the representation of the graph, using either of the two at

Table 1. Table (a) shows how the 1260 graphs generated are distributed over the features. The number of graphs of size 500 is 520 and those of size 1000 is 740. Table (b) and (c) show the number of graphs per class given by size, edge density and graph type. Table (d) gives the time limits in seconds for the instances differentiated by size (rows) and density (columns).

Size	Type	Density	Variability	Hidden colouring	Tot. graphs
1000	G	0.1	0	$\{5, 10, 20\}$	15
			1	$\{5, 10, 20\}$	15
			no	–	10
		0.5	0	$\{10i : i = 2, \dots, 14\}$	75
			1	$\{10i : i = 2, \dots, 14\}$	75
			no	–	10
		0.9	0	$\{20, 50i : i = 1, \dots, 5\}$	30
			1	$\{20, 50i : i = 1, \dots, 5\}$	30
			no	–	10
	U	0.1	0	$\{10i : i = 2, \dots, 5\}$	20
			1	$\{10i : i = 2, \dots, 5\}$	20
			no	–	10
		0.5	0	$\{10i : i = 2, \dots, 20\}$	95
			1	$\{10i : i = 2, \dots, 20\}$	95
			no	–	10
		0.9	0	$\{100i : i = 1, \dots, 6\}$	30
			1	$\{100i : i = 1, \dots, 6\}$	30
			no	–	10
	W	0.1	0	$\{10, 20\}$	10
			1	$\{10, 20\}$	10
			no	–	10
		0.5	0	$\{10i : i = 2, \dots, 9\}$	40
			1	$\{10i : i = 2, \dots, 9\}$	40
			no	–	10
		0.9	0	$\{30, 90, 150, 220\}$	20
			1	$\{30, 90, 150, 220\}$	20
			no	–	10

(a)

$n = 500$	$\rho = 0.1$	$\rho = 0.5$	$\rho = 0.9$
G	60	90	90
U	60	150	150
W	30	75	60

(b)

$n = 1000$	$\rho = 0.1$	$\rho = 0.5$	$\rho = 0.9$
G	90	180	180
U	120	150	180
W	30	120	60

(c)

	$\rho = 0.1$	$\rho = 0.5$	$\rho = 0.9$
$n = 500$	60	120	180
$n = 1000$	155	465	720

(d)

convenience. Similarly, we represent a colouring both as a mapping by means of an array and as a collection of colour classes, each one being implemented by a binary search tree. The other algorithms we implement use, as far as possible, the same data structures.

A common stopping criterion for TS_{N_1} is after $I_{\max} = 10^4 n$ iterations [8], which is also the one we used. In fact, we found this stopping criterion to be large enough so that the algorithm obtains limiting behaviour and that further improvements become unlikely. In particular, we found that setting the termination criterion to $I_{\max}$ corresponds to missing 26% of cases where a better colouring could still be found. This empirical probability drops below 3% after $10 \times I_{\max}$ and below 0.02% after $50 I_{\max}$. However, $10 \times I_{\max}$ implies an increase of a factor of 10 in computation time. For more detailed results on this analysis we refer to `http://www.imada.sdu.dk/~marco/gcp-study`. The actual time limits used for the heuristic algorithms are given in Table 1d; we divided them according to edge density since this property has a strong impact on computation time. For each density we took the median time observed.

3 Experimental Analysis

Exact Algorithms. The famous Brelaz's backtracking search algorithm with forward checking [2,17], here denoted as Ex-DSATUR, was shown to be substantially competitive with a more involved column generation approach [16].

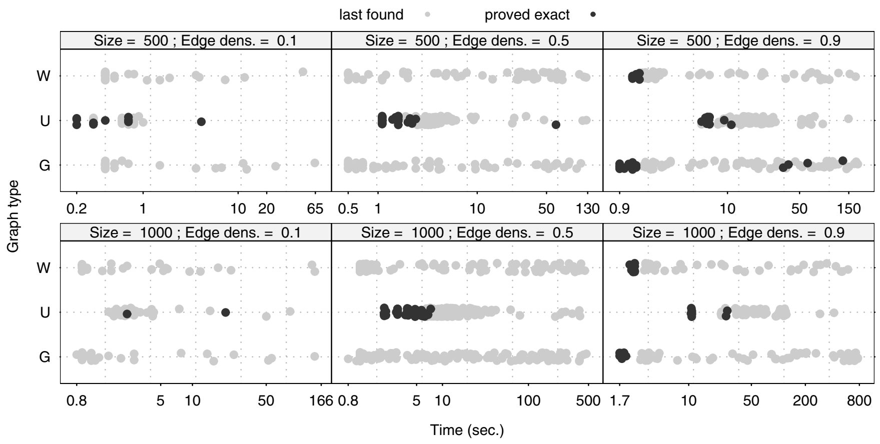


Fig. 1. The figure represents the exit status of Ex-DSATUR on the 1260 random graphs when the computation is truncated at the time limits of Table 1d. For each graph, the corresponding point indicates the computation time at which a last proper colouring was found. If the solution is proved optimal, the point is black, otherwise it is grey. A jitter effect is added to the y coordinates of the data to make points distinguishable. The plots are logarithmic in the x-axis.

An implementation of Ex-DSATUR by Mehrotra and Trick [16] including clique pre-colouring can be found online [18]. We tested this implementation of the algorithm on the 1260 instances using the same time limit as reported in Table 1d. In Figure 1, we summarise the behaviour of Ex-DSATUR. The plots give an account of the time at which the best proper colouring is found. If the colouring is proved optimal, the corresponding point is plotted in black.

The first observation is that an exact solution is found either quickly (in less than 10 sec.), or it is hardly found afterwards. Despite the large size, some graphs are easily solvable by Ex-DSATUR. Chances to find solvable graphs are highest for Geometric graphs and in particular for edge density equal to 0.5. Graphs of type G and W are solvable exactly only for the high edge density and graphs become harder to be solved exactly if the number of hidden colours increases. A closer look at the data reveals that the graphs solved to optimality have $\omega(G)$, the size of the largest clique, very close to $\chi(G)$. A graph for which $\omega(G) = \chi(G)$ is called *perfect* and can be recognised and coloured in polynomial time [5,10]. For perfect and quasi-perfect graphs, the large clique found heuristically by Ex-DSATUR can be used to prune effectively the search tree.

Construction heuristics. Construction heuristics are fast single pass heuristics that construct a proper colouring and finish. We compared the performance in terms of run-time and solution quality of the two most famous ones: DSATUR [2] and RLF [14], which we implemented to run in $O(n(n+k)+m)$ and $O(n^2+km)$, respectively. We include in the analysis also a random order greedy heuristic, ROS, that runs in $O(nk + m)$. Note that for dense graphs, like those in this work, and assuming $k = O(n)$, the complexity of these algorithms becomes $O(n^2)$, $O(n^3)$ and $O(n^2)$, respectively.

We run the three algorithms once on each instance. For run-times, we compute the 95% confidence intervals of the mean value by means of the Tukey test for all pairwise comparisons. For quality, we rank the results in terms of colours within each instance and we compute the 95% confidence intervals of the mean rank value by means of the Friedman test for unreplicated two-way designs with Bonferrori correction [6].

On Uniform and Weight Biased graphs, RLF is clearly, and by a large margin, the best algorithm in terms of solution quality. We do not show the results. This indication is somehow interesting because we might expect to see differences between RLF and DSATUR varying the size of the hidden colour classes, but this is not the case. On Geometric graphs, instead, differences in solution quality with respect to DSATUR are not always statistically significant. In Figure 2, we report the all-pairwise comparisons with an indication of the computation time on the y-axis. Only Geometric graphs of size 1000 are used to generate the plot because differences in computation time are more pronounced in this setting and no difference in solution quality due to size was observed. Interestingly, there are classes of graphs in which DSATUR performs better than RLF, but a clear pattern does not arise. Since Geometric graphs have a clique number close to the chromatic number, this result may indicate that coloring looking at the saturation number of vertices (DSATUR) may be a sufficiently good strategy when cliques are relevant for the final result. In addition, we observe that the performance of DSATUR with respect to RLF improves as edge density increases. Finally, in terms of run-time, RLF exhibits a much stronger dependency on edge density, as captured by the asymptotic analysis. This effect can be evinced in Figure 2 observing the row-wise growth of computation time for RLF. The same pattern is present also on Uniform and Weight Biased graphs thus trading off in those graphs with the outperformance in quality terms.

Stochastic Local Search algorithms. We implemented heuristic algorithms based on stochastic local search (SLS) concepts. Beside TS_{N_1}, we include $\mathsf{TS}_{\mathsf{VLSN}}$, a tabu search on a very large scale neighbourhood [3], SA_{N_6}, the simulated annealing by Johnson et al. (1991) based on Kempe chains [11], $\mathsf{MC\text{-}TS}_{N_1}$, a min-conflict heuristic [3], Nov^+, inspired by Novelty algorithms for satisfiability problems [3], HEA, the hybrid evolutionary algorithm by Galiner and Hao (1999) [8], GLS, a guided local search algorithm [3] and ILS, an iterated local search algorithm [3]. In addition, we include our reimplementation of XRLF [11], a parametrised version of RLF that uses an exact colouring when the set of remaining colours is sufficiently small. In order to reduce the variance of the results and to isolate the effect of these heuristics, we start all them, except XRLF, from the same initial solution produced by RLF.

The influence of hidden colours. For several generated graphs, the SLS algorithms were able to easily find much better colorings than the hidden colouring. In a few cases the difference reached 200 colours while, with few exceptions,

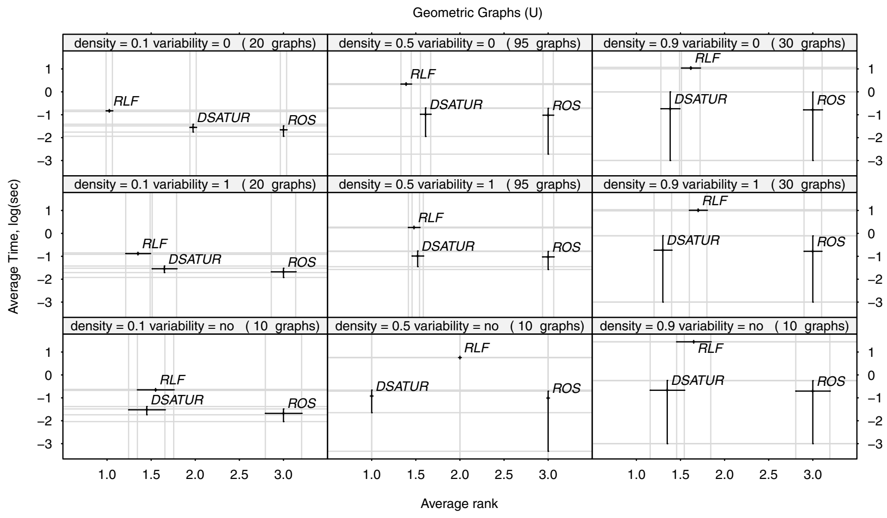


Fig. 2. All-pairwise comparisons in a run-time versus solution quality plot for construction heuristics on Geometric Graphs. Two algorithms are statistically significantly different in one of the two criteria if their intervals in the corresponding axis do not overlap. A base-10 logarithm transformation is applied on the time axis.

the best SLS algorithms always reach the hidden upper bound. Hence, the attempt to use hidden colourings to measure hardness of an instance is somehow a failure, due to the difficulty of hiding colourings. One interesting phenomenon is, however, worth reporting. On two of the 38 classes, we observed the phenomenon depicted in Figure 3. The plot shows the error relative to the hidden upper bound attained by the SLS heuristics on graphs of size 1000, density 0.5, variability 0, and type G and W. Since a hidden colouring exists, it should be possible to reach it on all those graphs, i.e., all curves should reach a zero error or approximate it. Nevertheless, the curves peak for some values of the hidden colourings, indicating that there are some values of k for which the instances are much harder to solve than usual. The region, which exhibits this phenomenon, arises at about 80 hidden colours for graphs of type G and between 70 and 80 colours for graphs of type W. It is interesting to note that this phenomenon affects also algorithms such as SA_{N_6} and XRLF, which do not solve sequences of k-colouring problems. The same effect was not observed in the corresponding graph classes of size 500 and it is unclear whether it exists for graphs of sizes not considered here.

Interaction between algorithms and graph features. We visualize the interactions between algorithms and graph features in Figure 4. Lines are added to emphasize the trend and not to interpolate data. The less parallel these lines are the stronger the interaction effect is. The strongest effect seems to be produced by edge density. Note that we omit hidden colours from now since this feature cannot be easily recognised a priori.

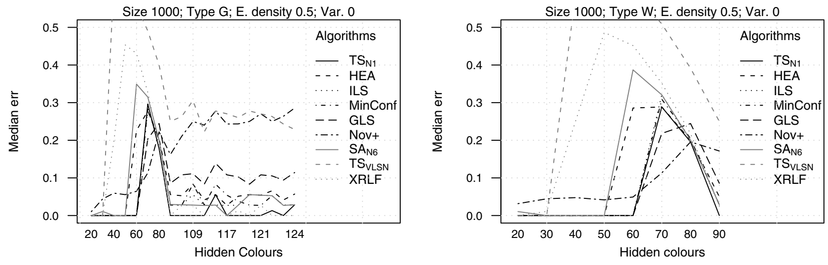


Fig. 3. The figure depicts the algorithm performance (expressed in terms of relative error computed over the hidden upper bound) and the number of hidden colours. A hidden colour indicates that a proper colouring with that or a lower number of colours certainly exists.

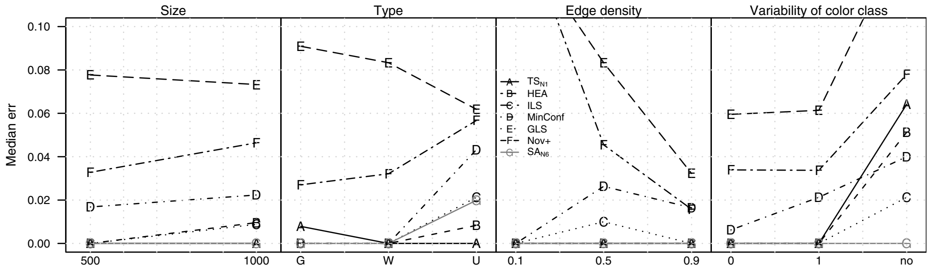


Fig. 4. Interaction plots between algorithms and stratification variables. The worst algorithms, XRLF and $\mathsf{TS}_{\mathsf{VLSN}}$, are removed in order to gain a clearer insight on the high performing algorithms.

Nonparametric analysis. We run all heuristics once on each graph until the corresponding time limit was exceeded. As a first step, we intended a parametric analysis fitting a linear model on a relative error transformation of the response. The analysis hinted at a strong significance of the interactions summarized in Figure 4. However, the necessary assumptions for a parametric analysis were found to be violated and therefore we proceeded with a nonparametric analysis by means of a rank transformation as mentioned above. An analysis based on permutation tests was also considered and results were in line with those presented below. Our preference for the rank-based analysis is due to its higher power. Due to the situation depicted in Figure 4, we separated the analysis into graph classes determined by the features: type, edge density, and variability. Graphs of size 500 and 1000 were instead aggregated, since the influence on the relative order of the algorithms is negligible. The results shown by means of confidence intervals on ranks derived by the Friedman test are reported in Figure 5. The following are the main conclusions from the analysis.

(i) On Uniform graphs, TS_{N_1} is the best algorithm on four scenarios and, except for one scenario, no other algorithm does significantly better. Therefore,

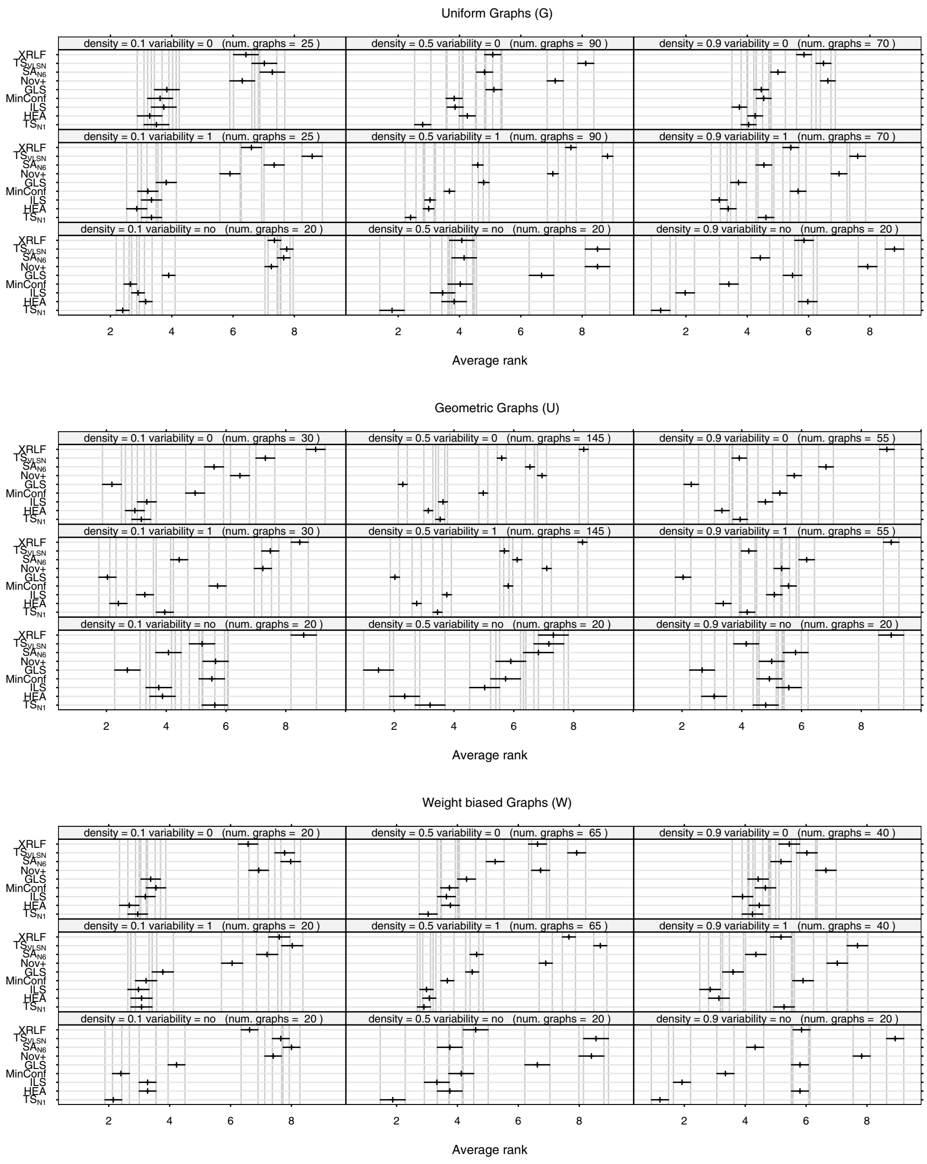


Fig. 5. The diagrams show average ranks and corresponding confidence intervals for the 9 algorithms considered. The analysis is divided into three main classes of graphs: Uniform graphs, Geometric graphs and Weight Biased graphs. Inside each class, subclasses are determined by the combinations of the stratification variables: edge density and independent set variability. Graphs of size 500 and 1000 are aggregated and the number of instances considered is reported in the strip text of the plots. Two algorithms are statistically significantly different if their intervals do not overlap.

we indicate TS_{N_1} as the preferable method for this class. It appears particularly powerful with graphs of density 0.5. The second best is ILS, which outperforms TS_{N_1} in the scenarios with density 0.9 and variability 1.

(ii) On the Weight Biased graphs, results are very similar to Uniform graphs. The best algorithm is TS_{N_1}, which is significantly the best in 3 scenarios, while ILS is the second best and again outperforms TS_{N_1} on graphs with density 0.9 and variability 1.

(iii) On the Geometric graphs, the overall best algorithm is clearly GLS. Differences from the second best are significant in 6 out of 9 scenarios, while in the other 3 scenarios it is not significantly dominated. The second best algorithm is HEA. The variability of the hidden colour classes and the edge density have a weak impact on these graphs, at least for the performance of the best algorithms, and results could be aggregated in a unique diagram, in which case GLS is significantly the best algorithm.

(iv) Scenarios with variability 0 and 1 exhibit very similar performance of the algorithms suggesting that this effect is not relevant.

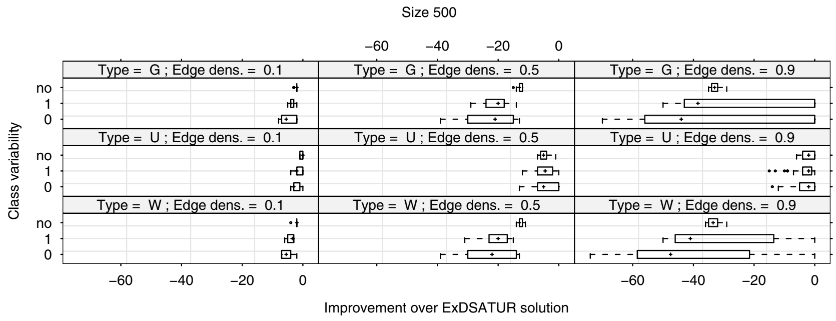
Improvement over Ex-DSATUR*.* Ex-DSATUR is an exact algorithm with exponential worst case, but it can be stopped at any time and it returns a feasible solution. In Figure 6 we compare the approximate solutions returned by SLS algorithms and Ex-DSATUR after the same run-time as that for TS_{N_1}. There is evidence that the SLS algorithms obtain colourings that are much better than those of Ex-DSATUR, reaching improvements by even more than 150 colours.

Improvement over RLF*.* The use of SLS heuristics gives a significant improvement over the initial solution of RLF. The results, reported in the online compendium, show that on Uniform and Weight Biased graphs the improvement increases considerably with size and edge density. In the case of Weight Biased graphs, the improvement can reach 105 colours. On Geometric graphs, the improvement is smaller, above all on graphs with edge density 0.9; a possible explanation is that RLF finds near-optimal solutions on these graphs.

4 Discussion

We reported our computational experience on algorithms for colouring large, general graphs. We considered 1260 graphs that were constructed by controlling several structural parameters in order to gain a better insight into the relationship between algorithm performance and graph features. We took into account graph size, edge density, type of graph, and characteristics of colour classes. These are features that may be recognised a priori in practical contexts.

We showed that a straightforward backtracking algorithm can solve instances of even 1000 vertices if these have the favourable property of being nearly *perfect*. The simple and fast construction heuristic RLF is considerably better than DSATUR in terms of solution quality, although its running time is more strongly affected by the number of edges in the graph. If a faster heuristic is needed and the graph is Geometric, then probably DSATUR is the best choice. If we can



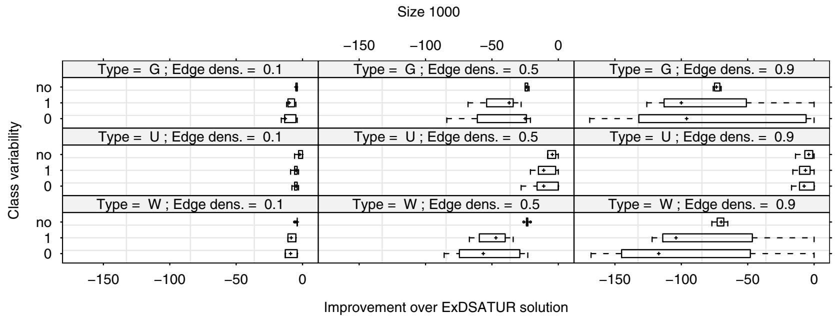


Fig. 6. Box-plots of differences for each graph class between the best solutions found by the SLS algorithms and the solution produced by the Ex-DSATUR heuristic

trade computation time for solutions quality, then SLS algorithms can lead to a large improvement. Our comparison includes some well known, high performing SLS algorithms for GCP. On the *Uniform and Weight Biased random graphs*, TS_{N_1} is the best algorithm with graphs of density 0.5 and it is not dominated at edge density 0.1 and 0.9. This result is important because it indicates TS_{N_1} as a relevant benchmark to be used when new algorithms are introduced and therefore we make its implementation available online. On *Geometric random graphs*, GLS is clearly the best algorithm. This class of graphs has the clique number very close to the chromatic number and the result leads us to conjecture that GLS works well for graphs with this property. Weight Biased graphs are, instead, graphs in which large cliques are avoided and indeed GLS is performing poorly.

References

1. Bondy, J., Murty, U.: Graph Theory. Graduate Texts in Mathematics, vol. 244. Springer, Heidelberg (2008)
2. Brélaz, D.: New methods to color the vertices of a graph. Communications of the ACM 22(4), 251–256 (1979)
3. Chiarandini, M., Dumitrescu, I., Stützle, T.: Stochastic local search algorithms for the graph colouring problem. In: Gonzalez, T.F. (ed.) Handbook of Approximation Algorithms and Metaheuristics, pp. 63-1–63-17. Chapman & Hall/CRC, Boca Raton (2007)

4. Chiarandini, M.: Bibliography on graph-vertex coloring (2010), http://www.imada.sdu.dk/~marco/gcp
5. Chudnovsky, M., Cornuéjols, G., Liu, X., Seymour, P., Vušković, K.: Recognizing Berge graphs. Combinatorica 25(2), 143–186 (2005)
6. Conover, W.: Practical Nonparametric Statistics, 3rd edn. John Wiley & Sons, New York (1999)
7. Culberson, J., Beacham, A., Papp, D.: Hiding our colors. In: Proceedings of the CP 1995 Workshop on Studying and Solving Really Hard Problems, Cassis, France, September 1995, pp. 31–42 (1995)
8. Galinier, P., Hao, J.: Hybrid evolutionary algorithms for graph coloring. Journal of Combinatorial Optimization 3(4), 379–397 (1999)
9. Galinier, P., Hertz, A.: A survey of local search methods for graph coloring. Computers & Operations Research 33, 2547–2562 (2006)
10. Grötschel, M., Lovász, L., Schrijver, A.: The ellipsoid method and its consequences in combinatorial optimization. Combinatorica 1(2), 169–197 (1981)
11. Johnson, D.S., Aragon, C.R., McGeoch, L.A., Schevon, C.: Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. Operations Research 39(3), 378–406 (1991)
12. Johnson, D.S., Mehrotra, A., Trick, M.A.: Special issue on computational methods for graph coloring and its generalizations. Discrete Applied Mathematics 156(2), 145–146 (2008)
13. Johnson, D.S., Trick, M. (eds.): Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993. DIMACS Series in DMTCS, vol. 26. American Mathematical Society, Providence (1996)
14. Leighton, F.T.: A graph coloring algorithm for large scheduling problems. Journal of Research of the National Bureau of Standards 84(6), 489–506 (1979)
15. Malaguti, E., Toth, P.: A survey on vertex coloring problems. International Transactions in Operational Research, 1–34 (2009)
16. Mehrotra, A., Trick, M.: A column generation approach for graph coloring. INFORMS Journal on Computing 8(4), 344–354 (1996)
17. Peemöller, J.: A correction to Brelaz's modification of Brown's coloring algorithm. Communications of the ACM 26(8), 595–597 (1983)
18. Trick, M.: Network resources for coloring a graph (1994), http://mat.gsia.cmu.edu/COLOR/color.html (last visited: February 2005)
19. Trick, M.: ROIS: Registry for optimization instances and solutions (2009), http://mat.tepper.cmu.edu/ROIS/ (last visited: December 2009)
20. de Werra, D.: Heuristics for graph coloring. Computing Supplement 7, 191–208 (1990)

Automatic Tuning of GRASP with Path-Relinking Heuristics with a Biased Random-Key Genetic Algorithm

Paola Festa[1], José F. Gonçalves[2], Mauricio G.C. Resende[3], and Ricardo M.A. Silva[4]

[1] University of Napoli "Federico II", Napoli, Italy
paola.festa@unina.it
[2] Universidade do Porto, Porto, Portugal
jfgoncal@fep.up.pt
[3] AT&T Labs Research, Florham Park, NJ, USA
mgcr@research.att.com
[4] Universidade Federal de Lavras, Lavras, MG, Brazil
rmas@dcc.ufla.br

Abstract. GRASP with path-relinking (GRASP+PR) is a metaheuristic for finding optimal or near-optimal solutions of combinatorial optimization problems. This paper proposes a new automatic parameter tuning procedure for GRASP+PR heuristics based on a biased random-key genetic algorithm (BRKGA). Given a GRASP+PR heuristic with n input parameters, the tuning procedure makes use of a BRKGA in a first phase to explore the parameter space and set the parameters with which the GRASP+PR heuristic will run in a second phase. The procedure is illustrated with a GRASP+PR for the generalized quadratic assignment problem with $n = 30$ parameters. Computational results show that the resulting hybrid heuristic is robust.

1 Introduction

A commonly cited drawback of heuristics is the large number of parameters that need to be tuned for good performance. These parameters are not limited to those that are numerically valued but can also be logical parameters that determine, for example, which sub-modules are activated in the heuristic and which ones are not. Heuristic parameters can consequently run into the tens and even hundreds and tuning them can be a labor intensive activity. Furthermore, the performance of a heuristic depends on the instance being solved, so a tuned set of parameters obtained for one instance may not result in a good performing heuristic for another instance. When documenting a heuristic, a description of the tuning process is often left out and therefore it is often difficult to reproduce computational results. These are some of the factors that point to the need for an algorithmic approach to parameter tuning.

In this paper we propose an automatic tuning procedure for GRASP with path-relinking (GRASP+PR) heuristics. In the first phase of this two-phase solution strategy a biased random-key genetic algorithm searches the space of parameters for a set of values that results in a good performance of the heuristic. In the second phase, the GRASP+PR heuristic is run using the parameters found in the first phase. We illustrate this procedure on a GRASP+PR heuristic for the generalized quadratic assignment problem. This GRASP+PR has 30 tunable parameters. Computational results show that the two-phase approach results in a robust hybrid heuristic.

The paper is organized as follows. In Section 2 we briefly describe the genetic algorithm. In Section 3 we summarize the solution strategy implemented in the GRASP+PR heuristic. The two-phase strategy is described in Section 4. Finally, computational experiments are reported in Section 5.

2 Biased Random-Key Genetic Algorithms

Genetic algorithms with random keys, or *random-key genetic algorithms* (RKGA), were first introduced by Bean [1] for solving combinatorial optimization problems involving sequencing. In a RKGA, chromosomes are represented as vectors of randomly generated real numbers in the interval $[0, 1]$. A deterministic algorithm, called a *decoder*, takes as input a solution vector and associates with it a solution of the combinatorial optimization problem for which an objective value or fitness can be computed.

A RKGA evolves a population of random-key vectors over a number of iterations, called *generations*. The initial population is made up of p vectors of random-keys. Each component of the solution vector is generated independently at random in the real interval $[0, 1]$. After the fitness of each individual is computed by the decoder in generation k, the population is partitioned into two groups of individuals: a small group of $p_e = 0.3p$ *elite* individuals, i.e. those with the best fitness values, and the remaining set of $p - p_e = 0.7p$ *non-elite* individuals. To evolve the population, a new generation of individuals must be produced. All elite individual of the population of generation k are copied without modification to the population of generation $k + 1$. RKGAs implement mutation by introducing *mutants* into the population. A mutant is simply a vector of random keys generated in the same way that an element of the initial population is generated. At each generation, a small number ($p_m = 0.2p$) of mutants is introduced into the population. With the p_e elite individuals and the p_m mutants accounted for in population $k + 1$, $p - p_e - p_m$ additional individuals need to be produced to complete the p individuals that make up the new population. This is done by producing $p - p_e - p_m$ offspring through the process of mating or crossover.

Bean [1] selects two parents at random from the entire population to implement mating in a RKGA. A *biased random-key genetic algorithm* (BRKGA) [2], differs from a RKGA in the way parents are selected for mating. In a BRKGA, each element is generated combining one element selected at random from the elite partition in the current population and one from the non-elite partition.

Repetition in the selection of a mate is allowed and therefore an individual can produce more than one offspring in the same generation. *Parameterized uniform crossover* [3] is used to implement mating in BRKGAs. Let $\rho_e = 0.7$ be the probability that an offspring inherits the vector component of its elite parent. Let n denote the number of components in the solution vector of an individual. For $i = 1, \dots, n$, the i-th component $c(i)$ of the offspring vector c takes on the value of the i-th component $e(i)$ of the elite parent e with probability ρ_e and the value of the i-th component $\bar{e}(i)$ of the non-elite parent $\bar{e}$ with probability $1 - \rho_e$. When the next population is complete, i.e. when it has p individuals, fitness values are computed for all of the newly created random-key vectors and the population is partitioned into elite and non-elite individuals to start a new generation.

A BRKGA searches the solution space of the combinatorial optimization problem indirectly by searching the continuous n-dimensional hypercube, using the decoder to map solutions in the hypercube to solutions in the solution space of the combinatorial optimization problem where the fitness is evaluated.

3 GRASP with Path-Relinking for the Generalized Quadratic Assignment Problem

A GRASP [4,5] is a multi-start metaheuristic where at each iteration a greedy randomized solution is constructed to be used as a starting solution for local search. The best local minimum found over all GRASP iterations is output as the solution. See [6,7,8,9] for recent surveys of GRASP.

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 [10,11,12]. In GRASP with path-relinking (GRASP+PR) [13,14], 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.

Mateus, Resende, and Silva [15] propose a GRASP with path-relinking heuristic for the generalized quadratic assignment problem (GQAP). In the GQAP, we are given n facilities and m locations and want to feasibly assign each facility to a location. Each facility uses a portion of the capacity of a location and each location has a fixed amount of capacity to distribute among facilities. An assignment is feasible if each location has sufficient capacity to accommodate the demands of all facilities assigned to it. Given nonnegative flows between all pairs of facilities and nonnegative distances between all pairs of locations, the GQAP seeks a feasible assignment that minimizes the sum of products of flows and distances in addition to a linear assignment component.

Algorithm 1 shows pseudo-code for the GRASP+PR heuristic proposed in [15] for the GQAP. The algorithm takes as input the set N of facilities, the set M of locations, the flow matrix A, the distance matrix B, the assignment cost matrix C, the facility demands $q_i,\ i \in N$, and the location capacities $Q_j,\ j \in M$, and outputs an assignment vector π^* specifying the location of each facility in the best solution found.

After initializing the elite set P as empty in line 1, the GRASP+PR iterations are computed in lines 2 to 24 until a stopping criterion is satisfied. 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 (ρ is an input parameter), then if π' is feasible and sufficiently different from all other elite set solutions, π' is added to the elite set in line 22. To define the term *sufficiently different* more precisely, let $\Delta(\pi', \pi)$ be defined as the minimum number of facility to location swaps needed to transform π' into π or vice-verse. For a given level of difference δ (δ is an input parameter), we say 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 19 are computed.

The greedy randomized construction procedure is not guaranteed to generate a feasible solution. If the greedy randomized procedure returns an infeasible solution, a feasible solution π' is selected uniformly at random from the elite set in line 6 to be used as a surrogate for the greedy randomized solution. An approximate local search is applied using π' as a starting point in line 8, resulting in an approximate local minimum, which we denote by π'. Since elite solutions are made up of approximate local minima, then applying an approximate local search to an elite solution will, with high probability, result in a different approximate local minimum.

The approximate local search is not guaranteed to find an exact local minimum. Since π' is an approximate local minimum, the application of an approximate local search to it will, with high probability, result in a different approximate local minimum. Next, path-relinking is applied in line 10 between π' and an elite solution π^+, randomly chosen in line 9. Solution π^+ is selected with probability proportional to $\Delta(\pi', \pi^+)$. In line 11, the approximate local search is applied to π'. If the elite set is full (the maximum number of solutions in the elite set is an input parameter), 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 14 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 18 if $\pi' \not\approx P$.

We next summarize procedures `GreedyRandomized`, `ApproxLocalSearch`, and `PathRelinking`. These procedures are described in detail in Mateus, Resende, and Silva [15].

```
procedure GRASP+PR
  Data   : $N, M, A, B, C, q_i, Q_j$.
  Result : Solution $\pi^* \in \chi$.
1  $P \leftarrow \emptyset$;
2  while stopping criterion not satisfied do
3      $\pi' \leftarrow$ GreedyRandomized($\cdot$);
4      if elite set $P$ has at least $\rho$ elements then
5          if $\pi'$ is not feasible then
6              Randomly select a new solution $\pi' \in P$;
7          end
8          $\pi' \leftarrow$ ApproxLocalSearch($\pi'$);
9          Randomly select a solution $\pi^+ \in P$;
10         $\pi' \leftarrow$ PathRelinking($\pi', \pi^+$);
11         $\pi' \leftarrow$ ApproxLocalSearch($\pi'$);
12         if elite set $P$ is full then
13             if $c(\pi') \leq \max\{c(\pi) \mid \pi \in P\}$ and $\pi' \not\approx P$ then
14                 Replace the element most similar to $\pi'$ among all
                      elements with cost worst than $\pi'$;
15             end
16
17         else if $\pi' \not\approx P$ then
18             $P \leftarrow P \cup \{\pi'\}$;
19         end
20
21     else if $\pi'$ is feasible and $\pi' \not\approx P$ then
22         $P \leftarrow P \cup \{\pi'\}$;
23     end
24 end
25 return $\pi^* = \min\{c(\pi) \mid \pi \in P\}$;
```

Algorithm 1. Pseudo-code for GRASP+PR: GRASP with path-relinking heuristic

Procedure GreedyRandomized attempts to construct a greedy randomized solution to serve at a starting solution for local search. It does so by attempting, at most $\bar{t} \in [1, 100]$ times, to construct a feasible solution. In the construction process facilities and locations are selected at random with bias. To implement the randomized selection three types of probabilities are computed:

- Probability of selecting new location j: $H_j / \sum_{l \in L} H_l$, where L is the set of currently unused locations and $H_j = \sum_{l \in CL} \frac{Q_j^{h_1} Q_l^{h_2}}{b_{jl}^{h_3}}$, where Q_j is the capacity of location j, b_{jl} is the distance between locations j and l, CL is the set of previously selected locations, and input parameters h_1, h_2, h_3 are real numbers in the interval $[0, 1]$.
- Probability of selecting new facility i: $W_i / \sum_{t \in T} W_t$, where T is the subset of currently unused facilities and $W_i = q_i^{w_1} \sum_{t \in N \setminus \{i\}} a_{it}^{w_2}$, where q_i is the demand of facility i, a_{it} is the flow between facilities i and t, N is the set of facilities, and input parameters w_1, w_2 are real numbers in the interval $[0, 1]$.

– Probability of selecting a used location j: $Z_j / \sum_{r \in R} Z_r$, where R is a subset of currently used locations and $Z_j = \sum_{l \in CL \setminus \{j\}} \frac{\sigma_j^{z_1} Q_l^{z_2}}{d^{z_3} b_{jl}^{z_4}}$, where σ_j is the available capacity of location j, d is the increase in the objective function resulting from the assignment to it of the chosen facility in T, and input parameters z_1, z_2, z_3, z_4 are real numbers in the interval $[0, 1]$.

These probabilities are used in one of five heuristic-biased stochastic sampling schemes of Bresina [16] determined by input parameter $s \in \{1, 2, 3, 4, 5\}$. If Bresina's polynomial bias is selected, parameter $g \in \{1, 2, \ldots, 10\}$ determines the degree of the polynomial used. Consequently procedure `GreedyRandomized` takes as input 12 parameters $\bar{t}, h_1, h_2, h_3, w_1, w_2, z_1, z_2, z_3, z_4, s$, and g.

Procedure `ApproxLocalSearch` applies an approximate local search scheme from a given starting solution. Given a current solution, the method samples solutions resulting from single and double facility-to-location moves to create a set of candidate solutions (*CLS*) to which to move to. At each iteration the method samples at most *MaxItr* moves, some improving, some not, and creates the set *CLS* with at most *MaxCLS* elements. The method either chooses the solution π from *CLS* in a greedy fashion or it selects it with probability

$$\frac{G_\pi}{\sum_{\pi' \in CLS} G_{\pi'}}, \text{ where } G_{\pi'} = 1/f(\pi'),$$

where $f(\cdot)$ is the objective function the GQAP. Input parameter *CLChoice* determines which option is used. In the former case, parameters $s \in \{1, 2, 3, 4, 5\}$ and $g \in \{1, 2, \ldots, 10\}$ determine, as in construction procedure, which of Bresina's stochastic sampling schemes will be used. Consequently, procedure `ApproxLocal-Search` uses as input 6 parameters: $MaxItr \in \{1, 2, \ldots, 1000\}$, $MaxCLS \in \{1, 2, \ldots, 100\}$, $CLChoice \in \{0, 1\}$, $s \in \{1, 2, 3, 4, 5\}$, $g \in \{1, 2, \ldots, 10\}$, which determines if the greedy or which randomized selection will be used, and the real-valued $neighborhoodBalance \in [0, 1]$ which determines the proportion of single and double facility-to-location moves sampled.

Procedure `PathRelinking` takes as input 8 parameters that determine, among many options, whether forward-, backward-, or mixed-path-relinking is used, whether truncated path-relinking is used (and if so, how many steps are carried out), whether greedy or greedy randomized path-relinking is used, and the maximum number of feasibility restoration steps that can be carried out.

In addition to the above 26 parameters, the main algorithm has the following 4 parameters: the maximum size of the elite set $maxES \in \{1, 2, \ldots, 50\}$, the minimum number of elements in the elite set required for path-relinking to be used $\rho \in \{2, 3, \ldots, maxES\}$, the minimum difference parameter $\delta \in \{0, 1, \ldots, n\}$, and parameter $selectFromES \in \{0, 1\}$ which determines whether in line 9 of Algorithm 1 element π^+ is selected uniformly at random or with bias. In total, there are 30 user-defined parameters than need to be tuned.

4 Two-Phase Hybrid Heuristic

The two-phase hybrid heuristic consists first of a tuning phase, where the BRKGA explores the GRASP+PR parameter space, followed by a solution phase, where the GRASP+PR heuristic using the parameters determined in the first phase explores the GQAP solution space seeking an optimal or near optimal assignment of facilities to locations. In the first phase, the BRKGA is run for Y_1 generations while in the second phase, the GRASP+PR heuristic is run for Y_2 iterations.

To describe the first phase of the two-phase hybrid heuristic, we first specify the encoding of the parameter-space solutions as well as the decoding of these solutions. The random-key solution vector x has $n = 30$ components, one for each tunable parameter. Each component is a random key generated in the real interval $[0, 1]$. If a parameter $i = 1, \ldots, n$ is in the real interval $[l, u]$, its random-key component $x(i)$ is decoded as $l+x(i)\cdot(u-l)$. On the other hand, if parameter i is in the discrete interval $[l, u]$, its random-key component $x(i)$ is decoded as $\lceil l - \frac{1}{2} + x(i) \cdot (u - l)\rceil$.

The fitness of a solution vector is obtained by carrying out V independent runs of the GRASP+PR heuristic using the parameters decoded from the solution vector, each run for U GRASP+PR iterations. The fitness is computed as the average objective function value of the V runs.

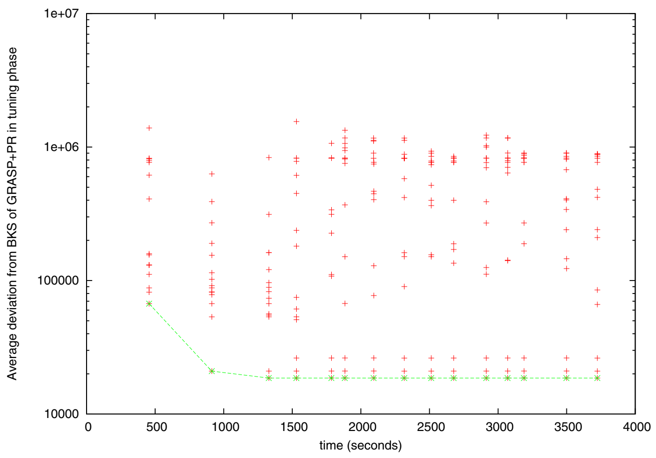
5 Computational Results

In this section, we report on preliminary computational results with the automatic parameter tuning scheme introduced in this paper. All experiments were done on a Dell PE1950 computer with dual quad core 2.66 GHz Intel Xeon processors and 16 Gb of memory, running Red Hat Linux nesh version 5.1.19.6 (CentOS release 5.2, kernel 2.6.18-53.1.21.el5). The two-phase BRKGA / GRASP-PR heuristic was implemented in Java and compiled into bytecode with `javac` version 1.6.0_05. The random-number generator is an implementation of the Mersenne Twister algorithm [17] from the COLT[1] library.

The objective of the experiments was to compare the performance of the GRASP+PR heuristic using the parameter obtained through manual tuning in [15] with the same heuristic using parameters automatically tuned with the BRKGA described in this paper. We consider five instances from Cordeau et al. [18]: 20-15-35, 20-15-55, 20-15-75, 30-07-75, and 30-08-55. Instance f-l-t in this class has f facilities and l locations. Parameter t controls the tightness of the constraints. The higher the value of t, the greater the tightness of the constraints. The tighter the constraints, the harder it is to find a feasible solution.

For each instance, we ran the first phase of the hybrid heuristic to automatically tune the 30 parameters of the GRASP+PR heuristic for the GQAP. The BRKGA used a population of 15 elements and ran for only 10 generations. The

[1] COLT is a open source library for high performance scientific and technical computing in Java. See `http://acs.lbl.gov/~hoschek/colt/`



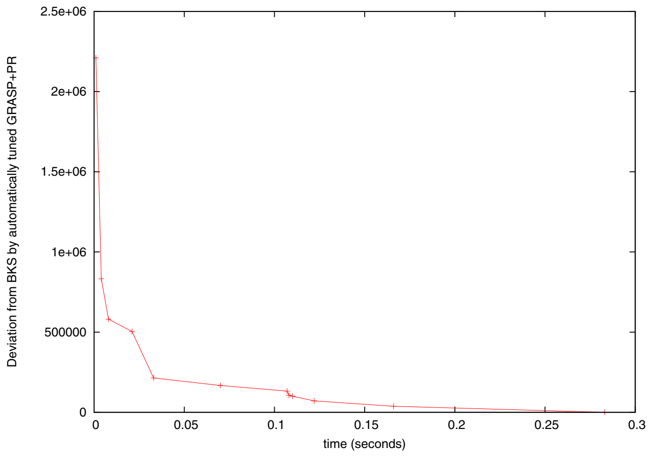


Fig. 1. The plot on the left shows, for each iteration of the BRKGA tuning procedure, the distribution of fitness values found for instance 20-15-75. The plot on the right shows the solutions found by the GRASP+PR heuristic using the automatically tuned parameters found by the tuning procedure. Both figures show deviations from the best known solution (BKS) for 20-15-75.

fitness computation was done over $V = 30$ independent runs of the GRASP+PR heuristic, each one for $U = 100$ iterations. With the automatically tuned parameters on hand, the heuristic found, in the second phase, the best solution for all five instances. In addition, 200 independent runs of the GRASP+PR heuristic (manually and automatically tuned variants) were carried out for each instance, stopping each time only after the best known solution for the instance was found. All 200 runs of each variant and for each instance found the best known solution.

The plots in Figure 1 show solutions found by the tuning procedure (on the left) and the GRASP+PR with the automatically tuned parameters (on

Table 1. Comparison of a GRASP+PR heuristic for the GQAP with manually and automatically tuned parameters. For each instance and each heuristic variant, the table lists minimum, maximum, and average times (in seconds) to find the best known solution, as well as standard deviations.

	Manually tuned				Automatically tuned			
problem	min	max	avg	sdev	min	max	avg	sdev
20-15-35	1.16	845.29	147.09	146.53	0.59	71.30	9.62	9.19
20-15-55	0.63	83.52	17.04	16.43	0.36	33.03	7.17	6.15
20-15-75	0.92	166.30	8.47	14.04	0.78	552.19	47.55	82.88
30-08-55	0.35	11.67	2.26	1.54	0.07	3.42	0.96	0.61
30-07-75	9.22	26914.03	716.08	2027.75	1.27	228.01	28.63	29.75

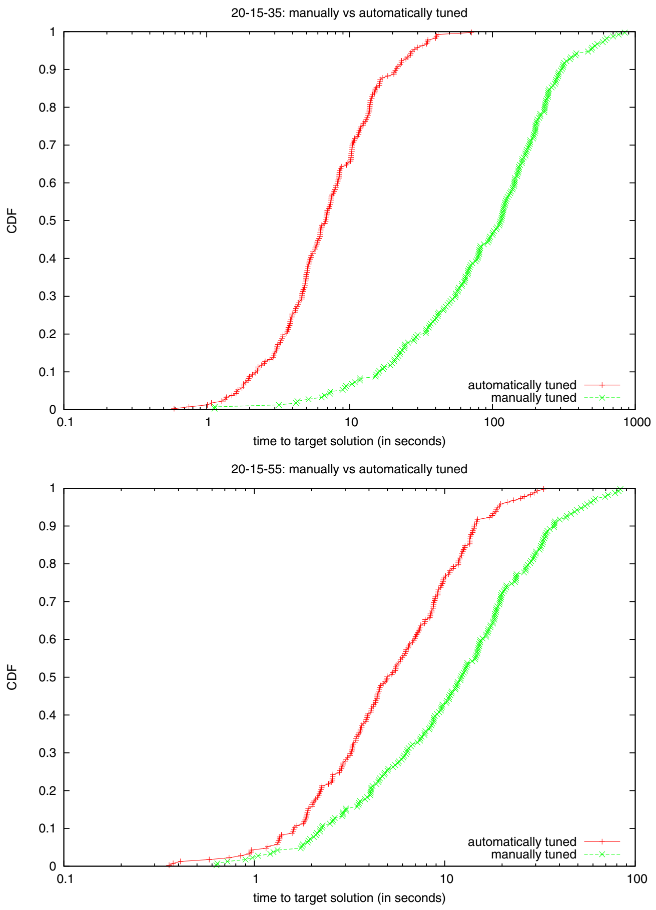


Fig. 2. Runtime distributions for manually and automatically tuned GRASP+PR heuristics for the GQAP on instances 20-15-35 and 20-15-55. 200 independent runs of each variant were carried out and running times to find the best known solution for the instances plotted.

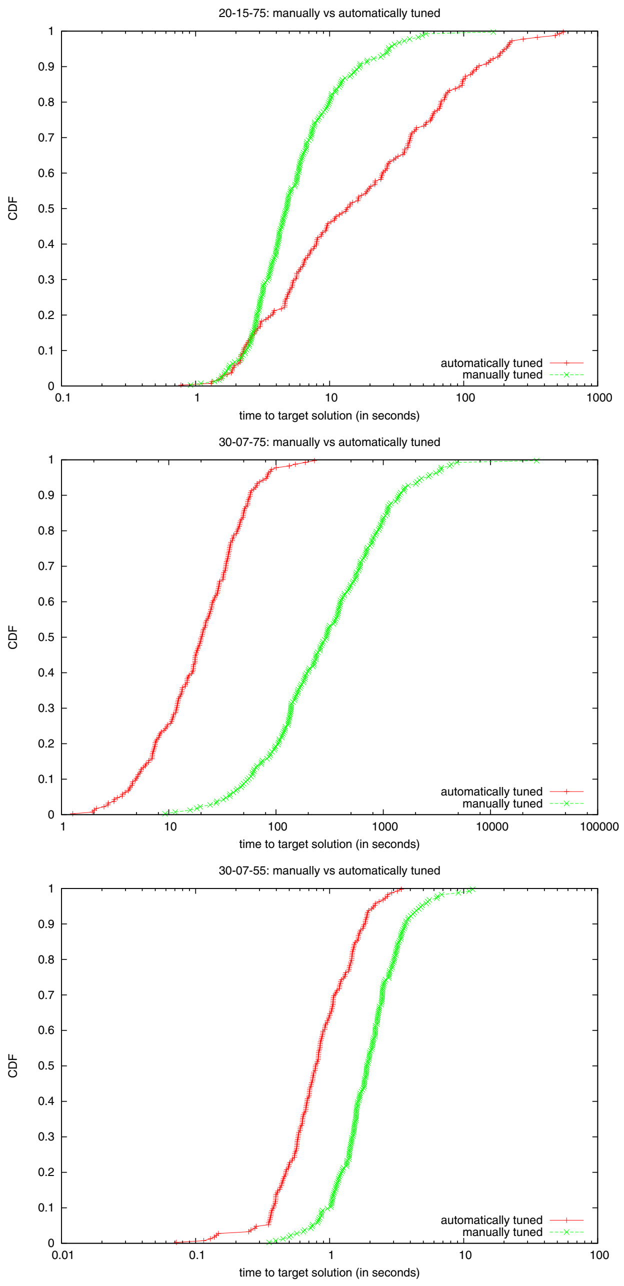


Fig. 3. Runtime distributions for manually and automatically tuned GRASP+PR heuristics for the GQAP on instances 20-15-75, 30-07-75, and 30-08-55. 200 independent runs of each variant were carried out and running times to find the best known solution for the instances plotted.

the right) on instance 20-15-75. As can be observed, the BRKGA finds a good parameter setting in a few generations and the GRASP+PR using these parameters quickly reaches the best known solution for the instance.

Table 1 summarizes the experiments. For each instance, the table lists statistics for both the manually and automatically tuned GRASP+PR heuristic. For each variant, the table shows the minimum, maximum, and average running times (in seconds) to find the best known solution for each instance, as well as the standard deviation computed over 200 independent runs of the GRASP+PR heuristic.

Figures 2 and 3 show runtime distribution plots for the manually and automatically tuned GRASP+PR heuristics for instances 20-15-35, 20-15-55, 20-15-75, 30-07-75, and 30-08-55.

Times on Table 1 as well as Figures 2 and 3 are limited to GRASP+PR and do not include the time taken by the BRKGA to automatically tune the parameters. Tuning times were, respectively, 10,739.2, 7,551.2, 3,690.3, 21,909.1, and 14,386.5 seconds for instances 20-15-35, 20-15-55, 20-15-75, 30-07-75, and 30-08-55. These times could be reduced considerably with a parallel implementation of the BRKGA as well as with the imposition of a maximum running time for the GRASP+PR heuristic run in the process of computing the fitness of the parameter settings. Poor settings often lead to configurations that struggle to find feasible assignments, thus leading to long running times. On the other hand, the times for the manually tuned heuristic do not reflect the weeks that it took for us to do the manual tuning.

The table as well as the figures clearly show that significant improvements can be obtained with automatic tuning of the parameters. On all instances except 20-15-75, the automatically tuned variant proved to find the best known solution in less time than the manually tuned variant. In the most difficult instance (30-07-75), the automatically tuned variant was on average about 25 times faster than the manually tuned variant. The ratio of maximum running times on this instance was over 118, in favor of the automatically tuned variant.

6 Concluding Remarks

In this paper, we have studied a new two-phase automatic parameter tuning procedure for GRASP+PR heuristics based on a biased random-key genetic algorithm. The robustness of the procedure has been illustrated through a GRASP+PR with $n = 30$ tunable parameters for the generalized quadratic assignment problem (GQAP) on five difficult GQAP instances from Cordeau et al. [18]. As future work, we intend to apply this automatic tuning procedure on other NP-hard problems beyond GQAP.

References

1. Bean, J.: Genetic algorithms and random keys for sequencing and optimization. ORSA J. on Computing 6, 154–160 (1994)
2. Gonçalves, J., Resende, M.: Biased random-key genetic algorithms for combinatorial optimization. Technical report, AT&T Labs Research, Florham Park, NJ 07932 (2009), http://www.research.att.com/~mgcr/doc/srkga.pdf

3. Spears, W., DeJong, K.: On the virtues of parameterized uniform crossover. In: Proceedings of the Fourth Int. Conference on Genetic Algorithms, pp. 230–236 (1991)
4. Feo, T., Resende, M.: A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters 8, 67–71 (1989)
5. Feo, T., Resende, M.: Greedy randomized adaptive search procedures. J. of Global Optimization 6, 109–133 (1995)
6. 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)
7. Resende, M., Ribeiro, C.: Greedy randomized adaptive search procedures: Advances and applications. In: Gendreau, M., Potvin, J.Y. (eds.) Handbook of Metaheuristics, 2nd edn. Springer Science+Business Media (2010)
8. Festa, P., Resende, M.: An annotated bibliography of GRASP – Part I: Algorithms. International Transactions on Operational Research 16, 1–24 (2009)
9. Festa, P., Resende, M.: An annotated bibliography of GRASP – Part II: Applications. International Transactions on Operational Research (in press, 2009)
10. 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)
11. Glover, F., Laguna, M., Martí, R.: Fundamentals of scatter search and path relinking. Control and Cybernetics 39, 653–684 (2000)
12. Resende, M., Ribeiro, C., Glover, F., Martí, R.: Scatter search and path-relinking: Fundamentals, advances, and applications. In: Gendreau, M., Potvin, J.Y. (eds.) Handbook of Metaheuristics, 2nd edn. Springer Science+Business Media (2010)
13. Laguna, M., Martí, R.: GRASP and path relinking for 2-layer straight line crossing minimization. INFORMS Journal on Computing 11, 44–52 (1999)
14. 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)
15. Mateus, G., Resende, M., Silva, R.: GRASP with path-relinking for the generalized quadratic assignment problem. Technical report, AT&T Labs Research Technical Report, Florham Park, NJ 07932 (2009), http://www.research.att.com/~mgcr/doc/gpr-gqap.pdf
16. Bresina, J.: Heuristic-biased stochastic sampling. In: Proc. AAAI, pp. 271–278 (1996)
17. 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)
18. Cordeau, J.F., Gaudioso, M., Laporte, G., Moccia, L.: A memetic heuristic for the generalized quadratic assignment problem. INFORMS Journal on Computing 18, 433–443 (2006)

Experiments with a Feasibility Pump Approach for Nonconvex MINLPs

Claudia D'Ambrosio[1], Antonio Frangioni[2], Leo Liberti[3], and Andrea Lodi[1]

[1] DEIS, University of Bologna, Italy
{c.dambrosio,andrea.lodi}@unibo.it

[2] Dipartimento di Informatica, University of Pisa, Italy
frangio@di.unipi.it

[3] LIX, École Polytechnique, France
liberti@lix.polytechnique.fr

Abstract. We present a new Feasibility Pump algorithm tailored for nonconvex Mixed Integer Nonlinear Programming problems. Differences with the previously proposed Feasibility Pump algorithms and difficulties arising from nonconvexities in the models are extensively discussed. The main methodological innovations of this variant are: (a) the first subproblem is a nonconvex continuous Nonlinear Program, which is solved using global optimization techniques; (b) the solution method for the second subproblem is complemented by a tabu list. We exhibit computational results showing the good performance of the algorithm on instances taken from the MINLPLib.

Keywords: Mixed-Integer Nonlinear Programming, nonconvex, heuristic method, experiments.

1 Introduction

Heuristic algorithms have always played a fundamental role in optimization, both as independent tools and as part of general-purpose solvers. Heuristics can be classified into two broad categories: those which are based on a specific problem structure (e.g., heuristics for Set Covering, or Knapsack, or Quadratic Assignment problems) and those which target a large class of problems, such as Mixed Integer Linear Programming (MILP), or Mixed Integer Nonlinear Programming (MINLP). Far fewer heuristics (including the present one) belong to the second class with respect to the first one because of the inherent difficulty of devising general-purpose methods. In the rest of the paper we focus on algorithms belonging to the second class.

Starting from MILPs, different kinds of heuristics have been proposed: their aim is finding a good feasible solution rapidly or improving the best solution found so far. Within a MILP solver context, both types are used. Examples are rounding heuristics, metaheuristics (e.g. [1]), Feasibility Pump (FP) [2,3,4], Local Branching [5] and Relaxation Induced Neighborhoods Search [6]. Since the early 1990's, MINLP has attracted rising interest from the operations research and the chemical engineering communities. Typically, MINLP solution techniques complement continuous Nonlinear Programming (NLP) algorithms with combinatorial-type searches. Chemical plant design and operations are amongst the early applications of this field. Special attention has been devoted to convex MINLPs, a class of MINLP problems whose objective function and constraints are convex (thus, any local optimum of the continuous relaxation is also its global optimum). Furthermore, standard linearization inequalities such as Outer Approximation (OA) cuts [7] are valid. OA cuts linearly approximate each nonlinear convex constraint, say $f(x) \leq 0$, at a given point $\bar{x}$:

$$f(\bar{x}) + \nabla f(\bar{x})^T (x - \bar{x}) \leq 0. \tag{1}$$

It is easy to show that (1) does not cut off any feasible point of the original convex MINLP. Heuristics have been proposed recently also for convex MINLPs. Basically the ideas originally tailored on MILP problems have been extended to convex MINLPs, for example, Feasibility Pump [8,9,10] and diving heuristics [10].

This paper proposes a heuristic algorithm for nonconvex MINLPs. These problems are in general very difficult to solve to optimality and, often, also finding a feasible solution is practically difficult (besides being NP-hard in theory, since they generalize NLP feasibility [11]). For this reason, heuristic algorithms are a fundamental part of any solution process. General-purpose nonconvex MINLP heuristics proposed so far are, for example, Variable Neighborhood Search [12] and Local Branching [13]. All existing exact convex MINLP methods [7,8,14,15,16,17,18,19] can be heuristically deployed on nonconvex MINLPs. This field, however, is still relatively young and current heuristics leave a lot of room for improvement.

We already mentioned FP for 0-1 MILP, introduced by Fischetti et al. [2]. The algorithm has been extended to general integer variables by Bertacco et al. [3] and improved with respect to solution quality by Achterberg and Berthold [4]. The idea is to iteratively solve subproblems of the original (difficult) problem with the aim of "pumping" the feasibility in the solution. More precisely, Feasibility Pump iteratively solves the continous relaxation of the problem trying to minimize the distance to a target (infeasible) integer solution, then rounding the fractional solution obtained to become the next integer target. Few years later a similar technique applied to convex MINLPs was proposed by Bonami et al. [8]. In that case the subproblems are a convex NLP and a MILP. The authors also prove the convergence of the algorithm and extend the same result to MINLP problems with nonconvex constraints, defining, however, a convex feasible region. More recently Bonami and Gonçalves [10] proposed a more efficient version in which the MILP solution process is replaced by a rounding phase similar to that originally proposed by Fischetti et al. [2] for MILPs. Finally, an enhancement for the MILP case has been recently studied by Fischetti and Salvagnin [20] by using domain propagation during rounding.

In this paper, we propose a FP algorithm for general nonconvex MINLPs. The remainder of the paper is organized as follows. In Section 2 we present the structure of the algorithm. Details on implementation issues are given in Section 3. In Section 4 we present computational results on MINLPLib instances. Section 5 concludes the paper.

2 The Algorithm

We address the following nonconvex MINLP:

$$(P) \qquad \min f(x,y) \tag{2}$$
$$g(x,y) \leq 0 \tag{3}$$
$$x \in X \cap \mathbb{Z}^n \tag{4}$$
$$y \in Y, \tag{5}$$

where X and Y are two polyhedra of appropriate dimension (possibly including variable bounds), $f : \mathbb{R}^{n+p} \to \mathbb{R}$ is *convex*, but $g : \mathbb{R}^{n+p} \to \mathbb{R}^m$ is *nonconvex*. We will denote by

$$\mathcal{P} = \{ \ (x,y) \mid g(x,y) \leq 0 \wedge x \in X \wedge y \in Y \ \} \subseteq \mathbb{R}^{n+p}$$

the (nonconvex) feasible region of the continuous relaxation of the problem, by $\mathcal{X}$ the set $\{1,\ldots,n\}$ and by $\mathcal{Y}$ the set $\{1,\ldots,p\}$. We will also denote by $\mathcal{N} \subseteq \{1,\ldots,m\}$ the subset of (indices of) nonconvex constraints, so that $\mathcal{C} = \{1,\ldots,m\} \setminus \mathcal{N}$ is the set of (indices of) convex constraints. Note that the convexity assumption on the objective function f does not involve a loss of generality: one can always introduce an additional variable v to be minimized, and add the $(m+1)-$th constraint $f(x,y) - v \leq 0$ to deal with the case where f is nonconvex.

Problem (P) presents two sources of nonconvexities:

1. integrality requirements on x variables;
2. constraints $g_j(x,y) \leq 0$ with $j \in \mathcal{N}$, defining a nonconvex feasible region.

The basic idea of FP is to decompose the original problem in two easier subproblems. One, called $(P1)$, is obtained by relaxing the integrality requirements; the other, called $(P2)$, by relaxing the nonlinear constraints. Both problems are solved at each iteration, yielding a pair of solutions $(\bar{x},\bar{y})$ and $(\hat{x},\hat{y})$ respectively. The aim of the algorithm is to make the trajectories of the two solutions converge to a unique point, satisfying all the constraints and the integrality requirements (see Algorithm 1).

In the next two sections we discuss the general framework by specializing it to our context, i.e., the nonconvex MINLP case.

Algorithm 1. The general scheme of Feasibility Pump

```
1. $i = 0$;
2. initialize $(\hat{x}^0, \hat{y}^0)$ and $(\bar{x}^0, \bar{y}^0)$;
3. while $((\hat{x}^i, \hat{y}^i) \neq (\bar{x}^i, \bar{y}^i)$ and CPU time $<$ limit) do
4.    increase $i$;
5.    solve $(P1)$ (minimize distance to $(\hat{x}^{i-1}, \hat{y}^{i-1})$ subject to $(x,y) \in \mathcal{P}$) to yield
      $(\bar{x}^i, \bar{y}^i)$;
6.    solve $(P2)$ (minimize distance to $(\bar{x}^i, \bar{y}^i)$ subject to $(x,y) \in (X \cap \mathbb{Z}^n) \times Y$) to
      yield $(\hat{x}^i, \hat{y}^i)$;
7. end while
```

2.1 Subproblem $(\boldsymbol{P}\mathbf{1})$

At iteration i the subproblem $(P1)$ is denoted by $(P1)^i$ and has the form

$$\min_{x \in X,\ y \in Y} ||x - \hat{x}^{i-1}|| \tag{6}$$

$$g(x, y) \leq 0, \tag{7}$$

where $(\hat{x}^{i-1}, \hat{y}^{i-1})$ is the solution of subproblem $(P2)^{i-1}$ which satisfies the integrality requirements on x (see Section 2.2) and in (6) we used the 2-norm. Either (a) the globally optimal objective function value of $(P1)^i$ is 0, implying a feasible solution of (P) with $x = \hat{x}^{i-1}$; or (b) no feasible point of (P) exists with $x = \hat{x}^{i-1}$. Unfortunately, solving $(P1)^i$ to global optimality is too computationally expensive to be considered as a viable option. Using a local NLP solver to solve $(P1)^i$ is not a viable alternative either: if $(\bar{x}^i, \bar{y}^i)$ is a local optimum of $(P1)^i$ with value > 0, (b) no longer holds and there might exist a feasible solution of (P) with $x = \hat{x}^{i-1}$. We would then erroneously consider the solution with $x = \hat{x}^{i-1}$ as infeasible and continue iterating. We therefore propose the following strategy:

1. Solve $(P1)^i$ using a simple multistart heuristic [21] to maximize chances of finding the global optimum.
2. If no solution yielding 0 as objective function value was found, solve the following problem denoted as $(P1fix)^i$:

$$\min f(\hat{x}^i, y) \tag{8}$$

$$g(\hat{x}^i, y) \leq 0. \tag{9}$$

Problem $(P1fix)^i$ differs with respect to problem $(P1)^i$ because the objective (6) (constant if x is fixed to $\hat{x}^i$ like in this case) is replaced by the original objective f, and it is solved in the attempt of finding a MINLP feasible solution with $\hat{x}^i$ values of x variables.

The solution proposed does not give any guarantee that the global optimum will be found and, consequentely, that no feasible solution of (P) will be ignored, but, since we propose a heuristic algorithm, we consider this simplification as a good compromise. Our computational experiments show that for some classes of nonconvex MINLP the approach is sound. Consider, for example, a problem (P) that, once variables x are fixed, is convex: in this case solving problem $(P1fix)^i$ would provide the global optimum.

2.2 Subproblem $(\boldsymbol{P}\mathbf{2})$

At iteration i subproblem $(P2)$, denoted as $(P2)^i$, has the form

$$\min ||x - \bar{x}^i|| \tag{10}$$

$$g_j(\bar{x}^k, \bar{y}^k) + \nabla g_j(\bar{x}^k, \bar{y}^k)^T \begin{pmatrix} x - \bar{x}^k \\ y - \bar{y}^k \end{pmatrix} \leq 0 \quad k = 1, \ldots, i; j \in M^k \tag{11}$$

$$x \in X \cap \mathbb{Z}^n \tag{12}$$

$$y \in Y, \tag{13}$$

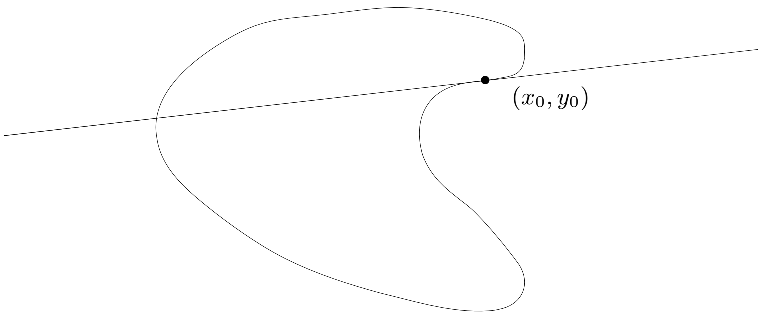


Fig. 1. Outer Approximation constraint cutting off part of the nonconvex feasible region

where $(\bar{x}^i, \bar{y}^i)$ is the solution of subproblem $(P1)^i$ and $M^k \subseteq \{1, \ldots, m\}$ is the set (possibly empty) of (indices of) constraints from which OA cuts are generated at point $(\bar{x}^k, \bar{y}^k)$. We remark that M^k will most usually be a proper subset of $\{1, \ldots, m\}$ because, when nonconvex constraints are involved, not all the possible OA cuts generated are "safe", i.e., do not cut off feasible solutions of (P) (see Figure 1). We remark that the OA cut generated from a convex constraint $g(z)$ is valid for (P). In order to model subproblem $(P2)^i$ as a MILP, in (10) we use the 1-norm.

Generation of OA cuts involves essentially two issues, one stemming from a practical consideration, the other from a theoretical point of view. The first issue is that discriminating convex and nonconvex constraints is a hard task in practice. We will describe in Section 4 how we simplified this step on the implementation side. The second issue is that OA cuts play a fundamental role on the convergence of the algorithm for convex MINLPs (see Bonami et al. [8]): if at one iteration no OA cut can be added, the algorithm may cycle. However, in the nonconvex case, even if an OA cut is added, there is no guarantee that it would cut off the solution of the previous iteration, as shown by the following example.

Example 1. In Figure 2 a nonconvex feasible region and its current linear approximation are depicted. Let us consider $\bar{x}$ being the current solution of subproblem $(P1)$. In this case, only one Outer Approximation cut can be generated, the one corresponding to convex constraint γ. However, this OA cut does not cut off solution $\hat{x}$, i.e., the solution of $(P2)$ at the previous iteration. In this example, the FP would not immediately cycle, because $\hat{x}$ is not the solution of $(P2)$ which is closest to $\bar{x}$. This shows that there is a distinction between cutting off and cycling. However, in the long(er) term not cutting off previously generated integer solutions might lead to cycling. □

A possible solution to this issue is using a tabu list for the last solutions obtained from $(P2)$: the MILP solver will discard integer feasible solutions in the tabu list. The integer part of the solution is compared with the one of the solutions in

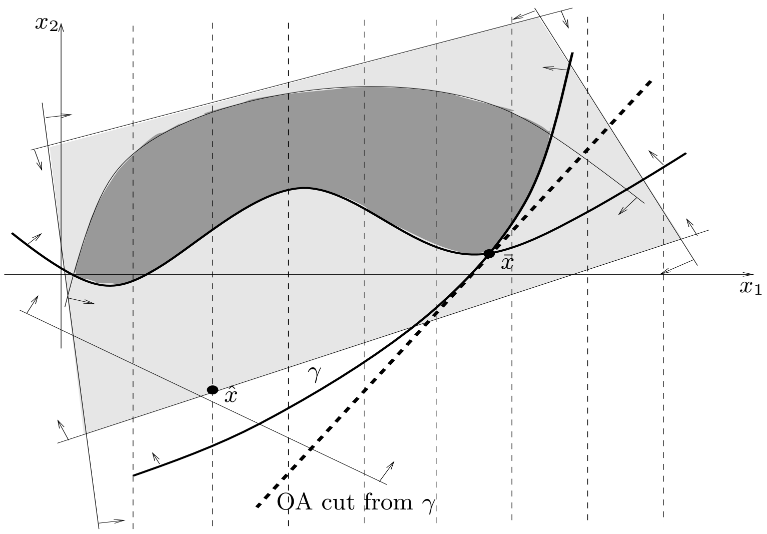


Fig. 2. The OA cut from γ does not cut off $\bar{x}$

the tabu list and it is accepted only if its integer part is different from that of the forbidden solutions. Otherwise it is discarded: this prevents algorithmic cycling as long as the cycle length is shorter than the tabu list length. This simple idea works with both binary and general integer variables.

Before ending Section 2, we discuss previous FP implementations with respect to the general framework described above.

First, when the original problem (P) is an MILP, $(P1)$ is simply the LP relaxation of the current problem and $(P2)$ is the original MILP itself but with a different objective function. However, because in such a case problem $(P2)$ is probably as difficult as (P), Fischetti et al. [2] iteratively solved a trivial relaxation in which all the constraints are relaxed, i.e., an integer solution is obtained by rounding the fractional solution of $(P1)$.

Moreover, when the original problem (P) is a convex MINLP, i.e., $\mathcal{N} = \emptyset$, $(P1)$ is the NLP relaxation of the problem and $(P2)$ is a MILP relaxation of (P). In this case, we know that: (a) $(P1)$ is convex as well and it can ideally be solved to global optimality; and (b) $(P2)$ can be defined as the OA of (P) (see, e.g., Bonami et al. [8]) or replaced by a rounding phase (see Bonami and Gonçalves [10]).

Finally, when $\mathcal{N} \neq \emptyset$, as previously discussed, things get much more complicated because we have two different sources of nonconvexity. This is the main difference with respect to the previous FP algorithms and both $(P1)$ and $(P2)$ require specialized algorithmic techniques.

3 Code Structure

The algorithm was implemented within the AMPL environment [22]. We chose to use this framework to make it easy to change subsolver. In practice, the user can select the preferred solver to solve NLPs or MILPs, exploiting their advantages. In our case, problems $(P1)$ and $(P1fix)$ are solved using IPOPT [23]. Problem $(P2)$ is solved by CPLEX [24] modified by a tabu list hooked up to the solver via the incumbent callback function. This allows the user to define a function which is called during execution whenever CPLEX finds a new integer feasible solution. The tabu list is stored in a text file which is then exchanged between AMPL and CPLEX. Every time CPLEX finds an integer feasible solution, the specialized incumbent callback checks whether the new solution appears in the tabu list. If this is the case, the solution is rejected, otherwise the solution is accepted. CPLEX continues executing until either the optimal solution (excluding those forbidden) is found or a time limit is reached. In the case where an integer solution (x', y') found by CPLEX at the root node appears in the tabu list, CPLEX stops and no new integer feasible solution is passed to FP. In such a case, we amended problem $(P2)$ with a no-good cut [25] which excludes (x', y') and we call CPLEX again.

We also use a new solver/reformulator called ROSE (Reformulation Optimization Software Engine, see [26]), of which we exploit the following features.

1. Model analysis: getting information about nonlinearity and convexity of the constraints and integrality requirements of the variables (necessary to define $(P1)$ and $(P2)$).
2. Solution feasibility analysis: necessary to verify feasibility of the provided solutions.
3. OA cut generation: necessary to update $(P2)$.

We remark that some of the above features were added to ROSE within the context of the present work. In order to determine whether a constraint is convex, ROSE performs a recursive analysis of its expression tree [27] to determine whether it is an affine combination of convex functions. We call such a function *evidently convex* [26]. Evident convexity is a stricter notion than convexity: evidently convex functions are convex but the converse may not hold. Thus, it might happen that a convex constraint is labelled nonconvex; the information provided is in any case safe for our purposes, i.e., we generate OA cuts only from constraints which are certified to be convex.

4 Computational Results

In this section we report the results of preliminary computational experiments performed on an Intel Xeon 2.4 GHz with 8 GB RAM running Linux. We present the results in Tables 1-3. The algorithm terminates after the first MINLP feasible solution is found or a time limit is reached. The parameters are set in the following way: time limit of 2 hours of user CPU time, the absolute feasibility

Table 1. Instances for which a feasible solution was found within the time limit (199/243). The solution found is also the best-known solution for the instances marked in boldface.

instance	time	# iter	value	instance	time	# iter	value	instance	time	# iter	value
alan	0	1	4.2222E+00	gear3	0	1	7.3226E-01	ortez	0	1	-3.9039E-01
batchdes	0	1	2.2840E+05	gear4	0	1	9.6154E+05	parallel	0	1	4.0000E+10
batch	0	1	3.5274E+05	gear	0	1	7.3226E-01	**prob02**	0	1	1.1224E+05
contvar	608	1	1.9442E+07	gkocis	0	1	-2.7840E-03	**prob03**	0	1	1.0000E+01
csched1a	0	1	-2.5153E+04	hmittelman	0	1	2.1000E+01	prob10	0	2	4.7854E+00
csched1	0	1	-2.1049E+04	johnall	615	1	-2.0129E+02	procsel	0	1	-6.7200E-04
csched2a	4	1	-1.0287E+05	m3	0	1	2.4000E+06	product	17	1	-1.7772E+03
csched2	203	1	-1.2007E+05	m6	0	1	6.4800E+06	qap	0	1	4.9951E+05
deb6	4	3	2.3710E+02	m7_ar2_1	1	1	7.8800E+06	qapw	610	1	4.6012E+05
deb7	13	2	3.6983E+02	m7_ar25_1	1	1	7.8800E+06	ravem	171	1	7.6441E+05
deb8	2	1	1.4515E+06	m7_ar3_1	0	1	7.8800E+06	ravempb	185	1	7.6441E+05
deb9	18	3	4.2657E+02	m7_ar4_1	0	1	7.8800E+06	risk2bpb	2	1	-1.0195E+01
detf1	128	1	1.5976E+04	m7_ar5_1	0	1	7.8800E+06	saa_2	169	1	1.5976E+04
du-opt5	0	1	9.7825E+03	m7	0	1	7.8800E+06	sep1	0	1	-3.6123E+02
du-opt	0	1	1.1988E+04	mbtd	5,058	1	9.8529E+01	space25a	134	17	6.5069E+02
eg_all_s	62	4	1.0000E+05	meanvarx	0	1	2.1362E+01	space25	446	17	6.5069E+02
eg_disc2_s	7	1	1.0000E+05	no7_ar25_1	2,986	879	4.0000E+06	spectra2	0	1	3.0479E+02
eg_disc_s	9	1	1.0001E+05	no7_ar3_1	14	1	4.0000E+06	spring	1	2	1.3073E+00
elf	0	1	2.3992E+06	no7_ar4_1	138	162	4.0000E+06	st_e13	0	1	2.6004E+00
eniplac	2	3	-1.0209E+05	no7_ar5_1	7	1	4.0000E+06	st_e14	0	1	1.4555E+01
enpro48	609	1	1.6422E+06	nous1	0	1	2.1055E+00	st_e15	0	1	8.4763E+00
enpro48pb	610	1	1.6422E+06	nous2	0	1	2.1328E+00	st_e27	0	1	2.0020E+00
enpro56	607	1	7.0730E+05	nuclear14a	1,839	130	-1.1136E+00	st_e29	0	1	-2.9550E-01
enpro56pb	607	1	7.0730E+05	nuclear14b	670	5	-1.1007E+00	st_e31	1	1	-4.1766E-01
ex1221	0	1	8.4763E+00	nuclear14	647	3	-1.1213E+00	**st_e32**	0	1	-1.4303E+00
ex1222	0	1	1.0800E+00	nuclear24a	1,826	130	-1.1136E+00	st_e35	0	1	1.3270E+05
ex1223a	0	1	1.4556E+01	nuclear24b	668	5	-1.0979E+00	st_e36	1	3	-1.6644E+02
ex1223b	0	1	1.4556E+01	nuclear24	830	15	-1.1145E+00	st_e38	0	1	7.4478E+03
ex1223	0	1	1.4555E+01	nuclear25a	902	32	-1.0927E+00	**st_miqp1**	0	1	2.8100E+02
ex1224	0	1	-2.9550E-01	nuclear25b	627	2	-1.0750E+00	st_miqp2	0	1	7.0000E+00
ex1225	0	2	3.4000E+01	nuclear25	1,213	26	-1.1050E+00	st_miqp3	0	1	-1.0900E-04
ex1226	0	1	-6.1179E+00	nuclearva	132	1	-1.0068E+00	st_miqp4	0	1	-1.8889E-01
ex1233	1	1	2.5338E+05	nuclearvb	111	1	-1.0248E+00	st_miqp5	0	1	1.6881E+03
ex1243	0	1	1.6850E+05	nuclearvc	119	3	-9.8701E-01	stockcycle	5	1	2.1821E+05
ex1244	0	1	1.3109E+05	**nuclearvd**	424	1	-1.0370E+00	**st_test1**	0	1	-2.2230E-03
ex1263a	1	11	3.1000E+01	**nuclearve**	406	1	-1.0344E+00	st_test2	0	1	4.5600E-04
ex1263	66	137	1.2100E+02	**nuclearvf**	373	1	-1.0195E+00	st_test3	0	1	3.1900E-04
ex1264a	0	3	1.2000E+01	nvs01	0	1	4.9199E+02	st_test4	0	1	7.0000E+00
ex1264	29	12	1.8300E+01	nvs02	0	3	7.0780E+00	**st_test5**	0	1	-1.1000E+02
ex1265a	2	5	1.6500E+01	**nvs03**	0	2	1.6000E+01	st_test6	0	1	5.6700E+02
ex1265	158	6	1.4305E+01	nvs04	0	1	1.0040E+10	st_test8	0	1	2.4728E+04
ex1266a	0	1	1.6300E+01	nvs06	0	1	1.1596E+01	st_testgr1	0	1	0.0000E+00
ex1266	628	36	3.4600E+01	nvs07	0	2	6.0000E+00	st_testgr3	0	1	-6.1000E-05
ex3	0	1	1.1501E+02	nvs08	0	1	2.4119E+04	st_testph4	1	1	-1.2820E-03
ex3pb	0	1	1.1501E+02	nvs09	0	1	1.2972E+01	synheat	0	1	2.4872E+05
ex4	0	1	2.5566E+06	nvs10	0	1	-1.0240E+02	synthes1	0	1	9.9980E+00
fac1	0	1	5.4299E+08	nvs11	0	1	-1.5300E+02	synthes2	0	1	1.3608E+02
fac2	1	1	1.9520E+09	nvs12	0	1	-1.8800E+02	synthes3	0	1	1.1074E+02
fac3	0	1	1.0497E+08	nvs13	0	1	-1.6640E+02	tln2	1	27	2.8300E+01
feedtray2	2	1	8.7100E-04	nvs14	0	3	-2.9220E+04	tln4	1	4	1.2000E+01
feedtray	0	1	-1.2414E+01	**nvs15**	0	1	1.0000E+00	tln5	0	6	1.6500E+01
fo7_2	11	28	1.2000E+06	nvs16	0	1	1.4203E+01	tln6	1	8	2.5100E+01
fo7_ar25_1	3,066	879	1.2000E+06	nvs17	0	1	-2.7900E+02	tln7	1,072	397	1.0780E+02
fo7_ar3_1	8	1	1.2000E+06	nvs18	0	1	-2.0900E+02	tloss	5	7	2.4100E+01
fo7_ar4_1	141	162	1.2000E+06	nvs19	0	1	-2.8240E+02	**tls2**	1	4	5.3000E+00
fo7_ar5_1	7	1	1.2000E+06	nvs20	0	1	1.3869E+08	tls4	22	7	1.0000E+01
fo8_ar4_1	430	237	1.4000E+06	nvs21	0	1	-2.0000E-05	tls5	60	20	2.2500E+01
fo8_ar5_1	13	11	1.4000E+06	nvs23	1	2	-4.5480E+02	**tltr**	0	1	4.8073E+01
fo8	1,330	532	1.4000E+06	nvs24	1	3	-5.3620E+02	uselinear	51	1	1.9514E+03
fo9_ar3_1	417	186	1.6000E+06	o7_2	10	28	4.8000E+06	util	1	1	4.3393E+03
fo9_ar4_1	3,986	698	1.6000E+06	o7_ar25_1	2,987	879	4.8000E+06	var_con10	10	4	4.6317E+02
fo9_ar5_1	15	1	1.6000E+06	o7_ar3_1	7	1	4.8000E+06	var_con5	7	2	3.1517E+02
fo9	196	152	1.6000E+06	o7_ar4_1	136	162	4.8000E+06	water4	5	6	3.3353E+06
fuel	0	1	1.5155E+04	o7_ar5_1	8	1	4.8000E+06	waterx	0	1	3.3362E+06
gasnet	62	1	1.0246E+07	o8_ar4_1	622	235	8.2000E+06	waterz	3	4	3.3555E+06
gbd	0	1	3.7264E+00	o9_ar4_1	3,953	697	8.2000E+06				
gear2	0	1	7.3226E-01	oaer	0	1	-6.0000E-06				

tolerance to evaluate constraints is 1e-6, and the relative feasibility tolerance is 1e-3 (used if absolute feasibility test fails). The tabu list length was set not to a fixed value, but to a value which was inversely proportional to the number of integer variables of the instance, i.e., the number of values to be stored for each solution of the tabu list. The value was 60,000 divided by the number of integer variables. The actual mean value of the solutions stored in the tabu list

Table 2. Instances for which no feasible solution was found within the time limit (19/243)

deb10	fo9_ar2_1	lop97icx	nuclear49	tls12
ex1252	fo9_ar25_1	nuclear10a	product2	tls6
fo8_ar25_1	gastrans	nuclear49a	space960	tls7
fo8_ar3_1	lop97ic	nuclear49b	tln12	

Table 3. Instances with numerical problems during the execution (25/243)

4stufen	fo_7_ar_2_1	nuclear104	o7	super2
beuster	fo7	nuclear10b	pump	super3
cecil_13	fo_8_ar_2_1	nvs05	risk2b	super3t
eg_int_s	minlphix	nvs22	st_e40	waste
ex1252a	no_7_ar_2_1	o_7_ar_2_1	super1	windfac

for the instances of the test bed was 35. The NLP solver used is IPOPT 3.5 trunk [23]. As a test set we use 243 instances taken from MINLPLib [28] (all those used in [12] excluding `oil` and `oil2` because the `log10` function is not supported by ROSE). We found an MINLP feasible solution for 199 of the instances as reported in Table 1. For each instance we report the CPU time (in seconds) and the number of iterations needed to find the (first) feasible solution (0 if it was found in less than 1 second) and the objective function value of such a solution. For 15 of the 199 solved instances, the algorithm found a feasible solution whose value is equal to the best-known solution reported in `http://www.gamsworld.org/minlp/minlplib/` within a 0.1% tolerance. The names of these instances are marked in boldface in Table 1. The instances for which the time limit is reached without finding any MINLP feasible solution are 19 and their names are reported in Table 2. The remaining 25 instances encounter some numerical problems during the execution (see Table 3). In general, finding a MINLP feasible solution for 199 of these 243 very difficult instances can be considered a very good performance for our algorithm. Of course the algorithm can be highly improved by taking into account the quality of the solution. First of all there is the possibility of continuing the execution of the algorithm instead of stopping it when the first feasible solution is found. Moreover, in most of the subproblems we solve, the original objective function is completely neglected. Using it in some way, i.e., combining it with the objective functions of subproblems $(P1)$ and $(P2)$ or adding a constraint which limits the value of the objective function, might lead to an improvement of the quality of the solutions obtained with the proposed algorithm. That would be in the spirit of [4].

5 Conclusions

In this paper we presented a Feasibility Pump algorithm aimed at solving nonconvex Mixed Integer Nonlinear Programming problems. The proposed algorithm is tailored to limit the impact of the nonconvexities in the MINLPs. These difficulties are extensively discussed. The preliminary results show that the algorithm behaves well with general problems on instances taken from MINLPLib.

References

1. Glover, F., Kochenberger, G. (eds.): Handbook of Metaheuristics. Kluwer Academic Publishers, Dordrecht (2003)
2. Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Mathematical Programming 104, 91–104 (2004)
3. Bertacco, L., Fischetti, M., Lodi, A.: A feasibility pump heuristic for general mixed-integer problems. Discrete Optimization 4, 63–76 (2007)
4. Achterberg, T., Berthold, T.: Improving the feasibility pump. Discrete Optimization 4, 77–86 (2007)
5. Fischetti, M., Lodi, A.: Local branching. Mathematical Programming 98, 23–47 (2002)
6. Danna, E., Rothberg, E., Pape, C.L.: Exploiting relaxation induced neighborhoods to improve MIP solutions. Mathematical Programming 102, 71–90 (2005)
7. Duran, M., Grossmann, I.: An outer-approximation algorithm for a class of mixed-integer nonlinear programs. Mathematical Programming 36, 307–339 (1986)
8. Bonami, P., Cornuéjols, G., Lodi, A., Margot, F.: A feasibility pump for mixed integer nonlinear programs. Mathematical Programming 119, 331–352 (2009)
9. Abhishek, K.: Topics in Mixed Integer Nonlinear Programming. PhD thesis, Lehigh University (2008)
10. Bonami, P., Gonçalves, J.: Primal heuristics for mixed integer nonlinear programs. Technical report, IBM Research Report RC24639 (2008)
11. Vavasis, S.: Nonlinear Optimization: Complexity Issues. Oxford University Press, Oxford (1991)
12. Liberti, L., Nannicini, G., Mladenovic, N.: A good recipe for solving MINLPs. In: Maniezzo, V., Stützle, T., Voß, S. (eds.) Matheuristics. Annals of Information Systems, vol. 10, pp. 231–244. Springer, US (2008)
13. Nannicini, G., Belotti, P., Liberti, L.: A local branching heuristic for MINLPs. ArXiv, paper 0812.2188 (2009)
14. Fletcher, R., Leyffer, S.: Solving mixed integer nonlinear programs by outer approximation. Mathematical Programming 66, 327–349 (1994)
15. Fletcher, R., Leyffer, S.: Numerical experience with lower bounds for MIQP branch-and-bound. SIAM Journal of Optimization 8, 604–616 (1998)
16. Westerlund, T., Pörn, R.: Solving pseudo-convex mixed integer optimization problems by cutting plane techniques. Optimization and Engineering 3, 235–280 (2002)
17. Westerlund, T.: Some transformation techniques in global optimization. In: Liberti, L., Maculan, N. (eds.) Global Optimization: from Theory to Implementation, pp. 45–74. Springer, Berlin (2006)
18. ARKI Consulting and Development. SBB Release Notes (2002)
19. Abhishek, K., Leyffer, S., Linderoth, J.: FilMINT: An outer-approximation based solver for nonlinear mixed-integer programs. Technical report, Argonne National Laboratory (2007)
20. Fischetti, M., Salvagnin, D.: Feasibility pump 2.0. Technical report, DEI, University of Padova (September 2008)
21. Schoen, F.: Two-phase methods for global optimization. In: Pardalos, P., Romeijn, H. (eds.) Handbook of Global Optimization, vol. 2, pp. 151–177. Kluwer Academic Publishers, Dordrecht (2002)
22. Fourer, R., Gay, D., Kernighan, B.: AMPL: A Modeling Language for Mathematical Programming, 2nd edn. Duxbury Press/Brooks/Cole Publishing Co. (2003)

23. Wächter, A., Biegler, L.T.: On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. Mathematical Programming 106, 25–57 (2006)
24. Ilog-Cplex (v. 11.0), http://www.ilog.com/products/cplex
25. D'Ambrosio, C., Frangioni, A., Liberti, L., Lodi, A.: On Interval-subgradient and No-good Cuts. Technical report, Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna (2009)
26. Liberti, L., Cafieri, S., Tarissan, F.: Reformulations in mathematical programming: a computational approach. In: Abraham, A., Hassanien, A.E., Siarry, P., Engelbrecht, A. (eds.) Foundations of Computational Intelligence. SCI, vol. 3, pp. 153–234. Springer, Berlin (2009)
27. Liberti, L.: Writing global optimization software. In: Liberti, L., Maculan, N. (eds.) Global Optimization: from Theory to Implementation, pp. 211–262. Springer, Berlin (2006)
28. Bussieck, M., Drud, A., Meeraus, A.: MINLPLib - a collection of test models for mixed-integer nonlinear programming. INFORMS Journal on Computing 15, 114–119 (2003)

Paging Multiple Users in Cellular Network: Yellow Page and Conference Call Problems

Amotz Bar-Noy[1,2], Panagiotis Cheilaris[3], and Yi Feng[1]

[1] Department of Computer Science, The Graduate Center of City University of New York, 365 Fifth Avenue, New York, NY 10016
[2] Department of Computer and Information Sciences, Brooklyn College of City University of New York, 2900 Bedford Avenue, Brooklyn, NY 11210
[3] Center for Advanced Studies in Mathematics, Department of Mathematics, Ben-Gurion University of the Negev, P.O. Box 653, Be'er Sheva 84105, Israel

Abstract. Mobile users are roaming in a zone of cells in a cellular network system. The probabilities of each user residing in each cell are known, and all probabilities are independent. The task is to find any one, or all, of the users, by paging the cells in a predetermined number of rounds. In each round, any subset of the cells can be paged. When a cell is paged, the list of users in it is returned. The paging process terminates when the required user(s) are found. The objective is to minimize the expected number of paged cells. Finding any one user is known as the yellow page problem, and finding all users is known as the conference call problem. The conference call problem has been proved NP-hard, and a polynomial time approximation scheme exists. We study both problems in a unified framework. We introduce three methods for computing the paging cost. We give a hierarchical classification of users. For certain classes of users, we either provide polynomial time optimal solutions, or provide relatively efficient exponential time solutions. We design a family of twelve fast greedy heuristics that generate competitive paging strategies. We implement optimal algorithms and non-optimal heuristics. We test the performance of our greedy heuristics on many patterns of input data with different parameters. We select the best heuristics for both problems based on our simulation. We evaluate their performances on randomly generated Zipf and uniform data and on real user data.

Keywords: cellular networks, paging, location management, experimental analysis of algorithms, heuristics.

1 Introduction

In cellular network systems, when a call arrives for a user, the system must know in which cell the user is located in order to establish a connection. Such a locating process is usually conducted by paging.

Let a user roam in a set of N cells: $\{C_1, \ldots, C_N\}$. With probability p_n the user is located in cell C_n. All p_is are independent. The system pages the cells in rounds. Once the cell that contains the user is paged, it reports to the system and the paging process terminates. To ensure the quality of service, the paging process must be conducted in at most D rounds. In each round, any subset of the cells could be paged. Thus, a paging strategy is an ordered D-partition of the cells, such that, in the dth round, the cells in dth part are paged. The *cost* of a paging strategy is the expected number of cells paged until the user is located. Our objective is to design algorithms that compute paging strategies that minimize paging costs.

In a variety of applications, e.g., to establish a conference call, we need to mutually page multiple users in a cellular network system. Suppose M users roam in a set of N cells, $\{C_1, \ldots, C_N\}$. With probability $p_{m,n}$, user m is located in cell C_n. All probabilities $p_{m,n}$s are independent. Our goal is find any one, some, or all of the users in at most D rounds. When a cell is paged, we become aware of the list of user(s) that reside in it. The paging process terminates as soon as the desired user(s) are found. The same objective remains: to minimize the expected number of paged cells.

In the multiple user paging problem, on one extreme, we want to find all the users so that a conference call can be established. We call this the *conference call* problem. On the other extreme, we only need to find any one of the users, no matter who. This is similar to when we look for information in a yellow page book: We stop after finding the first useful information in a category. We call this the *yellow page* problem. In this paper, we study both the yellow page and conference call problems, showing a kind of duality between the two problems.

Example: Suppose 2 users roam in 3 cells, C_1, C_2, C_3, with probabilities 0.5, 0.3, 0.2 (user 1) and 0.4, 0.1, 0.5 (user 2), respectively. For the paging strategy that pages C_1 and C_2 in the first round and C_3 in the second round, if we only search for user 1, by probability $(0.5+0.3)$ we page 2 cells; by probability 0.2 that we fail in the first round, we page 3 cells. The paging cost is $(0.5+0.3)\cdot 2+0.2\cdot 3 = 2.2$. For the same paging strategy, in the conference call problem, the probability that both users are in cells C_1, C_2 (i.e., paging stops after the first round and only 2 cells are paged) is $(0.5+0.3)\cdot(0.4+0.1) = 0.4$. Otherwise, all 3 cells are paged. The expected cost is $0.4\cdot 2 + (1-0.4)\cdot 3 = 2.6$. Again for the same strategy, in the yellow page problem, both users are in C_3 with probability $0.2\cdot 0.5 = 0.1$, in which case no user is found in the first round and we page all 3 cells. Otherwise, we only page the first 2 cells. The expected cost is $0.1\cdot 3 + (1-0.1)\cdot 2 = 2.1$.

Motivation: The cost of updating locations by users in Cellular Networks could be very expensive if users update their location each time they move from cell to cell. As a result, many systems partition the cells into zones where users report their new locations only when they enter a different zone. To locate a user, the system needs to page it in the zone of cells where it last reports its location. This scheme is part of one of the *location management* solutions (see survey [1]) where the paging step described above is a common task. It follows that any a priori knowledge of user locations, that can either be provided by the users or can be extracted from log files, will help the system to reduce the expected paging cost. There are several other applications to the paging problem. In wireless sensor networks, the system collects information from sensors by probing them. Such probes are costly (batteries of sensors) and therefore the system needs to arrange efficient information collection strategy such that the number of probes is minimized. Another application is the task of searching information on the Internet where search engines consume computational resource by accessing Internet cache databases. Since search request comes massively and frequently, the search engine system needs to better schedule the resource accessing by processing multiple search requests together, and arrange them in a good schedule.

Previous work: The problem of searching a single user in cellular networks under delay constraint D has been studied in [10,13,12,3]. An efficient, $O(ND)$ time, algorithm computing the optimal cost solution exists based on dynamic programming. The papers [14,9] considered the task of searching multiple users but as a collection of many single user search tasks that occur concurrently and therefore finding each one of the users is a success.

The conference call problem was introduced in [4]. The authors proved NP-hardness of the problem, and showed a natural greedy algorithm is an $\frac{e}{e-1}$-approximation. In [8], the authors introduced a PTAS to the conference call problem for any constant number of paging rounds. In [5], the authors studied the conference call problem with an additional bandwidth constraint, such that in each round, a limited number of users in each cell can be paged. The NP-hardness of the problem was shown, and a fast heuristic that minimized both delay constraint and paging cost was presented.

In [7], the authors explored another version of the conference call problem: instead of paging a cell and collecting all users in it, the system queries a cell by asking if a specific user is in it, and gets a boolean answer. The authors showed hardness and provided approximation algorithms in this setting.

The yellow page problem has not been studied to the best of our knowledge in the context of partitioning and scheduling. In [11], the authors explored a more general dynamic version of the problem. They showed it is NP-hard and provided a 4-approximation algorithm. In [6], the authors studied a similar problem. The problem differs from the yellow page problem in the parameter of number of users M. They proved its NP-hardness and provided an efficient approximation algorithm. In [15], the author discussed the yellow page problem in the context of efficiently finding alternative investment and provided heuristical solutions in a continuous model (in contrast to our discrete model).

Our contribution: In the problem of paging multiple users in a cellular network, computing the paging cost itself becomes an important task and is essential to understand the problem. We present three methods for computing the paging cost: two of them will be used in proving lemmas and constructing optimal algorithms, the other (the most efficient) one will be used in heuristics. We conjecture that, in addition to the conference call problem, the yellow page problem is also NP-hard, therefore we give a hierarchical classification of users and provide efficient optimal algorithms for certain interesting classes of users. When the delay constraint equals the number of cells, i.e., $D = N$, we present polynomial time algorithms for monotonic users and disjointed users, which are very representative cases for some applications. The optimal solution in the $D = N$ case is a permutation of the cells and thus a naive optimal algorithm has time complexity on the order of $N!$. However, by exploiting properties of the optimal solution, we design instead a dynamic programming algorithm of time complexity on the order of 2^N. This algorithm will be later used as a benchmark to evaluate our heuristics. In addition to optimal algorithms, we design a family of twelve greedy heuristics that belong to four groups. To evaluate their performance, we test them on several types of data with many settings and parameters. We also test them on real data with 171929 appearances of 996 users in 5625 cells in 31 consecutive days, acquired from a cell phone provider. We find the best heuristic for each problem that outperform the other heuristics on almost all instances. We also measure the running time of our algorithms on a real machine in addition to the theoretical analysis.

Open problems: The NP-hardness proof of the yellow page problem is still open. A natural generalization is to efficiently find k out of M users and remains open. In this paper, we assume that the costs of paging cells are all the same. However, they may vary from cell to cell due to different level of congestions. Another generalization, is to compute good paging strategies when cells have arbitrary paging costs. Finally, in the conference call problem, our paging strategies are static since they are predetermined before the paging process starts. In a dynamic setting, we may select the cells to be paged in the next round according to the users found in previous rounds.

2 Preliminaries

Let M users roam N cells and $p_{m,n}$ be the probability of user m being in cell C_n. Given a bound D on the number of rounds (with $1 \leq D \leq N$), a paging strategy $\mathcal{A} = \langle A_1, \ldots, A_D \rangle$ is an ordered partition of the set of cells $\{C_1 \ldots C_N\}$, such that, in the dth round, cells in part A_d are paged. Given a paging strategy $\mathcal{A} = \langle A_1, \ldots, A_D \rangle$, let $P_{m,d}$ be the probability of user m being in any cell in part A_d, i.e., $P_{m,d} = \sum_{C_n \in A_d} p_{m,n}$. Denote the suffix probability by $R_{m,d} = \sum_{i=d+1}^{N} P_{m,i}$ and the prefix probability by $Q_{m,d} = \sum_{i=1}^{d} P_{m,i}$. Let S_d be the number of cells in the first d parts, A_1, …, A_d. By convention, $S_0 = 0$.

We describe three methods that compute the paging cost of the yellow page problem and the conference call problem. The first two are used in our proofs while the third is used by our simulations since it is computationally the most efficient. Let $\mathrm{YP}(\mathcal{A})$ be the cost of the yellow page problem and $\mathrm{CC}(\mathcal{A})$ be the cost of the conference call problem, on paging strategy $\mathcal{A}$. Consider the vector $\mathbf{d} = (d_1, \ldots, d_M) \in \{1, \ldots, D\}^M$, which encodes in which part each user is (i.e., user m is in part d_m, for $1 \leq m \leq M$).

Combinatorial Computation: For a part location vector $(d_1, \ldots, d_M)$, which occurs with probability $\prod_{m=1}^{M} P_{m,d_m}$, the strategy pays a cost of $S_{\min\{d_1,\ldots,d_M\}}$ for the yellow page problem (i.e., it pages in parts until it finds the first part that contains some user) and therefore:

$$\mathrm{YP}(\mathcal{A}) = \sum_{\mathbf{d}\in\{1,\ldots,D\}^M} \left(S_{\min\{d_1,\ldots,d_M\}} \cdot \prod_{m=1}^{M} P_{m,d_m} \right) . \tag{1}$$

Similarly, the cost of the conference call problem is:

$$\mathrm{CC}(\mathcal{A}) = \sum_{\mathbf{d}\in\{1,\ldots,D\}^M} \left(S_{\max\{d_1,\ldots,d_M\}} \cdot \prod_{m=1}^{M} P_{m,d_m} \right) , \tag{2}$$

where the only difference is that for each part location vector, the strategy pays a cost of $S_{\max\{d_1,\ldots,d_M\}}$, because it has to page also all cells in the last part that contains a user.
Time Complexity: $\Theta(MN + (M+D)D^M)$.

Recursive computation: In the yellow page problem, we extend the definition of cost, so that $\mathrm{YP}(\langle A_d,\ldots,A_D\rangle)$ is the expected cost of paging parts $\langle A_d,\ldots,A_D\rangle$ given the condition that no user is found in parts $A_1,\ldots,A_{d-1}$. If no user has been found in the first $(D-1)$ rounds, we must page all the cells in A_D, i.e., $\mathrm{YP}(\langle A_D\rangle) = |A_D|$. The recursion step is

$$\mathrm{YP}(\langle A_d,\ldots,A_D\rangle) = |A_d| + \frac{\prod_{m=1}^{M} R_{m,d}}{\prod_{m=1}^{M} R_{m,d-1}} \mathrm{YP}(\langle A_{d+1},\ldots,A_D\rangle) \ , \tag{3}$$

because to page users in $A_d,\ldots,A_D$ given that no users are in $A_1,\ldots,A_{d-1}$, we must page cells in A_d by paying a cost of $|A_d|$ and if no user is found there (an event with probability $\prod_{m=1}^{M} R_{m,d}/\prod_{m=1}^{M} R_{m,d-1}$) we pay an extra cost of $\mathrm{YP}(\langle A_{d+1},\ldots,A_D\rangle)$.

Let $\mathrm{CC}(\langle A_1,\ldots,A_d\rangle)$ be the conference call cost of paging all users in parts $A_1,\ldots,A_d$. The recursion base is $\mathrm{CC}(\langle A_1\rangle) = \prod_{m=1}^{M} Q_{m,1}|A_1|$, since if all users are in cells of part A_1, we page $|A_1|$ cells. The recursion step is

$$\mathrm{CC}(\langle A_1,\ldots,A_d\rangle) = \mathrm{CC}(\langle A_1,\ldots,A_{d-1}\rangle) + \left(\prod_{m=1}^{M} Q_{m,d} - \prod_{m=1}^{M} Q_{m,d-1} \right) |A_d|, \tag{4}$$

because to page all users in parts $A_1,\ldots,A_d$, we must page parts $A_1,\ldots,A_{d-1}$ first and pay a cost of $\mathrm{CC}(\langle A_1,\ldots,A_{d-1}\rangle)$, and with probability that at least one user is in A_d, we pay an extra cost of $|A_d|$.
Time Complexity: $\Theta(MN + MD)$.

Exclusive Computation: With probability $\prod_{m=1}^{M} R_{m,d-1}$, all users are in parts $\{A_d \ldots A_D\}$; with probability $\prod_{m=1}^{M} R_{m,d}$, all users are in parts $\{A_{d+1} \ldots A_D\}$. Thus, with the difference of the above probabilities, at least one user is in part A_d but no user is in parts $A_1 \ldots A_{d-1}$, in which case we need to page exactly S_d cells.

Summing through $d = 1, \ldots, D$, we have the cost for the yellow page problem.

$$\mathrm{YP}(\mathcal{A}) = \sum_{d=1}^{D} \left(S_d \cdot \left(\prod_{m=1}^{M} R_{m,d-1} - \prod_{m=1}^{M} R_{m,d} \right) \right) \tag{5}$$

Similarly, in the conference call problem, with probability $\prod_{m=1}^{M} Q_{m,d}$, all users are in parts $\{A_1 \ldots A_d\}$ and with probability $\prod_{m=1}^{M} Q_{m,d-1}$, all users are in parts $\{A_1 \ldots A_{d-1}\}$. Thus, with the difference of the above probabilities, at least one user is in part A_d and all users are in parts $A_1 \ldots A_d$, in which case we need to page exactly S_d cells.

$$\mathrm{CC}(\mathcal{A}) = \sum_{d=1}^{D} \left(S_d \cdot \left(\prod_{m=1}^{M} Q_{m,d} - \prod_{m=1}^{M} Q_{m,d-1} \right) \right) \tag{6}$$

Time Complexity: $\Theta(MN + MD)$

Types of Users: In [4], the authors proved that the conference call problem is NP-hard. We conjecture that the yellow page problem is also NP-hard. In Sec.3, we observe some "duality" between the two problems. Since the general setting is hard to tackle, we study some interesting restricted classes of instances, for which we provide more efficient optimal solutions – some have polynomial running time and some have improved exponential running time. Toward that goal, we present a hierarchical classification of types of users.

We define a few properties for a set of M users according to their probabilities in the set of cells. A set of users is *identical* if for any cell $C_n \in \{C_1, \ldots, C_N\}$, $p_{1,n} = \cdots = p_{M,n}$. A set of users is *uniform* if for each user $m = 1 \ldots M$, $p_{m,n}$ is either 0 or $1/k$, where k is the number of non-zero entries in $\{p_{m,1}, \ldots, p_{m,N}\}$. A set of users is *similar*, if for all users $m = 2, \ldots, M$, $\{p_{m,1}, \ldots, p_{m,N}\}$ is some permutation of user 1's probabilities $\{p_{1,1}, \ldots, p_{1,N}\}$. A set of users is *disjoint* if for each cell $C_n \in \{C_1, \ldots, C_N\}$, there is exactly one non-zero entry $p_{m,n}$, for some $m = 1, \ldots, M$.

3 Optimal Solutions

In this section, we first present efficient algorithms for both problems to compute the paging cost for a predetermined order of the cells. Based on these algorithms, we describe relatively efficient optimal solutions to some types of users. The proofs of our lemmas and theorems can be found in our technical report [2].

Given an order of the cells, say $\langle C_1, \ldots, C_N \rangle$ (without loss of generality), a paging strategy $\mathcal{A} = \langle A_1, \ldots, A_D \rangle$ is said to *respect the above order* if for any cells C_i, C_j with $i < j$, we have $C_i \in A_{d_i}$ and $C_j \in A_{d_j}$ with $d_i \leq d_j$. Given an order of the cells, Algorithm 1 compute the optimal paging cost and corresponding strategy that respects this order, in polynomial time.

In Algorithm 1, for the yellow page problem, let $h^{\mathrm{yp}}_{n,d}$ denote the optimal cost of paging cells $\{C_n, \ldots, C_N\}$ in d rounds given the condition that no user locates in cells $\{C_1, \ldots, C_{n-1}\}$. Our objective is to find $h^{\mathrm{yp}}_{1,D}$. It is not difficult to see that $h^{\text{yp}}_{n,1} = N - n + 1$. To compute $h^{\text{yp}}_{n,d}$, we need to search through all possible js with $n+1 \leq j \leq n-d+1$ for the strategy of minimum cost that pages cells $\{C_j, \dots, C_N\}$ in the last $(d-1)$ rounds and pages cells $\{C_n, \dots, C_{j-1}\}$ in the previous round; the inner loop $(*)$ follows equation (3)). For the conference call problem, let $h^{\text{cc}}_{n,d}$ denote the optimal cost of paging cells $\{C_1, \dots, C_n\}$ in d rounds. Our objective is to find $h^{\text{cc}}_{N,D}$. It is not difficult to see that $h^{\text{cc}}_{n,1} = n$. To compute $h^{\text{cc}}_{n,d}$, we need to search through all possible js with $(d-1) \leq j < n$ for the strategy of minimum cost that pages cells $\{C_1, \dots, C_j\}$ in the first $(d-1)$ rounds and pages cells $\{C_{j+1}, \dots, C_n\}$ in the dth round; the inner loop $(*)$ follows equation (4).

Algorithm 1. Dynamic programming algorithm, respect order $\langle C_1, \dots, C_N \rangle$, cost $= \text{DP}(p[M][N], D)$

```
for $n = 1 \dots N$ do
  YP: $h^{\text{yp}}_{n,1} \leftarrow N - n + 1$
  CC: $h^{\text{cc}}_{n,1} \leftarrow n$
for $d = 2 \dots D$ do
  for $n = 1 \dots (N-d)$ do
    YP: $h^{\text{yp}}_{n,d} \leftarrow \min_{j=n+1}^{n-d+1} |j-n| + (\prod_{m=1}^{M} R_{m,n} - \prod_{m=1}^{M} R_{m,j}) / \prod_{m=1}^{M} R_{m,n} \cdot h^{\text{yp}}_{j,d-1}$
    CC: $h^{\text{cc}}_{n,d} \leftarrow \min_{j=d-1}^{n-1} h^{\text{cc}}_{j,d-1} + \left(\prod_{m=1}^{M} Q_{m,n} - \prod_{m=1}^{M} Q_{m,j}\right) |n-j|$ $(*)$
return YP:$h^{\text{yp}}_{1,D}$ / CC:$h^{\text{cc}}_{N,D}$
```

The correctness of Algorithms 1 follows from the fact that, under the particular order constraint, any sub-partition of an optimal paging strategy must be sub-optimal within itself; otherwise, replacing the sub-partition with the alternative sub-optimal paging strategy would gain a better paging strategy than optimal. We omit details of a proof here, but a formal proof can be adapted from [12].

Lemma 1. *The running time of Algorithm 1 is $\Theta(MDN^2)$*

Monotonic Users: A set of users is called *monotonic* if there is a permutation of cells, w.l.o.g., say $\langle C_1, \dots, C_N \rangle$, such that $p_{m,1} \geq \dots \geq p_{m,N}$ for every $m \in \{1, \dots, M\}$. Let this permutation be the monotonic order of the cells for the monotonic users.

Lemma 2. *For monotonic users, the optimal paging strategies for both the yellow page and conference call problems follow the monotonic order of the cells.*

Applying Algorithm 1 on the monotonic order yields:

Corollary 1. *The optimal paging strategies for both the yellow page problem and the conference call problem for monotonic users can be computed in polynomial time for any D, M, and N.*

D=N, Duality: An interesting case is when $D = N$, i.e., *sequential* searching, in which a paging strategy is a permutation of the cells. The conference call problem has been proved NP-Hard in [4]. Although we have not yet proved the NP-hardness for yellow page problem, we show that there is some kind of *duality* between the two problems.

Lemma 3. *Let $\mathcal{A}$ be a paging strategy that pages M users in N cells in $D = N$ rounds. Let an instance of yellow page with probabilities $p_{m,n}$. Let another instance for the conference problem with probabilities $q_{m,n} = p_{m,N+1-n}$. Then, $YP(\mathcal{A}, p_{m,n}) + CC(\mathcal{A}, q_{m,n}) = N + 1$.*

Corollary 2. *When $D = N$, the maximization problem of yellow page is equivalent to the minimization problem of conference call, and vice versa.*

D=N, Arbitrary User: When $D = N$, a brute-force approach to find the optimal strategy is to test all permutations of the cells. This method requires $\Theta(N!)$ time. We present lemmas, which allow us to give instead a $\Theta(2^N)$ algorithm that generates the optimal permutation.

Lemma 4. *In the yellow page problem, if $\langle C_{i_1}, \dots C_{i_n} \rangle$ is an optimal paging strategy of paging cells $\{C_{i_1}, \dots C_{i_n}\}$ given the condition that no user is located in the rest of cells, then any suffix of it, $\langle C_{i_k}, \dots C_{i_n} \rangle$, for $1 \leq k \leq n$, is an optimal paging strategy that pages cells $\langle C_{i_k}, \dots C_{i_n} \rangle$ given no user is located in any other cells (except in $\{C_{i_k}, \dots C_{i_n}\}$).*

Similarly, we have the following lemma for the conference call problem:

Lemma 5. *In the conference call problem, if $\langle C_{i_1}, \dots C_{i_n} \rangle$ is an optimal paging strategy of paging cells $\{C_{i_1}, \dots C_{i_n}\}$ in the first i_n rounds, then any prefix of it, $\langle C_{i_1}, \dots C_{i_k} \rangle$, for $1 \leq k \leq n$, is an optimal paging strategy that pages cells in $\{C_{i_1}, \dots C_{i_k}\}$ in the first i_k rounds.*

In light of Lemmas 4 and 5, Algorithm 2 computes an optimal paging strategy. A dedicated array Best$[2^N]$ is used in the algorithm. Best$[k]$ records the optimal sub-strategy of paging cells $\{C_{i_1}, \dots, C_{i_l}\}$ where $i_1, \dots, i_l$ are the bits of 1 after converting k into binary. For the yellow page problem, in the first *for* loop, we initialize the optimal paging strategy of a single cell given no user is found in other cells which is to page the cell itself in the only around. For paging l cells in l rounds, we search through all possible cases that page one of the l cells in the first round, and page the other $(l-1)$ cells optimally in the remaining $(l-1)$ rounds. The data structure Best is set up for random access any optimal sub-strategy that has been already computed. Similarly, we construct the optimal algorithm for the conference call problem. The only difference with the yellow page algorithm is the recursive computation of the cost according to (4).

Theorem 1. *The time complexity of Algorithm 2 is $\Theta(MN \cdot 2^N)$. The space complexity is $\Theta(N \cdot 2^N)$.*

Algorithm 2. $D = N$, arbitrary users: compute the optimal cost and strategy using dynamic programming; opt = DP(A)

```
for $\forall A$, that $|A| = 1$ do
    YP: $\text{Best}[A]_{\text{cost}} \leftarrow 1$; $\text{Best}[A]_{\text{strategy}} \leftarrow \langle A \rangle$
    CC: $\text{Best}[A]_{\text{cost}} \leftarrow \prod_{m=1}^{M} P_{m,A}$; $\text{Best}[A]_{\text{strategy}} \leftarrow \langle A \rangle$
for $\forall A$ that $|A| = 2 \dots N$ do
    YP: $\text{Best}[A]_{\text{cost}} \leftarrow \min_{C_i \in A}\{1 + \frac{\prod_{m=1}^{M} \sum_{C_n \in A \setminus C_i} p_{m,i}}{\prod_{m=1}^{M} \sum_{C_n \in A} p_{m,i}} \cdot \text{Best}[A \setminus C_i]_{\text{cost}}\}$
    YP: $\text{Best}[A]_{\text{strategy}} \leftarrow \langle \arg\min\{C_i | \text{Best}[A \setminus C_i]_{\text{cost}}\}, \text{Best}[A \setminus C_i]_{\text{strategy}} \rangle$
    CC: $\text{Best}[A]_{\text{cost}} \leftarrow \min_{C_i \in A}\{\text{Best}[A \setminus C_i]_{\text{cost}} + \left(\prod_{m=1}^{M} P_A - \prod_{m=1}^{M} P_{A \setminus C_i}\right) \cdot |A|\}$
    CC: $\text{Best}[A]_{\text{strategy}} \leftarrow \langle \arg\min\{\text{Best}[A \setminus C_i]_{\text{cost}}, \text{Best}[A \setminus C_i]_{\text{strategy}}, C_i\} \rangle$
return $\{\text{Best}[A] |$ that $|A| = N\}$
```

D=N, Disjoint users: For disjoint users we can reduce the running time of optimal algorithm to $O(N^M)$ based on the following lemma.

Lemma 6. *For disjoint users, in an optimal strategy, for every user, the cells where the user is located must be paged by order of non-increasing probability.*

Based on the above partial order of optimal paging strategies, when $D = N$, we can compute an optimal paging strategy for disjoint users in $\Theta(N^M \text{Poly}(M, N))$ time using dynamic programming techniques.

4 Experiments

We design a family of 12 heuristics that compute good paging strategies in practice. All our heuristics are of the following form: First, we compute an order of the cells (according to some greedy method) and then, we apply Algorithm 1 to find the best strategy that follows this order of the cells. We have four criteria to order the cells. Define $X_n = \prod_{m=1}^{M} p_{m,n}$ (the probability all users are in cell C_n). Define $Y_n = \prod_{m=1}^{M}(1 - p_{m,n})$ (the probability no user is in cell C_n). Define $S_n = \sum_{m=1}^{M} p_{m,n}$ (the sum of user probabilities being in cell C_n). Define $Z_n = \max_{m=1}^{M} p_{m,n}$ (the maximum user probability in cell C_n). Heuristics X, Y, S and Z page the cells in the orders $X_1 \geq \dots \geq X_N$, $Y_1 \leq \dots \leq Y_N$, $S_1 \geq \dots \geq S_N$[1] and $Z_1 \geq \dots \geq Z_N$, respectively.

We use the above four basic heuristics to compute paging strategies for both problems. For each basic greedy heuristic $\mathcal{G} \in \{X, Y, Z, S\}$, we design two adaptive versions, $BF\mathcal{G}$ and $WL\mathcal{G}$. In the *Best First* (BF) version, each time we select a cell to form an order, we select the next available cell that has the best value (max. for X, S, Z and min. for Y), and then normalize the probabilities among unselected cells. In the *Worst Last* (WL) version, we select the available cell that has the worst value (min. for X, S, Z and max. for Y) as the last cell in the order, and then normalize probabilities among unselected cells.

[1] [4] Showed Y is an $\frac{e}{e-1}$-approximation for any $D \leq N$.

Select the Best Heuristics: We test our heuristics for both the yellow page and the conference call problem. For each problem, we check regular users and disjoint users. We also conduct our simulation on small instances and large instances with respect to number of cells. For small instances, we try all possible inputs (exhaustive search) up to some granularity (values of probabilities are integer multiples of some small value), then we compute strategies for Zipf and Uniform distributed user location data. We check the cases of $D = 2$ and $D = N$ (for efficient optimal algorithms). We compare the paging costs between different greedy heuristics and between greedy heuristics and OPT. For large instances, we test on Zipf and Uniform distributed data, for several values of D, and then we compare the paging costs between the heuristics. We measure the average and worst case ratio of costs on the instances that belongs to every setting and parameter combination and on real data. We use a kind of "voting system" to generate a ranking of heuristics. Details of the setup and results of our experiments can be found at [2].

Our results show that the ranking of heuristics is the same for large and small instances, when doing exhaustive search, for Zipf and uniform data, for average case and worst case measurement, and when comparing heuristics among themselves or when comparing heuristics with OPT. Table 1 shows the ranking of heuristics and of the three versions of the best heuristic (Y, BFY, WLY) for the two problems.

Table 1. Performance ranking of heuristics. $\mathcal{G} > \mathcal{H}$: $\mathcal{G}$ is consistently better than $\mathcal{H}$; $\mathcal{G} \geq \mathcal{H}$: $\mathcal{G}$ is better than or comparable to $\mathcal{H}$; $\mathcal{G} \sim \mathcal{H}$: $\mathcal{G}$ is comparable to $\mathcal{H}$.

Yellow Page	Conference Call
$Y \geq S > Z \geq X$	$Y \geq S > Z \geq X$
$BFY > Y \geq WLY$	$Y > BFY \sim WLY$

Next we evaluate the performance of our best greedy heuristics. Our simulation found some "bad" instances. Based on these instances, we are able to craft some examples that show lower bounds on the competitiveness of our heuristics for both problems [2].

Performances of Best Greedy Heuristics: We evaluate the performances of the best heuristics (BFY for yellow page and Y for conference call) on small instances, large instances. We measure their average case and worst case cost ratios over OPT for small instances and real data. We measure the average and worst case cost ratios of other heuristics over BFY or Y for large instances.

Tables 2(a), 2(c) and 2(d) show the results. We observe that our selected heuristics perform well in the worst case and average performance for all types of input data. We also present the results in bar chart in Fig. 7 in [2].

Simulation on Real Data: We obtain from a cellular phone company 996 users' 171929 appearances in 5625 cells in 31 consecutive days. For each user, we extract (from the above real data) the number of appearances in every cell. We randomly

Table 2. Cost Ratios of BFY for Yellow Page (YP) and Y for Conference Call (CC)

(a) Cost Ratios Over OPT: Small Instances

Problem	Average	Worst
YP (BFY)	1.00638	1.19415
CC (Y)	1.00173	1.03609

(b) Cost Ratios Over OPT: Real Data

Problem	Average	Worst
YP (BFY)	1.03352	1.54384
CC (Y)	1.00643	1.05930

(c) Cost Ratios Over BFY: Large Instances, YP

Heuristic	Average	Worst
BFX	1.19102	2.23126
BFZ	1.28360	2.51777
BFS	1.00003	1.00961

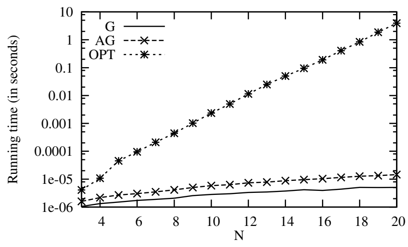
(d) Cost Ratios Over Y: Large Instances, CC

Heuristic	Average	Worst
X	1.15884	1.62346
Z	1.00626	1.03000
S	1.00000	1.00316

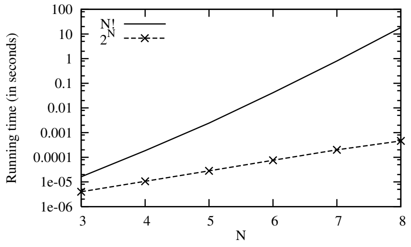
pick two ($M = 2$), three ($M = 3$), and four ($M = 4$) users and compute the probabilities $p_{m,n}$s. For each $M = 2, 3, 4$, we pick 10,000 instances as above. We conduct the same simulations as in the non-real data. We observe that each real user is close to Zipf distributed and users are almost disjoint. The results coincide with the results from non-real data as shown in Tables. 1 and 2(b).

Running Time: We compare the running time of our greedy heuristics and efficient optimal algorithms. For accurate time measurement, for each N, we run our algorithms for many iterations and compute the average running times.

For $D = 2$ and $D = N$, we measure the running time of the static version of the greedy heuristics G (i.e., Y) of complexity $\Theta(MDN \log N)$, the adaptive version of the greedy heuristics AG (i.e., BFY and WLY) of complexity $\Theta(MDN^2 \log N)$, and the D^N optimal algorithm of complexity $\Theta(MND^N)$. The results are similar. We show the case of $D = N$ in Fig. 1(a) In the two experiments, we observe a complexity hierarchy of the algorithms and a tradeoff between complexity and optimality. For $D = N$, we also compare the running time of the straight forward optimal $N!$ algorithm and fast 2^N algorithm. The result is in Fig. 1(b). As expected, the $N!$ algorithm is super exponential and



(a) $D = N$, heuristics and optimal algorithm



(b) $D = N$, two optimal algorithms

Fig. 1. Running time

the 2^N algorithm is comparably much faster, which allows us do experiments on larger problem instances.

Acknowledgement

The work of Amotz Bar-Noy and Yi Feng was partially sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. Yi Feng also likes to thank the Research Computing at the Graduate Center of CUNY for providing computational resources.

References

1. Akyildiz, I.F., Mcnair, J., Ho, J., Uzunalioglu, H., Wang, W.: Mobility management in next-generation wireless systems. In: Proc. IEEE, pp. 1347–1384 (1999)
2. Bar-Noy, A., Cheilaris, P., Feng, Y.: Paging multiple users in cellular network: Yellow page and conference call problems. Tech. Rep. TR-2010003, Department of Computer Science, The Gradaute Center of CUNY, New York, NY (2010)
3. Bar-Noy, A., Feng, Y., Golin, M.J.: Paging mobile users efficiently and optimally. In: Proc. IEEE Conference on Computer Communications, pp. 1910–1918 (2007)
4. Bar-Noy, A., Malewicz, G.: Establishing wireless conference calls under delay constraints. J. Algorithms 51(2), 145–169 (2004)
5. Bar-Noy, A., Naor, Z.: Efficient multicast search under delay and bandwidth constraints. Wireless Networks 12(6), 747–757 (2006)
6. Cohen, E., Fiat, A., Kaplan, H.: Efficient sequences of trials. In: Proc. of the ACM-SIAM symposium on Discrete algorithms (SODA), pp. 737–746 (2003)
7. Epstein, L., Levin, A.: The conference call search problem in wireless networks. Theor. Comput. Sci. 359(1-3), 418–429 (2006)
8. Epstein, L., Levin, A.: A PTAS for delay minimization in establishing wireless conference calls. Discrete Optimization 5(1), 88–96 (2008)
9. Gau, R.H., Haas, Z.J.: Concurrent search of mobile users in cellular networks. IEEE/ACM Trans. Netw. 12(1), 117–130 (2004)
10. Goodman, D.J., Krishnan, P., Sugla, B.: Minimizing queuing delays and number of messages in mobile phone location. Mobile Netw. and Appl. 1(1), 39–48 (1996)
11. Kaplan, H., Kushilevitz, E., Mansour, Y.: Learning with attribute costs. In: Proc. of ACM Symposium on Theory of Computing (STOC), pp. 356–365 (2005)
12. Krishnamachari, B., Gau, R.H., Wicker, S.B., Haas, Z.J.: Optimal sequential paging in cellular wireless networks. Wireless Netw. 10(2), 121–131 (2004)
13. Rose, C., Yates, R.D.: Minimizing the average cost of paging under delay constraints. Wireless Netw. 1(2), 211–219 (1995)
14. Rose, C., Yates, R.D.: Ensemble polling strategies for increased paging capacity in mobile communication networks. Wireless Netw. 3(2), 159–167 (1997)
15. Weitzman, M.L.: Optimal search for the best alternative. Econometrica 47(3), 641–654 (1979)

Realtime Classification for Encrypted Traffic

Roni Bar-Yanai[1], Michael Langberg[2,⋆], David Peleg[3,⋆⋆], and Liam Roditty[4]

[1] Cisco, Netanya, Israel
rbaryana@cisco.com
[2] Computer Science Division, Open University of Israel, Raanana, Israel
mikel@openu.ac.il
[3] Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel
david.peleg@weizmann.ac.il
[4] Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel
liamr@macs.biu.ac.il

Abstract. Classifying network flows by their application type is the backbone of many crucial network monitoring and controlling tasks, including billing, quality of service, security and trend analyzers. The classical "port-based" and "payload-based" approaches to traffic classification have several shortcomings. These limitations have motivated the study of classification techniques that build on the foundations of learning theory and statistics. The current paper presents a new statistical classifier that allows real time classification of encrypted data. Our method is based on a hybrid combination of the k-means and k-nearest neighbor (or k-NN) geometrical classifiers. The proposed classifier is both fast and accurate, as implied by our feasibility tests, which included implementing and intergrading statistical classification into a realtime embedded environment. The experimental results indicate that our classifier is extremely robust to encryption.

1 Introduction

Classifying network flows by their application type is the backbone of many crucial network monitoring and controlling tasks. Basic network management functions such as billing, quality of service, network equipment optimization, security and trend analyzers, are all based on the ability to accurately classify network traffic into the right corresponding application.

Historically, one of the most common forms of traffic classification has been the *port-based* classification, which makes use of the port numbers employed by the application on the transport layer. However, many modern applications use dynamic ports negotiation making *port-based* classification ineffective [10,17] with accuracy ranges between 30% and 70%.

The next step in the evolution of classification techniques was *Deep Packet Inspection* (DPI) or *payload-based* classification. DPI requires the inspection of

⋆ Supported in part by The Open University of Israel's Research Fund (grant no. 46109) and Cisco Collaborative Research Initiative (CCRI).

⋆⋆ Supported in part by Cisco Collaborative Research Initiative (CCRI).

the packets' payload. The classifier extracts the application payload from the TCP/UDP packet and searches for a signature that can identify the flow type. Signatures usually include a sequence of bytes/strings and offsets that are unique to the application and characterize it. DPI is widely used by today's traffic classifier vendors. It is very accurate [11,17] but suffers from a number of drawbacks.

Recently, we have witnessed a dramatic growth in the variety of network applications. Some of these applications are transmitted in an encrypted manner, posing a great challenge to the DPI paradigm. Such applications may choose to use encryption both for security and to avoid detection. Common P2P applications such as BitTorrent and eMule have recently added encryption capabilities (primarily to avoid detection). As a significant share of the total bandwidth is occupied by P2P applications and since current DPI based classifiers must see the packet's payload, encryption may become a real threat for ISP's in the near future. The inability of port-based and payload-based analysis to deal with the wide range of new applications and techniques used in order to avoid detection has motivated the study of other classification techniques. Two examples include *behavior* based classification and classification based on a combination of learning theory and statistics.

In the *behavioral* paradigm, traffic is classified by identifying a certain behavior that is unique to the application at hand. In this setting, the signature is not syntactic (as in DPI classification) but rather a combination of events. For example, it is possible to identify encrypted BitTorrent by intercepting the torrent file (a file used to start the downloading process in BitTorrent clients) [3]. The torrent file includes a list of BitTorrent hosts, each possessing certain parts of the downloaded file. The classifier processes the file and saves these hosts. Now, an encrypted flow that is destined to one of these hosts will be marked immediately as BitTorrent. The drawback of such behavioral solutions is that they are too specific, and it will not take long before the P2P community strikes back by encrypting torrent files [3].

This paper presents a new method for statistical classification. Our method is based on a hybrid combination of the well known k-means and k-nearest neighbor geometrical classifiers. The proposed classifier is both fast and accurate, as implied by our feasibility tests, which included implementing and intergrading our classifier into a realtime embedded environment. The experimental results indicate that our classifier is extremely robust to encryption and other DPI flaws, such as asymmetric routing and packet ordering. Finally, we show how to boost the performance of our classifier even further, by enhancing it with a simple *cache-based* mechanism that combines elements of port-based and statistical classification. In what follows, we elaborate on statistical classification in general and specify our contribution.

Related work. The statistical approach to classification is based on collecting statistical data on properties of the network flow, such as the mean packet size, flow duration, number of bytes, etc. The statistical paradigm relies on the assumption that each application has a unique distribution of properties that represents it and can be used to identify it. This approach has been the subject of intensive research in the recent years.

The work of Paxson [15] from 1994 established a relationship between flow application type and flow properties (such as the number of bytes and flow duration). A methodology for separating chat traffic from other Internet traffic, which uses statistical properties such as packet sizes, number of bytes, duration and inter arrival times of packets, was developed in [5]. Mcgregor et al. [12] explored the possibility of forming clusters of flows based on flow properties such as packet size statistics (e.g., minimum and maximum), byte count, and idle times etc. Their study used an *expectation maximization (EM)* algorithm to find the clusters' distribution density functions.

A study focusing on identifying flow application *categories* rather than specific individual applications was presented in [16]. While limited by a small dataset, they showed that the k-nearest neighbor algorithm and other techniques can achieve rather good results, correctly identifying around 95% of the flows. Zander *et al.* [18], using an EM based clustering algorithm, obtained an average success rate of 87% in the separation of individual applications. The basic Navie Bayes algorithm, enhanced by certain refinements, was studied by Moore *et al.* [13] and was shown to achieve an accuracy level of 95%.

Realtime classification, in which the flow is to be classified based mainly on its first few packets' size and direction, was addressed in [2,4,7]. It is important to note that these algorithms were tested only against basic application protocols. Encrypted BitTorrent and Gnuttela, for example, use packet padding in the beginning of the flow start, to avoid such detection methods. For more details on related work see [10,14].

Our contribution. The current paper introduces a hybrid statistical algorithm that integrates two basic and well known machine learning algorithms, known as k-nearest neighbors and k-means. The algorithm is fast, accurate and most important it is insensitive to encrypted traffic. Moreover, the strength of our algorithm is precisely in overcoming several weaknesses of the DPI approach, which is the leading technology used by current network classifiers. In particular, our algorithm overcomes asymmetric routing[1] and packet ordering. To the best of our knowledge, our study is the first to demonstrate the potential of statistical methods on encrypted traffic in realtime classification.

To put our results in perspective, we note that most previous statistical classification methods were tested in an off-line environment [14]. The results on realtime classification [2,4,7] are all based on inspecting the first five initial packets of the flow, and thus work well only when these packets represent the application under study. Note that encrypted Bittorrent and EMule, which use padding on their initial packets, cannot be classified using such techniques.

The strength of our algorithm is demonstrated on Encrypted BitTorrent, one of the hardest applications to identify. The BitTorrent development community puts a lot of effort into detection avoidance and uses port alternation, packet padding (on initial flow packets) and encryption as part of this effort. Actually, as

[1] Occurring when incoming and outgoing flows use different routers.

our algorithm is insensitive to encryption, it turns out that it identifies encrypted and non-encrypted BitTorrent flows with the exact same accuracy.

The data set used for the experiments reported in this paper was recorded in 2009 (full payload) on a real ISP network edge router on two different geographical locations, and contains millions of flows. The record is unique in its relevance and reflects the distribution and behavior of contemporary application flows. We integrated our statistical classifier into a realtime embedded environment. The feasibility test included a full implementation of our algorithm on SCE2020, which is one of the leading Cisco platforms specialized to classification. The algorithm was tested in full line rate, and the experiment has demonstrated that our algorithm can be implemented and integrated on platforms that are limited on resources, memory and cpu, and require realtime responses.

In addition, we show that using a simple LRU cache drastically reduces classification time and memory of more than 50% of the flows, and also increases the classification accuracy of some of the protocols.

2 Methodology

The general paradigm followed by our classifier is a machine learning one. Roughly speaking, we first build a training set and use it to train our classifier; we then turn to the task of classification. In what follows we briefly elaborate on the techniques we use to obtain labeled traffic (for our training set), we then address some special properties of the data sets we use.

Collecting labeled data. To train our classifier, we require a collection of reliably classified traffic flows[2]. We were provided such a database generated using Endace [6] for real-time traffic recording and injecting, and labeled using the Cisco SCE 2020 box (a professional tool for classifying and controlling network traffic) coupled with manual inspection and verification. The database included 12 million flows recorded in 2009 on ISP network edge routers on two different geographical locations.

As mentioned earlier, one of the major challenges in flow classification is identifying encrypted flows. In the current study we used two different sources to obtain encrypted flows. Our first database, which was recorded in 2009, contains some encrypted flows of BitTorrent and Skype. Our second source was a manual recording of a BitTorrent application taken in a controlled environment.

Special properties of the data set. Short flows, namely, flows with fewer than 15 payload packets, were removed from the dataset. The rational behind ignoring short flows is that we are using the statistical properties of the flow for classification, and measuring such properties on short flows is unreliable. Hence short flows require a different approach. Note that in many practical scenarios, classifying short flows is of lower priority, as they account for an insignificant

[2] The term *flow* refers to a single data flow connection between two hosts, defined uniquely by its five-tuple (source IP address, source port, destination IP address, destination port, protocol type TCP/UDP).

fraction of the overall utilized bandwidth. We remark that flows with fewer than 15 packets account for 87% of the total flows but only 7% of the total bytes. Using the algorithm refinement of an LRU cache for heavy hosts, we were able to classify around fifty percent of the short flows as well, thus reducing the total bytes that were actually ignored to approximately 3.5%.

Another property of the data set is that only applications with sufficiently significant representation in the data traces were considered. Specifically, we considered applications that had at least 4000 flow instances in our records. The flow distribution was as follows: Http flows accounted for 59% of the flows, BitTorrent for 17.1%, SMTP 13%, EDonkey for 8.5%, and POP3, Skype, Encrypted BitTorrent, RTP and ICQ were each responsible for less than 1% of the flows.

3 The Classification Algorithm

We now specify our machine learning based classification algorithm. The host initiating the flow is defined as the *client* and the host accepting the flow - as the *server*. We consider only packets that contain payload.

Feature extraction. The use of classification algorithms based on machine learning requires us to parameterize the flow, turning each flow x into a vector of features $\bar{V}_x = \langle V_1, \ldots, V_d \rangle$, where each coordinate V_i contains some statistical parameter of the flow x (e.g., its packet mean size). Our study focused on realtime classification, making it necessary to concentrate on features that are both cheap to calculate and can be calculated in streaming mode (namely, inspecting a single packet at a time and seeing each packet only once).

The feature extraction stage consists of two phases. In the first phase, we consider basic traffic flow properties and collect the corresponding parameters for each flow. The statistics are collected until we reach *classification point* (the point in time upon we decide on the flow's application type). All the experimental results reported in this paper used an inspection length parameter of $m = 100$ packets, that is, all flows were classified upon seeing packet 100 (or earlier, if the flow size was less than 100 packets). In the second phase, once the classifier reaches classification point, it turns the statistics collected into a feature vector, which is then used as the input for the classifier.

Feature Set. Our complete feature set included the following 17 different parameters: Client number of packets; Server number of packets; Total number of packets; Client packet size expectation; Server packet size expectation; Client average 'packets per second' rate; Server average 'packet per second' rate; Client packet size variance; Server packet size variance; Total client bytes; Total server bytes; Download to upload ratio; Server average number of bytes per bulk[3]; Client average number of bytes for bulk; Server average number of packets for bulk; Client average number of packets for bulk; and Transport protocol (TCP or UDP). Note that in the asymmetric setting, some of our features take zero value. Moreover, unidirectional flows exist in a symmetric routing as well, for example during FTP download.

[3] Contiguous parts of a flow, separated by idle periods of 1sec or more.

The k-nearest neighbors algorithm. This is one of the simplest and most well-known classification algorithms. It relies on the assumption that nearby data sets have the same label with high probability. In its simplest form, the algorithm classifies flows as follows. Upon receiving the feature vector $\bar{V}_x$ of a new flow x, the algorithm finds its k-nearest neighboring flows (in Euclidean distance) in the training set. The flow x is then assigned the label associated with the majority of those neighbors. In our experiments we used single neighbors ($k = 1$); we discuss the use of multiple neighbors in the discussion section. Our preliminary tests show a reasonably good classification rate of above 99% for $k = 1$.

The main problem with the k-nearest neighbors algorithm is that its time complexity grows linearly with the training set size, which is problematic as the training set may contain thousands of samples. Algorithm accuracy, learning time and complexity are compared in [10]. To address this disadvantage, we combined the k-nearest neighbors algorithm with the k-means algorithm.

The k-means algorithm. Another component in our classifier is the k-means algorithm. In the training phase of k-means classification, the flows are divided into k clusters (according to geometrical similarity of their corresponding vectors). We then label each cluster based on the majority of flow types that have been assigned to the cluster. Now, a new flow x is classified by finding the cluster C whose center is nearest to x. The flow x is assigned with the label of C. The algorithm's accuracy is only 83%, but it requires considerably less computational resources compared to the k-nearest neighbors algorithm.

An analysis of the distances between flows and their cluster centers[4] reveals that the distance distribution is Gaussian, so in each cluster most of the flows are placed close the cluster center. This suggests that the flow's nearest neighbor is likely to fall in the same cluster with high probability, and only instances very far from the cluster center can have their nearest neighbor placed in another cluster. Our hybrid algorithm, presented next, relies on this fact. We start by presenting the hybrid algorithm as a whole, and then discuss its properties.

Our hybrid algorithm. The core of our final classifier is a hybrid algorithm that integrates the above two algorithms, thus combining the light-weight complexity of the k-means algorithm, with the accuracy of the k-nearest neighbors algorithm. The hybrid algorithm also features additional refinements in the k-means clustering phase. As mentioned previously, our algorithm has two phases: the training phase and the classification phase.

In the training phase, using the labeled data in our training set, we construct a set of clusters in two stages. In the first stage, for each protocol (HTTP, SKYPE, ...), we run the k-means algorithm on the flows in our training set that are labeled as the protocol being considered. We note, that it is common that a collection of flows all generated by the same protocol may have very diverse behavior (and thus a diverse cluster structure). This follows by the fact that certain protocols (such as HTTP) may behave in different manners depending on the precise setting in which they are used (e.g., a HTTP flow carrying streaming

[4] Omitted for space considerations; also noted independently in [2], although different parameters were used.

video might not look similar to one carrying text only). The result of our first stage clustering, is a set of cluster centers (k centers for each protocol), where each cluster is labeled naturally by the protocol in which it was constructed.

We now turn to the second stage of our clustering. After stage 1, clusters generated from different protocols might overlap, which might cause classification errors. To overcome this, in our second clustering stage, we redistributes the entire sample set of cluster centers defined in stage 1. Namely, using the same cluster centers that were found in stage 1, each flow in our training set is associated with the cluster center closest to it. Our two stage clustering is presented as Algorithm 1 below. In what follows, X_i is the dataset of flows generated by application i, C_i is the set of k cluster centers of application i, and C is the set of centers after stage 1 of our clustering (k centers for each application).

Algorithm 1. Pseudo code for our two stage clustering

```
for-each $X_i \in \{X_1, ..., X_l\}$
    $C_i = \mathbf{k\text{-}means}(X_i, k)$
$C = C_1 \bigcup C_2 \bigcup .... \bigcup C_l$
$X = X_1 \bigcup X_2 \bigcup .... \bigcup X_l$
for-each $x_i \in X$
    associate $x_i$ with the closest center from $C$.
```

This concludes the training phase of our algorithm. Now, for a given flow x, the online classification is also done in two stages. First, we find the cluster center c nearest to (the geometrical representation of) x. Note that this may not be enough. Namely, recall that after the second stage of our training, the clusters may not be homogeneous, and thus it is not clear how to label x given c. For this reason, we use the second stage of our online classification, which runs the k-nearest neighbors algorithm (for $k = 1$) over the members of the cluster corresponding to c. The resulting Algorithm 2 is presented below.

Algorithm 2. Pseudo code for hybrid classification

```
$j = \text{argmin}_j \|x - c_j\|$, where $c_j \in C$
$nb = \text{argmin}_{x_i} \|x_i - x\|$, where $x_i$ is associated with cluster center $c_j$
return label($nb$)
```

Some remarks are in order. The design of our algorithm was guided by the observation that nearest neighbor classification is very accurate but slow in running time, while k-means classification is fast but has relatively weak accuracy. This naturally leads to the idea of combining the two algorithms. However, one may first consider a seemingly more natural way of combining the two algorithms, namely, for training take the entire training set and cluster it using the k means algorithm (here, one would take a large k), and then perform the two-stage classification suggested above. We have checked this simpler hybrid technique, and indeed it yields very good results. Namely, on the one hand the accuracy remains almost identical to that of the k-nearest neighbor algorithm, while on the other, the performance resembles that of the k-means algorithm. However, we have noticed that our two-stage training technique improves the overall accuracy (without modifying the running time). This follows from the fact that in our two-stage clustering, flows of the same protocol tend to be clustered together. Thus in classification, using the nearest neighbor approach, we are able to ovoid mislabelings. We also note that our two-stage training procedure is more efficient in running time than the naive single stage training, despite the fact that we run the k-means algorithm multiple times (once for each application type). This can be explained by the use of a much smaller data set on each separate run.

Setting parameters and running time analysis. The complexity of the algorithm highly depends on the clusters that were formed and on the distribution of the inspected flows. In the worst case, the outcome could be an unbalanced clustering, with few large clusters and many small ones. The hybrid algorithm uses the nearest-neighbor procedure within the nearest cluster as the final stage of classification, and therefore the complexity is directly affected by the cluster size. It is not unreasonable to assume that the distribution of the inspected flows correlates strongly with the cluster distribution, i.e., most of the flows will likely be assigned to a large cluster. In this case we lose accuracy without achieving the desired performance improvement.

We overcome this difficulty by setting a maximum size for each cluster. Then, in the end of our training phase, we reduce the size of a large cluster by removing random flows from it until reaching the desired size. We found it useful to bound cluster sizes by $4n/c$, where c is the number of clusters and n is the training set size. Our experiments show that this restriction does not affect the overall accuracy. On the other hand, the complexity of our classification can now be bounded by $4\frac{n}{c} + c$. Namely, it takes c comparisons to find the closest cluster center, and then $4\frac{n}{c}$ comparisons to find the nearest neighbor in the cluster at hand. We minimize the running time of $4\frac{n}{c} + c$ by setting c to $4\sqrt{n}$.

The complexity of the hybrid algorithm is thus much better than that of the nearest neighbors algorithm (which is n). In fact, this is just a worst-case upper bound, and the actual experimental results are even better; as seen in Section 4, the practical complexity is almost as good as the complexity of k-means.

Leveraging Internet "heavy host" nature to save performance. Another useful component of our hybrid classifier makes use of a simple and relatively small cache in order to save more than 50% of the classification time and memory. This component relies on the *heavy host* phenomenon. Heavy hosts are hosts that consume considerably more network resources compared to other hosts. Both the Web and P2P systems are known to have heavy hosts [1,8,9]. Inspecting P2P and HTTP traffic usually reveals a small percentage of hosts that account for a large percentage of the total flows and used bandwidth. In the Web, this behavior is mainly driven by content popularity [8], as popular content is often held at a small number of servers. In P2P systems, heavy hosts behavior is caused by a different reason, namely, the distribution of P2P traffic, which is dominated by "free riders" (i.e., clients that do not contribute content and mainly consume) and "benefactors" (i.e., clients that mainly contribute and do not consume) [1].

Our findings show that the top ranked host accounts for about 0.7% of the total flows, and the same more or less goes for the second ranked host. The third accounts for 0.5%, the tenth accounts for 0.3% and these figures continue to drop sharply. All in all, we find that the 1000 top ranked hosts account for more than 50% of the flows. This distribution suggests the use of an LRU cache in our classification process. For each server stored in the LRU cache, the cache keeps the host IP and port of the flow server as its key and the flow class as the value. The classification works as follows. On receiving a new flow, first check if its server's host IP and port are stored in the LRU. If the information exists in the cache, then classify the flow according to the LRU cache, else use the classification algorithm and store the result in the LRU cache.

This classification caching scheme has a serious flaw: if the algorithm misclassifies one of our top ranked servers, then all flows destined to it would be misclassified as well. To overcome this problem, we keep in the LRU cache the last ℓ classification results destined to a given server, for some parameter ℓ. Once we have ℓ results concerning a given server in the LRU cache, we apply a majority vote to decide its class. This improves our accuracy, as the probability of misclassification drops exponentially with ℓ. Misclassification can be reduced even further by adding checkpoints, and rerunning the classification algorithm every ρ classifications, replacing the oldest classification in the last classification list. This improves the overall accuracy of the LRU cache by an additional 1%-2%. Our experiments indicate that using the LRU cache, more than half of the flows are classified on the basis of their first packet, in $O(1)$ time. Finally, we remark that it is possible to use such LRU caching within any classifier to boost both its performance and accuracy.

4 Results

In this section we present our evaluation methods and the experimental results obtained by our algorithms. We also discuss a unique aspect of our work, namely, the implementation and testing of our algorithm in line rate in the SCE 2020, the network traffic controller box of Cisco.

Algorithm evaluation. We used two data sets in our validation process, one small and the other much larger. For the small data set we extracted 4000 flows from each application type (32k flows in total). On each test we partitioned the data set into two: a training set consisting of 1000 randomly selected flows and a validation set (to be classified) containing the rest of the flows. We repeated the test several times and took the average result. For the large data set we used the entire data set available, where again 1000 flows of each application type were chosen randomly into the training set and the rest of the flows were taken into the validation set (1.5M flows in total). The basic experiments were done using the small data set, while some of the major experiments were repeated on the large data set. The results were very consistent, and the main added value of the large data set turned out to be the ability to test one of our algorithm refinements, namely, the use of the LRU cache.

Algorithm accuracy. Our results are presented in Table 1. BitTorrent (BT) and Encrypted BitTorrent flows were grouped together, namely, classifying encrypted BitTorrent into non-encrypted BitTorrent was counted as a success and the same for the other way around. As mentioned, the k-means algorithm is somewhat less accurate and achieves a modest average accuracy rate of 83%. The k-nearest neighbors (k-NN) algorithm achieves the best results, with overall accuracy of 99.1%. Its accuracy on traditional applications is very close to 100%, but as mentioned, the algorithm is too expensive for realtime applications. The accuracy of the hybrid algorithm is very similar to that of the k-nearest neighbors algorithm, implying that very little accuracy is lost by combining the two approaches.

Table 1. Algorithm accuracy

Algorithm	Http	SMTP	POP3	Skype	EDonkey	BT	Encrypted BT	RTP	ICQ
k-means	0.78	0.93	0.93	0.85	0.80	0.75	0.74	0.93	0.71
k-NN	0.997	0.999	1.0	0.945	0.947	0.96	0.98	0.997	0.962
hybrid	0.997	0.999	0.998	0.94	0.94	0.963	0.974	0.992	0.954

One of the main purposes of this study was dealing with encrypted flows in realtime. Indeed, encrypted BitTorrent exhibited results similar to non-encrypted BitTorrent. Also note that Skype (which is encrypted) exhibits an accuracy similar to BitTorrent. These results look very promising and indicate that our algorithm is insensitive to encryption and can classify encrypted traffic as easily as non-encrypted one. The experiments conducted using our own generated records (recorded manually, see Section 2) yielded even a higher accuracy. We note that this may be attributed in part to localization effects of the records.

Table 2. Complexity

Data Set Size	k-means	k-nearest neighbor	hybrid
100	153 Sec	177 Sec	150 Sec
1000	153 Sec	900 Sec	151 Sec
9000	153 Sec	7300 Sec	172 Sec

Table 2 presents a comparison of the classification time. We ran the classifiers on the same environment with the same data and similar configurations (cluster numbers). The classification was done using the LRU cache refinement. The results indicate that the k-nearest neighbors algorithm is by far the most time consuming. The time requirements of our hybrid algorithm are almost as low as those of the k-means algorithm, which is the most efficient.

Realtime evaluation. We tested the feasibility of our algorithm in a realtime embedded environment, by implementing the k-nearest neighbors and hybrid algorithms on the SCE2020 platform, one of Cisco's network traffic controllers. More specifically, the algorithms were developed and implemented as a stand-alone component (in C++, on a PPC dual core) on SCE 2020. We tested the accuracy of the algorithms by injecting the records (flows) using the Endace tool [6] and comparing the classification results.

The Endace tool injected the traffic in line rate as it was recorded, while the SCE2020 was configured to send a report on each classified flow. The report contained the flow five-tuple and the assigned application. The accuracy results were similar to our off-line tests, as expected. We offer the following conclusions.

Technical Limitations. The algorithm employs basic mathematical calculations, mostly simple additions and multiplications. Implementing such an algorithm may be more challenging on a platform that does not support floating point primitives, although this difficulty is of course solvable in software.

Memory. The algorithm used 76 bytes per flow on the statistics collection phase and one Megabyte for the training set. Taking into account a concurrency level of a few thousand flows (in the classification phase) and the use of an LRU table, the classifier uses only 4-5Mb. This is a very low figure (for core classifiers) and fits our realtime low memory usage requirement.

Performance. Running the algorithm did not appear to exert any stress on the CPU. This is not surprising considering, by comparison, the amount of work required by a DPI classifier. However, one must keep in mind that the SCE2020 box runs many tasks besides the classification, and hence it is expected of the classifier not to load the CPUs. This should be tested further.

Summarizing, there appear to be no technical, memory or performance limitations in implementing our algorithm in a real-world professional classifier. The algorithm has some practical limitations, such as a somewhat high average classification point. For further discussion of usability issues see Sect. 5.

5 Discussion

Conclusions and directions for future study. This paper presents a statistical algorithm enhancing and complementing traditional classification methods. Its strength is in points where traditional methods are relatively weak, most importantly in handling encryption, but also in asymmetric routing and packet disordering. The proposed algorithm is shown to be fast and accurate, and has no limitations to implementing it in professional realtime embedded devices. Hence it can be implemented as a complementary method for dealing with encrypted and other problematic flows.

Misclassification. Flow classification is usually employed by traffic controllers to enforce some policy on the traffic flow. The result of imposing a wrong policy may significantly affect the user experience. Thus, in some cases it is better to classify flow as "unknown" than to misclassify it. We propose to label traffic as unknown based on the notion of *homogeneous neighborhoods*. Namely, define the flow neighborhood (k-nearest neighbors, for $k > 1$) as *homogeneous* if all neighbors have the same label. Then classify a given flow as follows. In case its neighborhood is homogeneous, give it the neighborhood label; otherwise, mark it as "unknown". Our results show that this approach reduces misclassification levels but results in many more "unknown" flows. This tradeoff poses an interesting direction for further investigation.

Combining DPI and statistical approaches. Our algorithm has several advantages over traditional methods but still, the fastest and most accurate way to classify simple HTTP traffic is by using DPI, relying on a simple string signature. Yet, our algorithm can be used in situations where traditional methods fail. Indeed, the strengths of the statistical approach correspond to the weaknesses of DPI. For example, flows that were not identified by the traditional classifier (such as encrypted flows), and were labeled as 'unknown', may now be classified correctly using our algorithm, with little additional computational cost. Combining DPI and our proposed algorithm in such a way also allows a quick and efficient way to cope with new applications (one of the major drawbacks of DPI classification). Specifically, until a DPI signature is generated for the new application, our algorithm may give the customer a quick solution.

References

1. Basher, N., Mahanti, A., Mahanti, A., Williamson, C.L., Arlitt, M.F.: A comparative analysis of web and peer-to-peer traffic. In: Proc. 17th WWW, pp. 287–296 (2008)
2. Bernaille, L., Teixeira, R., Salamatian, K.: Early application identification. In: Proc. ACM CoNEXT, p. 6 (2006)
3. BitTorrent. Tracker peer obfuscation, http://bittorrent.org/beps/bep_0008.html
4. Crotti, M., Dusi, M., Gringoli, F., Salgarelli, L.: Traffic classification through simple statistical fingerprinting. Computer Commun. Review 37(1), 5–16 (2007)
5. Dewes, C., Wichmann, A., Feldmann, A.: An analysis of Internet chat systems. In: Proc. 3rd ACM SIGCOMM Internet Measurement Conf. (IMC), pp. 51–64 (2003)
6. Endace. The dag tool, http://www.endace.com/
7. Este, A., Gringoli, F., Salgarelli, L.: Support Vector Machines for TCP traffic classification. Computer Networks 53(14), 2476–2490 (2009)
8. Gummadi, P.K., Dunn, R.J., Saroiu, S., Gribble, S.D., Levy, H.M., Zahorjan, J.: Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In: Proc. SOSP, pp. 314–329 (2003)
9. Karagiannis, T., Papagiannaki, K., Faloutsos, M.: BLINC: multilevel traffic classification in the dark. In: Proc. ACM SIGCOMM, pp. 229–240 (2005)
10. Kim, H., Claffy, K.C., Fomenkov, M., Barman, D., Faloutsos, M., Lee, K.-Y.: Internet traffic classification demystified: myths, caveats, and the best practices. In: Proc. ACM CoNEXT, p. 11 (2008)
11. Madhukar, A., Williamson, C.L.: A Longitudinal Study of P2P Traffic Classification. In: Proc. IEEE MASCOTS, pp. 179–188 (2006)

12. McGregor, A., Hall, M., Lorier, P., Brunskill, J.: Flow Clustering Using Machine Learning Techniques. In: Barakat, C., Pratt, I. (eds.) PAM 2004. LNCS, vol. 3015, pp. 205–214. Springer, Heidelberg (2004)
13. Moore, A.W., Zuev, D.: Internet traffic classification using bayesian analysis techniques. In: Proc. ACM SIGMETRICS, pp. 50–60 (2005)
14. Nguyen, T.T., Armitage, G.J.: A survey of techniques for internet traffic classification using machine learning. IEEE Comm. Surv. & Tutor. 10, 56–76 (2008)
15. Paxson, V.: Empirically derived analytic models of wide-area TCP connections. IEEE/ACM Trans. Networking 2(4), 316–336 (1994)
16. Roughan, M., Sen, S., Spatscheck, O., Duffield, N.G.: Class-of-service mapping for QoS: a statistical signature-based approach to IP traffic classification. In: Proc. 4th ACM SIGCOMM Internet Measurement Conf. (IMC), pp. 135–148 (2004)
17. Sen, S., Spatscheck, O., Wang, D.: Accurate, scalable in-network identification of p2p traffic using application signatures. In: Proc. 13th WWW, pp. 512–521 (2004)
18. Zander, S., Nguyen, T.T., Armitage, G.J.: Automated Traffic Classification and Application Identification using Machine Learning. In: Proc. 30th IEEE LCN, pp. 250–257 (2005)
19. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification. Wiley, Chichester (2001)
20. Guyon, I., Elisseeff, A.: An Introduction to Variable and Feature Selection. J. Machine Learning Research 3, 1157–1182 (2003)

Data Propagation with Guaranteed Delivery for Mobile Networks

Hakob Aslanyan, Pierre Leone, and Jose Rolim

Computer Science Department, University of Geneva,
Battelle Batiment A, Route de Drize 7, 1227 Geneva, Switzerland

Abstract. In this paper, we consider wireless sensor networks where nodes have random and changeable mobility patterns. We study the problem where a particular node, called the base station, collects the data generated by the sensors/nodes. The nodes deliver the data to the base station at the time when they are close enough to the base station to ensure a direct transmission. While the nodes are too far to transmit to the base station, they store the data in a limited capacity internal FIFO queue. In the case where the queue is full, the new generated data are inserted in the queue and the oldest data are lost. In order to ensure, with a high probability, that the base station receives the generated data, the nodes disseminate the generated data in the network. The dissemination process consists in transmitting the data to others mobile nodes which are close enough to ensure a direct transmission. The nodes must control the dissemination process. Indeed, if the nodes send systematically the data to the neighbouring nodes then, the FIFO queues are going to be quickly saturated and the data lost (the dissemination process duplicate the generated data). On the other hand if the nodes do not disseminate the data, the data queued first are prone to be systematically lost if the capacity of the queue is too limited.

We propose a protocol based on the estimate of the delivery probabilities of the data. Each node estimates the delivery probabilities of all the queued data. These probabilities depend on the position of the data in the queue and, on the dissemination process. The lower is the delivery probability the more the nodes disseminate the data to increase the delivery guarantee to the base station. In that way, all the messages get a high probability to be delivered to the base station (higher that some predefined threshold). Experimental validations of the protocol show that the protocol performs well and outperforms an existing protocol.

Keywords: Sensor networks, Mobility, Guaranteed delivery, Data propagation.

1 Introduction

Wireless sensor networks (WSN) are composed of a large number of sensor nodes with sensing, processing and wireless communication capabilities. Usually, the nodes are spatially distributed in a given region that they monitor. They use their sensing capabilities to monitor the environment, collecting data like temperature, pressure, vibration, sound, etc. The sensed data (that the nodes generate) have to be delivered to a particular node called the base station.

Usually the nodes are battery powered and it is crucial to limit the energy consumption due to the transmissions in order to increase the operability time of the network (network lifetime). In some settings, the nodes are able to transmit the messages directly to the base station by using wireless transmissions. However, it is known that the energy required to transmit data over distance d is proportional d^{α} where α is usually in the interval $[2, 4]$ (see [13]). Hence, long-range transmissions are energy-costly. A way to reduce the energy consumption is to use intermediate nodes to convey the data to the base station with multi-hops. There is a large amount of research on energy aware data gathering in wireless sensor networks with static nodes, see for instance [1,4,5,10,7,14] and the references therein.

In this paper, we consider the case where the nodes are mobile. Many of the routing protocols for wireless sensor networks with static nodes use information about the network topology. One of the first works on data gathering with mobile WSN is presented in [8] that considers the case where the nodes have a reduced mobility pattern and the base station is mobile. The protocol presented in [15] is similar to the one we present in the present paper: Each node locally calculates a delivery probability and decides whether the data are forwarded. However, it is hard to relate the computed estimate with ours and then to proceed to fair comparison at this stage. In [11] the authors present a data gathering protocol for networks where the position of each node is a known function of time. Finally, in [9] the authors present a protocol for networks with randomly moving nodes with different mobility patterns. The nodes are able to change their mobility patterns during the time. The authors define a mobility level index that captures the node speed, dislocation and mobility changes. Based on this index, the authors suggest to evaluate the probability that a given node will deliver data to the base station.

The main idea of the algorithm presented in this paper is to transmit (diffuse) to many nodes a data m generated by the sensor nodes, in such a way that the probability that at least one of the nodes will get close enough to the base station and delivers the data is larger than a predefined threshold. We consider that the nodes have a limited memory and after getting their memory full, they need to drop data for saving newly generated ones. The nodes manage the memory as a FIFO queue. The main advantage of our algorithm is that the nodes do not use information about their positions; they also do not need to know where the base station is located. The nodes take the decisions whether to forward the data to another node or not by using only the count of the previously (successfully) delivered and dropped data. We proceed to the simulation of the algorithm and, we show the effectiveness of our protocol. We also compare the performance with the algorithm presented in [9]. Both algorithms consider that the nodes are randomly moving with varying mobility patterns and with limited memory.

We compare the performance of both algorithms in terms of data delivery rate, average data delivery delay and average number of sent messages per node.

The rest of the paper is organized as follows. We describe the proposed algorithm in Section 2. The theoretical validation is described in Section 3 while we present the experimental validation in Section 4.

2 Description of the Algorithm

We consider that the memory capacity of the nodes is limited. The nodes convey data to the base station coming from two sources. The first type of data, called generated data, are the data that the nodes acquire with their sensing devices. The second type of data, called received data, are the data that are transmitted by others nodes, i.e. disseminated. Thus, after spending some time away from the base station the generated and received data might saturate the memory of some nodes. In this case, the nodes no longer accept received data. However, the node inserts the generated data in the FIFO queue and the data in the head of the queue are lost.

Let us assume that each node i knows the probability $p_i(m)$ that the data m will be successfully delivered to the base station. We point out that this probability depends on the position in the queue where the data are first inserted. Because the position of the data might change with time, we assume that this information is attached to the data m. Then, $q_i(m) = 1 - p_i(m)$ is the probability that the node i will not deliver the data m to the base station. If the data m in some way appear on nodes $j_1, \ldots, j_k$, the probability that at least one of those nodes will deliver m to the base station is

$$P_m = 1 - \prod_{i=1}^{k} q_{j_i}(m). \tag{1}$$

We call (1) the delivery probability of data m, and consequently

$$Q_m = \prod_{i=1}^{k} q_{j_i}(m), \tag{2}$$

is the probability that the data m are not delivered.

The main goal of our algorithm is to diffuse the data in the network in such a way that the delivery probability P_m satisfies $P_m \geq d$ for some predefined threshold d (for example we put $d = 0.993$ in our simulations). Or equivalently, $Q_m \leq 1 - d$.

The discussion above shows that if we are able to compute the delivery probability $p_i(m)$, then the diffusion process ensures that the delivery probability is larger than d.

The nodes manage the incoming data (generated or transmitted) in a FIFO queue. The new accepted data are stored at the end of the queue. Once the memory is full, a node drops the data from the beginning of the queue to make space for the new generated ones. In this case, no new forthcoming received data are accepted. Data m have a supplementary field where the probability Q_m that the data will not be delivered by the nodes to the base station is stored. Newly generated data before being saved to the node memory have $Q_m = 1$. When node i accepts (generated or received from other nodes) data and stores it in the queue, the probability is updated with $Q_m \cdot q_i(m)$ to take into account the probability that the data will be delivered. Although it is not explicitly denoted, the probability $q_i(m)$ depends on the index where the data are inserted in the queue.

In a second phase, the node i proceeds to the diffusion of the data in the network to ensure that the probability of delivery is large enough. If the new probability that m is not delivered satisfies $Q_m \geq 1 - d$ and the node encounters another node j then it transmits the data to j. When node j accepts the data, i marks the data as *diffused* and stops the diffusion process. Node j updates the probability Q_m by $Q_m \cdot q_j(m)$ and stores the data in its queue. Node j diffuses the data further if $Q_m \geq 1 - d$.

From the point of view of the node j that is requested to convey the data, the diffusion process consists in: 1. j refuses the data if its queue is full 2. else, accepts the data, updates the probability of delivery and diffuses the data further if $Q_m \geq 1 - d$.

Finally when a node meets the base station it forwards all the data from its queue. The node does not remove the delivered data from the queue but, simply keeps them to prevent the multiple acceptance of already delivered data. However, the node inserts new data in the queue as if it was empty by ignoring the already delivered data.

The algorithm that we present in the preceding section uses the probabilities $p_i(m)$ that data m will be delivered to the base station. These probabilities depend on many parameters such as: The index where the data are inserted in the queue, the size of the queue, the mobility pattern and the size of the area covered by the mobile nodes. In order to ensure the flexibility and robustness of the protocol we suggest that the nodes estimate themselves these probabilities. Basically, we propose that the nodes use two counters $C_1[k]$ and $C_2[k]$ per queue's entry k. The counter $C_1[k]$ counts the number of data inserted in position k that are delivered by the node to the base station (the value of these counters depend with the time t but we do not introduce this dependency in order to simplify the notation). The counter $C_2[k]$ counts the total number of data inserted in the queue at position k. We then suggest to estimate the probability of delivery with the estimation

$$p_i(m) \approx \frac{C_1[k]}{C_2[k]}. \qquad (3)$$

On the left side of Figure 1, we can observe the time the nodes needs to estimate a suitable delivery probability. We observe that the nodes improve the estimates in order to provide a nearly 100% data delivery rate.

Although the purpose of this paper is to provide evidence that the protocol is suitable and a full theoretical analysis of the performance is beyond the scope of this paper, we provide some theoretical evidence here.

3 Theoretical Analysis of the Performance of the Algorithm

We first notice that the mobility pattern of a node is independent of the data generated and received. The random mobility patterns that we consider are proposed in [9] in a similar setting that ours. In [3] the authors classify such random mobility patterns as Random Direction Mobility Patterns since the nodes choose a direction and a travel time repetitively. The aim of such mobility patterns is to make the distribution of the nodes as even as possible in the covered area as well as to ensure that the encounters between mobile nodes are as constant as possible. Indeed, the mean number of neighboring nodes is rather constant, compared, for instance, to the Random Waypoint Mobility Model, see [3].

We use this property and assume that for a given node the expected number of received data on a time span T is μT. Alternatively, we may define $\mu = \lim_{t\to\infty} E(\#received\ data\ in\ [0,t])/t$, and prove that the limit exists by using the stationarity of the nodes' motion. In particular it is independent of the position of the nodes.

The expected number of generated data is also assumed to evolve linearly with time and we denote λT this number. This corresponds to the situation where the nodes collect data repetitively in a deterministic way or in a random but stationary way.

In the following, we discuss how to prove that the estimates (3) converge (see equation (5)). In order to ensure the convergence, it is necessary that that the number of data generated and received by a node, denoted M_n, during the travel time does not depend on the time (time homogeneous) [2]. Although this has to be proved formally, the discussion above shows that this is a reasonable assumption.

Instead of considering the probability $p_i(m)$ that a node i delivers the data m, we consider an averaged value p. This is equivalent to consider the complete set of data and compute the average probability of delivery. Equivalently, we replace the value $p_i(m)$ by an average value p given that the nodes' encounters as well as the index where the data are queued are random. Given the probability p, data m are diffused an expected number α times, ensuring that $(1-p)^{\alpha+1} < 1-d$ ($\alpha = \alpha(p) = log(1-d)/log(1-p)-1$). Notice that we implicitly assume that the probabilities of delivery are independent of the nodes and the index of the queue.

Each time a node transmits the data to the base station, the node updates the probability of delivery using (3). We denote by p_n the n-th estimate. Let us denote by N_n the total number of data received by the node at the time step n. N_n is the value of C_2 in (3) where we removed the dependencies in k and in time. By the definition of p_n ($p_n = C_1/C_2$), C_1 is then given by $N_n p_n$. Between the time steps n and $n+1$ the node travels during a time T and has to convey M_n generated and received data. Then, the new estimate p_{n+1} is given by

$$p_{n+1} = \frac{1}{N_n + M_n}\Big(N_n p_n + (M \wedge M_n)\Big)^{1}.$$

The value M_n is the total number of data collected by the node between the time steps n and $n+1$, $N_n p_n$ is the expected total number of data delivered at time step n and $M \wedge M_n$ is the total number of data delivered to the base station at the time step $n+1$.

After some algebraic manipulations using that $N_n \to \infty$ we get that

$$p_{n+1} \approx p_n - \frac{1}{N_n}\Big(M_n p_n - (M \wedge M_n)\Big). \tag{4}$$

This last equation shows how the estimate of $p = \lim_{n\to\infty} p_n$ evolves. Consider first that the queue is never full, i.e. $M \wedge M_n = M_n$, $\forall n$. In this case, p_n increases up to the time when $p_n = 1$. The behaviour is correct since no data are lost and then, the probability of delivery is 1. On the other case, if some data are lost, i.e. $M \wedge M_n = M$, p_n might converge towards the value ensuring that $p_{n+1} = p_n$. One can prove that under some mild assumptions, we have

$$p = \lim_{n\to\infty} p_n = \frac{E\big(M \wedge M_n\big)}{E\big(M_n\big)}. \tag{5}$$

This last equation shows that the estimates p_n are converging to the right limit since the right side of the equation is the fraction of data that are delivered to the total number of data received and generated by the node.

The rate of convergence of the estimate (3) is difficult to compute. However, we observe on the left of Figure 1 that after a simulation period of $50'000$ seconds the estimate are good enough to provide nearly the maximum delivery guarantee. In the conditions of the simulations, this is the time corresponding to going back to the base station 25 times in average.

With the definition of the parameters λ and α provided in the beginning of the section, we obtain that $E(M_n) = T(\lambda + \mu)$, with T is the expected travel time. Notice that we depart from our implementation of the algorithm since the expression for $E(M_n)$ given here counts the received data even if the queue is full. Using (5), we observe that once the convergence occurs, the nodes can estimate the value of T by using only the statistics related to the queue occupation, λ and μ with

$$T = \frac{E\big(M \wedge M_n\big)}{p(\lambda + \mu)}. \tag{6}$$

Notice that we also expect that $\lambda + \mu = \lambda(1+\alpha)$ since the expected total number of data diffused in the network is $\alpha\lambda$, these data have to be carried by the nodes and we assume that the diffusion process distribute the data uniformly among the nodes.

The analysis we have proposed is likely to be rooted in some formal framework. Indeed, we postulate the existence of the constant λ, μ and, we conjecture that

[1] $a \wedge b$ is the minimum of a and b, recall that M is the capacity of the queue.

this existence can be asserted by using a renewal argument [6]. The renewal theory framework is natural in our setting, since each time a node gets by the base station corresponds to a renewal. In our short investigations, we assume that the estimate p_n is more or less similar for all nodes since we define α as a function of p_n. However, from the numerical experiments that we conducted, it appears that the probability of delivery depends on the mobility pattern. The 'mean-field' analysis that we propose here is prone to be more accurate in the case where the mobility patterns of all the nodes are the same. On the other case, the value we obtain is an averaged value. Moreover, the analysis of the convergence of the estimate (3) can be conducted with the ODE method [12,2].

We point out that the estimate (6) counts the data received after the queue is full. In our implementation of the algorithm we do not accept data from others nodes while the queue is full, only generated data are inserted in the queue.

In the next section we validate experimentally our protocol where nodes use (3) for the estimation of the delivery probability.

4 Experimental Validations

In Figure 1 we present the time evolution of some parameters. Under our simulation conditions the average travel time is about 2000 seconds (not very dependent on the simple or complex mobility pattern). The figure on the left shows how the data delivery rate evolves. We observe that after a period of $50'000$ seconds, the behaviour stabilizes and the network delivers the data with the required guarantees. In the simulation conditions, the period of $50'000$ seconds corresponds to going back an average of 25 times. This means that by applying formula (3) 25 times, we obtain some estimates that are sufficiently accurate to ensure the delivery guarantees. The figure on the center shows the average delay. We observe that the delay is much lower, about 1000 seconds. This means that the diffusion process is really participating efficiently to convey the data to the base station. On the right of Figure 1 we display the average total number of data

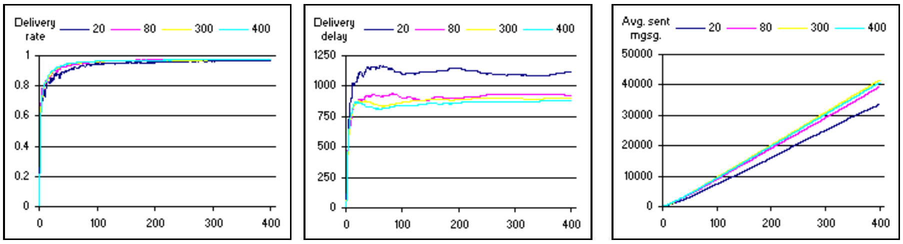


Fig. 1. Time evolution of the algorithm performance with the complex mobility pattern and 20, 80, 300 and 400 nodes. From left to right: The data delivery probability, the average message delay and the average number of messages sent per node. Axis X is a time scale in 1000 seconds.

sent by node. We first observe that this increases linearly with time. This supports our assumption that the network dynamics is stationary. Moreover, the 40′000 transmissions are due to an average of 10′000 generated data and the diffusion of an average of 15′000 data. Because the average travel time is about 2000 seconds, each node delivers about 125 data to the base station. This shows that on average the network can support the load of conveying the generated data and, that the network is also able to adapt to the period where more data are produced than in average. We suspect that the conditions of the simulations are close to the limit, in the sense that increasing the rate of generated data might lead to a decrease in the data delivery rate.

We proceed to a set of simulations with different network configurations in order to evaluate the performance of the protocol that we propose in this article. We also proceed to a set of simulations with the same set parameters to the mobility level based protocol (local adapt with random neighbor selection) presented in [9] in order to compare the performance. Actually, both protocols are comparable since they consider nodes with random motions and limited memories. To simplify the presentation, we use the well defined mobility patterns introduced in [9]. The four simple mobility patterns defined are *Working Mobility*, *Walking Mobility*, *Biking Mobility* and *Vehicular Mobility*. These mobility patterns are similar to the motion of a human who is working in his office, walking outside, biking or driving. Figure 3 presents some traces of the motions of the above-defined simple mobility patterns. For each mobility pattern, a node selects a direction and a speed and moves a random time in the direction at

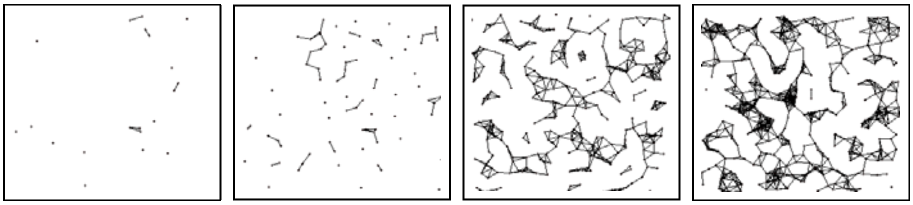


Fig. 2. Network connectivity at random time. Left to right 20 nodes, 80 nodes, 300 nodes, 400 nodes.

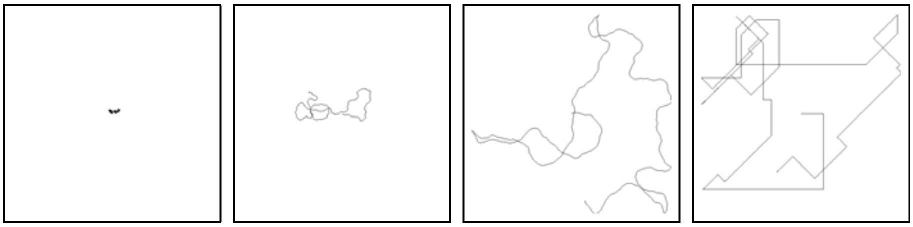


Fig. 3. Motions of simple mobility patterns. Left to right *Working Mobility*, *Walking Mobility*, *Biking Mobility*, *Vehicular Mobility*.

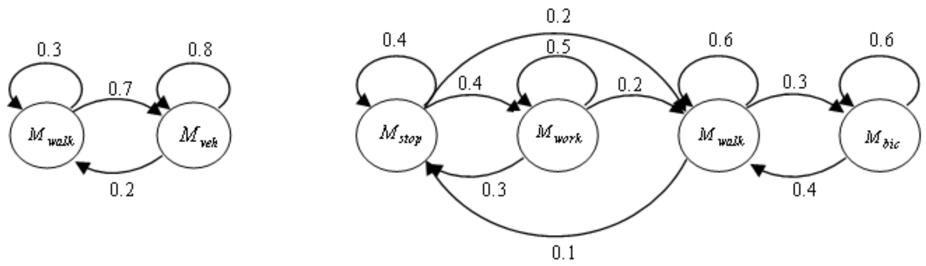


Fig. 4. Transition graphs of complex mobility patterns used in our simulations (in M_{stop} state node has no motion). On the left *C1 Mobility* on the right *C2 Mobility*.

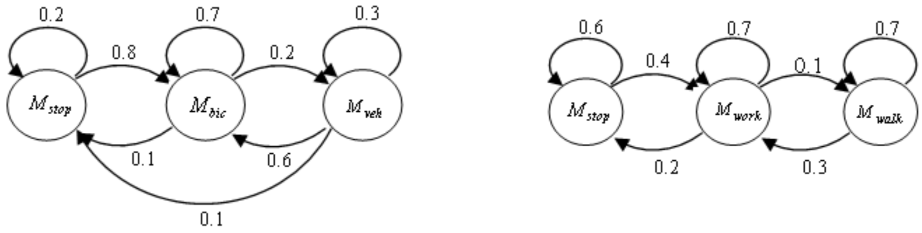


Fig. 5. Transition graphs of complex mobility patterns used in our simulations (in M_{stop} state node has no motion). On the left *C3 Mobility* on the right *C4 Mobility*.

the given speed. Basically, the speeds range makes the difference between the various mobility patterns. Using these simple mobility patterns, we define more complex ones where a node changes its mobility pattern by passing from one pattern to another one with some probability. These complex mobility patterns are easy to present with the transition graphs on the Figures 4 and 5. Each vertex of a graph corresponds to a simple mobility pattern, and two vertices are connected by a directed edge, on the top of which is written the probability of passing from on pattern to the other one. $C1 - C4$ complex mobility patterns, presented on Figures 4 and 5 are similar to the ones in [9] and detailed information about the mobility patterns can be found therein. In our simulations all the nodes have the same size of memory, which is enough to accommodate 128 messages, the transmission range of nodes and base station is $70m$ and network is a $1000 \times 1000m^2$ square. Also, each node in average generates one message per 40 seconds ($0.025msg/second$).

We present two sets simulations. In the first, we assign each simple mobility pattern to 1/4 of total number of nodes in network. In the second round, we use the complex mobility patterns in the same portions. Every round contains four simulations with different numbers of nodes in networks 20, 80, 300 and 400, in total eight different simulations. In Figure 2 we show the network connectivity at random time for different numbers of nodes. For each of eight simulations, we

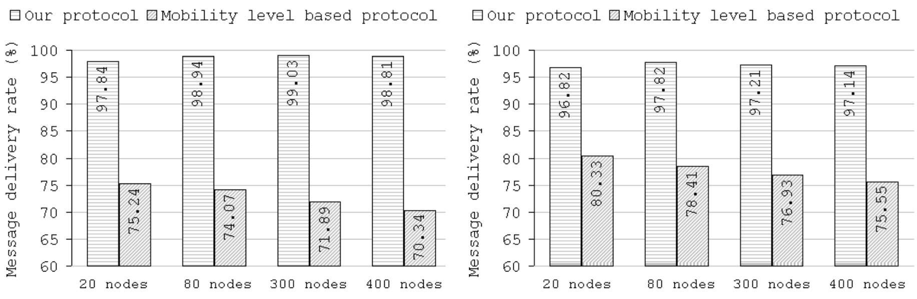


Fig. 6. Data delivery rate comparison. On the left *simple mobility* on the right *complex mobility*.

simulate the protocol for 400.000 seconds (111 hours) and, we chose the required delivery rate $d = 0.993$ and use (3) for the estimation of the delivery probability.

We compare the performance of our algorithm with the one presented in [9]. We consider three criteria. The first is the data delivery rate, which is the percent of delivered data to the base station. The second is average data delivery delay which is the average time the data are delivered to base station after being generated. And as the main part of energy goes for data transmissions, we compare the average number of sent data per node which will roughly represent the energy consumption of the protocols. In Figure 6 the delivery rate comparison of protocols are presented, respectively for networks where nodes have simple and complex mobility patterns. We observe that both protocols have stable delivery rates, according to number of nodes in network. And in all eight cases our protocol ensures the requested delivery rates. The comparisons of the average data delivery delay of both protocols are presented in Figure 7. Here we observe that as the number of nodes composing the network increases, the data delivery delay tends to a constant. The delivery delay decreases as the number of nodes increases. The algorithm proposed in [9] behaves similarly and the delivery delay is shorter for this algorithm than for ours. It is the only criterion for which that happens.

Figure 8 presents the protocol comparisons in terms of average sent data per node, we observe that the success of mobility level based protocol in dense networks in terms of average message delivery delay is due to the high number of sent data (replication). However, this requires a larger amount of energy. We observe that our algorithm ensures that the number of data sent does not increase as the number of nodes becomes large. Indeed, we observe that the number of data sent tends to a constant. We observe that if the number of nodes in the network is not large enough, the diffusion process does not manage to provide the guaranteed delivery of data. This is due to the fact that the nodes do not encounter others nodes. However, our experimental validations show that with 20 nodes, see left of Figure 2, we do not manage to ensure that $Q_m < 1-d$. However, the performance are still valuable since the algorithm ensures 97.84% and 96.82% of data delivery for respectively simple and complex mobility patterns. Figure 8

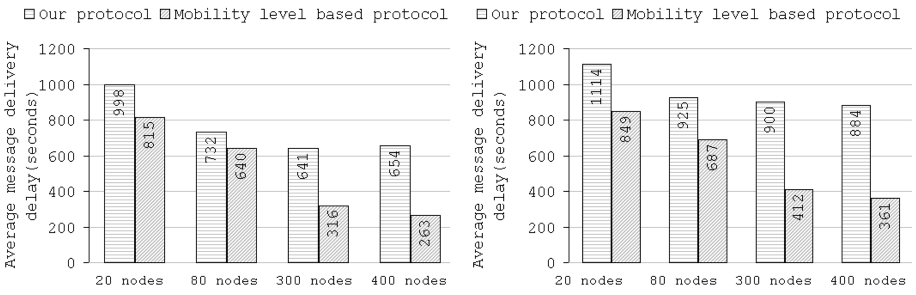


Fig. 7. Average data delivery delay comparison. On the left *simple mobility* on the right *complex mobility.*

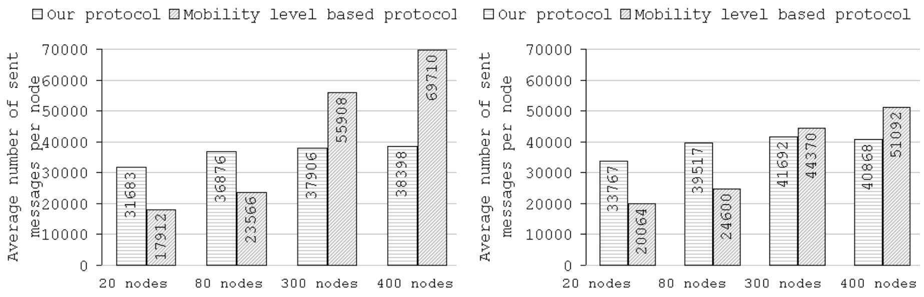


Fig. 8. Average sent data per node comparison. On the left *simple mobility* on the right *complex mobility.*

shows that with 20 nodes the number of data sent is small and, this confirms that the nodes' encounters are not sufficiently frequent. This has also an impact on the data delivery delay that is larger than for networks with more nodes.

References

1. Al-Karaki, J.N., Kamal, A.E.: Routing techniques in wireless sensor networks. IEEE Wireless Communications 11(6), 6–28 (2004)
2. Borkar, V.S.: Stochastic Approximation: A Dynamical Systems Viewpoint. Cambridge University Press, Cambridge (2008)
3. Camp, T., Boleng, J., Davies, V.: A survey of mobility models for ad hoc network research. Wireless communications & Mobile Computing 2(3), 531–541 (2002)
4. Dagher, J.C., Marcellin, M.W., Neifeld, M.A.: A theory for maximizing the lifetime of sensor networks. IEEE Transactions on Communications 55(2), 323–332 (2007)
5. Giridhar, A., Kumar, P.R.: Maximizing the functional lifetime of sensor networks. In: IPSN, pp. 5–12 (2005)
6. Heyman, D.P., Sobel, M.J.: Stochastic Models in Operations Research, vol. I. Dover Publications, Inc., New York (1982)

7. Jarry, A., Leone, P., Powell, O., Rolim, J.D.P.: An optimal data propagation algorithm for maximizing the lifespan of sensor networks. In: Gibbons, P.B., Abdelzaher, T., Aspnes, J., Rao, R. (eds.) DCOSS 2006. LNCS, vol. 4026, pp. 405–421. Springer, Heidelberg (2006)
8. Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L., Rubenstein, D.: Energy-efficient computing for wildlife tracking: Design tradeoffs and early experience with zebranet. In: 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLO 2002 (2002)
9. Kinalis, A., Nikoletseas, S.: Adaptive redundancy for data propagation exploiting dynamic sensory mobility. In: International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems, pp. 149–156 (2008)
10. Leone, P., Nikoletseas, S.E., Rolim, J.D.P.: An adaptive blind algorithm for energy balanced data propagation in wireless sensors networks. In: Prasanna, V.K., Iyengar, S.S., Spirakis, P.G., Welsh, M. (eds.) DCOSS 2005. LNCS, vol. 3560, pp. 35–48. Springer, Heidelberg (2005)
11. Lui, C., Wu, J.: Scalable routing in delay tolerant networks. In: Mobihoc (2007)
12. Meyn, S.: Control Techniques for Complex Networks. Cambridge University Press, Cambridge (2008)
13. Pahlavan, K., Levesque, A.: Wireless Information Networks. John Wiley and Sons, Chichester (1995)
14. Powell, O., Leone, P., Rolim, J.D.P.: Energy optimal data propagation in wireless sensor networks. J. Parallel Distrib. Comput. 67(3), 302–317 (2007)
15. Wang, Y., Wu, H.: Dft-msn: The delay/fault-tolerant mobile sensor network for pervasive information gathering. In: INFOCOM (2006)

Data Structures Resilient to Memory Faults: An Experimental Study of Dictionaries

Umberto Ferraro-Petrillo[1], Fabrizio Grandoni[2], and Giuseppe F. Italiano[2]

[1] Dipartimento di Statistica, Probabilità e Statistiche Applicate, Sapienza Università di Roma, P.le Aldo Moro 5, 00185 Rome, Italy
umberto.ferraro@uniroma1.it

[2] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via del Politecnico 1, 00133 Roma, Italy
{grandoni,italiano}@disp.uniroma2.it

Abstract. We address the problem of implementing data structures resilient to memory faults which may arbitrarily corrupt memory locations. In this framework, we focus on the implementation of dictionaries, and perform a thorough experimental study using a testbed that we designed for this purpose. Our main discovery is that the best-known (asymptotically optimal) resilient data structures have very large space overheads. More precisely, most of the space used by these data structures is not due to key storage. This might not be acceptable in practice since resilient data structures are meant for applications where a huge amount of data (often of the order of terabytes) has to be stored. Exploiting techniques developed in the context of resilient (static) sorting and searching, in combination with some new ideas, we designed and engineered an alternative implementation which, while still guaranteeing optimal asymptotic time and space bounds, performs much better in terms of memory without compromising the time efficiency.

1 Introduction

Memories in modern computing platforms are not always fully reliable, and sometimes the content of a memory word may be corrupted. This may depend on manufacturing defects, power failures, or environmental conditions such as cosmic radiation and alpha particles [14,19]. This type of phenomena can seriously affect the computation, especially when the amount of data to be processed is huge and the storage devices are inexpensive. This is for example the case for Web search engines, that store and process terabytes of dynamic data sets, including inverted indices which have to be maintained sorted for fast document access. For such large data structures, even a small failure probability can result in bit flips in the index, which may become responsible of erroneous answers to keyword searches [15,16].

The classical way to deal with memory faults is via error detection and correction mechanisms, such as redundancy, Hamming codes, etc. These traditional approaches imply non-negligible costs in terms of time and money, and thus they are not always adopted in large-scale clusters of PCs. Hence, it makes sense to try to solve the problem at the application level, i.e., to design algorithms and data structures which are resilient to memory faults. Dealing with unreliable information has been addressed in the algorithmic community in a variety of different settings, including the liar model [1,4,8,17,20], fault-tolerant sorting networks [2,18,21], resiliency of pointer-based data structures [3], and parallel models of computation with faulty memories [7].

The Faulty-RAM Model. In this paper we focus on the *faulty-RAM* model introduced in [12,13][1]. In this model, an adaptive adversary can corrupt any memory word, at any time, by overwriting its value. Corrupted values cannot be (directly) distinguished from correct ones. An upper bound δ is given on the total number of memory faults that can occur throughout the execution of an algorithm or during the lifetime of a data structure. However, we can exploit $O(1)$ *safe* memory words, whose content never gets corrupted. The adaptive adversary knows the algorithm and the state of safe and unsafe memory at any time. Furthermore, it can react to the actions of the algorithm. In other terms, corruptions do not need to be scheduled *a priori* (this is relevant for randomized algorithms). However, the adversary cannot access the sequence of random bits used by a randomized algorithm. Furthermore, read operations are considered atomic, i.e., the adversary cannot corrupt a memory word right after the algorithm starts to read it.

A natural approach to the design of algorithms and data structures in the faulty-RAM model is data replication. Informally, a *resilient variable* consists of $(2\delta + 1)$ copies x_1, x_2,…, $x_{2\delta+1}$ of a standard variable. The value of a resilient variable is given by the majority of its copies (which can be computed in linear time and constant space [5]). Observe that the value of x is reliable, since the adversary cannot corrupt the majority of its copies. The approach above induces a $\Theta(\delta)$ multiplicative overhead in terms of both space and running time. For example, a trivially-resilient implementation of a standard dictionary based on AVL trees would require $O(\delta n)$ space and $O(\delta \log n)$ time for each `search`, `insert` and `delete` operation. Thus, it can tolerate only $O(1)$ memory faults while maintaining optimal time and space asymptotic bounds.

This type of overhead seems unavoidable if one wishes to operate correctly in the faulty-RAM model. For example, with less than $2\delta + 1$ copies of a key, we cannot avoid that its correct value gets lost. Since a $\Theta(\delta)$ multiplicative overhead could be unacceptable in several applications even for small values of δ, the next natural thing to do is relaxing the notion of correctness. We say that an algorithm or data structure is *resilient* to memory faults if, despite the corruption of some memory location during its lifetime, it is nevertheless able to operate correctly (at least) on the set of uncorrupted values.

In [10,13], the *resilient sorting* problem is considered. Here, we are given a set K of n keys. A key is a (possibly negative) real value. We call a key *faithful* if it

[1] Due to space constraints, we refer to [12,13] for a detailed description of the model.

is never corrupted, and *faulty* otherwise. The problem is to compute a *faithfully sorted* permutation of K, that is a permutation of K such that the subsequence induced by the faithful keys is sorted. This is the best one can hope for, since the adversary can corrupt a key at the very end of the algorithm execution, thus making faulty keys occupy wrong positions. This problem can be trivially solved in $O(\delta\, n \log n)$ time. In [13], an $O(n \log n + \delta^3)$ time algorithm is described together with a $\Omega(n \log n + \delta^2)$ lower bound. A sorting algorithm `ResSort` with optimal $O(n \log n + \delta^2)$ running time is later presented in [10]. In the special case of polynomially-bounded integer keys, an improved running time of $O(n + \delta^2)$ can be achieved [10]. In [9] an experimental study of resilient sorting algorithms is presented. The experiments show that a careful algorithmic design can have a great impact on the performance and reliability achievable in practice.

The *resilient searching* problem is studied in [10,13]. Here we are given a faithfully sorted sequence K of n keys, and a search key κ. The problem is to return a key (faulty or faithful) of value κ, if K contains a faithful key of that value. If there is no faithful key equal to κ, one can either return *no* or return a (faulty) key equal to κ. Note that, also in this case, this is the best possible: the adversary may indeed introduce a corrupted key equal to κ at the very beginning of the algorithm, such that this corrupted key cannot be distinguished from a faithful one. Hence, the algorithm might return that corrupted key both when there is a faithful key of value κ (rather than returning the faithful key), and when such faithful key does not exist (rather than answering *no*). There is a trivial algorithm which solves this problem in $O(\delta \log n)$ time. A lower bound of $\Omega(\log n + \delta)$ is described in [13] for deterministic algorithms, and later extended to randomized algorithms in [10]. A $O(\log n + \delta^2)$ deterministic algorithm is given in [13]. An optimal $O(\log n + \delta)$ randomized algorithm `ResSearch` is provided in [10]. An optimal $O(\log n + \delta)$ deterministic algorithm is eventually given in [6]. For both resilient sorting and searching, the space usage is $O(n)$.

Resilient Dictionaries. More recently, the problem of implementing resilient data structures has been addressed. A *resilient dictionary* is a dictionary where the `insert` and `delete` operations are defined as usual, while the `search` operation must be resilient as described before. In [11], Finocchi et al. present a resilient dictionary using $O(\log n + \delta^2)$ amortized time per operation. In [6], Brodal et al. present a simple randomized algorithm `Rand` achieving optimal $O(\log n + \delta)$ time per operation. Using an alternative, more sophisticated approach, they also obtain a deterministic resilient dictionary `Det` with the same asymptotic performances. For all the mentioned implementations, the space usage is $O(n)$, which is optimal. However, as we will see, the constant hidden in the latter bound is not negligible in practice.

We next give some more details about `Rand`, since it will be at the heart of our improved implementation `RandMem`, which is described in Section 3. The basic idea is maintaining a dynamically evolving set of intervals spanning $(-\infty, +\infty)$, together with the corresponding keys. Intervals are merged and split so that at any time each interval contains $\Theta(\delta)$ keys. More precisely, each interval is implemented as a buffer of size 2δ, which contains between $\delta/2$ and 2δ keys at any time. Intervals are stored in a classical AVL tree, where standard variables (pointers, interval boundaries, etc.) are replaced by resilient variables. The number of intervals is at most $1 + \frac{n}{\delta/2}$. For each interval, we store a buffer of size 2δ plus a constant number of resilient variables, each one of size $2\delta + 1$. Hence, the overall space usage is $O(n + \delta)$. A `search` is performed in the standard way, where, instead of reading the $2\delta + 1$ copies of each relevant variable, the algorithm only reads one of those copies uniformly at random. At the end of the search, the algorithm reads reliably (in $\Theta(\delta)$ time) the boundaries of the final interval, in order to check whether they include the searched key. If not, the process is started from scratch. Otherwise, the desired key is searched for by linearly scanning the buffer associated to the interval considered. Operations `insert` and `delete` are performed analogously. In particular, the insertion of a key already present in the dictionary is forbidden (though duplicated keys might be inserted by the adversary). When, after one `insert`, one interval contains 2δ keys, it is split in two halves. When, after one `delete`, one interval contains $\delta/2$ keys, it is merged with a boundary interval. If the resulting interval contains more than $3\delta/4$ keys, it is split in two halves. The modifications of the interval set above involve a modification of the search tree of cost $O(\delta \log n + \delta^2)$. However, this cost is amortized over sequences of $\Omega(\delta)$ `insert` and `delete` operations. We remark that, for $\delta > n$, it is sufficient to maintain all the keys as an unsorted sequence into a buffer of size $O(n)$. In this case, each operation can be trivially implemented in $O(n) = O(\delta)$ time. Hence, the space usage can be reduced to $O(n)$ without increasing the running time.

The Experimental Framework. In this paper we focus our attention on resilient dictionaries. We perform an experimental evaluation of the optimal dictionaries `Det` and `Rand`, together with an improved implementation `RandMem` developed by ourselves. In order to evaluate the drawbacks and benefits of resilient implementations, we also consider a standard (non-resilient) implementation of a search tree. In particular, we implemented an AVL binary search tree called `Avl`, in order to make a more direct comparison with `Rand` (which builds upon the same data structure).

In order to test different data structures, we use the same testbed to simulate the faulty-RAM model as in [9][2]. Shortly, we model the data structure and the adversary as two separate parallel threads. The adversary thread is responsible for injecting $\alpha \leq \delta$ faults during the lifetime of the data structure. In order to inject one fault, the adversary selects one unsafe memory word (among the ones used by the data structure) uniformly at random, and overwrites it with a random value. In order to inject α faults, the adversary samples α operations uniformly at random over a given sequence of operations, and injects exactly one random fault during each sampled operation. We performed experiments both

[2] The reader is referred to [9] for a more detail description of the testbed.

on random inputs and on real-world inputs[3]. In random inputs, the instances consist of a sequence of random operations. A `random insert` simply inserts a random value in a given range $[\ell, r]$ (the actual range is not really relevant). In a `random search` we search for a key κ, where, with probability $1/2$, κ is chosen uniformly at random among the keys currently in the dictionary, and otherwise is set to a random value in $[\ell, r]$. In a `random delete`, we delete a random key κ, where κ is generated as in the case of the `random search`. We also performed experiments with non-random instances, involving the set of words in one English dictionary and a few English books.

2 Evaluation of Existing Dictionaries

In this section we report the results of our experimental study on the asymptotically optimal resilient dictionaries `Det` and `Rand`, plus a standard (non-resilient) implementation `Avl` of an AVL binary search tree. All the results reported here are averaged over 20 (random) input instances. We observed that those results do not change substantially by considering a larger number of instances. For each experiment discussed here, we let $\delta = 2^i$ for $i = 2, 3, \ldots, 10$. This range of values of δ includes both cases where the running time is dominated by the $O(\log n)$ term and cases where the $O(\delta)$ term dominates.

The Importance of Being Resilient. First of all, we wish to test how much the lack of resiliency affects the accuracy of a non-resilient dictionary. To that aim, we measured the fraction of `search` operations which fail to provide a correct answer in `Avl` (which is not resilient), for increasing values of δ. Since `Avl` is affected only by the actual number of faults, in all the experiments we assumed $\alpha = \delta$.

We observed experimentally that even a few memory faults make `Avl` crash very soon, due to corrupted pointers. In order to make a more meaningful test, we implemented a variant of `Avl` which halts the current operation without crashing when that operation tries to follow a corrupted pointer. Even with this (partially resilient) variant of `Avl`, very few faults make a large fraction of the `search` operations fail. This is not surprising, since the corruption of a pointer at top levels in the AVL tree causes the loss of a constant fraction of the keys.

In order to illustrate this phenomenon, let us consider the following input sequence. First of all, we populate the dictionary via a sequence of 10^6 `random insert` operations. Then we corrupt the data structure by injecting $\alpha = \delta$ faults. Finally, we generate a sequence of 10^5 `random search` operations, and count how many operations fail because of a corrupted pointer.

[3] Our experiments have been carried out on a workstation equipped with two Opteron processors with 2 GHz clock rate and 64 bit address space, 2 GB RAM, 1 MB L2 cache, and 64 KB L1 data/instruction cache. The workstation runs Linux Kernel 2.6.11. All programs have been compiled through the GNU `gcc` compiler version 3.3.5 with optimization level `O3`. The full package, including algorithm implementations, and a test program, is publicly available at the URL: `http://www.statistica.uniroma1.it/users/uferraro/experim/faultySearch/`

Our results for this case are shown in Figure 1(a). As expected, the number of incorrect `search` operations grows with $\alpha = \delta$. More interestingly, the expected number of wrong operations is roughly one order of magnitude larger than the number of faults. For example, in the case $\delta = 1024$ (i.e., δ is roughly 0.1% of the number of keys), roughly 1100 `search` operations fail (i.e., roughly 1% of the operations). This suggests that using resilient implementations can be really worth the effort.

The Cost of Resiliency. In order to evaluate how expensive resiliency is, we experimentally compared the time and space performance of `Det`, `Rand` and `Avl`. In order to test the sensitivity of resilient algorithms to the actual number of faults α (besides to δ), we considered different values of the ratio α/δ: we next provide results only for the extreme cases $\alpha = 0$ and $\alpha = \delta$. For `Avl` we only considered $\alpha = 0$, since in the presence of faults the space and time usage of this non-resilient dictionary is not very meaningful (as shown in previous subsection).

The results obtained for the following input instance summarize the qualitative behaviors that we observed. We bulk-load the dictionary via a random sequence of 10^6 `random insert` operations. Then we generate a sequence of 10^5 operations, where each operation is uniformly chosen to be a `random insert`, `random search`, or `random delete`. The adversary injects α faults during the last 10^5 operations. We consider the average time per operation of the latter operations. The space usage is measured at the end of the process.

The results concerning the running time for the above instance are shown in Figure 1(b)-(c). As expected, `Avl` is much faster than the resilient dictionaries. This suggests that resilient implementations should be used only when the risk of memory faults is concrete.

Interestingly enough, the time performance of `Rand` and `Det` is very sensitive to δ, but it is almost not affected by α. This suggests that, in any practical implementation, δ should be chosen very carefully: underestimating δ compromises the resiliency of the dictionary while overestimating it might increase the running time drammatically.

`Rand` is rather faster than `Det` for large values of δ, but it is much slower than `Det` when δ is small. Not surprisingly, the running time of `Det` grows with δ. More interestingly, the running time of `Rand` initially quickly decreases with δ and then slowly increases. This behavior arises from the fact that, for small values of δ, `Rand` often restructures the AVL search tree in order to keep the number of keys in each interval within the range $[\delta/2, 2\delta]$: this operation dominates the running time. This interpretation is confirmed in next section, where we consider a variant of `Rand` which maintains intervals with a number of keys in $[a\delta/2, 2a\delta]$, for a proper constant $a > 1$. For example, in the case $a = 32$ the running time of this variant of `Rand` is monotonically increasing in δ.

The results concerning the space usage for the mentioned case are shown in Figure 1(d). The space usage in the figure are given as multiples of the total space occupied by keys (which is a lower bound on the space needed). We first of all observe that the space usage of `Rand` and `Det` is almost not affected by δ (this is obvious in the case of `Avl`). In particular, the space usage initially decreases with

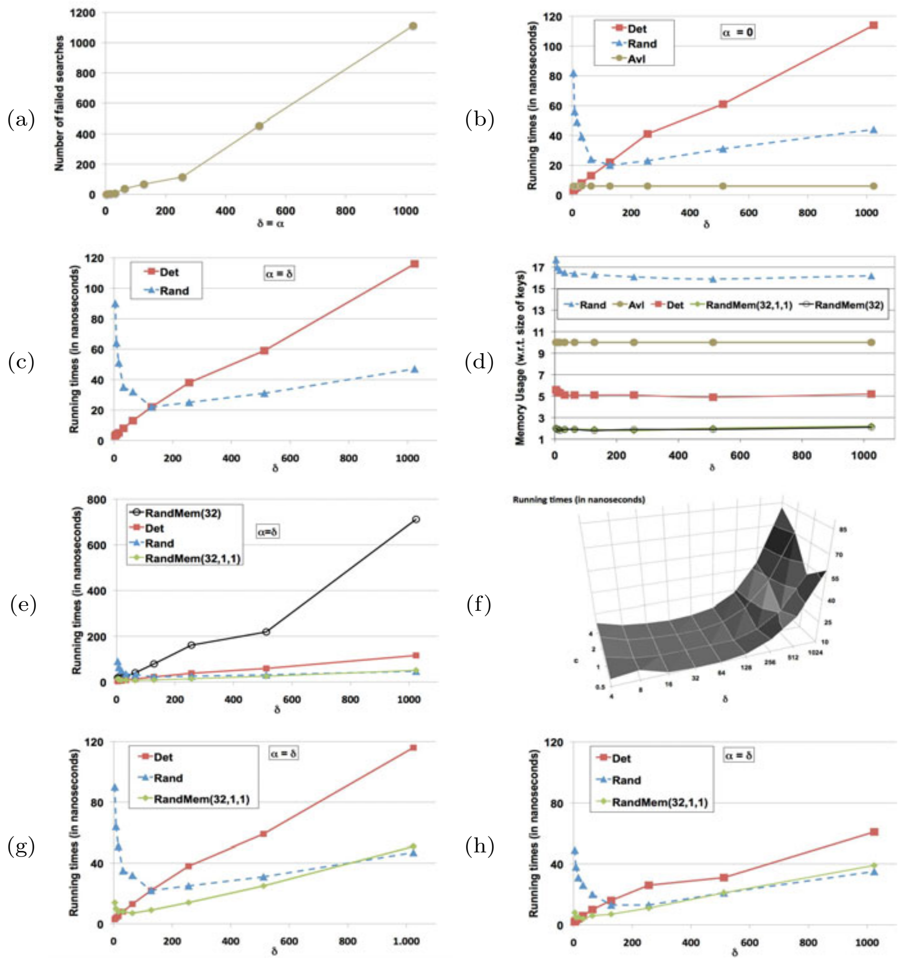


Fig. 1. We use $\alpha = 0$ in (b) and (d), and $\alpha = \delta$ in the other cases. **(a)** Number of failed `search` operations of `Avl` when processing a sequence of 10^5 random searches on a dictionary containing 10^6 random keys. **(b)+(c)** Average running time per operation of `Rand`, `Det` and `Avl`, when processing a sequence of 10^5 random operations on a dictionary initially containing 10^6 random keys. **(d)+(e)** Average memory usage, as multiples of the total size of the keys, and running time per operation of `Rand`, `Det` and `Avl`, `RandMem(32)` and `RandMem(32,1,1)`, when processing a sequence of 10^5 random operations on a dictionary initially containing 10^6 random keys. **(f)** Average running time per operation of `RandMem`(32,1,c), for different combinations of c, when processing a sequence of 10^5 random operations on a dictionary initially containing 10^6 random keys. **(g)+(h)** Average running time per operation of `Rand`, `Det` and `RandMem(32,1,1)`, when processing a sequence of 10^5 random operations on a dictionary initially containing 10^6 random keys, and when searching for all the 81965 words in "The Picture of Dorian Gray" on a dictionary initially containing 234936 distinct English words.

δ, and then reaches a stationary value rather quickly. This might seem surprising at a first glance. The reason for this behavior is that both dictionaries exploit a data structure containing $\Theta(n/\delta)$ nodes, each one of size $\Theta(\delta)$. Hence the space usage is $\Theta(n)$, irrespectively of δ. The experiments show that even the constants in the asymptotic notation are weakly related to δ.

Not surprisingly, `Rand` uses much more space than `Avl`. What is more surprising is that `Det` uses much less space than `Avl` (despite the fact that `Avl` does not need to take care of faults). This behavior might be explained by the fact that `Det` builds upon data structures developed in the context of algorithms for external memory. In more detail, the combination of buffering techniques and lazy updates in `Det` reduces the use of pointers with respect to `Avl`.

We remark that all dictionaries use much more space than the space occupied by keys only. In particular, the space usage of `Rand`, `Avl`, and `Det` is roughly 16, 10, and 5 times the total size of the keys, respectively. This space overhead is determined by the use of pointers for all those implementations. Furthermore, in the case of `Rand` and `Det` part of the space is wasted due to buffers which are partially empty. Such a large space overhead may not be acceptable in the applications, where keys alone already occupy huge amounts of memory. This is also the main motivation for the refined implementation `RandMem` described in next section.

3 A Refined Resilient Dictionary

Motivated by the large space overhead of `Rand`, in this section we describe a new randomized resilient dictionary `RandMem`, which is a (non-trivial) variant of `Rand`. `RandMem` has optimal asymptotic time and space complexity, but it performs better in practice. In particular, it uses an amount of space closer to the lower bound given by the total space occupied by keys. Furthermore, it is sometimes slightly slower and often even faster than `Rand` and `Det`.

Our refined data structure is based on a careful combination of the results developed in the context of static sorting and searching in faulty memories [10,13], together with some new, simple ideas. This machinery is exploited to implement more efficiently the part of each operation which involves the keys in a given interval. The rest of the implementation is exactly as in `Rand`. In particular, the structure of the AVL tree and the way it is explored and updated is the same as before. Rather than describing directly `RandMem`, we illustrate the logical steps which led us to its development.

Reducing Space Usage. A simple way to reduce the space overhead of `Rand` is modifying the algorithm so that, for a proper parameter $a > 1$, the number of keys in each interval is in the range $[a\delta/2, 2a\delta]$ rather than $[\delta/2, 2\delta]$ (adapting the update operations consequently). Intuitively, the larger is a, the smaller is the number of intervals and hence the space overhead. This allows one to reach asymptotically a space usage of at most 4 times the total space occupied by keys, the worst case being when all the intervals contain $a\delta/2$ keys each (while the space reserved is roughly $2a\delta$ per interval). In practice, the space usage might even be smaller since we expect to see an intermediate number of keys in each interval (rather than a number close to the lower bound).

We tested this variant of `Rand`, that we called `RandMem`(a), for growing values of a. As expected, the space usage is a decreasing function of a. In Figure 1(d) we report on the space usage of `RandMem`(32) on the same input instances used in Section 2. The memory usage is much smaller than the one of `Rand` and `Det`, and it is roughly twice the space occupied by keys. In our experiments, larger values of a do not provide any substantial reduction of the space usage.

We remark that it is not hard to achieve a space usage arbitrarily close to the total size of the keys (for growing values of a). The idea is requiring that each interval contains between $(1-\epsilon)a\delta$ and $(1+\epsilon)a\delta$ keys, for a small constant $\epsilon > 0$. Of course, this alternative implementation implies a more frequent restructuring of the search tree, which a consequent negative impact on the running time (in particular, the running time is an increasing function of $1/\epsilon$). We do not discuss here the experimental results for this alternative implementation due to space constraints.

In the following A denotes the buffer (of size $2a\delta$) containing the keys associated to the interval I under consideration. We implicitly assume that empty positions of A (to the right of the buffer) are marked with a special value ∞, which stands for a value larger than any feasible key. Note that the adversary is allowed to write ∞ in a memory location. In the next paragraphs we show how to speed up each operation.

Reducing Searching Time. The main drawback of the approach described above is that, in each operation, `RandMem`(a) needs to linearly scan a buffer A of size $\Theta(a\delta)$: for large values of a this operation is very expensive. This is witnessed by the experiment shown in Figure 1(e), where we report on the running time of `RandMem`(32) for the same input instances as in Section 2.

One way to reduce the `search` time is keeping all the keys in A faithfully sorted. Each time we insert or delete a key from A, we faithfully sort its keys with a (static) resilient sorting algorithm: here we used the optimal resilient sorting algorithm `ResSort` described in [10]. In order to search for a key, we exploit a resilient (static) searching algorithm. In particular, we decided to implement the simple randomized searching algorithm `ResSearch` in [10].

Reducing Insertion Time. Of course, keeping A faithfully sorted makes `insert` and `delete` operations more expensive.

In order to reduce the `insert` time, without increasing substantially the time needed for the other two operations, we introduce a secondary buffer B of size $b\delta$, for a proper constant $a > b > 0$. All the keys are initially stored in B (which is maintained as an unsorted sequence of keys). Like for A, empty positions of B are set to ∞. When B is full, buffers A and B are merged into A, so that the resulting buffer A is faithfully sorted. In order to perform this merging, first of all we faithfully sort B by means of the classical `SelectionSort` algorithm (which turns out to be a resilient sorting algorithm [10]). For small enough b, `SelectionSort` is faster than `ResSort`. Then we apply to A and B the procedure `UnbalancedMerge`, which is one of the key procedures used in

ResSort. This procedure takes as input two faithfully sorted sequences, one of which is much shorter than the other, and outputs a faithfully sorted sequence containing all the keys of the original sequences. We remark that merging two faithfully sorted sequences is faster than sorting everything from scratch (using, e.g., ResSort). Of course, now search and delete operations have to take buffer B into consideration. In particular, those operations involve a linear scan of B.

The main advantage of B is that it allows to spend $O(b\delta)$ time per insertion, and only after $\Omega(b\delta)$ insertions one needs to merge A and B, which costs $O((b\delta)^2 + a\delta + \delta^2)$ time. However, this also introduces a $\Theta(b\delta)$ overhead in each search and delete operation. Henceforth, b has to be tuned carefully.

Reducing Deletion Time. It remains to reduce the delete time, without increasing too much the time needed for the other operations. Deleting an element in B is a cheap operation, therefore we focus on the deletion of an element in A.

A natural approach to delete an element is replacing it with the special value ∞ (recall that empty positions are set to ∞ as well). When A and B are merged after one insert, we can easily get rid of this extra values ∞. Note that the ∞ entries introduced by deletions are not correctly sorted despite the fact that they are not faults introduced by the adversary. As a consequence, ResSearch is not guaranteed to work correctly. However, we can solve the problem by letting ResSearch run with respect to a number $\delta' = x + \delta$ of faults, where x is the current number of deleted elements. In other terms, we can consider the x deleted entries as faults introduced by the adversary. When x is large, the search (and hence delete) operation becomes slower. Hence, we fix a threshold $\tau = c\delta$ for a proper constant $c > 0$. When x reaches τ, we *compress* A so that the values different from ∞ occupy the first positions in the buffer (while the final ∞ entries correspond to empty positions). The $\Theta(a\delta)$ cost of this compression is amortized over $\Omega(c\delta)$ deletions.

It remains to explain how we keep track of x. One natural way to do that is using a resilient variable (i.e., $2\delta + 1$ copies of one variable) as a counter. Each time a new deletion occurs, we increment the counter and we reset it when it reaches τ (this costs roughly 4δ per delete). Here we adopt an alternative, novel approach, which is better for small values of c both in terms of time and of space. We define an array C of size τ, which is used as a *unary counter*. In particular, each time a new deletion occurs, we linearly scan C, searching for a 0 entry, and we replace it with a 1. If no 0 entry exists, C is reset to 0 and A is compressed. The number of 1's in C, i.e. the number of deletions in A, is denoted by $|C|$. Note that, differently from the case of the resilient counter, the adversary might reset some entries of C: in that case $|C|$ underestimates the actual number of deletions in A. In principle, this might compromise the correctness of ResSearch. However, running ResSearch with $\delta' = |C| + \delta$ is still correct. In fact, let α_C and α_A be the total number of faults occurring in C and A, respectively. The number of deletions in A is at most $|C| + \alpha_C$ (each deletion in A not counted by C corresponds to a fault in C). Hence, the total number of unsorted elements in A (faults plus deletions) is at most $|C| + \alpha_C + \alpha_A \leq |C| + \delta$. Of course, the adversary might as well set some 0 entries of C to 1, hence anticipating the compression of

A. However, $\Omega(c\delta)$ such faults are needed to force a compression of cost $O(a\delta)$. Hence, this type of phenomenon does not affect the running time substantially. We remark that unary counters were not used before in the literature on reliable algorithms and data structures.

We next call `RandMem`(a,b,c) the variant of `RandMem`(a) which exploits the secondary buffer B and the unary counter C. It is worth to remark that the secondary buffer and the deletions might compromise the asymptotic optimality of the dictionary. In particular, it might happen that the set of intervals (and hence the AVL tree) is modified more often than every $\Omega(\delta)$ operations. By a simple computation, it turns out that $b \leq a/2$ and $c \leq a/8$ are sufficient conditions to avoid this situation, hence maintaining optimal $O(n)$ space complexity and $O(\log n + \delta)$ time per operation.

Experimental Evaluation. We tested `RandMem`(a,b,c) in different scenarios and for different values of the triple (a, b, c). We next restrict our attention to the input instances as considered in Section 2, with $\alpha = \delta$. Furthermore, we assume $a = 32$, which minimizes the space usage.

We experimentally observed that, for a given value of a, the running time of the data structure may vary greatly according to the choice of c while it is only partially influenced by the choice of b. For fixed values of a and b, the running time first decreases and then increases with c. This is the result of two opposite phenomena. On one hand, small values of c imply more frequent compressions of the primary buffer A, with a negative impact on the running time. On the other hand, small values of c reduce the maximum and average value of $\delta' = |C| + \delta$, hence making the searches on A faster. This behavior is visible in Figure 1(f), where we fixed $a = 32$ and $b = 1$, and considered different combinations of δ and c. In most cases, the best choice for c is 1.

We next focus on the case $(a, b, c) = (32, 1, 1)$. Figure 1(d) shows that that the space usage of `RandMem`(32,1,1) is essentially the same as `RandMem`(32) (hence, much better than both `Rand` and `Det`). Figure 1(g) shows that `RandMem`(32,1,1) is much faster than `Rand` for small values of δ, and slightly slower for large values of δ. The improvement of the performance for small values of δ is due to the use of a larger primary buffer, as mentioned in previous section. The fact that `RandMem`(32,1,1) becomes slower than `Rand` for large values of δ is not surprising, since the first data structure is more complicated (in order to save space). `RandMem`(32,1,1) is much faster than `Det` unless δ is very small.

Non-Random Data Sets. We tested resilient dictionaries also on non-random data sets, observing the same qualitative phenomena as with random instances. In particular, we considered the following experiment. First, we bulk-load the resilient dictionary considered with all the words of one English dictionary, and then we search for all the words in one English book. Figure 1(h) shows the average time per `search` of `Rand`, `Det` and `RandMem(32,1,1)`, when searching for the words in "The Picture of Dorian Gray". The high-level behavior of the running time is the same as with random instances. The smaller running time with respect to Figure 1(g) is due to the fact that in this experiment we considered `search` operations only: these operations turn out to be faster than `insert` and `delete` operations (which contribute to the average running time in Figure 1(g)).

The above results suggest that `RandMem` is probably the data structure of choice for practical applications.

References

1. Aslam, J.A., Dhagat, A.: Searching in the presence of linearly bounded errors. In: STOC, pp. 486–493 (1991)
2. Assaf, S., Upfal, E.: Fault-tolerant sorting networks. SIAM J. Discrete Math. 4(4), 472–480 (1991)
3. Aumann, Y., Bender, M.A.: Fault-tolerant data structures. In: FOCS, pp. 580–589 (1996)
4. Borgstrom, R.S., Rao Kosaraju, S.: Comparison based search in the presence of errors. In: STOC, pp. 130–136 (1993)
5. Boyer, R., Moore, S.: MJRTY - A fast majority vote algorithm. University of Texas Tech. Report (1982)
6. Brodal, G.S., Fagerberg, R., Finocchi, I., Grandoni, F., Italiano, G.F., Jorgensen, A.G., Moruz, G., Molhave, T.: Optimal Resilient Dynamic Dictionaries. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 347–358. Springer, Heidelberg (2007)
7. Chlebus, B.S., Gambin, A., Indyk, P.: Shared-memory simulations on a faulty-memory DMM. In: Meyer auf der Heide, F., Monien, B. (eds.) ICALP 1996. LNCS, vol. 1099, pp. 586–597. Springer, Heidelberg (1996)
8. Feige, U., Raghavan, P., Peleg, D., Upfal, E.: Computing with noisy information. SIAM J. Comput. 23, 1001–1018 (1994)
9. Ferraro-Petrillo, U., Finocchi, I., Italiano, G.F.: The Price of Resiliency: a Case Study on Sorting with Memory Faults. Algorithmica 53(4), 597–620 (2009)
10. Finocchi, I., Grandoni, F., Italiano, G.F.: Optimal sorting and searching in the presence of memory faults. Theor. Comput. Sci. 410(44), 4457–4470 (2009)
11. Finocchi, I., Grandoni, F., Italiano, G.F.: Resilient search trees. In: SODA, pp. 547–553 (2007)
12. Finocchi, I., Grandoni, F., Italiano, G.F.: Designing reliable algorithms in unreliable memories. Computer Science Review 1(2), 77–87 (2007)
13. Finocchi, I., Italiano, G.F.: Sorting and searching in faulty memories. Algorithmica 52(3), 309–332 (2008)
14. Hamdioui, S., Al-Ars, Z., de Goor, J.V., Rodgers, M.: Dynamic faults in random-access-memories: Concept, faults models and tests. J. of Electronic Testing: Theory and Applications 19, 195–205 (2003)
15. Henzinger, M.: The Past, Present, and Future of Web Search Engines. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, p. 3. Springer, Heidelberg (2004)
16. Henzinger, M.R.: Combinatorial algorithms for web search engines - three success stories. In: SODA (2007) (invited talk)
17. Kleitman, D.J., Meyer, A.R., Rivest, R.L., Spencer, J., Winklmann, K.: Coping with errors in binary search procedures. J. Comp. Syst. Sci. 20, 396–404 (1980)

18. Leighton, T., Ma, Y.: Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. SIAM J. Comput. 29(1), 258–273 (1999)
19. May, T.C., Woods, M.H.: Alpha-Particle-Induced Soft Errors In Dynamic Memories. IEEE Trans. Elect. Dev. 26(2) (1979)
20. Pelc, A.: Searching games with errors: Fifty years of coping with liars. Theoret. Comp. Sci. 270, 71–109 (2002)
21. Yao, A.C., Yao, F.F.: On fault-tolerant networks for sorting. SIAM J. on Comput. 14, 120–128 (1985)

Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure

Md. Mostofa Ali Patwary[1], Jean Blair[2], and Fredrik Manne[1]

[1] Department of Informatics, University of Bergen, N-5020 Bergen, Norway
`Mostofa.Patwary@ii.uib.no`, `fredrikm@ii.uib.no`
[2] Department of EE and CS, United States Military Academy,
West Point, NY 10996, USA
`Jean.Blair@usma.edu`

Abstract. The disjoint-set data structure is used to maintain a collection of non-overlapping sets of elements from a finite universe. Algorithms that operate on this data structure are often referred to as Union-Find algorithms. They are used in numerous practical applications and are also available in several software libraries. This paper presents an extensive experimental study comparing the time required to execute 55 variations of Union-Find algorithms. The study includes all the classical algorithms, several recently suggested enhancements, and also different combinations and optimizations of these. Our results clearly show that a somewhat forgotten simple algorithm developed by Rem in 1976 is the fastest, in spite of the fact that its worst-case time complexity is inferior to that of the commonly accepted "best" algorithms.

Keywords: Union-Find, Disjoint Set, Experimental Algorithms.

1 Introduction

Let U be a set of n distinct elements and let S_i denote a subset of U. Two sets S_1 and S_2 are disjoint if $S_1 \cap S_2 = \emptyset$. A disjoint-set data structure maintains a dynamic collection $\{S_1, S_2, \ldots, S_k\}$ of disjoint sets that together cover the universe U. Each set is identified by a representative x, which is usually some member of the set. The two main operations are to Find which set a given element belongs to by locating its representative element and to replace two existing sets with their Union. In addition, there is a Makeset operation which adds a new element to U as a singleton set.

The underlying data structure of each set is typically a rooted tree represented by a parent function $p(x) \in S_i$ for each $x \in U$; the element in the root of a tree satisfies $p(x) = x$ and is the representative of the set. Then Makeset(x) is achieved by setting $p(x) \leftarrow x$ and the output of Find(x) is the root of the tree containing x. This is found by following x's *find-path*, which is the path of parent pointers from x up to the root of x's tree. A set of algorithms that operate on this data structure is often referred to as a Union-Find algorithm.

This disjoint-set data structure is frequently used in practice, including in application areas such as image decomposition, clustering, sparse matrix computations, and graph algorithms. It is also a standard subject taught in most algorithms courses.

Early theoretical work established algorithms with worst-case time complexity $\Theta(n + m \cdot \alpha(m, n))$ for any combination of m Makeset, Union, and Find operations on n elements [2,3,15,16,17,18], where α is the very slowly growing inverse of Ackermann's function. These theoretically best classical algorithms include a standard Union method using either Link-by-Rank or Link-by-Size and a Find operation incorporating one of three standard compression techniques: Path-Compression, Path-Splitting, or Path-Halving. Other early algorithms either use a different compression technique like Collapsing or interleave the two Find operations embedded in a Union operation, as was the case with Rem's algorithm and a variant of it designed by Tarjan and van Leeuwen. The worst case time complexity of these variations are not optimal [18].

The current work presents an extensive experimental study comparing the time required to execute a sequence of Union operations, each with two embedded Find operations. Altogether, we consider 55 variations; 29 of these had been well studied in the theoretical literature by 1984. We call these the classical algorithms. The remaining 26 variations implement a number of improvements. Our results clearly show that a somewhat forgotten simple algorithm developed by Rem in 1976 [6] is the fastest, in spite of the fact that its worst-case complexity is inferior to that of the commonly accepted "best" algorithms.

Related experimental studies have compared only a few Union-Find algorithms, usually in the context of a specific software package. In particular, [10] and [8] compared only two and six Union-Find algorithms, respectively, in the context of sparse matrix factorization. The works in [19] and [20] compared eight and three Union-Find algorithms, respectively, in the setting of image processing. More recently, [13] compared a classic algorithm with a variation described here in the ipc subsection of Section 2.2. The most extensive previous experimental study was Hynes' masters thesis [9] where he compared the performance of 18 Union-Find algorithms used to find the connected components of a set of Erdös-Rényi style random graphs.

2 Union-Find Algorithms

This section outlines, to the best of our knowledge, the primary Union-Find algorithms. We also include a number of suggested enhancements designed to speed up implementations of these algorithms. Our presentation is from the viewpoint of its use in finding connected components of a graph $G(V, E)$ as shown in Algorithm 1. In this case, the Union-Find algorithm computes a minimal subset $S \subseteq E$ such that S is a spanning forest of G.

Algorithm 1. Use of Union-Find

```
1: $S \leftarrow \emptyset$
2: for each $x \in V$ do
3:     Makeset($x$)
4: for each $(x, y) \in E$ do
5:     if Find($x$) $\neq$ Find($y$) then
6:         Union($x, y$)
7:         $S \leftarrow S \cup \{(x, y)\}$
```

2.1 Classical Algorithms

Here we discuss the classical Union techniques and then present techniques for compressing trees during a Find operation. Finally, we describe classical algorithms that interleave the Find operations embedded in a Union along with a compression technique that can only be used with this type of algorithm.

Union techniques. The Union(x, y) operation merges the two sets containing x and y, typically by finding the roots of their respective trees and then linking them together by setting the parent pointer of one root to point to the other.

Clearly storing the results of the two Find operations on line 5 and then using these as input to the Union operation in line 6 will speed up Algorithm 1. This replacement for lines 5–7 in Algorithm 1 is known as Quick-Union (Quick) in [7]. Throughout the remainder of this paper we use Quick.

Three classic variations of the Union algorithm center around the method used to Link the two roots. Let r_x and r_y be the roots of the two trees that are to be merged. Then Union with Naive-Link (nl) arbitrarily chooses one of r_x and r_y and sets it to point to the other. This can result in a tree of height $O(n)$.

In Union with Link-by-Size (ls) we set the root of the tree containing the fewest nodes to point to the root of the other tree, arbitrarily breaking ties. To implement ls efficiently we maintain the size of the tree in the root. For the Union with Link-by-Rank (lr) operation we associate a rank value, initially set to 0, with each node. If two sets are to be merged and the roots have equal rank, then the rank of the root of the combined tree is increased by one. In all other lr operations the root with the lowest rank is set to point to the root with higher rank and all ranks remain unchanged. Note that when using lr the parent of a node x will always have higher rank than x. This is known as the *increasing rank property*. The union algorithm presented in most textbooks uses the Quick and lr enhancements to implement lines 5–7 of Algorithm 1.

Both ls and lr ensure that the find-path of an n vertex graph will never be longer than $\log n$. The alleged advantage of lr over ls is that a rank value requires less storage than a size value, since the rank of a root in a set containing n vertices will never be larger than $\log n$ [3]. Also, sizes must be updated with every Union operation whereas ranks need only be updated when the two roots have equal rank. On the other hand lr requires a test before each Link operation.

Compression techniques. Altogether we describe six classical compression techniques used to compact the tree, thereby speeding up subsequent Find operations. The term nf will represent a Find operation with no compression.

Using Path-Compression (pc) the find-path is traversed a second time after the root is found, setting all parent pointers to point to the root. Two alternatives to pc are Path-Splitting (ps) and Path-Halving (ph). With ps the parent pointer of every node on the find-path is set to point to its grandparent. This has the effect of partitioning the find-path nodes into two disjoint paths, both hanging off the root. In ph this process of pointing to a grandparent is only applied to every other node on the find-path. The advantage of ps and ph over

PC is that they can be performed without traversing the find-path a second time. On the other hand, PC compresses the tree more than either of the other two.

Note that when using ranks PC, PS, and PH all maintain the increasing rank property. Furthermore, any one of the three combined with either LR or LS has the same asymptotic time bound of $O(m \cdot \alpha(m, n))$ for any combination of m MAKESET, UNION, and FIND operations on n elements [3,16,18].

Other compression techniques include REVERSAL-OF-TYPE-k. With this the first node x on the find-path and the k last nodes are set to point to the root while the remaining nodes are set to point to x. In a REVERSAL-OF-TYPE-0 (R0) every node on the find-path from x, including the root, is set to point to x and x becomes the new root, thus changing the representative element of the set. Both R0 and REVERSAL-OF-TYPE-1 (R1) can be implemented efficiently, but for any values of $k > 1$ implementation is more elaborate and might require a second pass over the find-path [18]. We limit $k \leq 1$. Using either R0 or R1 with any of NL, LR, or LS gives an asymptotic running time of $O(n + m \log n)$ [18].

In COLLAPSING (CO) every node of a tree will point directly to the root so that all find-paths are no more than two nodes long. When merging two trees in a UNION operation, nodes of one of the trees are changed to point to the root of the other tree. To implement this efficiently the nodes are stored in a linked list using a sibling pointer in addition to the parent pointer. The asymptotic running time of CO with either LS or LR is $O(m + n \log n)$; CO with NL is $O(m + n^2)$ [18].

It is possible to combine any of the three different UNION methods with any of the seven compression techniques (including NF), thus giving rise to a total of 21 different algorithm combinations. We denote each of these algorithms by combining the abbreviation of its UNION method with the abbreviation of its compression technique (e.g., LRPC). The asymptotic running times of these classical algorithms are summarized in the tables on page 280 of [18].

Classical interleaved algorithms. INTERLEAVED (INT) algorithms differ from the UNION-FIND algorithms mentioned so far in that the two FIND operations in line 5 of Algorithm 1 are performed as one interleaved operation. The main idea is to move two pointers r_x and r_y alternatively along their respective find-paths such that if x and y are in the same component then $p(r_x) = p(r_y)$ when they reach their lowest common ancestor and processing can stop. Also, if x and y are in different components, then in certain cases the two components are linked together as soon as one of the pointers reaches a root. Thus, one root can be linked into a non-root node of the other tree. The main advantage of the INT algorithms is that they can avoid traversing portions of find-paths.

The first INT algorithm is Rem's algorithm (REM) [6]. Here it is assumed that each node has a unique identifier (*id*) that can be ordered (typically in the range 1 through n). In our presentation we let this identifier be the index of the node.

The constructed trees always have the property that a lower numbered node either points to a higher numbered node or to itself (if it is a root). Instead of performing FIND(x) and FIND(y) separately, these are executed simultaneously by first setting $r_x \leftarrow x$ and $r_y \leftarrow y$. Then whichever of r_x and r_y has the smaller parent value is moved one step upward in its tree. In this way it follows that if x

and y are in the same component then at some stage of the algorithm we will have $p(r_x) = p(r_y) =$ the lowest common proper ancestor of x and y. The algorithm tests for this condition in each iteration and can, in this case immediately stop.

As originally presented, Rem integrates the Union operation with a compression technique known as Splicing (sp). In the case when r_x is to be moved to $p(r_x)$ it works as follows: just before this operation, r_x is stored in a temporary variable z and then, just before moving r_x up to its parent $p(z)$, $p(r_x)$ is set to $p(r_y)$, making the subtree rooted at r_x a sibling of r_y. This neither compromises the increasing parent property (because $p(r_x) < p(r_y)$) nor invalidates the set structures (because the two sets will have been merged when the operation ends.) The effect of sp is that each new parent has a higher value than the value of the old parent, thus compressing the tree. The full algorithm is given as Algorithm 2. The running time of Rem with sp (RemSP) is $O(m \log_{(2+m/n)} n)$ [18].

Algorithm 2. RemSP(x, y)

```
1: $r_x \leftarrow x$, $r_y \leftarrow y$
2: while $p(r_x) \neq p(r_y)$ do
3:     if $p(r_x) < p(r_y)$ then
4:         if $r_x = p(r_x)$ then
5:             $p(r_x) \leftarrow p(r_y)$, break
6:         $z \leftarrow r_x$, $p(r_x) \leftarrow p(r_y)$, $r_x \leftarrow p(z)$
7:     else
8:         if $r_y = p(r_y)$ then
9:             $p(r_y) \leftarrow p(r_x)$, break
10:        $z \leftarrow r_y$, $p(r_y) \leftarrow p(r_x)$, $r_y \leftarrow p(z)$
```

Tarjan and van Leeuwen present a variant of Rem that uses ranks rather than identifiers. This algorithm is slightly more complicated than Rem, as it also checks if two roots of equal rank are being merged and if so updates the rank values appropriately. Details are given on page 279 of [18]. We label this algorithm as TvL. The running time of TvL with sp (TvLsp) is $O(m \cdot \alpha(m, n))$.

Note that sp can easily be replaced in either Rem or TvL with either pc or ps. However, it does not make sense to use ph with either because ph might move one of r_x and r_y past the other without discovering that they are in fact in the same tree. Also, since r0 and r1 would move a lower numbered (or ranked) node above higher numbered (ranked) nodes, thus breaking the increasing (rank or *id*) property, we will not combine an Int algorithm with either r0 or r1.

2.2 Implementation Enhancements

We now consider three different ways in which the classical algorithms can be made to run faster by: *i)* making the algorithm terminate faster, *ii)* rewriting so that the most likely case is checked first, and *iii)* reducing memory requirements.

Immediate parent check (ipc). This is a recent enhancement that checks before beginning Quick if x and y have the same parent. If they do, Quick is not executed, otherwise execution continues. This idea is motivated by the fact that trees often have height one, and hence it is likely that two nodes in the same tree will have the same parent. The method was introduced by Osipov et al. [13] and used together with lr and pc in an algorithm to compute minimum weight spanning trees. ipc can be combined with any classical algorithm except Rem, which already implements the ipc test.

Better interleaved algorithms. The TvL algorithm, as presented in [18], can be combined with the IPC enhancement. However, the TvL algorithm will already, in each iteration of the main loop, check for $p(r_x) = p(r_y)$ and break the current loop iteration if this is the case. Still, three comparisons are needed before this condition is discovered. We therefore move this test to the top of the main loop so that the loop is only executed while $p(r_x) \neq p(r_y)$. (This is similar to the IPC test.) In addition, it is possible to handle the remaining cases when $rank(p(r_x)) = rank(p(r_y))$ together with the case when $rank(p(r_x)) < rank(p(r_y))$. For details see [14]. This will, in most cases, either reduce or at least not increase the number of comparisons; the only exception is when $rank(p(r_x)) < rank(p(r_y))$, which requires either one or two more comparisons. We call this new enhanced implementation eTvL.

A different variation of the TvL algorithm, called the ZIGZAG (ZZ) algorithm was used in [11] for designing a parallel UNION-FIND algorithm where each tree could span across several processors on a distributed memory computer. The main difference between the ZZ algorithm and eTvL is that the ZZ algorithm compares the ranks of r_x and r_y rather than the ranks of $p(r_x)$ and $p(r_y)$. Due to this it does not make sense to combine the ZZ algorithm with SP.

Memory smart algorithms. We now look at ways to reduce the amount of memory used by each algorithm. In the algorithms described so far each node has a parent pointer and, for some algorithms, either a size or rank value. In addition, for the CO algorithm each node has a sibling pointer. It follows that we will have between one and three fields in the record for each node. (Recall that we use the corresponding node's index into the array of records as its "name").

It is well known, although as far as we know undocumented, that when the parent pointer values are integers one can eliminate one of the fields for most UNION-FIND implementations. The idea capitalizes on the fact that usually only the root of a tree needs to have a rank or size value. Moreover, for the root the parent pointer is only used to signal that the current node is in fact a root. Thus it is possible to save one field by coding the size or rank of the root into its parent pointer, while still maintaining the "root property." This can be achieved by setting the parent pointer of any root equal to its negated rank (or size) value.

This MEMORY-SMART (MS) enhancement of combining the rank/size field with the parent pointer can be incorporated into any of the classical algorithms except those using an INT algorithm (because they require maintaining the rank at every node, not just the root) or PH (because PH changes parent pointers to the grandparent value, which, if negative, will mess up the structure of the tree.) MS can also be combined with the IPC enhancement. Because Rem does not use either size or rank, we will also classify it as an MS algorithm.

3 Experiments and Results

For the experiments we used a Dell PC with an Intel Core 2 Duo 2.4 GHz processor and 4MB of shared level 2 cache, and running Fedora 10. All algorithms were implemented in C++ and compiled with GCC using the -O3 flag.

We used three test sets. The first consists of nine real world graphs (rw) of varying sizes drawn from different application areas such as linear programming, medical science, structural engineering, civil engineering, and the automotive industry [4]. The second includes five synthetic small world graphs (sw) and the third contains six synthetic Erdös-Rényi style random graphs (er). For each synthetic graph (sw or er), we generated five different random graphs with the same number of vertices and with the same edge probability using the GTGraph package [1]. Statistics reported here about these graphs are an average for the five different random graphs of that type and size. For structural properties of the test sets as well as additional figures see [14].

To compute the run-time of an algorithm for a given graph, we execute the algorithm five times using each of five different random orderings of the edges, taking the average time as the result. For test sets sw and er this is also done for each graph. Hence we compute the average run-time of each graph in rw by taking the average of 25 total runs and for sw and er by taking the average of 125 total runs. Algorithms stop if and when they find that the entire graph is a single connected component. The time for all runs of reasonable algorithms (not including the extremely slow algorithms NL with no compression and NL with CO) ranged from 0.0084 seconds to 28.7544 seconds.

We now present the results of experiments from 55 different algorithms. We first consider the classical algorithms and then using the enhancements presented in Section 2.2. Finally, we compare and discuss the 10 overall fastest algorithms. For each type of algorithm we give a table in which each cell represents an algorithm that combines the row's union method with the column's compression technique. The combinations for crossed out cells are either not possible or nonsensical. The rows with gray background are repeated from an earlier table.

Throughout we will say that an algorithm X *dominates* another algorithm Y if X performs at least as well as Y (in terms of run-time) on every input graph. For illustrative purposes we will pick four specific dominating algorithms numbered according to the order in which they are first applied. These will be marked in the tables with their number inside a colored circle, as in ❶. If algorithm X dominates algorithm Y, an abbreviation for X with its number as a subscript will appear in Y's cell of the table, as in LRPC$_1$. Algorithms that are not dominated by any other algorithm are marked as undominated.

Classical Algorithms (Table 1). The two algorithms LRPC and LRPH are generally accepted as best, and so we begin by examining these. Although they dominate a large majority, RemSP dominates still more. This gives us the first three dominating algorithms. LRPC$_1$ dominates 14 algorithms. Three additional algorithms are dominated by LRPH$_2$, including LRPC; hence, LRPH also dominates all algorithms dominated by LRPC. Figure 1(a) shows the relative performance of the ten remaining algorithms dominated by RemSP$_3$. Note that RemSP dominates LRPH. Because LRPC dominates both LSPC and TvLPC, our experiments show that LR is a better UNION method to combine with PC. Only two algorithms are undominated. In the following we do not report results for algorithms using NF,

Table 1. Relative performance of the classical Union-Find algorithms

	NF	PC	PH	PS	CO	R0	R1	SP
NL	LRPC$_1$	LRPH$_2$	RemSP$_3$	RemSP$_3$	LRPC$_1$	LRPC$_1$	LRPC$_1$	╳
LR	LRPC$_1$	❶ LRPH$_2$	❷ RemSP$_3$	RemSP$_3$	RemSP$_3$	LRPC$_1$	LRPC$_1$	╳
LS	LRPC$_1$	LRPC$_1$	RemSP$_3$	RemSP$_3$	RemSP$_3$	LRPC$_1$	LRPC$_1$	╳
Rem	LRPC$_1$	RemSP$_3$	╳	undom.	╳	╳	╳	❸ undom.
TvL	LRPC$_1$	LRPC$_1$	╳	RemSP$_3$	╳	╳	╳	LRPH$_2$

R0, or R1 as these compression techniques consistently require several orders of magnitude more time than the others.

IPC Algorithms (Table 2). The algorithms for which using IPC always performed better than the corresponding non-IPC version are marked in the table with an uparrow (↑). Note that Rem is, by definition, an IPC algorithm, and hence, there are no changes between its “IPC” version and its “non-IPC” version; they are the same algorithm. In each of the other five cases, using IPC is sometimes better than not, and vice versa. No IPC version is consistently worse than its non-IPC version.

Table 2. IPC relative performance

	PC	PH	PS	SP
IPC-LR	IPC-LRPS$_4$ ↑	IPC-LRPS$_4$	❹ RemSP$_3$	╳
IPC-LS	RemSP$_3$ ↑	RemSP$_3$	RemSP$_3$	╳
Rem	RemSP$_3$	╳	undom.	❸ undom.
IPC-TvL	IPC-LRPS$_4$ ↑	╳	RemSP$_3$	RemSP$_3$ ↑

One algorithm, IPC-LRPS$_4$, dominates three others. Also, RemSP dominates that algorithm and others. Figure 1(b) shows the performance of the remaining six dominated algorithms relative to Rem–SP. Among the IPC--LR algorithms, PS dominates the other two compression techniques. Note also that unlike the results of the previous subsection where LRPC dominated LSPC, neither IPC-LRPC nor IPC-LSPC dominates the other. In general IPC-LSPC performed better than IPC-LRPC.

Interleaved Algorithms (Table 3). RemSP dominates the five new Int algorithms. Figure 1(c) shows the performance of each of these relative to RemSP. The performance of the Int algorithms was impacted more by the compression technique than by the union method. Also, PC is considerably worse (by approximately 50%) than either PS or SP.

Memory-smart Algorithms (Table 4). RemSP dominates six of the MS algorithms. Figure 1(d) shows the relative performance of these relative to RemSP.

Note that MS-IPC-LSPS and MS-IPC-LSPC come close to RemSP, and, in fact, these are the only two dominated algorithms that are among the 10 fastest algorithms using the metric described next.

The Fastest Algorithms. We now compare all algorithms using a different metric than the "dominates" technique. We begin by calculating, for each input graph and each algorithm, its run-time relative to the best algorithm for that graph (Global-Min). For each algorithm and each type of input (rw, sw, er) we then compute the average relative time over all input of that type. The average of these three averages is then used to rank order the algorithms.

The results for the top ten ranked algorithms are given in Table 5. Each cell contains both the algorithm's rank for the given type of graph and its relative timing reported as a percent. The last row in the table is included to show how far the last ranked algorithm is from the algorithms that are not in the top 10.

When considering rw, er, and "all graphs," the next best algorithm is ranked 11 and is 8.68%, 10.68%, and 8.12% worse than the slowest algorithm reported in the table, respectively. For the sw graphs, however, the fastest algorithm not in the table is faster than four of the algorithms in the table and only 2.66% slower than the algorithm ranked 6 (MS-LRCO). For sw graphs five algorithms not in the table were faster than the slowest of those reported in the table.

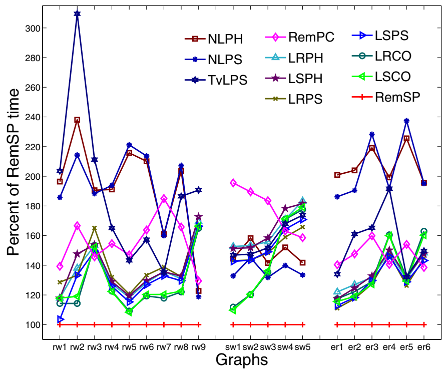
Table 3. Int relative performance

	PC	PS	SP
Rem	RemSP$_3$	undom.	❸ undom.
TvL	LRPC$_1$	RemSP$_3$	LRPH$_2$
IPC-TvL	IPC-LRPS$_4$	RemSP$_3$	RemSP$_3$
eTvL	RemSP$_3$	RemSP$_3$	RemSP$_3$
ZZ	RemSP$_3$	RemSP$_3$	╳

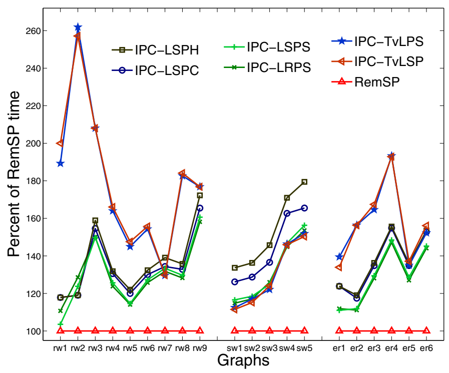
The relative performance of the five algorithms with ranks 1-5 on the overall average is plotted in Figure 1(e). The figure clearly shows RemSP outperformed all other algorithms. Notice that LS and LR with all other variations on the algorithm remaining constant tend to have similar performance trends. Furthermore, neither LS nor LR consistently outperformed the other. Of the top 10 algorithms all use the MS enhancement and out of these all but MS-IPC-LRPC and MS-IPC-LSPC are one-pass algorithms. However, out of the top five algorithms two use PC. We note that PS has the advantage over PH, perhaps because it is easier to combine with MS.

Table 4. MS relative performance

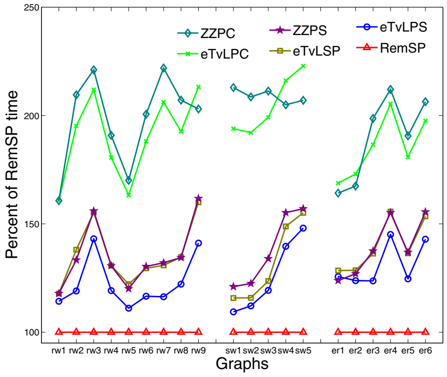
	PC	PS	CO
MS-NL	RemSP$_3$	RemSP$_3$	╳
MS-LR	RemSP$_3$$\uparrow$	undom.$\uparrow$	undom.$\uparrow$
MS-LS	RemSP$_3$$\uparrow$	undom. $\uparrow$	undom. $\uparrow$
MS-IPC-LR	undom.$\uparrow$	undom.	╳
MS-IPC-LS	RemSP$_3$	RemSP$_3$	╳
Rem	RemSP$_3$	undom.	╳



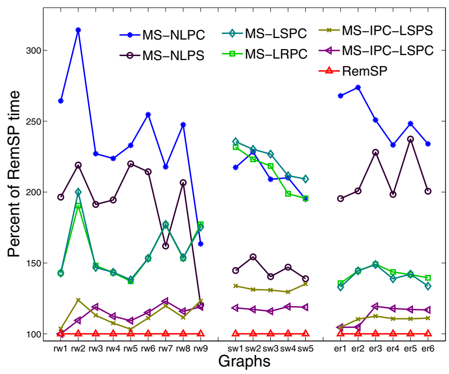
(a) Classical dominated by RemSP



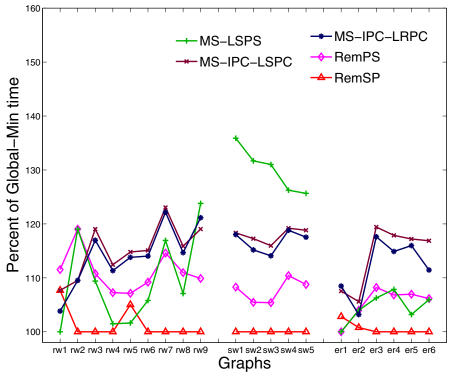
(b) IPC dominated by RemSP



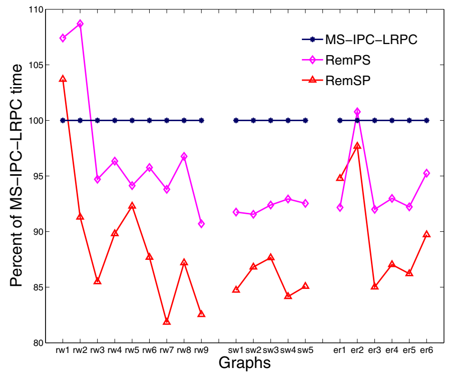
(c) Int dominated by RemSP



(d) Memory-Smart dominated by RemSP



(e) The five fastest algorithms



(f) Improvement over best enhanced classical algorithm (MS-IPC-LRPC)

Fig. 1. Relative performance of Union-Find algorithms

Table 5. Rank order(% of GLOBAL-MIN) of the fastest algorithms based on graph type

	All graphs	Real-World	Small-World	Erdős-Rényi
RemSP	1 (100.67)	1 (101.41)	1 (100.00)	1 (100.61)
RemPS	2 (108.07)	5 (111.16)	2 (107.67)	4 (105.36)
MS-IPC-LRPC	3 (114.28)	6 (114.17)	3 (116.73)	7 (111.94)
MS-LSPS	4 (114.71)	2 (109.47)	10 (130.1)	2 (104.55)
MS-IPC-LSPC	5 (115.73)	8 (115.17)	4 (117.92)	8 (114.09)
MS-LRPS	6 (115.90)	4 (111.02)	13 (131.65)	3 (105.05)
MS-IPC-LRPS	7 (116.90)	3 (111.00)	11 (131.22)	5 (108.48)
MS-LSCO	8 (118.29)	9 (115.21)	6 (123.05)	9 (116.61)
MS-LRCO	9 (118.91)	10 (115.47)	5 (123.04)	10 (118.22)
MS-IPC-LSPS	10 (119.08)	7 (114.47)	15 (132.15)	6 (110.62)
Fastest not listed	11 (127.20)	11 (124.15)	7 (125.71)	11 (128.90)

Unlike the sparse matrix studies in [8,10,20] we found that using either LR or LS does pay off. Hynes [9] and later Wassenberg et al. [19] found that CO was the best choice. Our results also show that CO used with either LR or LS is one of the faster algorithms, but only if used with the MS enhancement. However, on average it is outperformed by several classical algorithms (such as LSPC and LRPC) if these are enhanced with IPC and MS.

The clear winner in our study was RemSP. This was the fastest algorithm for all types of graphs. On average it was 13.52% faster (with a range of −3.57% to 22.18%) compared to the best non-Rem algorithm and 7.34% faster (with a range of −2.75% to 19.05%) than the second best Rem algorithm. We believe that this is due to several factors: it has low memory overhead; INT algorithms perform less operations than other classical algorithms; it incorporates the IPC enhancement at every step of traversal, not only for the two initial nodes; and even when integrated with SP the algorithm is relatively simple with few conditional statements.

4 Concluding Remarks

This paper reports the findings of 1600 experiments on each of 53 different UNION-FIND algorithms: 27 classical variations that were studied from a theoretical perspective up through the 1980s, nine IPC variations, five more INT variations, and 12 additional MS variations. We also ran two experiments using the very slow algorithms NLNF and NLCO. In order to validate the results, we reran the 84,800 experiments on the same machine. While the absolute times varied somewhat, the same set of algorithms had the top five ranks as did the set of algorithms with the top 10 ranks, and RemSP remained the clear top performer.

We have also constructed spanning forests using both DFS and BFS, finding that use of the UNION-FIND algorithms are substantially more efficient.

The most significant result is that RemSP substantially outperforms LRPC even though RemSP is theoretically inferior to LRPC. This is even more surprising because LRPC is both simple and elegant, is well studied in the literature, is often implemented for real-world applications, and typically is taught as best in standard algorithms courses. In spite of this RemSP improved over LRPC by an average of 52.88%, with a range from 38.53% to 66.05%. Furthermore, it improved over LRPH (which others have argued uses the best one-pass compression technique) by an average of 28.60%, with a range from 15.15% to 45.44%.

Even when incorporating the MS and IPC enhancements, RemSP still improved over these other two classical algorithms on all inputs except one (rw1), where MS-IPC-LRPC improved over RemSP by only 3.57%. On average, RemSP improves over MS-IPC-LRPC by 11.91%, with a range from -3.70% to 18.15%. The savings incurred over the MS-IPC-LRPC are illustrated in Figure 1(f) where the times for the top two ranked algorithms are plotted relative to the time for MS-IPC-LRPC.

To verify that our results hold regardless of the cache size, we ran experiments using twice, three times, and four times the memory for each node (simulating a smaller cache). The relative times for the algorithms under each of these scenarios were not significantly different than with the experiments reported here.

We believe that our results should have implications for developers of software libraries like [5] and [12], which currently only implement LRPC and LRPH in the first case and LSPC in the second. Initial profiling experiments show that RemSP gives both fewer cache misses and fewer parent jumps than the classical algorithms. We postpone discussion of these results until the full version of the paper.

These UNION-FIND experiments were conducted under the guise of finding the connected components of a graph. As such, the sequences of operations tested were all UNION operations as defined by the edges in graphs without multiedges. It would be interesting to study the performances of these algorithms for arbitrary sequences of intermixed MAKESET, UNION, and FIND operations.

References

1. Bader, D.A., Madduri, K.: GTGraph: A synthetic graph generator suite (2006), http://www.cc.gatech.edu/~kamesh/GTgraph
2. Banachowski, L.: A complement to Tarjan's result about the lower bound on the complexity of the set union problem. Inf. Proc. Letters 11, 59–65 (1980)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
4. Davis, T.A.: University of Florida sparse matrix collection. Submitted to ACM Transactions on Mathematical Software
5. Dawes, B., Abrahams, D.: The Boost C++ libraries (2009), www.boost.org
6. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
7. Galil, Z., Italiano, G.F.: Data structures and algorithms for disjoint set union problems. ACM Comput. Surv. 23(3), 319–344 (1991)

8. Gilbert, J.R., Ng, E.G., Peyton, B.W.: An efficient algorithm to compute row and column counts for sparse Cholesky factorization. SIAM J. Matrix Anal. Appl. 15(4), 1075–1091 (1994)
9. Hynes, R.: A new class of set union algorithms. Master's thesis, Department of Computer Science, University of Toronto, Canada (1998)
10. Liu, J.W.H.: The role of elimination trees in sparse factorization. SIAM J. Matrix Anal. Appl. 11, 134–172 (1990)
11. Manne, F., Patwary, M.M.A.: A scalable parallel union-find algorithm for distributed memory computers. To appear in proc. PPAM 2009. LNCS. Springer, Heidelberg (2009)
12. Melhorn, K., Näher, S.: LEDA, A Platform for Combinatorial Geometric Computing. Cambridge University Press, Cambridge (1999)
13. Osipov, V., Sanders, P., Singler, J.: The filter-Kruskal minimum spanning tree algorithm. In: ALENEX, pp. 52–61 (2009)
14. Patwary, M.M.A., Blair, J., Manne, F.: Efficient Union-Find implementations. Tech. Report 393, Dept. Computer Science, University of Bergen (2010), http://www.uib.no/ii/forskning/reports-in-informatics/reports-in-informatics-2010-2019
15. Poutré, J.A.L.: Lower bounds for the union-find and the split-find problem on pointer machines. J. Comput. Syst. Sci. 52, 87–99 (1996)
16. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM 22, 215–225 (1975)
17. Tarjan, R.E.: A class of algorithms which require nonlinear time to maintain disjoint sets. J. Comput. Syst. Sci. 18, 110–127 (1979)
18. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM 31, 245–281 (1984)
19. Wassenberg, J., Bulatov, D., Middelmann, W., Sanders, P.: Determination of maximally stable extremal regions in large images. In: Proc. of Signal Processing, Pattern Recognition, and Applications (SPPRA). Acta Press (2008)
20. Wu, K., Otoo, E., Suzuki, K.: Optimizing two-pass connected-component labeling algorithms. Pattern Anal. Appl. 12, 117–135 (2009)

Policy-Based Benchmarking of Weak Heaps and Their Relatives[⋆,⋆⋆]

Asger Bruun[1], Stefan Edelkamp[2], Jyrki Katajainen[1], and Jens Rasmussen[1]

[1] Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen East, Denmark
[2] TZI, Universität Bremen, Am Fallturm 1, 28357 Bremen, Germany

Abstract. In this paper we describe an experimental study where we evaluated the practical efficiency of three worst-case efficient priority queues: 1) a weak heap that is a binary tree fulfilling half-heap ordering, 2) a weak queue that is a forest of perfect weak heaps, and 3) a run-relaxed weak queue that extends a weak queue by allowing some nodes to violate half-heap ordering. All these structures support *delete* and *delete-min* in logarithmic worst-case time. A weak heap supports *insert* and *decrease* in logarithmic worst-case time, whereas a weak queue reduces the worst-case running time of *insert* to $O(1)$, and a run-relaxed weak queue that of both *insert* and *decrease* to $O(1)$. As competitors to these structures, we considered a binary heap, a Fibonacci heap, and a pairing heap. Generic programming techniques were heavily used in the code development. For benchmarking purposes we developed several component frameworks that could be instantiated with different policies.

1 Introduction

In this paper, we study addressable priority queues which store dynamic collections of elements and support the operations *find-min*, *insert*, *decrease* (or *decrease-key*), *delete*, *delete-min*, and *meld*. For addressable priority queues, *delete* and *decrease* take a handle to an element as an argument, and *find-min* and *insert* return a handle. These handles must be kept valid even though elements are moved around inside the data structures.

The most prominent priority queues described in textbooks (see, e.g. [7,25]) include binary heaps [31], binomial queues [29], Fibonacci heaps [16], and pairing heaps [15]. Of these, a Fibonacci heap is important for many applications since it supports *decrease* in $O(1)$ amortized time. Also, it supports the other priority-queue operations in optimal amortized bounds: *find-min*, *insert*, and *meld* in $O(1)$ time; and *delete* and *delete-min* in $O(\lg n)$ time, n being the number of elements stored prior to the operation.

⋆ Research partially supported by DFG grant ED 74/8-1.
⋆⋆ Partially supported by the Danish Natural Science Research Council under contract 09-060411 (project "Generic programming—algorithms and tools").

Table 1. Worst-case number of element comparisons performed by the most important operations on a weak heap and its variants (when *find-min* takes $O(1)$ worst-case time). Here n denotes the size of the data structure just prior to an operation.

Framework	Structure	*delete*	*insert*	*decrease*
single heap	weak heap	$\lceil \lg n \rceil$	$\lfloor \lg n \rfloor + 1$	$\lceil \lg n \rceil$
multiple heap	weak queue	$2 \lg n + O(1)$	2	$\lfloor \lg n \rfloor$
relaxed heap	run-relaxed weak queue	$3 \lg n + O(1)$	2	4

After the publication of Fibonacci heaps, two questions were addressed: 1) Can the same time bounds be achieved in the worst case? 2) Can the time bounds be achieved by a simpler data structure? The first question was settled by Brodal [3] in a practically unsatisfactorily manner since in his solution, when considering the number of element comparisons performed, the constant factor involved in the complexity of *delete*-*min* is much higher than $\lg n$ for all reasonable values of n [18]. Relaxed heaps proposed by Driscoll et al. [9] are more practical, but they support *meld* in logarithmic worst-case time, which is suboptimal. The second question has been studied by several authors, but there does not seem to be an agreement, whether the question has been settled or not. For more information about this issue, consult any of the recent articles [6,13,17,26] and the references mentioned therein.

The research reported in this paper is related to both of the foregoing questions. Our primary objective was to evaluate the usefulness of various implementation strategies when programming weak heaps [10] and their close relatives: weak queues and relaxed weak queues [11,14]. Our secondary objective was to get a complete picture of the field and compare the performance of these structures to that of some well-known competitors: binary heaps [31], Fibonacci heaps [16], and pairing heaps [15,28]. Of the studied data structures, a weak heap has the same asymptotic performance as a binary heap [31], a weak queue the same as a binomial queue [29], and a rank/run-relaxed weak queue the same as a rank/run-relaxed heap [9]. To get an insight of the performance characteristics of the studied data structures, in Table 1 we list the number of element comparisons performed by the most important operations.

To make the experimental comparison as fair as possible, we relied on policy-based design (see, for example, the book by Alexandrescu [2]). For similar priority queues, a separate component framework was developed. Three parameterized frameworks were written: 1) a single-heap framework that can realize a binary heap, relying on top-down or bottom-up heapifying, and a weak heap (Section 3); 2) a multiple-heap framework that can realize a weak queue and a binomial queue (Section 4); and 3) a relaxed-heap framework that can realize a run-relaxed and rank-relaxed weak queue (Section 5).

In a popular-scientific form, our results could be summarized as follows:

1) Read the masters! The original implementation of a binomial queue [29], in essence a weak queue, turned out to be one of the best performers mainly because of the focus put on implementation details in its description.

2) Priority queues that guarantee good performance in the worst-case setting have difficulties in competing against solutions that guarantee good performance in the amortized setting. Hence, worst-case efficient priority queues should only be used in applications where worst-case efficiency is essential.
3) Memory management is expensive. In our early code many unnecessary memory allocations were performed. Micro benchmarking revealed that memory management caused a significant performance slowdown.
4) In current computers, caching effects are significant. Memory-saving and bit-packing techniques turned out to be effective.
5) For most practical values of n, the difference between $\lg n$ and $O(1)$ is small! Often in the literature, in particular in theoretical papers, the significance of $O(1)$-time *insert* and *decrease* has been exaggerated. For heaps, for which *decrease* requires logarithmic time, the loop sifting up an element is extremely tight. Unless we make element comparisons noticeable expensive, it is difficult to come up with a faster solution.
6) For random data, the typical running time of *insert*, *decrease*, and *delete* (but not *delete-min*) is $O(1)$ for binary heaps, weak heaps, and weak queues. Hence, more advanced data structures can only beat these data structures for pathological input instances.
7) Generic component frameworks help algorithm engineers to carry out unbiased experiments. Changing policies helped us to tune the programs significantly while keeping the code base small.

2 Parameterized Design

The frameworks written for this study have been made part of the CPH STL [8]. In this section we give a brief overview of the overall design of our programs. As to the actual code, we refer to the CPH STL design documents [5,20,27].

When doing the implementation work, we followed the conventions set for the CPH STL project. For example, the application programming interface (API) for a meldable priority queue is specified in [19] (and corrected in [20]). All *containers*, as they are called in STL parlance, are interfaces that are decoupled from their actual implementations. These interfaces are designed to be user friendly, but to implement them only a smaller *realizator* is needed. There is a clear division of labour between the container and its realizator: 1) A client gives an element to the container, which allocates a node and puts the element into that node, and gives the node further to the realizator. When a realizator extracts a node, it gives the node back to the container which takes care of the deallocation of the node. 2) The container also provides (unidirectional) *iterators* to traverse through the elements. Iterators can also be used as handles to elements.

As an example, the single-heap framework is parameterized to accept seven type arguments: the type of the elements (or values) manipulated; the type of the comparator used in element comparisons; the type of the allocator providing an interface to allocate, construct, destroy, and deallocate objects; the type of the nodes (or encapsulators) used for storing the elements; the type of the heapifier used when re-establishing heap order after an element update; the type of the resizable array used for storing the heap; and the type of the surrogate proxy used (by iterators) for referring to the realizator (this is needed for supporting *swap* in $O(1)$ worst-case time).

Our goal was to make the frameworks generic such that they only use the methods provided by the policies given as type parameters and make as few assumptions on their functionality as possible. The key point is that the implementation of priority-queue operations *find-min*, *insert*, *decrease*, *delete*, *delete-min*, and *meld* is exactly the same for all realizators that a framework can create.

Our parameterized design has several advantages: a high level of code reuse, and ease of maintenance and benchmarking. By changing the parameters, one can easily see what the effect of a particular change is. The parameterized design also has its disadvantages: component frameworks can be difficult to understand, the design and development can be time consuming compared to quick-and-dirty programming, and sometimes generic programming can be difficult because of inadequate tool support. Moreover, a framework can become a hindrance for code optimizations, even though we did not experience any performance slowdown because of this type of design. However, we did experience that sometimes it was a challenge to make a change to a framework; this required that the programmer knew many data structures well and understood consequences of the change.

3 Single-Heap Framework

Recall that in a *heap-ordered tree* the element stored at a node is no smaller than the element stored at the parent of that node. The main difference between a binary heap [31] and a weak heap [10] is that the latter is only partially ordered. A *weak heap* has the following properties: 1) The element stored at a node is smaller than or equal to any element stored in the right subtree of that node (half-heap ordering). 2) The root of the entire structure has no left child. 3) The right subtree of the root is a complete binary tree (in the meaning defined in [22, Section 2.3.4.5]). In a *perfect weak heap*, the right subtree of the root is a perfect binary tree (i.e. a complete binary tree where even the last level is full).

A weak heap of size n has a clever array embedding that utilizes n auxiliary bits r_i, $i \in \{0, \ldots, n-1\}$. For location i, the left child is found at location $2i + r_i$ and the right child at location $2i + 1 - r_i$. For this purpose, r_i is interpreted as an integer in the range $\{0, 1\}$, initialized to 0. By flipping the bit, the status of being a left or a right child can be exchanged, which is an essential property to join two weak heaps in $O(1)$ worst-case time. It is possible to construct a weak heap of size n using $n - 1$ element comparisons, while for weak-heapsort the number of element comparisons performed is at most $n \log n + 0.09n$ [12], a value remarkably close to the lower bound of $n \log n - 1.44n$ element comparisons required by any sorting algorithm [23, Section 5.3.1].

Similar to binary heaps, array-embedded weak heaps can be extended to work as priority queues. For *delete-min*, after exchanging the element stored at the root with that stored at the last location, half-heap ordering is restored bottom-up, joining the weak heaps that lie on the left spine of the subtree rooted at the right child of the root. For *insert*, we sift up the element until half-heap ordering is re-established. Similarly, for *decrease*, we start at the node that has changed its value, and propagate the change upwards. For *delete* we move the element at the last location into the place of the deleted element, and sift that element down as in *delete-min* and up as in *decrease*.

Framework engineering. Instead of using a linked representation, we decided to use a resizable array. To keep the iterators valid at all times, we store the elements indirectly and maintain pointers between the array and the elements as proposed in [7, Chapter 6]. This does not destroy the worst-case complexity, since for a resizable array the worst-case running time of the grow and shrinkage operations can be kept constant (see, for example, [21]). By accepting different heapifier policies, which provide methods for sifting down and up, we can easily switch between weak heaps and different implementations of binary heaps. For example, one may use an alternative bottom-up sift-down strategy (see [23, Section 5.2.3, Exercise 18] or [30]).

4 Multiple-Heap Framework

The key idea behind an improved worst case of *insert* is to maintain a sequence of perfect weak heaps instead of keeping all elements in a single heap. As this makes relocation of subheaps frequent, we rely on a pointer-based representation. Recall that a *binomial queue* is a collection of heap-ordered binomial trees [29], and that a *binomial tree* is a multiary tree that stores 2^h elements for some integer $h \geq 0$. A *weak queue* is just like a binomial queue, but each multiary tree is transformed into a binary tree by applying the standard child-sibling transformation (see, e.g. [22, Section 2.3.2]). A binary-tree variant of a binomial tree was already utilized by Vuillemin [29] in his tuned implementation of binomial queues, even though he did not give any name for the data structure.

For brevity, we call the perfect weak heaps maintained just heaps. Furthermore, we call the data structure used to keep track of the heaps a *heap store*. The heaps are maintained in size order, starting from the smallest. The basic operations to be supported include *inject* which inserts a new heap into the heap store, and *eject* which removes the smallest heap from the heap store. For *inject* it is essential that the size of the new heap is no greater than that of the smallest heap currently in a weak queue.

After *delete-min*, when determining the root that contains the new minimum, we have to iterate over the heaps. Therefore, the number of heaps has to be kept low; the worst-case minimum for the number of heaps is $\lfloor \lg n \rfloor + 1$. That is, when new heaps arrive, occasional joins are necessary. A simple way to maintain a heap store is to utilize the connection to binary numbers: If a weak queue stores n elements and the binary representation of n is $\langle b_0, b_1, \ldots, b_{\lfloor \lg n \rfloor} \rangle$, the heap store contains a heap of size 2^i if and only if $b_i = 1$. The main problem with this invariant is that sometimes *inject* has to perform a logarithmic number of joins, each taking $O(1)$ worst-case time.

There are several alternative strategies to implement the heap store such that both *inject* and *eject* take $O(1)$ time in the worst case still keeping the number of heaps logarithmic. One of the simplest approaches, mentioned already in [4], is to rely on redundant numbers. If d_i denotes the number of heaps of height i and $d_i \in \{0, 1, 2\}$, the heap store can keep the *cardinality sequence* $\langle d_0, d_1, \ldots, d_{\lfloor \lg n \rfloor} \rangle$ *regular*, i.e. in the form $\bigl(0 \mid 1 \mid 01^{*}2\bigr)^{*}$ using the normal syntax for regular expressions (see, for example, [1]). If after each *inject* the first two heaps of the same size are joined, the regularity of the cardinality sequence will be preserved and the number of heaps will never be larger than $\lfloor \lg n \rfloor + 1$ [14]. For *eject*, the smallest heap is extracted and the cardinality sequence is updated accordingly.

Framework engineering. There were several alternative ways of implementing the nodes. In our baseline version, every node stores an element and pointers to its left child, right child, and parent. To make swapping of nodes cheaper, we also tried variants where elements were stored indirectly, but these versions turned out to be slower than the baseline version. In an extreme case, only two pointers per node would be necessary to cover the parent-child relationships, as observed by Brown [4], but this space optimization did not pay off; the space optimized versions were considerably slower than the baseline version.

The framework supports two types of heap stores: one that maintains a proxy for each heap and keeps these proxies in a linked list, and another that maintains the roots in a linked list by reusing the pointers at the nodes. The latter idea goes back to Vuillemin [29]. Also, following Vuillemin's original proposal the heights of the heaps are maintained in a bit vector, which can be stored in a single word, since the heights are between 0 and $\lfloor \lg n \rfloor + 1$. Both types of heap stores could be equipped with the binary number system or redundant binary number system. For the redundant system, different strategies for maintaining the information about the pairs of heaps having the same size were tried. The best alternative turned out to a preallocated stack storing pointers to the first member of each pair. In general, all solutions relying on dynamic storage management were noticeable slower than the versions that avoided it. Overall, the overhead incurred by the redundant system turned out to be negligible.

We observed that there was a huge difference in the typical running times for the two known ways of dealing with *delete*. Brown [4] called the two strategies top-down and bottom-up. The *top-down strategy* sifts up the node being deleted to the root and removes the root, whereas the *bottom-up strategy* finds a replacement node for the node being deleted, makes the replacement, and sifts down or up the new node. In a typical case, assuming that we are not deleting the minimum, the amount of work done by the bottom-up approach is $O(1)$, whereas the amount of work involved in the top-down approach is logarithmic.

5 Relaxed-Heap Framework

In relaxed weak queues the new ingredient is that the half-ordering violations incurred by *decrease* operations are resolved by marking. When there are too many marked nodes, the number of marked nodes is reduced. Driscoll et al. [9] introduced this idea in their relaxed heaps, and Elmasry et al. [14] observed that the idea carries over into the binary-tree setting. The other operations *find-min*, *insert*, *delete*, *delete-min*, and *meld* can be implemented as for weak queues.

We call the data structure used to keep track of all markings a *mark store.* The fundamental operations to be supported include *mark* which marks a node to denote that a half-ordering violation may occur at that node, *unmark* which removes a marking, and *reduce* which removes one or more unspecified markings. A *run* is a maximal sequence of two or more marked nodes that are consecutive on the left spine of a subtree. More formally, a node is a *member* of a run if it is marked, a left child, and its parent is marked. A node is the *leader* of a run if it is marked, its left child is marked, and it is either a right child or a left child of a non-marked parent. A marked node that is neither a member nor a leader of a run is called a *singleton.* If at some height there are more than one singleton, these singletons form a *team.* To summarize, the set of all nodes is divided into four disjoint categories: non-marked nodes, members, leaders, and singletons.

A pair $(type, height)$ with *type* being either non-marked, member, leader, or singleton; and *height* being a value in the range $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$ denotes the *state* of a node. The states are stored explicitly at the nodes. Transformations used when reducing the number of marked nodes induce a constant number of state transitions. A simple example of such a transformation is a join, where the height of the new root is increased by one.

Other transformations (see Fig. 1) are cleaning, parent, sibling, and pair transformations. A *cleaning transformation* rotates a marked left child to a marked right one, provided that its sibling and parent are non-marked. A *parent transformation* reduces the number of marked nodes or pushes the marking one level up. A *sibling transformation* reduces the number of markings by eliminating two markings, while generating a new marking one level up. A *pair transformation* has a similar effect, but it operates on disconnected trees. These four transformations are combined to perform a *singleton* or *run transformation.*

In a *run-relaxed weak queue* [14], which is similar to a run-relaxed heap [9], an invariant is maintained that, after each priority-queue operation, the number of markings is never larger than $\lfloor \lg n \rfloor$. When this bound is exceeded, a singleton or a run transformation is applied to restore the invariant. The running time of *decrease* can be guaranteed to be $O(1)$ in the worst case. In a *rank-relaxed weak queue* [11], which is similar to a rank-relaxed heap [9], the transformations are applied in an eager way by performing as many *reduce* operations as possible after each priority-queue operation that introduces new markings. The worst-case cost of *decrease* can be logarithmic, but the amortized cost is a constant. From a practical perspective, amortization leads to a slightly more efficient implementation, as verified in [11].

Framework engineering. The first implementation of a mark store that supports *mark*, *unmark*, and *reduce* in $O(1)$ worst-case time was described in [9]. In this solution it was necessary to maintain a doubly-linked list of leaders, a doubly-linked list of teams, a doubly-linked list of singletons at each height, and a resizable array of pointers to the beginning of each singleton list. In our engineered implementation we use no lists, but keep pointers to the marked nodes in a preallocated array and maintain another preallocated array of bit vectors, each occupying a single word, indicating which of the marked nodes are singletons of

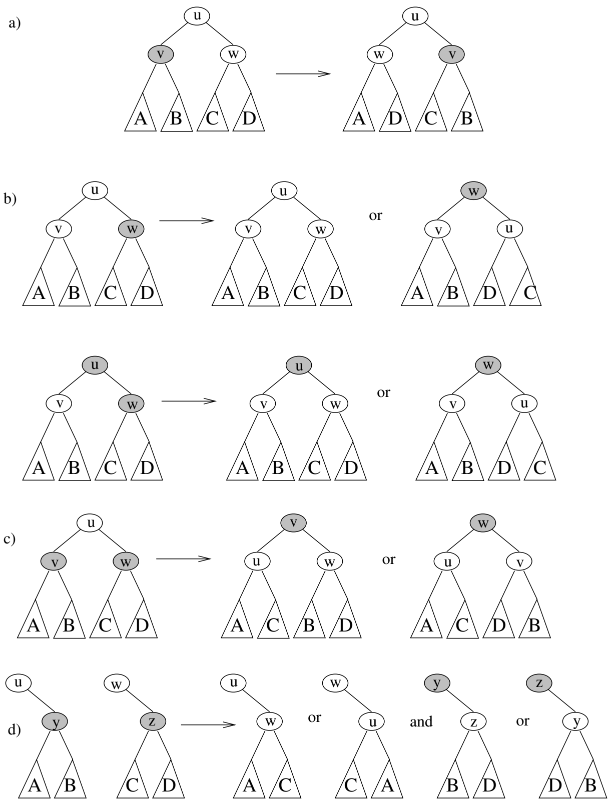


Fig. 1. Primitives used by *reduce*: a) cleaning transformation, b) parent transformation, c) sibling transformation, and d) pair transformation

particular height. Additionally, we need one bit vector to denote which of the marked nodes are leaders and another to indicate which of the singleton sets have more than one node. To allow fast *unmark*, every marked node stores an index referring to the pointer array maintained in the mark store. The bit-vector class features the fast selection of the most significant 1-bit.

6 Experiments

In our benchmarks we compared priority queues from the weak-heap family (weak heap, weak queue, and run-relaxed weak queue) to their closest competitors (binary heap [31], Fibonacci heap [16], and pairing heap [28]). The last two were taken from LEDA (version 6.2) [24]. The other data structures were the engineered versions that we implemented for the CPH STL [8].

We carried out several experiments for different types of input data (worst-case and randomly-generated instances) in different environments (compilers and computers varied) considering different kinds of performance indicators (number of element comparisons, clock cycles, and CPU time). The results obtained did not vary much across the tested environments. Also, the results obtained by the clock-cycle and CPU-time measurements were similar.

When engineering our implementations we carried out several micro-benchmarks. Due to space limitations, we do not report any detailed results on them, but refer to [5]. For randomly-generated data, the average running time of *insert*, *decrease*, and *delete* (but not *delete-min*) is $O(1)$ for binary and weak heaps. Since these structures are simple, other more advanced structures have difficulties in beating them. For the structures having good amortized time bounds, *insert* and *decrease* are fast because most work is delayed till *delete* and *delete-min*. Due to space limitations, we leave out the results for randomly-generated input data, but present them in the full version of this paper.

We find the results of synthetic benchmarks involving the basic operations interesting and report these results here. These benchmarks were conducted on a laptop computer (model Intel® Core$^{\text{TM}}$2 CPU T5600 @ 1.83GHz) under Ubuntu 9.10 operating system (Linux kernel 2.6.31-19-generic) using `g++` C++ compiler (`gcc` version 4.4.1 with options `-DNDEBUG -Wall -std=c++0x -pedantic -x c++ -fno-strict-aliasing -O3`). The size of L2 cache of this computer was about 2 MB and that of the main memory 1 GB. The input data was integers of type `long long` and, for the LEDA data structures, pairs of type (`long long`, `struct empty`) since in LEDA the elements are expected to be (priority, information) pairs. Besides comparing integer elements with their built-in comparison function, the comparator increased a global counter to gather comparison counts.

In order to avoid the problem caused by a bounded clock granularity, which in the test computer was 10 milliseconds, for given n we repeated each experiment $\lceil 10^6/n \rceil$ times, each time with a new priority queue. The standard dual-loop strategy was used to eliminate the time taken by all initializations. All running times are reported in microseconds, and they are average times per operation.

In Fig. 2, 3, 4, and 5 we give the average running times used and the number of element comparisons performed per *insert*, *decrease*, *delete*, and *delete-min*, respectively. In the *insert* experiment, the integers between 0 and $n - 1$ were inserted in reversed sorted order which forced binary and weak heaps to use

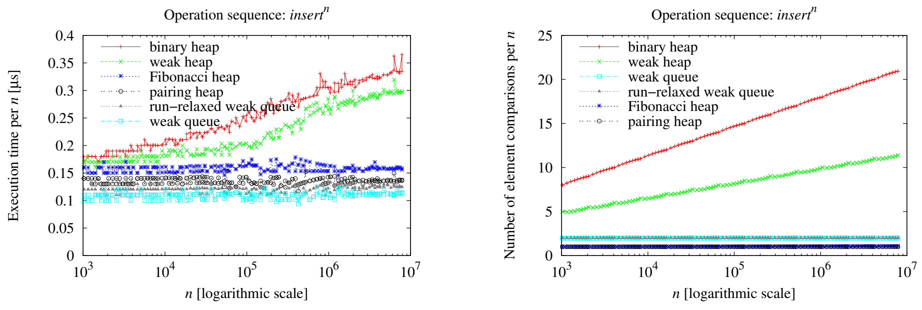


Fig. 2. *insert*: CPU times and comparison counts for different priority queues

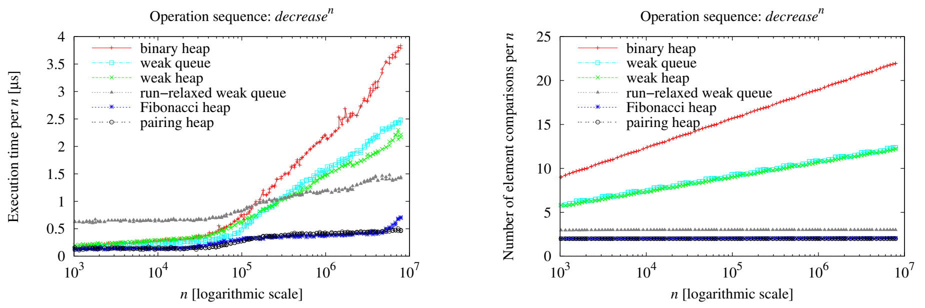


Fig. 3. *decrease*: CPU times and comparison counts for different priority queues

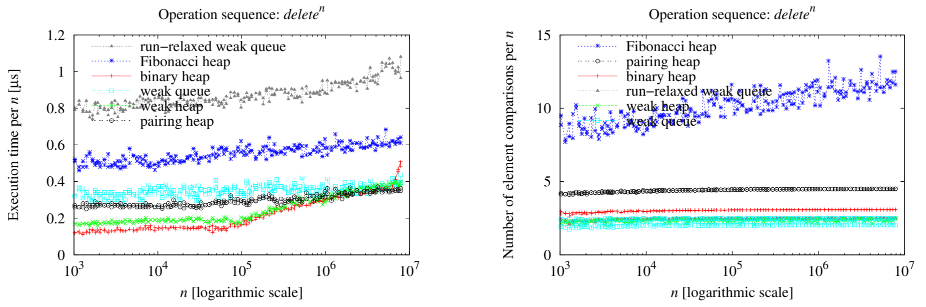


Fig. 4. *delete*: CPU times and comparison counts for different priority queues

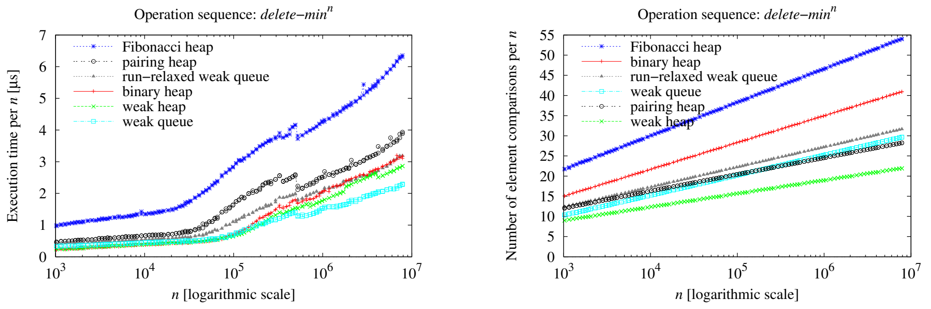


Fig. 5. *delete-min*: CPU times and comparison counts for different priority queues

logarithmic time for each operation. In spite of this, the running times were competitive. In the *decrease* experiment, the integers were inserted in random order, and thereafter the values were updated such that each new value became the new minimum element; the time used by *decrease* operations was measured. This arrangement guaranteed that *decrease* took logarithmic time on an average for a binary heap, weak heap, and weak queue. Even in such extreme situation, these three data structures were competitive against theoretically more robust solutions. In the *delete* experiment, the integers were inserted in random order and extracted in their insertion order; the time used by *delete* operations was measured. All our implementations relied on the bottom-up deletion strategy; this experiment confirmed that this was a good choice. In the *delete*-*min* experiment, the integers were inserted in random order, and the minimum was extracted until the data structure became empty; the time used by *delete*-*min* operations was measured. Even if a weak heap is optimal with respect to the number of element comparisons, a weak queue was faster. For all problem sizes, a Fibonacci heap used about 3 times more time than a weak queue.

The average running times reported can be used to estimate the overhead caused by the worst-case behaviour. For data structures that do not provide good performance in the worst-case setting, the running times of individual operations can fluctuate considerably. For example, for binary and weak heaps the worst-case running time of a single *insert* was linear since we relied on `std::vector`, not on a worst-case efficient resizable array. For Fibonacci and pairing heaps the worst-case running time of a single *delete*-*min* is linear. Even for Vuillemin's implementation of a weak queue the running times of *insert* and *decrease* can vary between $\Theta(1)$ and $\Theta(\lg n)$. In applications, where such fluctuations are intolerable, only a run-relaxed weak queue can guarantee stable behaviour, but as shown, this stability has its price.

In particular for randomly-generated input data, the performance of simple data structures like binary and weak heaps is good. These simple data structures fall in short only when melding has to be efficient. Namely, for our implementations, melding two weak heaps (or binary heaps) of size m and n, $m \leq n$, takes $\Theta(m \lg n)$ time in the worst case. Even though more efficient implementations are possible, one should consider using some of other data structures instead.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools, 2nd edn. Pearson Education, Inc., London (2007)
2. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, Reading (2001)
3. Brodal, G.S.: Worst-case efficient priority queues. In: Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 52–58. ACM/SIAM (1996)
4. Brown, M.R.: Implementation and analysis of binomial queue algorithms. SIAM Journal on Computing 7(3), 298–319 (1978)
5. Bruun, A.: Effektivitetsmåling på krydsninger af svage og binomiale prioritetskøer, CPH STL Report 2010-2, Department of Computer Science, University of Copenhagen (2010)
6. Chan, T.M.: Quake heaps: A simple alternative to Fibonacci heaps (2009) (unpublished manuscript)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
8. Department of Computer Science, University of Copenhagen, The CPH STL (2000–2010), Website accessible at `http://cphstl.dk/`

9. Driscoll, J.R., Gabow, H.N., Shrairman, R., Tarjan, R.E.: Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. Communications of the ACM 31(11), 1343–1354 (1988)
10. Dutton, R.D.: Weak-heap sort. BIT 33(3), 372–381 (1993)
11. Edelkamp, S.: Rank-relaxed weak queues: Faster than pairing and Fibonacci heaps? Technical Report 54, TZI, Universität Bremen (2009)
12. Edelkamp, S., Wegener, I.: On the performance of Weak-Heapsort. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 254–266. Springer, Heidelberg (2000)
13. Elmasry, A.: Violation heaps: A better substitute for Fibonacci heaps, E-print 0812.2851v1, arXiv.org (2008)
14. Elmasry, A., Jensen, C., Katajainen, J.: Relaxed weak queues: An alternative to run-relaxed heaps, CPH STL Report 2005-2, Department of Computer Science, University of Copenhagen (2005)
15. Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E.: The pairing heap: A new form of self-adjusting heap. Algorithmica 1(1), 111–129 (1986)
16. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM 34(3), 596–615 (1987)
17. Haeupler, B., Sen, S., Tarjan, R.E.: Rank-pairing heaps. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 659–670. Springer, Heidelberg (2009)
18. Jensen, C.: Private communication (2009)
19. Katajainen, J.: Project proposal: A meldable, iterator-valid priority queue, CPH STL Report 2005-1, Department of Computer Science, University of Copenhagen (2005)
20. Katajainen, J.: Priority-queue frameworks: Programs, CPH STL Report 2009-7, Department of Computer Science, University of Copenhagen (2009)
21. Katajainen, J., Mortensen, B.B.: Experiences with the design and implementation of space-efficient deques. In: Brodal, G.S., Frigioni, D., Marchetti-Spaccamela, A. (eds.) WAE 2001. LNCS, vol. 2141, pp. 39–50. Springer, Heidelberg (2001)
22. Knuth, D.E.: Fundamental Algorithms, 3rd edn. The Art of Computer Programming, vol. 1. Addison Wesley Longman, Amsterdam (1997)
23. Knuth, D.E.: Sorting and Searching, 2nd edn. The Art of Computer Programming, vol. 3. Addison-Wesley Longman, Amsterdam (1998)
24. Mehlhorn, K., Näher, S.: The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (1999)
25. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox. Springer, Heidelberg (2008)
26. Paredes, R.: Graphs for metric space searching, Ph.D. Thesis, Department of Computer Science, University of Chile (2008)
27. Rasmussen, J.: Implementing run-relaxed weak queues, CPH STL Report 2008-1, Department of Computer Science, University of Copenhagen (2008)
28. Stasko, J.T., Vitter, J.S.: Pairing heaps: Experiments and analysis. Communications of the ACM 30(3), 234–249 (1987)
29. Vuillemin, J.: A data structure for manipulating priority queues. Communications of the ACM 21(4), 309–315 (1978)
30. Wegener, I.: Bottom-up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). Theoretical Computer Science 118, 81–98 (1993)
31. Williams, J.W.J.: Algorithm 232: Heapsort. Communications of the ACM 7(6), 347–348 (1964)

Modularity-Driven Clustering of Dynamic Graphs[⋆]

Robert Görke[1], Pascal Maillard[2], Christian Staudt[1], and Dorothea Wagner[1]

[1] Institute of Theoretical Informatics
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{robert.goerke,dorothea.wagner}@kit.edu, christian.staudt@ira.uka.de
[2] Laboratoire de Probabilités et Modèles Aléatoires
Université Pierre et Marie Curie (Paris VI), Paris, France
pascal.maillard@upmc.fr

Abstract. Maximizing the quality index *modularity* has become one of the primary methods for identifying the clustering structure within a graph. As contemporary networks are not static but evolve over time, traditional static approaches can be inappropriate for specific tasks. In this work we pioneer the NP-hard problem of online *dynamic modularity* maximization. We develop scalable dynamizations of the currently fastest and the most widespread static heuristics and engineer a heuristic dynamization of an optimal static algorithm. Our algorithms efficiently maintain a *modularity*-based clustering of a graph for which dynamic changes arrive as a stream. For our quickest heuristic we prove a tight bound on its number of operations. In an experimental evaluation on both a real-world dynamic network and on dynamic clustered random graphs, we show that the dynamic maintenance of a clustering of a changing graph yields higher *modularity* than recomputation, guarantees much smoother clustering dynamics and requires much lower runtimes. We conclude with giving recommendations for the choice of an algorithm.

1 Introduction

Graph clustering is concerned with identifying and analyzing the group structure of networks. Generally, a partition (i.e., a clustering) of the set of nodes is sought, and the size of the partition is a priori unknown. A plethora of formalizations for what a *good* clustering is exist, good overviews are, e.g., [21,3]. In this work we set our focus on the quality function *modularity*, coined by Girvan and Newman [4], which has proven itself feasible and reliable in practice, especially as a target function for maximization (see [2] for further references), which follows the paradigm of parameter-free community discovery [5]. The foothold of this work is that most networks in practice are not static. Iteratively clustering snapshots of a dynamic graph from scratch with a static method has several disadvantages: First, runtime cannot be neglected for large instances

[⋆] This work was partially supported by the DFG under grant WA 654/15-1. The full version of this extended abstract is available as a technical report [1].

or environments where computing power is limited [6], even though very fast clustering methods have been proposed recently [7,8]. Second, heuristics for the NP-hard [2] optimization of *modularity* suffer from local optima—this might be avoided by dynamically maintaining a good solution. Third, static heuristics are known not to react in a continuous way to small changes in a graph. The lefthand figure illustrates the general situation for updating clusterings. A graph G is updated by some change Δ, yielding G'. We investigate procedures $\mathcal{A}$ that update the clustering $\mathcal{C}(G)$ to $\mathcal{C}'(G')$ without reclustering from scratch, but work towards the same aim as a static technique $\mathcal{T}$ does.

$$\begin{tikzcd} G \arrow[r, "\Delta"] \arrow[d, "\mathcal{T}"'] & G' \arrow[d, "\mathcal{T}"] \\ \mathcal{C}(G) \arrow[r, dashed, "\mathcal{A}"] & \mathcal{C}'(G') \end{tikzcd}$$

Fig. 1. Problem setting

Related Work. Dynamic graph clustering has so far been a rather untrodden field. Recent efforts [9] yielded a method that can provably dynamically maintain a clustering that conforms to a specific bottleneck-quality requirement. Apart from that, there have been attempts to track communities over time and interpret their evolution, using static snapshots of the network, e.g. [10,11], besides an array of case studies. In [12] a parameter-based dynamic graph clustering method is proposed which allows user exploration. Parameters are avoided in [13] where the minimum description length of a graph sequence is used to determine changes in clusterings and the number of clusters. In [14] an explicitly bicriterial approach for low-difference updates and a *partial* ILP are proposed, the latter of which we also discuss. To the best of our knowledge no fast procedures for updating *modularity*-based clustering in general dynamic graphs have been proposed yet. Beyond graph theory, in data mining the issue of clustering an evolving data set has been addressed in, e.g., [15], where the authors share our goal of finding a smooth dynamic clustering. The literature on static *modularity*-maximization is quite broad and we recommend [2,3,16] for further reading. Spectral methods, e.g., [17], and techniques based on random walks [18,19], do not lend themselves well to dynamization due to their non-continuous nature. Variants of greedy agglomeration [20,7], however, work well, as we shall see.

Our Contribution. In this work we present, analyze and evaluate a number of concepts for efficiently updating *modularity*-driven clusterings. We prove the NP-hardness of dynamic *modularity* optimization and develop heuristic dynamizations of the most widespread [20] and the fastest [7] static algorithms, alongside apt strategies to determine the search space. For our fastest procedure, we can prove a tight bound of $\Theta(\log n)$ on the expected number of operations required. We then evaluate these and a heuristic dynamization of an ILP. We compare the algorithms with their static counterparts and evaluate them experimentally on random preclustered dynamic graphs and on large real-world instances. We reveal that the dynamic maintenance of a clustering yields higher quality than recomputation, smoother clustering dynamics and lower runtimes.

Notation. Throughout this paper, we will use the notation of [21]. We assume that $G = (V, E, \omega)$ is an undirected, weighted, and simple graph with the edge weight function $\omega\colon E \to \mathbb{R}_{\geq 0}$. We set $|V| =: n, |E| =: m$ and $\mathcal{C} = \{C_1, \dots, C_k\}$ to be a partition of V. We call $\mathcal{C}$ a *clustering* of G and sets C_i *clusters*. $\mathcal{C}(v)$ is $C \ni v$. A clustering is *trivial* if either $k = 1$ ($\mathcal{C}^{\mathbf{1}}$), or all clusters contain only one element, i.e., are *singletons* ($\mathcal{C}^V$). We identify a cluster C_i with its node-induced subgraph of G. Then $E(\mathcal{C}) := \bigcup_{i=1}^{k} E(C_i)$ are *intra-cluster* edges and $E \setminus E(\mathcal{C})$ *inter-cluster* edges, with cardinalities $m(\mathcal{C})$ and $\overline{m}(\mathcal{C})$, respectively. Further, we generalize degree $\deg(v)$ to clusters as $\deg(C) := \sum_{v \in C} \deg(v)$. When using edge weights, all the above definitions generalize naturally by using $\omega(e)$ instead of 1 when counting edge e. Weighted node degrees are called $\omega(v)$. A *dynamic graph* $\mathcal{G} = (G_0, \dots, G_{t_{\max}})$ is a sequence of graphs, with $G_t = (V_t, E_t, \omega_t)$ being the state of the dynamic graph at time step t. The *change* $\Delta(G_t, G_{t+1})$ between timesteps comprises a sequence of b *atomic* events on G_t, which we detail later. We have the sequence of changes arrive as a stream.

The Quality Index *Modularity*. In this work we set our focus on *modularity* [4], a measure for the goodness of a clustering. Just like any other quality index for clusterings (see, e.g., [21,3]), *modularity* does have certain drawbacks such as *non-locality* and *scaling behavior* [2] or *resolution limit* [22]. However, being aware of these peculiarities, *modularity* can very well be considered a useful measure that closely agrees with intuition on a wide range of real-world graphs, as observed by myriad studies. *Modularity* can be formulated as

$$\operatorname{mod}(\mathcal{C}) := \frac{m(\mathcal{C})}{m} - \frac{1}{4m^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} \deg(v) \right)^2 \quad \text{(weighted analogous)} \ . \qquad (1)$$

Roughly speaking, *modularity* measures the fraction of edges which are covered by a clustering and compares this value to its expected value, given a random rewiring of the edges which, on average, respects node degrees. This definition generalizes in a natural way as to take edge weights $\omega(e)$ into account, for a discussion thereof see [23] and [24]. ModOpt, the problem of optimizing *modularity* is NP-hard [2], but *modularity* can be computed in linear time and lends itself to a number of greedy maximization strategies. For the dynamic setting, the following corollary corroborates the use of heuristics (see [1] for a proof).

Corollary 1 (DynModOpt is NP-hard). *Given graph G, a modularity-optimal clustering $\mathcal{C}^{\text{opt}}(G)$ and an atomic event Δ to G, yielding G'. It is NP-hard to find a modularity-optimal clustering $\mathcal{C}^{\text{opt}}(G')$.*

Measuring the Smoothness of a Dynamic Clustering. By comparing consecutive clusterings, we quantify how smooth an algorithm manages the transition between two steps, an aspect which is crucial to both readability and applicability. An array of measures exist that quantify the (dis)similarity between two partitions of a set; for an overview and further references, see [25]. Our results strongly suggest that most of these widely accepted measures are qualitatively equivalent in all our (non-pathological) instances (see full version [1]). We thus restrict our view to the *(graph-structural) Rand* index [25], being a well known representative; it maps two clusterings into the interval $[0, 1]$, i.e., from equality to maximum dissimilarity: $\mathcal{R}_g(\mathcal{C}, \mathcal{C}') := 1 - (|E_{11}| + |E_{00}|)/m$, with

$E_{11} = \{\{v, w\} \in E : \mathcal{C}(v) = \mathcal{C}(w) \wedge \mathcal{C}'(v) = \mathcal{C}'(w)\}\}$, and E_{00} the analog for inequality. We use the intersection of two graphs when comparing their clusterings. *Low* distances correspond to *smooth* dynamics.

2 The Clustering Algorithms

Formally, a dynamic clustering algorithm is a procedure which, given the previous state of a dynamic graph G_{t-1}, a sequence of graph events $\Delta(G_{t-1}, G_t)$ and a clustering $\mathcal{C}(G_{t-1})$ of the previous state, returns a clustering $\mathcal{C}'(G_t)$ of the current state. While the algorithm may discard $\mathcal{C}(G_{t-1})$ and simply start from scratch, a good dynamic algorithm will harness the results of its previous work. A natural approach to dynamizing an agglomerative clustering algorithm is to break up those local parts of its previous clustering, which are most likely to require a reassessment after some changes to the graph. The half finished instance is then given to the agglomerative algorithm for completion. A crucial ingredient thus is a *prep strategy* S which decides on the search space which is to be reassessed. We will discuss such strategies later, until then we simply assume that S breaks up a reasonable part of $\mathcal{C}(G_{t-1})$, yielding $\tilde{\mathcal{C}}(G_{t-1})$ (or $\tilde{\mathcal{C}}(G_t)$ if including the changes in the graph itself). We call $\tilde{\mathcal{C}}$ the *preclustering* and nodes that are chosen for individual reassessment *free* (can be viewed as singletons).

Formalization of Graph Events. We describe our test instances in more detail later, but for a proper description of our algorithms, we now briefly formalize the graph events we distinguish. Most commonly *edge creations* and *removals* take place, and they require the incident nodes to be present before and after the event. Given edge *weights*, changes require an edge's presence. *Node creations* and *removals* in turn only handle degree zero nodes, i.e., for an intuitive node deletion we first have to remove all incident edges. To summarize such compound events we use *time step* events, which indicate to an algorithm that an updated clustering must now be supplied. Between time steps it is up to the algorithm how it maintains its intermediate clustering. Additionally, *batch* updates allow for only running an algorithm after a scalable number of b timesteps.

2.1 Algorithms for Dynamic Updates of Clusterings

Alg. 1. Global$(G, \mathcal{C})$

```
1 while $\exists C_i, C_j \in \mathcal{C} : \mathrm{dQ}(C_i, C_j) \geq 0$ do
2     $(C_1, C_2) \leftarrow \arg\max_{C_i, C_j \in \mathcal{C}} \mathrm{dQ}(C_i, C_j)$
3     merge$(C_1, C_2)$
```

The Global Greedy Algorithm. The most prominent algorithm for *modularity* maximization is a global greedy algorithm [20], which we call Global (Alg. 1). Starting with singletons, for each pair of clusters, it determines the increase in *modularity* dQ that can be achieved by merging the pair and performs the most beneficial merge. This is repeated until no more improvement is possible. As the **static** (pseudo-dynamic) algorithm sGlobal[1], we let this algorithm cluster from scratch at each timestep for comparison. By passing a *preclustering* $\tilde{\mathcal{C}}(G_t)$ to Global we can define the properly **dynamic** algorithm dGlobal. Starting from $\tilde{\mathcal{C}}(G_t)$ this algorithm lets Global perform greedy agglomerations of clusters.

The Local Greedy Algorithm. In a recent work [7] the simple mechanism of the aforementioned Global has been modified as to rely on local decisions (in terms of graph locality), yielding an extremely fast and efficient maximization. Instead of looking globally for the best merge of two clusters, Local, as sketched out in Alg. 2, repeatedly lets each node consider moving to one of its neighbors' clusters, if this improves *modularity*; this potentially merges clusters, especially when starting with singletons. As soon as no more nodes move, the current clustering is *contracted*, i.e., each cluster is contracted to a single node, and adjacencies and edge weights between them are summarized. Then, the process is repeated on the resulting graph which constitutes a higher level of abstraction; in the end, the highest level clustering is decisive about the returned clustering: The operation unfurl assigns each elementary node to a cluster represented by the highest level cluster it is contained in. We again sketch out an algorithm which serves as the core for both a static and a dynamic variant of this approach, as shown in Alg. 2. As the input, this algorithm takes a hierarchy of graphs and clusterings and a search space policy P. Policy P affects the graph *contractions*, in that P decides which nodes of the next level graph should be free to move. Note that the input hierarchy can also be flat, i.e., $h_{\max} = 0$, then line 11 creates all necessary higher levels. Again posing as a pseudo-dynamic algorithm, the **static variant** (as in [7]), sLocal, passes only $(G_t, \tilde{\mathcal{C}}^V)$ to Local, such that it starts with singletons and all nodes freed, instead of a proper *preclustering*. Policy P is set to tell the algorithm to start from scratch on all higher levels and to not work on previous results in line 11, i.e., in $\tilde{\mathcal{C}}^{h+1}$ again all nodes in the contraction are free singletons. The **dynamic variant** dLocal remembers its old results. It passes the changed graph, a current *preclustering* of it and all higher-level contracted structures from its previous run to Local: $(G_t,\, G_{\text{old}}^{1,\dots,h_{\max}},\, \tilde{\mathcal{C}},\, \mathcal{C}_{\text{old}}^{1,\dots,h_{\max}}, P)$. In level 0, the preclustering $\tilde{\mathcal{C}}$ defines the set of free nodes. In levels

Alg. 2. Local$(G^{0\dots h_{\max}}, \mathcal{C}^{0\dots h_{\max}}, P)$

```
1  $h \leftarrow 0$
2  repeat
3      $(G, \mathcal{C}) \leftarrow (G^h, \mathcal{C}^h)$
4      repeat
5          forall free $v \in V$ do
6              if $\max_{v \in N(u)} \mathrm{dQ}_{uv} \geq 0$ then
7                  $w \leftarrow \arg\max_{v \in N(u)} \mathrm{dQ}_{uv}$
8                  move$(u, \mathcal{C}(w))$
9      until no more changes
10     $\mathcal{C}^h \leftarrow \mathcal{C}$
11     $(G^{h+1}, \tilde{\mathcal{C}}^{h+1}) \leftarrow$ contract$_P(G^h, \mathcal{C}^h)$
12     $h \leftarrow h + 1$
13 until no more real contractions
14 $\mathcal{C}(G^0) \leftarrow$ unfurl$(\mathcal{C}^{h-1})$
```

[1] For historical reasons, sGlobal appears in plots as StaticNewman, dGlobal as Newman, sLocal as StaticBlondel and dLocal as Blondel, based on the algorithms' authors.

beyond 0, policy P is set to have the contract-procedure free only those nodes of the next level, that have been affected by lower level changes (or their neighbors as well, tunable by policy P). Roughly speaking, dLocal starts by letting all free (elementary) nodes reconsider their cluster. Then it lets all those (super-)nodes on higher levels reconsider their cluster, whose content has changed due to lower level revisions.

ILP. While optimality is out of reach, the problem *can* be cast as an ILP [2]. A distance relation X_{uv} indicates whether elements u and v are in the same cluster, and simple constraints keep these X-variables consistent. Since runtimes for the full ILP reach days for more than 200 nodes, a promising idea pioneered in [14] is to solve a *partial ILP* (pILP). Such a program takes a *preclustering*—of much smaller complexity—as the input, and solves this instance, i.e., finishes the clustering, optimally via an ILP; a singleton *preclustering* yields a full ILP. We introduce two variants, (i) the argument noMerge prohibits merging *pre-clusters*, and only allows free nodes to join clusters or form new ones, and (ii) merge allows existing clusters to merge. For both variants we need to add constraints and terms to the standard formulation using solely variables X_{uv}. Roughly speaking, for (i), variables Y_{uC} indicating the distance of node u to cluster C are introduced constraints ensure their consistency with the X-variables; for (ii), we additionally need variables $Z_{CC'}$ for the distance between clusters, constrained just as X_{uv}. See the full paper [1] for details on all these ILP formulations. The dynamic clustering algorithms which first solicit a *preclustering* and then call pILP are called dILP. Note that they react on any edge event; accumulating events until a timestep occurs can result in prohibitive runtimes.

Elemental Optimizer. The *elemental operations optimizer*, EOO, performs a limited number of operations, trying to increase the quality. Specifically, we allow moving or splitting off nodes and merging clusters, as listed in Table 1. Although rather limited in its options, EOO or very similar tools for local optimization are often used as post-processing tools (see [26] for a discussion). Our algorithm dEOO simply calls EOO at each time step.

Table 1. EOO operations, allowed/disallowed via parameters

Operation	Effect
merge(u,v)	$\mathcal{C}(u) \cup \mathcal{C}(v)$
shift(u,v)	$\mathcal{C}(u) - u, \mathcal{C}(v) + u$
split(u)	$(\{u\}, \mathcal{C}(u) \setminus u) \leftarrow \mathcal{C}(u)$

2.2 Strategies for Building the Preclustering

We now describe *prep strategies* which generate a *preclustering* $\tilde{\mathcal{C}}$, i.e., define the search space. We distinguish the *backtrack strategy*, which refines a clustering, and *subset strategies*, which free nodes. The rationale behind the *backtrack strategy* is that selectively backtracking the clustering produced by Global enables it to respect changes to the graph. On the other hand, *subset strategies* are based on the assumption that the effect of a change on the clustering structure is necessarily local. Both output a half-finished *preclustering*.

The *backtrack strategy* (BT) records the merge operations of Global and backtracks them if a graph modification suggests their reconsideration. We detail in the full paper [1] what we mean by "suggests", but for brevity we just state that the actions listed for BT provably require very little asymptotic effort and offer Global a good chance to find an improvement. Speaking intuitively, the reactions to a change in (non-)edge $\{u, v\}$ are as follows (weight changes are analogous): For intra-cluster additions we backtrack those merge operations that led to u and v being in the same cluster and allow Global to find a tighter cluster for them, i.e., we separate them. For inter-cluster additions we track back u and v individually, until we isolate them as singletons, such that Global can re-classify and potentially merge them. Inter-cluster deletions are not reacted on. On intra-cluster deletions we again isolate both u and v such that Global may have them find separate clusters. Note that this strategy is only applicable to Global; conferring it to Local is neither straightforward nor promising as Local is based on node *migrations* in addition to *agglomerations*. Anticipating this strategy's low runtime, we can give a bound on the expected number of backtrack steps for a single call of the crucial operation isolate (proven in the full paper [1]).

Theorem 1. *Assume that a backtrack step divides a cluster randomly. Then, for the number I of steps isolate(v) requires, it holds: $E\{I\} \in \Theta(\ln n)$.*

A *subset strategy* is applicable to all dynamic algorithms. It frees a subset $\tilde{V}$ of individual nodes that need reassessment and extracts them from their clusters. We distinguish three variants which are all based on the hypothesis that local reactions to graph changes are appropriate. Consider an edge event involving $\{u, v\}$. The *breakup strategy* (BU) marks the affected clusters $\tilde{V} = \mathcal{C}(u) \cup \mathcal{C}(v)$; the *neighborhood strategy* (N_d) with parameter d marks $\tilde{V} = N_d(u) \cup N_d(v)$, where $N_d(w)$ is the d-hop neighborhood of w; the *bounded neighborhood strategy* (BN_s) with parameter s marks the first s nodes found by a breadth-first search simultaneously starting from u and v.

3 Experimental Evaluation of Dynamic Algorithms[2]

Instances. We use both generated graphs and real-world instances. We briefly describe them here, but for more details please see [27] and [14].

Random Graphs {*ran*}. Our Erdős-Rényi-type generator builds upon [28] and adds to this dynamicity in all graph elements and in the clustering, i.e., nodes and edges are inserted and removed and ground-truth clusters merged and split, always complying with sound probabilities. The generator's own clustering serves as a reference to compare our algorithms to, see [27] for details. In later plots we use selected random instances, however, descriptions apply to all such graphs.[2]

[2] For many more experimental results and plots as well as for implementation notes see the full paper [1], supplementary information is stored at i11www.iti.uni-karlsruhe.de/projects/spp1307/dyneval

EMail Graph $\mathcal{G}_e$. The network of email contacts at the department of computer science at KIT is an ever-changing graph with an inherent clustering: Workgroups and projects cause increased communication. We weigh edges by the number of exchanged emails during the past seven days, thus edges can completely time out; degree-0 nodes are removed from the network. $\mathcal{G}_e$ has between 100 and 1500 nodes depending on the time of year, and about 700K events spanning about 2.5 years. It features a strong power-law degree distribution.

arXiv Graphs {arx}. Since 1992 the *arXiv.org e-Print archive*[3] is a popular repository for scientific e-prints, stored in several categories alongside timestamped metadata. We extracted networks of collaboration between scientists based on coauthorship. E-prints induce equally weighted clique-edges among the contributors such that each author gains a total edge weight of 1.0 per e-print contributed to. E-prints time out after two years and disconnected authors are removed.[5] As these networks are ill-natured for local updates, we use them as tough trials. We show results on two categories with large connected components.

Fundamental Results. For the sake of readability, we use a moving average in plots for distance and quality in order to smoothen the raw data. We consider the criteria quality (*modularity*), smoothness ($\mathcal{R}_g$) and runtime (ms), and additionally $|\mathcal{C}|$ as a structural indicator.

Discarding dEOO. In a first feasibility test, dEOO immediately falls behind all other algorithms in terms of quality (see full paper [1]), an observation substantiated by the fact that dEOO works better if related to some base algorithm [26]. Moreover, runtimes for dEOO as the sole technique are infeasible for large graphs.

Local Parameters. It has been stated in [7] that the order in which Local considers nodes is irrelevant. In terms of average runtime and quality we can confirm this for sLocal, though a random order tends to be less smooth; for dLocal the same observation holds (see full version [1]). However, since node order *does* influence specific values, a random order can compensate the effects this might have in pathological cases. Considering only affected nodes or also their neighbors in higher levels, does not affect any criterion on average.

pILP Variants. Allowing the ILP to merge existing clusters takes longer, and clusters coarser and with a slightly worse *modularity*; we therefore reject it.

Heuristics vs. dILP. A striking observation about dILP is the fact that it yields worse quality than dLocal and sLocal with identical *prep strategies*. Being locally optimal seems to overfit, a phenomenon that does not weaken over time and persists throughout most instances. Together with its high runtime and only small advantages in smoothness, dILP is ill-suited for updates on large graphs.

Static Algorithms. Briefly comparing sGlobal and sLocal we can state that sLocal consistently yields better quality and a finer yet less smooth clustering (see full version [1]). This generally applies to the corresponding dynamic algorithms as well. In terms of speed, however, sGlobal hardly lags behind sLocal, especially for small graphs with many connected components, where sLocal cannot capitalize on its strength of quickly reducing the size of a large instance.

[3] Website of e-print repository: arxiv.org

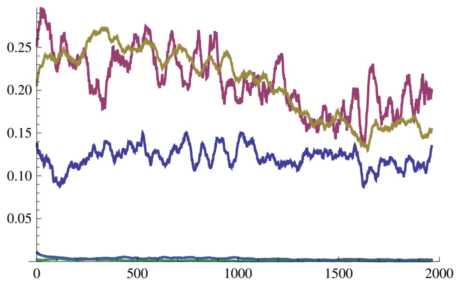


Fig. 2. $\mathcal{R}_g$, {ran} (top to bottom at right end): **sGlobal** (1st) and **sLocal** (2nd) are less smooth (factor 100) than **dLocal@BN**$_4$, **dGlobal@BN**$_{16}$ (bottom); **dGlobal@BT** (3rd) competes well

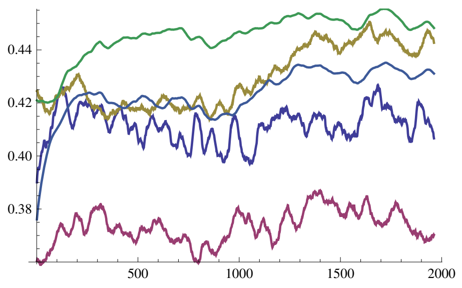


Fig. 3. *Modularity*, {ran} (top to bottom at right end): **dGlobal@BT** (4th) and **dGlobal@BN**$_{16}$ (3rd) beat **sGlobal** (5th); **dLocal@BN**$_4$ (1st) beats **sLocal** (2nd)

For such instances, separately maintaining and handling connected components could thus reasonably speed up sLocal, but would also do so for sGlobal.

Prep Strategies. We now determine the best choice of *prep strategies* and their parameters for dGlobal and dLocal. In particular, we evaluate N_d for $d \in \{0, 1, 2, 3\}$ and BN_s for $s \in \{2, 4, 8, 16, 32\}$, alongside BU and BT. Throughout our experiments $d = 0$ (or $s = 2$) proved insufficient, and is therefore ignored in the following. For dLocal, increasing d has only a marginal effect on quality and smoothness, while runtime grows sublinearly, which suggests $d = 1$. For dGlobal, N_d risks high runtimes for depths $d > 1$, especially for dense graphs. In terms of quality N_1 is the best choice, higher depths seem to deteriorate quality—a strong indication that large search spaces contain local optima. Smoothness approaches the bad values of sGlobal for $d > 2$. For BN, increasing s is essentially equivalent to increasing d, only on a finer scale. Consequently, we can report similar observations. For dLocal, BN_4 proved slightly superior. dGlobal's quality benefits from increasing s in this range, but again at the cost of speed and smoothness, so that BN_{16} is a reasonable choice. BU clearly falls behind in terms of all criteria compared to the other strategies, and often mimics the static algorithms. dGlobal using BT is by far the fastest algorithm, confirming our theoretical predictions from Sec. 2.2, but still produces competitive quality. However, it often yields a smoothness in the range of sGlobal. Summarizing, our best dynamic candidates are the algorithms dGlobal@BT and dGlobal@BN_{16} (achieving a speedup over sGlobal of up to 1k and 20 at 1k nodes, respectively) and algorithm dLocal@BN_4(speedup of 5 over sLocal).

Comparison of the Best. As a general observation, as depicted in Fig. 3, each dynamic candidate beats its static counterpart in terms of *modularity*. On the generated graphs, dLocal is superior to dGlobal, and faster. In terms of smoothness (Fig. 2), dynamics (except for dGlobal@BT) are superior to statics by a factor of ca. 100, but even dGlobal@BT beats them.

Trials on *arXiv* Data. As an independent data set, we use our *arXiv* grahps for testing our results from $\mathcal{G}_e$ and the random instances. These graphs consist solely of glued cliques of authors (papers), established within single timesteps where potentially many new nodes and edges are introduced. Together with modularity's *resolution limit* [22] and its fondness of balanced clusters and a non-arbitrary number thereof in large graphs [30], these degenerate dynamics are adequate for fooling local algorithms that cannot regroup cliques all over as to modularity's liking: Static algorithms constantly reassess a growing component, while dynamics using N or BN will sometimes have no choice but to further enlarge some growing cluster. Locally this is a good choice, but globally some far-away cut might qualify as an improvement over pure componentwise growth.

However, we measured that dGlobal@BT easily keeps up with the static algorithms' *modularity*, being able to adapt its number of clusters appropriately. The dynamic algorithms using other *prep strategies* do struggle to make up for their inability to re-cluster; however, they still only lag behind by about 1%. Figures 4 and 5 show *modularity* for coarse and fine batches, respectively, using the *arXiv* category *Nuclear Theory* (1992-2010, 33K e-prints, 200K elementary events, 14K authors). As before, dynamics are faster and smoother. For the coarse batches, speedups of 10 to 2K (BT) are attained; for fine batches, these are 100 to 2K. In line with the above observations, their clusterings are slightly coarser (except for dGlobal@BT) (see full paper [1] for further insights).

Summary of Insights. The outcomes of our evaluation are very favorable for the dynamic approach in terms of all three criteria. Furthermore, the dynamics exhibit the ability to react quickly and adequately to changes in the random generator's ground-truth clustering (see full paper [1]).

We observed that dLocal is less susceptible to an increase of the search space than dGlobal. However, our results argue strongly for the locality assumption in both cases—an increase in the search space beyond a very limited range is not

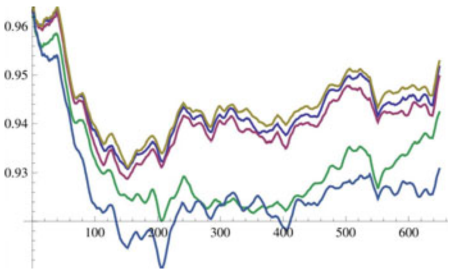


Fig. 4. *Modularity*, {arx}, batch size 50 e-prints (top to bottom at right end): Backtracking (**dGlobal@BT**) (2nd) easily follows the static algorithms (**sLocal** (1st) and **sGlobal** (3rd)); even **dLocal@BN**$_4$ (4th) and **dGlobal@BN**$_{16}$ (5th) lag behind by only $\sim 1\%$

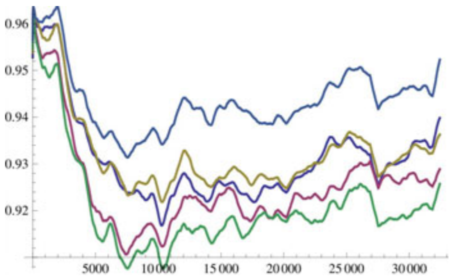


Fig. 5. *Modularity*, {arx}, batch size 1 e-print, dynamics only (top to bottom at right end): **dGlobal@BT** (1st) excels, followed by **dLocal@N**$_1$ (3rd) and **dLocal@BN**$_4$ (2nd) and then **dGlobal@BN**$_{16}$ (4th) and **dGlobal@N**$_1$ (5th) whom finer batches don't help

justified when trading off runtime against quality. On the contrary, quality and smoothness may even suffer for dLocal. Consequently, N and BN strategies with a limited range are capable of producing high-quality clusterings while excelling at smoothness. The BT strategy for dGlobal yields competitive quality at unrivaled speed, but at the expense of smoothness. For dLocal a gradual improvement of quality and smoothness over time is observable, which can be interpreted as an effect reminiscent of *simulated annealing*, a technique that has been shown to work well for *modularity* maximization [29]. Our data indicates that the best choice for an algorithm in terms of quality may also depend on the nature of the target graph. While dLocal surpasses dGlobal on almost all generated graphs, dGlobal is superior on our real-world instance $\mathcal{G}_e$. We speculate that this is due to $\mathcal{G}_e$ featuring a power law degree distribution in contrast to the Erdős-Rényi-type generated instances. In turn, our *arXiv* trial graphs, which grow and shrink in a volatile but local manner, allow a for a small margin of quality improvement, if the clustering is regularly adapted globally (re-balanced and coarsened/refined). Only the statics and dGlobal@BT are able to do this, however, at the cost of smoothness. Universally, the latter algorithm is the fastest. Concluding, some dynamic algorithm always beats the static algorithms; backtracking is preferable for locally concentrated or monotonic graph dynamics and a small search space is to be used for randomly distributed changes in a graph.

4 Conclusion

As the first work on *modularity*-driven clustering of dynamic graphs, we deal with the NP-hard problem of updating a *modularity*-optimal clustering after a change in the graph. We developed dynamizations of the currently fastest and the most widespread heuristics for *modularity*-maximization and evaluated them and a dynamic partial ILP for local optimality. For our fastest update strategy, we can prove a tight bound of $\Theta(\log n)$ on the expected number of backtrack steps required. Our experimental evaluation on real-world dynamic networks and on dynamic clustered random graphs revealed that dynamically maintaining a clustering of a changing graph does not only save time, but also yields higher *modularity* than recomputation—except for degenerate graph dynamics—and guarantees much smoother clustering dynamics. Moreover, heuristics are better than being locally optimal at this task. Surprisingly small search spaces work best, avoid trapping local optima well and adapt quickly and aptly to changes in the ground-truth clustering, which strongly argues for the assumption that changes in the graph ask for local updates on the clustering.

References

1. Görke, R., Maillard, P., Staudt, C., Wagner, D.: Modularity-Driven Clustering of Dynamic Graphs. Technical report, Universität Karlsruhe (TH), Informatik, TR 2010-5 (2010)

2. Brandes, U., Delling, D., Gaertler, M., Görke, R., Höfer, M., Nikoloski, Z., Wagner, D.: On Modularity Clustering. IEEE TKDE 20(2), 172–188 (2008)
3. Fortunato, S.: Community detection in graphs. Elsevier Phys. R 486(3-5) (2009)
4. Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Physical Review E 69(026113) (2004)
5. Keogh, E., Lonardi, S., Ratanamahatana, C.A.: Towards Parameter-Free Data Mining. In: Proc. of the 10th ACM SIGKDD Int. Conf., pp. 206–215. ACM, New York (2004)
6. Schaeffer, S.E., Marinoni, S., Särelä, M., Nikander, P.: Dynamic Local Clustering for Hierarchical Ad Hoc Networks. In: Proc. of Sensor and Ad Hoc Communications and Networks, vol. 2, pp. 667–672. IEEE, Los Alamitos (2006)
7. Blondel, V., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: The. and Exp. 2008(10)
8. Delling, D., Görke, R., Schulz, C., Wagner, D.: ORCA Reduction and ContrAction Graph Clustering. In: Goldberg, A.V., Zhou, Y. (eds.) AAIM 2009. LNCS, vol. 5564, pp. 152–165. Springer, Heidelberg (2009)
9. Görke, R., Hartmann, T., Wagner, D.: Dynamic Graph Clustering Using Minimum-Cut Trees. In: Dehne, F., et al. (eds.) WADS 2009. LNCS, vol. 5664, pp. 339–350. Springer, Heidelberg (2009)
10. Hopcroft, J.E., Khan, O., Kulis, B., Selman, B.: Tracking Evolving Communities in Large Linked Networks. Proceedings of the National Academy of Science of the United States of America 101 (April 2004)
11. Palla, G., Barabási, A.L., Vicsek, T.: Quantifying social group evolution. Nature 446, 664–667 (2007)
12. Aggarwal, C.C., Yu, P.S.: Online Analysis of Community Evolution in Data Streams. In: [31]
13. Sun, J., Yu, P.S., Papadimitriou, S., Faloutsos, C.: GraphScope: Parameter-Free Mining of Large Time-Evolving Graphs. In: Proc. of the 13th ACM SIGKDD Int. Conference, pp. 687–696. ACM Press, New York (2007)
14. Hübner, F.: The Dynamic Graph Clustering Problem - ILP-Based Approaches Balancing Optimality and the Mental Map. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik (May 2008)
15. Chakrabarti, D., Kumar, R., Tomkins, A.S.: Evolutionary Clustering. In: Proc. of the 12th ACM SIGKDD Int. Conference, pp. 554–560. ACM Press, New York (2006)
16. Schaeffer, S.E.: Graph Clustering. Computer Science Review 1(1), 27–64 (2007)
17. White, S., Smyth, P.: A Spectral Clustering Approach to Finding Communities in Graphs. In: [31], pp. 274–285
18. Pons, P., Latapy, M.: Computing Communities in Large Networks Using Random Walks. Journal of Graph Algorithms and Applications 10(2), 191–218 (2006)
19. van Dongen, S.M.: Graph Clustering by Flow Simulation. PhD thesis, University of Utrecht (2000)
20. Clauset, A., Newman, M.E.J., Moore, C.: Finding community structure in very large networks. Physical Review E 70(066111) (2004)
21. Brandes, U., Erlebach, T. (eds.): Network Analysis: Methodological Foundations. LNCS, vol. 3418. Springer, Heidelberg (2005)
22. Fortunato, S., Barthélemy, M.: Resolution limit in community detection. PNAS 104(1), 36–41 (2007)
23. Newman, M.E.J.: Analysis of Weighted Networks. P. R. E 70(056131), 1–9 (2004)
24. Görke, R., Gaertler, M., Hübner, F., Wagner, D.: Computational Aspects of Lucidity-Driven Graph Clustering. JGAA 14(2) (2010)

25. Delling, D., Gaertler, M., Görke, R., Wagner, D.: Engineering Comparators for Graph Clusterings. In: Fleischer, R., Xu, J. (eds.) AAIM 2008. LNCS, vol. 5034, pp. 131–142. Springer, Heidelberg (2008)
26. Noack, A., Rotta, R.: Multi-level Algorithms for Modularity Clustering. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 257–268. Springer, Heidelberg (2009)
27. Görke, R., Staudt, C.: A Generator for Dynamic Clustered Random Graphs. Technical report, Universität Karlsruhe (TH), Informatik, TR 2009-7 (2009)
28. Brandes, U., Gaertler, M., Wagner, D.: Experiments on Graph Clustering Algorithms. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 568–579. Springer, Heidelberg (2003)
29. Guimerà, R., Amaral, L.A.N.: Functional Cartography of Complex Metabolic Networks. Nature 433, 895–900 (2005)
30. Good, B.H., de Montjoye, Y., Clauset, A.: The performance of modularity maximization in practical contexts. arxiv.org/abs/0910.0165 (2009)
31. Proceedings of the fifth SIAM International Conference on Data Mining. SIAM, Philadelphia (2005)

Gateway Decompositions for Constrained Reachability Problems

Bastian Katz[1], Marcus Krug[1], Andreas Lochbihler[2], Ignaz Rutter[1], Gregor Snelting[2], and Dorothea Wagner[1]

[1] Institute of Theoretical Informatics
[2] Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
firstname.lastname@kit.edu

Abstract. *Given a directed graph whose vertices are labeled with propositional constraints, is there a variable assignment that connects two given vertices by a path of vertices that evaluate to true?* Constrained reachability is a powerful generalization of reachability and satisfiability problems and a cornerstone problem in program analysis. The key ingredient to tackle these computationally hard problems in large graphs is the efficient construction of a short *path condition*: A formula whose satisfiability is equivalent to constrained reachability and which can be fed into a state-of-the-art constraint solver.

In this work, we introduce a new paradigm of decompositions of digraphs with a source and a target, called *gateway decompositions*. Based on this paradigm, we provide a framework for the modular generation of path conditions and an efficient algorithm to compute a fine-grained gateway decomposition. In benchmarks, we show that especially the combination of our decomposition and a novel arc filtering technique considerably reduces the size of path conditions and the runtime of a standard SAT solver on real-world program dependency graphs.

1 Introduction

Constrained reachability (CR) is a straightforward, yet powerful generalization of the well-known satisfiability problem: Given a digraph with propositional formulas as vertex or arc labels, it asks whether for some truth assignment there is a path connecting two given vertices s and t such that all formulas along the path evaluate to true.

CR problems are vital to Information Flow Control (IFC), a central problem in program analysis. In particular, IFC aims at answering for a given piece of code under which conditions the (confidential) outcome of a statement s can influence the execution of an (observable) statement t. Since this problem is undecidable in general, an approximation is sought that must report all such influences and raises as few *false* alarms as possible. Recent works address this problem by formalizing *execution conditions* which are necessary conditions for the execution of each statement [16]. These approaches then look for a path in the *program dependency graph* modeling all immediate influences between statements by control or data dependencies. The existence of a variable assignment—including input variables—such that all execution conditions of such a path between s and t are satisfied at the same time is a very strong necessary condition for information flow. Supporting alarms by a critical variable assignment also tells *under what conditions* a security leak might occur.

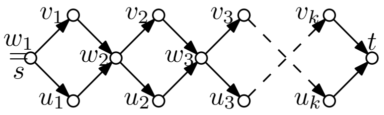


Fig. 1. Acyclic graph with an exponential number of s-t-paths

Solvers for propositional formulas (e. g. MiniSat for Boolean logic [5,4]) have been greatly improved during the last decade and made more and more complex satisfiability problems feasible. Nevertheless the naive way of testing every single s-t-path for satisfiability is far beyond inefficient since there can be an infinite number of such paths in the presence of cycles. Restricting the tests to *simple s-t-paths* is sufficient—every satisfiable path contains a simple s-t-path—but there still can be an exponential number of simple s-t-paths even in acyclic digraphs.

A different approach that benefits from the development of SAT solvers is to reduce constrained reachability problems to single propositional formulas, for which the satisfiability problem is equivalent to the CR problem. Finding a *short* such formula potentially reduces the runtime of the back-end solver. However, since the smallest possible path condition are either $PC_{\mathrm{OPT}} = \top$ (true) or $PC_{\mathrm{OPT}} = \bot$ (false), optimizing such a condition's length is obviously NP-hard. Also, to preserve the information about critical variable assignments, we look for formulas whose satisfying variable assignments also satisfy the original CR problem. Such a formula is called a *path condition* (PC). The efficient construction of short path conditions is a key to solving large CR problems. For the above example, with c_v denoting the label of vertex v, $c_{w_1} \wedge (c_{v_1} \vee c_{u_1}) \wedge c_{w_2} \wedge (c_{v_2} \vee c_{u_2}) \wedge \cdots \wedge c_{w_k} \wedge (c_{v_k} \vee c_{u_k}) \wedge c_t$ is such a path condition. Unfortunately, not all graphs have such a straightforward path condition. Snelting et al. were the first to use graph decompositions to exploit the structure of program dependency graphs to find *modular* path conditions [16]. Although this idea has been applied to CR problems with great success, the issue of tailoring decompositions to this task has hardly has been looked at. So far, all decompositions considered have originally been designed for loop detection and not even been proven to be a correct basis for PC generation.

Our contribution: We introduce a novel paradigm of decomposing digraphs with designated source and target vertices into nested *gateway* subgraphs. Applied to CR problems, they allow for the first characterization of provably correct modular PC generation. We give an efficient algorithm to compute a fine-grained gateway decomposition based on connectivity and dominance relations. We further introduce a novel technique to filter irrelevant vertices and arcs from the input, which significantly improves the quality of decompositions. In combination these two techniques reduce the size of path conditions by almost 70% on average and reduces the running time of a SAT solver by roughly a factor of 4 in real-world program dependency graphs. All algorithms have been implemented in C++ and decompose real-world instances within seconds.

2 Preliminaries and Problem Statement

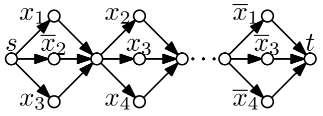


Fig. 2. Instance of CR corresponding to the 3SAT instance $(x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge \cdots \wedge (\overline{x}_1 \vee \overline{x}_3 \vee \overline{x}_4)$

Throughout this work, we assume a digraph $D = (V, A)$ with a start vertex s, a target vertex t and propositional constraints c_v for all $v \in V$. For the sake of simplicity, we assume the c_v to be from truth-valued propositional logics, although our approach naturally extends to other domains. The problem ConstrainedReachability (CR) is to decide whether there is an s-t-path P such that there is a variable assignment f with $f(c_v) = \top$ (true) for all $v \in P$. As stated above, this is equivalent to deciding whether there is a *simple* such s-t-path. Not surprisingly, CP is NP-hard, even if vertex constraints are restricted to literals: It generalizes the classic 3Sat problem by drawing instances of 3Sat as depicted in Fig. 2. NP-completeness is straightforward; a similar proof for labeled arcs can be found in [10].

A *path condition* is a propositional formula PC that can be satisfied if and only if the corresponding CR problem has a solution and a truth assignment f with $f(PC) = \top$ satisfies $\wedge_{v\in P} c_v$ for some s-t-path P. Since SAT solvers typically take formulas in *conjunctive normal form* (CNF) as input, we focus on the construction of such a formula ϕ and measure its *complexity* by the number of variable occurrences, denoted by $|\phi|$. We assume the c_v to be in CNF, i. e., to be of the form $c_v^1 \wedge \cdots \wedge c_v^{m_v}$ where each c_v^i is a disjunction of literals.

We can assume that every vertex is on some s-t-path, possibly containing a cycle. If this is not the case, all vertices violating this condition are missed by either a forward search from s or a backward search from t. They can be removed from the input without losing any s-t-path. This process is sometimes referred to as "chopping", and more generally, we will call a digraph $D = (V, A)$ an *S-T-chop* for some $S, T \subset V$ if every $v \in V$ is on some s-t-path with $s \in S$ and $t \in T$. In this sense, we assume the input to be an $\{s\}$-$\{t\}$-chop, or, in a slight abuse of notation, an s-t-chop. From now on we deal with the following problem:

Given an s-t-chop, find a path condition PC in CNF whose complexity is as small as possible.

3 Linear Path Conditions and Acyclicity Constraints

In the introductory example (cf. Fig. 1), we have seen that linear-size path conditions are possible in some cases. Notably, even in CNF, any *acyclic* digraph has a path condition whose complexity is in $O(n+m+k)$ for $n := |V|$, $m := |A|$ and $k := \sum_{v\in V} |c_v|$, i. e., linear in the input. It can be obtained by introducing additional variables x_v which can be read as "there is a true-valued path from s to v":

$$PC^{\text{DAG}} := c_s \wedge x_t \bigwedge_{v\in V-s} \big((x_v \to c_v) \wedge (x_v \to (\vee_{uv\in A} x_u))\big)$$

Note that implications of the form $(a_1 \wedge \cdots \wedge a_i) \rightarrow (b_1 \vee \cdots \vee b_i)$ can be written in CNF as $(\overline{a}_1 \vee \cdots \vee \overline{a}_i \vee b_1 \vee \cdots \vee b_i)$ with the same complexity. We use $(x_v \rightarrow c_v)$ as a shorthand for adding $\overline{x}_v$ to every clause in c_v, yielding a complexity of at most $2|c_v|$. Hence, the whole formula can then be read as: "A valid truth assignment must satisfy c_s, and there must be a true-valued path to t. To have a valid path to any node v other than s, it is necessary to satisfy c_v and to have a path to some of its predecessors."

Unfortunately, this does not hold for arbitrary digraphs: If there is a cycle C and a v-t-path P for some $v \in C$ such that $c_s \wedge_{v \in C \cup P} c_v$ is satisfiable, we can extend such a truth assignment to also satisfy PC^{DAG} by setting x_v to true if and only if $v \in C \cup P \cup \{s\}$. We remedy this problem by modeling the selection of certain acyclic subgraphs of D as an *acyclicity condition*, introducing new variables y_{uv} for every $u, v \in V$. For a later use, we generalize acyclicity conditions to the case of multiple source and target vertices.

Definition 1. *Let $D = (V, A)$ be an S-T-chop. A boolean formula ϕ is called an S-T-*acyclicity condition *if (a) for every truth assignment f that satisfies ϕ, the set of arcs $A_f := \{uv \in A \mid f(y_{uv}) = \top\}$ is acyclic and (b) for every acyclic S-T-chop $(V', A') \subset D$, there is a truth assignment f satisfying ϕ with $A' \subset A_f$.*

To give an example, the *partial ordering constraint* (antisymmetry+transitivity)

$$\phi^{\mathrm{PO}} := \bigwedge_{u,v \in V} (\overline{y}_{uv} \vee \overline{y}_{vu}) \bigwedge_{u,v,w \in V} \big((y_{uv} \wedge y_{vw}) \rightarrow y_{uw}\big)$$

is an acyclicity condition for every S-T-chop, but with a cubic complexity. For acyclic graphs on the other hand, $\phi^{\mathrm{DAG}} := \top$ is an acyclicity condition. The following lemma integrates acyclicity conditions into PC generation for general graphs. Due to a lack of space, we omit proofs.

Lemma 1. *Let $D = (V, A)$ be an s-t-chop and ϕ an s-t-acyclicity condition for D. Then the following is a path condition for D:*

$$PC^{\phi} := \phi \wedge c_s \wedge x_t \bigwedge_{v \in V - s} \big(x_v \rightarrow (c_v \wedge \vee_{uv \in A} x_{uv})\big) \bigwedge_{uv \in A} \big(x_{uv} \rightarrow (x_u \wedge y_{uv})\big)$$

Given an s-t-acyclicity condition ϕ, constructing PC^{ϕ} adds only linear complexity, i.e., $|PC^{\phi}| \in O(|\phi| + k + m)$. The complexity of PC^{ϕ} can further be decreased by the following observation: If for some $uv \in A$, the variable y_{uv} does not occur in ϕ, we can replace occurrences of x_{uv} and y_{uv} by x_u and hence remove clauses of the form $(\overline{x}_{uv} \vee x_u)$ or $(\overline{x}_{uv} \vee y_{uv})$. If A_{ϕ} denotes the set of $uv \in A$ for which the variable y_{uv} occurs in ϕ, we can hence use

$$\phi \wedge c_s \wedge x_t \bigwedge_{v \in V - s} \big(x_v \rightarrow (c_v \wedge \vee_{uv \in A \setminus A_{\phi}} x_u \vee_{uv \in A_{\phi}} x_{uv})\big) \bigwedge_{uv \in A_{\phi}} \big(x_{uv} \rightarrow (x_u \wedge y_{uv})\big)$$

instead of PC^{ϕ}, which yields exactly PC^{DAG} for an acyclic digraph D and ϕ^{DAG}.

Using ϕ^{PO} as suggested above yields a path condition with complexity in $\Theta(k + n^3)$. As a first step towards small acyclicity conditions and hence small path conditions, we introduce a shorter acyclicity condition, $\phi^{\star}$:

Lemma 2. *Let $D = (V, A)$ be an S-T-chop. The following is an S-T-acyclicity condition for D:*

$$\phi^\star_D := \bigwedge_{uv \in A} (\overline{y}_{uv} \vee \overline{y}_{vu}) \quad \bigwedge_{uv \in A, w \in V \setminus \{u,v\}} ((y_{wu} \wedge y_{uv}) \rightarrow y_{wv})$$

Using $\phi^\star := \phi^\star_D$ instead of the partial ordering constraint as above yields a path condition with complexity $\Theta(k + mn)$ for general digraphs.

4 Gateway Decompositions

So far, we only distinguish between acyclic graphs, which allow for a path condition with linear complexity, and general digraphs, for which a path condition takes possible cycles into account at the expense of their complexity. In this section, we introduce a novel paradigm for the nested decomposition of digraphs into subgraphs, "gateways", with a special property. Among other possible applications, such gateway decompositions allow us to modularize the construction of acyclicity constraints to critical portions of the graph. The motivating observation for this section is the following: Let D be an s-t-chop, and let $D_1, \ldots, D_q$ be its strongly connected components. Each possible cycle lies within one of the D_i, and every acyclic s-t-chop in D can be composed by taking all arcs not covered by a component and picking some acyclic arc set within the D_i. Even stronger, any acyclic s-t-chop in D can be composed by taking all arcs not covered by a D_i and S_i-T_i-chops for each of the D_i with S_i and T_i denoting D_i's *entry* and *exit vertices*, i. e., vertices with in- or out-neighbors not part of that component, respectively. Applied to the problem of PC generation, this means that $\phi^\star_{D_1} \wedge \cdots \wedge \phi^\star_{D_q}$ is an acyclicity condition for D. We will give the formal proof for this observation as part of a much more powerful decomposition framework, which focuses on the relevant function of strongly connected components in the above observation:

Definition 2. *Let $D = (V, A)$ be an s-t-chop and let G be a node-induced subgraph. We call G a* gateway *if no simple s-t-path P enters G after leaving it.*

Obviously, strongly connected components are gateways, but Fig. 3 shows another example where a gateway can be found within a strongly connected component. For a gateway G, we denote by $S_G := \{v \in V_G : \exists u \notin V_G, uv \in A\}$ the *entry vertices* and by $T_G := \{u \in V_G : \exists v \notin V_G, uv \in A\}$ the *exit vertices* of G.

We now turn to the main theorem of this work, which will allow us later to turn a hierarchy of nested gateways into a modular and compact acyclicity condition. It considers how a subgraph $D' = (V', A')$ of an s-t-chop D can be decomposed into the result of contracting the vertices of a gateway $G = (V_G, A_G)$, denoted by D'/G and the intersection with G, denoted by $D' \cap G$. It also considers the somehow reverse *composition*, which, given a subgraph C of D/G and a subgraph G' of G selects a subgraph $C \oplus G'$ of D containing all vertices and arcs of G' plus all vertices and arcs in D mapped to vertices or arcs in C by contraction

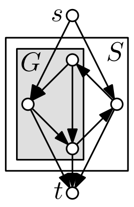


Fig. 3. A gateway G inside a strongly connected component S

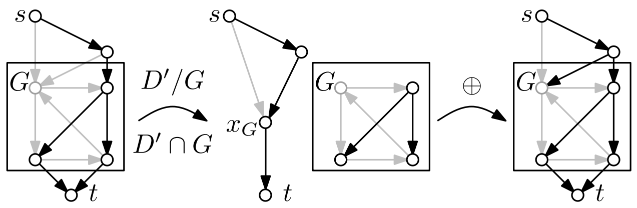


Fig. 4. Digraph D with acyclic s-t-chop D' (black), decomposed into contracted chop D'/G and projected chop $D' \cap G$ and re-composition $D'/G \oplus (D' \cap G)$

of G. These two operations are not inverse in general, since contraction can map different arcs of D to the same arc in D/G, see Fig. 4 for an example.

Theorem 1 (Composition Theorem). *Let $D = (V, A)$ be an s-t-chop and let G a gateway.*

1. *The* decomposition *of any acyclic s-t-chop D' with respect to G yields an acyclic s-t-chop D'/G in D/G and an acyclic S_G-T_G-chop $D' \cap G$ in G and $D' \subseteq (D'/G) \oplus (D' \cap G)$.*
2. *The* composition *$D^* \oplus G'$ of any acyclic s-t-chop D^* in D/G and any acyclic S_G-T_G-chop G' in G is acyclic with $(D^* \oplus G')/G = D^*$, $(D^* \oplus G') \cap G = G'$.*

Its main contribution is the following: For every acyclic s-t-chop D' in D, a supergraph can be composed from an acyclic s-t-chop C in D/G and an acyclic S_G-T_G-chop G' in G, and any such composition is acyclic. This immediately allows a modularization of acyclicity conditions with the help of additional variables to model the identification of arcs mapped to the same image under contraction.

Corollary 1. *Let $D = (V, A)$ be an s-t-chop and let G be a gateway. Let $\phi_{D/G}$ be an s-t-acyclicity condition for D/G and ϕ_G be an S_G-T_G-acyclicity condition for G. Then the following is an s-t-acyclicity condition for D:*

$$\phi_{D/G} \wedge \phi_G \bigwedge_{uv \in A: u \notin G, v \in G} (y_{uv} \doteq y_{ux_G}) \bigwedge_{uv \in A: u \in G, v \notin G} (y_{uv} \doteq y_{x_G v})$$

A nested gateway decomposition of a digraph D is a rooted tree T such that the vertices of T are gateways and (a) D is the root, (b) a gateway is a subgraph of all ancestors, super-graph of all its descendants, and disjoint to any other gateway. Given such a decomposition, Corollary 1 allows us to construct a path condition bottom-up. Note that the resulting path condition is a conjunction not only of disjunctive clauses, but also of the equivalences introduced in Corollary 1. Once the construction is completed, variables aliased by these equivalences may be replaced, e. g., an equivalence $(x_1 \doteq x_2)$ can be used to replace each occurrence of x_1 by x_2. When all equivalences are of the form $(x \doteq x)$, they can be removed from the formula, leaving a less complex path condition in pure CNF. In fact, with n_S and m_S being the number of vertices and arcs of a gateway after contraction of inner gateways, respectively, the resulting path condition has a complexity of less than

$$2k + n + m + \sum_{S \in V_T \text{ not acyclic}} (6m_S + 3m_S(n_S - 2)) \in O(k + m + \sum_{S \in V_T \text{ not acyclic}} n_S m_S) \ .$$

5 Decomposition Paradigms

Finding a good hierarchy of gateways can dramatically reduce the complexity of the acyclicity condition and hence of a path condition. Since only non-acyclic nodes of such a decomposition contribute to the acyclicity condition, a good hierarchy not only has to be fine-grained, but should foremost reduce the size and number of the gateways containing cycles. Snelting et al. proposed to apply *loop decomposition* in a very similar setting, where path conditions were constructed by enumerating simple paths in components that are not acyclic [16]. In this approach, components with cycles can contribute exponentially to the resulting path condition's complexity, and avoiding large such components is even more important than with acyclicity constraints. Loop decompositions primarily aim at the construction of a hierarchy of strongly connected subgraphs and have a long tradition in program analysis. Robschink compares a variety of loop decomposition algorithms on program dependency graphs [14] that extend Tarjan's *interval analysis* [19] to irreducible graphs. Among them, he compared algorithms due to Havlak [9], Steensgaard [18], and Sreedhar, Gao, and Lee (SGL) [17]. A good overview on loop decomposition algorithms and their implementation is due to Ramalingam [13]. Currently, SGL is the state-of-the-art decomposition applied to PC problems [14,16]. Although not explicitly proven, the "loops" computed by the SGL algorithm are gateways. Hence, using SGL as base for hierarchical acyclicity conditions is correct and—like any nontrivial decomposition—improves the complexity of the resulting path condition. Since SGL is designed to identify loops in programs, all subgraphs in the decomposition are strongly connected, i. e., SGL does not identify acyclic subgraphs within strongly connected components.

The Connectivity-Dominance Decomposition

The identification of nested strongly connected subgraphs is not the only reasonable way to compute a gateway decomposition. The following lemma provides the basis for a much more natural recursive *connectivity-dominance* (CD) decomposition paradigm: It extensively uses *dominator* and *postdominator* relations: A vertex d *dominates* a vertex x, if any s-x-path must contain d. Analogously, a vertex p *postdominates* a vertex x, if any x-t-path must contain p. We write d *dom* x or p *pdom* x, respectively. The transitive reduction of both dominator and postdominator relation in chopped digraphs are rooted trees. They can be computed using the Lengauer-Tarjan algorithm [12], which is implemented in most libraries for graph algorithms and runs in almost linear time, or the linear time algorithm of Buchsbaum et al. [2].

Algorithm 1. CDDECOMPOSITION(D, G)

```
1  $G' \leftarrow \perp$;
2  if $G$ is neither strongly connected nor acyclic then
3      $G' \leftarrow$ any SCC of $G$;
4  if $G$ is strongly connected then
5      if $V_G \supsetneq V_u^d(G) \supsetneq \{u\}$ for some $u \in V_G$ then
6          $G' \leftarrow D[V_u^d(G)]$;
7      else if $V_G \supsetneq V_u^p(G) \supsetneq \{u\}$ for some $u \in V_G$ then
8          $G' \leftarrow D[V_u^p(G)]$;
9  if $G' = \perp$ then return new Tree($G$);
10 tree $\leftarrow$ CDDECOMPOSITION($D, G/G'$);
11 tree.APPENDTOROOT(CDDECOMPOSITION($D, G'$));
12 return tree;
```

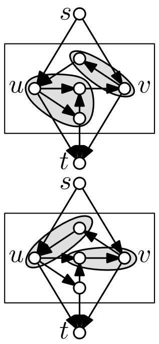


Fig. 5. Decomposition by V^d (top) and V^p (bottom)

Lemma 3. *Let D be an s-t-chop and G a gateway. (1) If G is not strongly connected then all strongly connected subgraphs of G are gateways of D. (2) If G is strongly connected, then for any $u \in V_G$, the subgraphs induced by $V_u^d(G) := \{v \in V_G : u \text{ dom } v\}$, $V_u^p(G) := \{v \in V_G : u \text{ pdom } v\}$, or $V_u(G) := V_u^d \cup V_u^p$ are gateways of D.*

In other words, if some gateway is not a strongly connected component, we can iteratively contract strongly connected components until the gateway becomes acyclic. If a gateway is strongly connected, we can iteratively contract (nontrivial) gateways induced by V_u^d, V_u^p, or V_u—those gateways may or may not be strongly connected, until no two remaining vertices dominate or postdominate each other. In either case, we can recursively process the identified gateways. This process is depicted in Algorithm 1. Note that decomposition of strongly connected components in Algorithm 1 is completely independent of the order in which the components are identified and processed. In fact, it is safe to identify them at once and process them subsequently. This is not true in general for the identification of gateways in Algorithm 1: Figure 5 gives an example where only two vertices u, v in a strongly connected component of five vertices define non-trivial gateways. Only the gateways induced by V_u^p and V_v^p *or* V_u^d and V_v^d are disjoint, such that contraction of one of them does not alter the other. This is true in general: Decomposition of all V_u^d for all vertices u which do not have a dominator in D' can be done independently, analogously the decomposition of all V_u^p for all vertices without a postdominator in D'. When mixing the relations or using the union V_u, contracting one subgraph may affect other induced subgraphs and candidates need to be recomputed. We hence consider only the two variants which either decompose strongly connected components along the V_u^d as long as possible and then along the V_u^p, or vice versa. Figure 6 shows an exemplary decomposition after applying an additional arc filtering step (cf. next section).

6 Filtering Techniques

An obvious step to further reduce the complexity of path conditions is a preprocessing to remove irrelevant vertices and arcs. Unfortunately, it is NP-hard for both vertices and arcs to decide whether they lie on any simple s-t-path: Given two pairs s_1, t_1 and s_2, t_2, it is hard to decide whether there are two disjoint paths connecting s_1 to t_1 and s_2 to t_2 [6]. Asking for a simple s_1-t_2-path in the same graph augmented by an arc $t_1 s_2$ also answers this question.

Nevertheless, we can efficiently identify sets of arcs and vertices that cannot be part of a *simple* s-t-path. As a first step, we applied chopping to remove vertices and arcs that do not belong to *any* s-t-path. Based on dominance and postdominance, we can filter additional arcs from the input.

Lemma 4 (Arc Filtering). *Let $D = (V, A)$ be a digraph with start vertex s and target vertex t. If for some $uv \in A$ there is an $x \in V$ with x dom u and x pdom v, then every path condition for $D - uv$ is also a path condition for D.*

More generally, a path condition for $D - A'$ is a path condition for D if all arcs in A' are candidates according to Lemma 4. Using interval labeling [15] and interval trees [3], all candidates for arc filtering can be found in $O(m \log n)$ time.

Unfortunately, removing arcs can change reachability and domination relations. Although in all experiments (cf. Section 7), repeating chopping and arc filtering once was sufficient, it is possible to construct graphs that require a linear number of filtering steps to reach a fixpoint. Also, with the decision problem being hard in general, we cannot hope to filter all irrelevant arcs.

Combining arc filtering with SGL can have ambivalent effects. Removing arcs and vertices can decrease the size of strongly connected components, but at the same time, breaking cycles within those components can hinder identification of nested loops. For CD decomposition on the other hand, arc filtering has a very helpful effect: When a gateway G' is identified within a strongly connected gateway G applying Lemma 3, G' is never strongly connected. Thus, a CD decomposition can nest acyclic gateways (after contraction of inner gateways) and strongly connected gateways (cf. Fig 6).

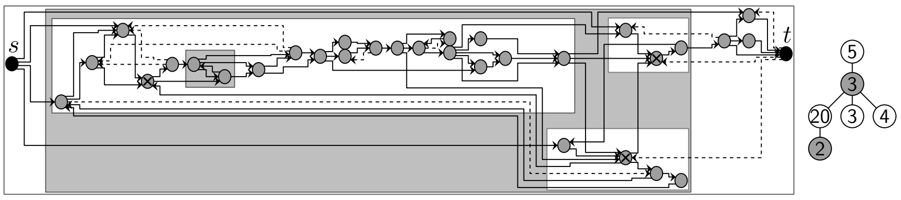


Fig. 6. CD decomposition after arc filtering applied to a small program dependency graph (filtered arcs dotted): Resulting PC has 52 variables, 67 clauses, and a complexity of 164 plus the complexity of the c_v. Not applying a decomposition/applying SGL after (without) arc filtering yields a formula with 1147(1158)/784(916) variables, 2005(2379)/1273(1765) clauses, and a complexity of 5833(6933)/3671(5105) plus the complexity of the c_v.

7 Experiments

To evaluate our approach we compare the formula size generated by our dominance-based decomposition with the formula sizes obtained from various decomposition methods. Moreover, we also compare the running time a standard SAT solver needs to solve the formulas generated by SGL and CD decompositions. Decomposition and PC generation algorithms have been implemented in C++. For SAT solving we use MiniSat2 [5,4], a standard reference in SAT solving. All experiments were performed on an AMD Opteron@2.6Ghz with 16GB of RAM.

In the following we describe our experimental setup and data in detail. We use several program dependency graphs (PDGs) of real world programs. Namely, we use programs from *The Java Grande Forum Benchmark Suite*[1] from the University of Edinburgh and the PACAP study [1], two well-known benchmark sets in program analysis. For the generation of the PDGs from the source files we use the SDG generator of the JOANA project [7]. Details on the generation process can be found in the literature on program analysis [11,8].

From each graph we pick 200 random *s*-*t*-pairs and compute their corresponding chops. We then compare the reduction of PC-complexity resulting from CD decomposition and SGL decomposition, both with and without arc-filtering. Almost all decompositions could be computed in less than 10 seconds, in all cases the running time was negligible compared to PC generation and SAT solving.

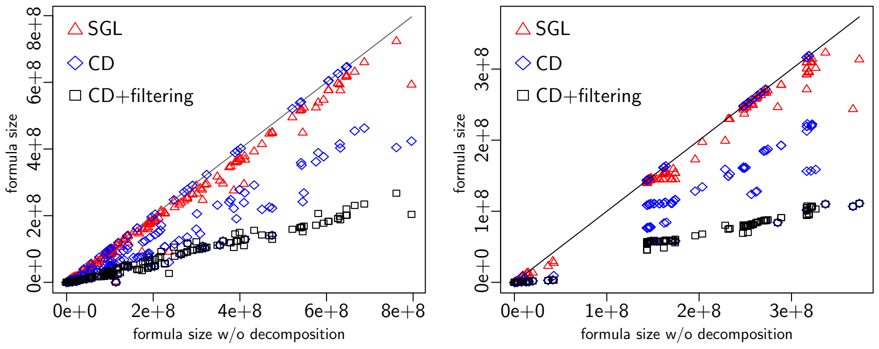


Fig. 7. Formula sizes for different decomposition techniques on benchmark sets Crypt (left) and JavaCard (right)

Figure 7 shows the results for two of our benchmark sets as scatter plots. The black line indicates the PC size if no decomposition is used at all. It can be seen that SGL reduces the formula size only marginally in most cases. The effect of applying arc filtering before SGL is almost negligible and is therefore not shown in the plot. Clearly, CD yields an improvement of up to 50% in many cases although sometimes it does not have any effect and the reduction has a

[1] www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html

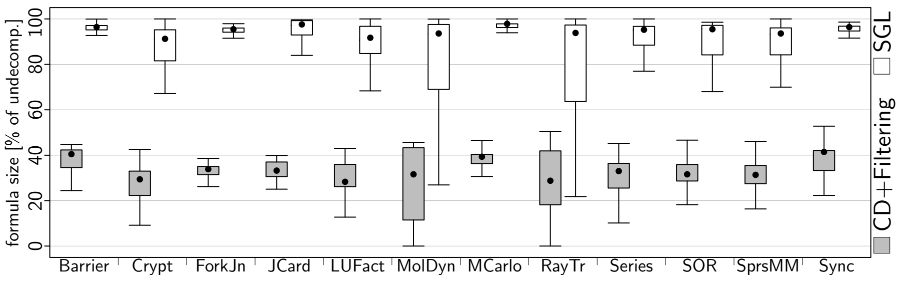


Fig. 8. Formula sizes of the complete benchmark set of our technique and the best previous competitor SGL

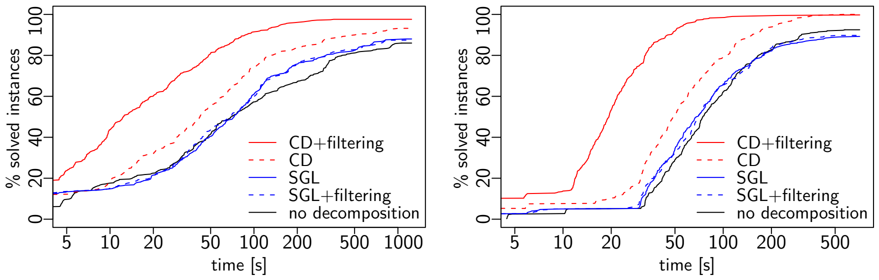


Fig. 9. Running times necessary to solve SAT-formulas generated for PDGs of the benchmark sets Crypt (left) and JavaCard (right)

high variance. Using CD after arc filtering gives consistently good results with an average reduction of almost 70% with a rather low variance; for 75% of the instances the reduction is at least 62%. Especially the effect that arc filtering improves CD, which was already justified theoretically in the previous section, seems to have a strong effect for practical instances. The results are actually prototypical for all our benchmark sets, see Fig. 8.

We further demonstrate that better decompositions and thus smaller PCs lead to faster solving by a standard SAT solver. For this time-consuming task, we restrict our test data to two of the PDGs aforementioned, namely to the PDGs generated from the JGF Crypt Bench and from the JavaCard study. In smaller samples, other PDGs showed a very similar behavior. For each we take the first 50 of the generated random chops and generate ten CR problems from each of them by attaching to each vertex randomly 5 to 15 clauses each containing 3 literals over $0.05 \cdot n$ (or at least 6) variables. We then compare the MiniSat2 running times for solving using a cut-off time of 1200 seconds per instance.

Figure 9 shows the fraction of instances that can be solved individually within a given time for different decompositions. Our experiments clearly show a correlation between the formula size and the running time of a SAT solver. Especially using CD with arc-filtering gives a significant improvement in running times over the SGL decomposition: CD solves 83% of all instances generated from the JGF Crypt benchmark within 50 seconds each, whereas SGL solves only 51% of all instances within this time and needs more than 160 seconds to solve the same amount of instances. Similarly, CD solves 91% of all instances generated from the JavaCard Benchmark within 50 seconds each, whereas SGL solves only 44% within this time and needs 119 seconds to solve the same amount of instances.

Acknowledgements. We thank Jürgen Graf for providing us with test data and Carsten Sinz for helpful discussions on SAT solving.

References

1. Attali, P.I.I., Jensen, T., Cards, J.O.S., Bieber, P., Cazin, J., El-marouani, A., Girard, P., louis Lanet, J., Wiels, V., Zanon, G.: The PACAP Prototype: a Tool for Detecting Java Card Illegal Flow (2001)
2. Buchsbaum, A.L., Kaplan, H., Rogers, A., Westbrook, J.R.: Linear-Time Pointer-Machine Algorithms for Least Common Ancestors, MST Verification, and Dominators. ACM TOPLAS 20(6), 1265–1296 (1998)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge (2001)
4. Een, N., Mishchenko, A., Sörensson, N.: Applying Logic Synthesis for Speeding Up SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 272–286. Springer, Heidelberg (2007)
5. Een, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 333–336. Springer, Heidelberg (2004)
6. Fortune, S., Hopcroft, J., Wyllie, J.: The Directed Subgraph Homeomorphism Problem. Theoretical Computer Science 10(2), 111–121 (1980)
7. Giffhorn, D., Hammer, C.: Precise Analysis of Java Programs using JOANA (Tool Demonstration). In: 8th IEEE Int'l. Working Conference on Source Code Analysis and Manipulation, pp. 267–268 (2008)
8. Hammer, C.: Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs. PhD thesis, Universität Karlsruhe (TH), Fak. f. Informatik (July 2009) ISBN 978-3-86644-398-3
9. Havlak, P.: Nesting of Reducible and Irreducible Loops. ACM Trans. Program. Lang. Syst. 19(4), 557–567 (1997)
10. Hermann, M.: Constrained Reachability is NP-complete (March 1998) (manuscript)
11. Larsen, L., Harrold, M.J.: Slicing Object-Oriented Software. In: ICSE 1996: Proc. of the 18th Int'l. Conf. on Software Engineering, pp. 495–505. IEEE, Los Alamitos (1996)
12. Lengauer, T., Tarjan, R.E.: A Fast Algorithm for Finding Dominators in a Flowgraph. ACM TOPLAS 1(1), 121–141 (1979)
13. Ramalingam, G.: On Loops, Dominators, and Dominance Frontier. In: PLDI 2000: Proc. of the ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation (2000)
14. Robschink, T.: Pfadbedingungen in Abhängigkeitgraphen und ihre Anwendung in der Softwaresicherheitstechnik. PhD thesis, Universität Passau (2004)

15. Santoro, N., Khatib, R.: Labelling and Implicit Routing in Networks. The Computer Journal 28(1), 5–8 (1985)
16. Snelting, G., Robschink, T., Krinke, J.: Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. ACM TOSEM 14(4), 410–457 (2006)
17. Sreedhar, V.C., Gao, G.R., Lee, Y.-F.: Identifying Loops Using DJ Graphs. ACM Transactions on Programming Languages and Systems 18(6), 649–658 (1996)
18. Steensgaard, B.: Sequentializing Program Dependence Graphs for Irreducible Programs. Technical report, Microsoft Research, Redmond (1993)
19. Tarjan, R.E.: Testing flow graph reducibility. J. Comput. Sci. 9, 355–365 (1974)

Robust and Efficient Delaunay Triangulations of Points on Or Close to a Sphere[⋆]

Manuel Caroli[1], Pedro M.M. de Castro[1], Sébastien Loriot[1], Olivier Rouiller[1], Monique Teillaud[1], and Camille Wormser[2]

[1] INRIA Sophia Antipolis – Méditerranée, France
{Manuel.Caroli,Pedro.Machado,Monique.Teillaud}@sophia.inria.fr
[2] ETH Zürich, Switzerland
Camille.Wormser@inf.ethz.ch

Abstract. We propose two ways to compute the Delaunay triangulation of points on a sphere, or of *rounded* points close to a sphere, both based on the classic incremental algorithm initially designed for the plane. We use the so-called space of circles as mathematical background for this work. We present a fully robust implementation built upon existing generic algorithms provided by the CGAL library. The efficiency of the implementation is established by benchmarks.

1 Introduction

The CGAL project [3] provides users with a public discussion mailing list, where they are invited to post questions and express their needs. There are recurring requests for a package computing the Delaunay triangulation of points on a sphere or its dual, the Voronoi diagram. This is useful in many domains such as geology, geographic information systems, information visualization, or structural molecular biology, to name a few. An easy and standard solution to the problem of computing such a Delaunay triangulation consists in constructing the 3D convex hull of the points: They are equivalent [13,38]. The convex hull is one of the most popular structures in computational geometry [20,11]; optimal algorithms and efficient implementations are available [1,2].

Another fruitful way to compute Delaunay on a sphere consists of reworking known algorithms designed for computing triangulations in $\mathbb{R}^2$. Renka adapts the distance in the plane to a geodesic distance on a sphere and triangulates points on a sphere [37] through the well-known flipping algorithm for Delaunay triangulations in $\mathbb{R}^2$ [30]. As a by-product of their algorithm for arrangements of circular arcs, Fogel et al. can compute Voronoi diagrams of points lying exactly on the sphere [26]. Using two inversions allows Na et al. to reduce the computation of a Voronoi diagram of sites on a sphere to computing two Voronoi diagrams

⋆ This work was partially supported by the ANR *(Agence Nationale de la Recherche)* under the "Triangles" project of the *Programme blanc* ANR-07-BLAN-0319 http://www.inria.fr/geometrica/collaborations/triangles/

in $\mathbb{R}^2$ [33], but no implementation is available. Note that this method assumes that data points are lying exactly on a sphere.

As we are motivated by applications, we take practical issues into account carefully. While data points lying exactly on the sphere can be provided either by using Cartesian coordinates represented by a number type capable of handling algebraic numbers exactly, or by using spherical coordinates, in practice data-sets in Cartesian coordinates with double precision are most common. In this setting, the data consists of rounded points that do not exactly lie on the sphere, but close to it.

In Section 4, we propose two different ways to handle such rounded data. Both approaches adapt the well-known incremental algorithm [12] to the case of points on, or close to the sphere. It is important to notice that, even though data points are rounded, we follow the exact geometric computation paradigm pioneered by C. K. Yap [39]. Indeed, it is now well understood that simply relying on floating point arithmetic for algorithms of this type is bound to fail (see [29] for instance).

The first approach (Section 4.1) considers as input the projections of the rounded-data points onto the sphere. Their coordinates are algebraic numbers of degree two. The approach computes the Delaunay triangulation of these points exactly lying on the sphere.

The second approach (Section 4.2) considers circles on the sphere as input. The radius of a circle (which can alternatively be seen as a *weighted* point) depends on the distance of the corresponding point to the sphere. The approach computes the weighted Delaunay triangulation of these circles on the sphere, also known as the *regular* triangulation, which is the dual of the Laguerre Voronoi diagram on the sphere [38] and the convex hull of the rounded-data points.

These interpretations of rounded data presented in this work are supported by the space of circles [10,24] (Section 3).

We implemented both approaches, taking advantage of the genericity of CGAL. In Section 5, we present experimental results on very large data-sets, showing the efficiency of our approaches. We compare our code to software designed for computing Delaunay triangulations on the sphere, and to convex-hull software [28,35,1,6,2,37,25]. The performance, robustness, and scalability of our approaches express their added value.

2 Definitions and Notation

Let us first recall the definition of the *regular triangulation* in $\mathbb{R}^2$, also known as *weighted Delaunay triangulation*. A circle c with center $p \in \mathbb{R}^2$ and squared radius r^2 is considered equivalently as a *weighted point* and is denoted by $c = (p, r^2)$. The *power product* of $c = (p, r^2)$ and $c' = (p', r'^2)$ is defined as $pow(c, c') = \|pp'\|^2 - r^2 - r'^2$, where $\|pp'\|$ denotes the Euclidean distance between p and p'. Circles c and c' are orthogonal iff $pow(c, c') = 0$. If $pow(c, c') > 0$ (i.e., the disks

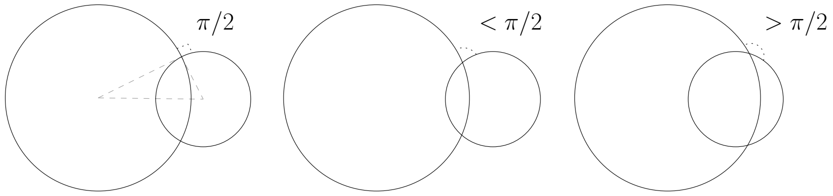


Fig. 1. From left to right: orthogonal ($pow(s_0, s_1) = 0$), suborthogonal ($pow(s_0, s_1) > 0$), and superorthogonal ($pow(s_0, s_1) < 0$) circles in $\mathbb{R}^2$

defined by c and c' do not intersect, or the circles intersect with an angle strictly smaller than $\frac{\pi}{2}$), we say that c and c' are *suborthogonal*. If $pow(c, c') < 0$, then we say that c and c' are *superorthogonal* (see Figure 1). Three circles whose centers are not collinear have a unique common orthogonal circle.

Let $\mathcal{S}$ be a set of circles. Given three circles of $\mathcal{S}$, $c_i = (p_i, r_i^2)$, $i = 1 \ldots 3$, whose centers are not collinear, let T be the triangle whose vertices are the three centers p_1, p_2, and p_3. We define the *orthogonal circle* of T as the circle that is orthogonal to the three circles c_1, c_2, and c_3. T is said to be *regular* if for any circle $c \in \mathcal{S}$, the orthogonal circle of T and c are not superorthogonal. A *regular triangulation* $\mathcal{RT}(\mathcal{S})$ is a partition of the convex hull of the centers of the circles of $\mathcal{S}$ into regular triangles formed by these centers. See Figure 2 for an example. The dual of the regular triangulation is known as the *power diagram*, *weighted Voronoi diagram*, or *Laguerre diagram*.

If all radii are equal, then the power test reduces to testing whether a point lies inside, outside, or on the circle passing through three points; the regular triangulation of the circles is the Delaunay triangulation $\mathcal{DT}$ of their centers.

More background can be found in [8]. We refer the reader to standard textbooks for algorithms computing Delaunay and regular triangulations [20,11].

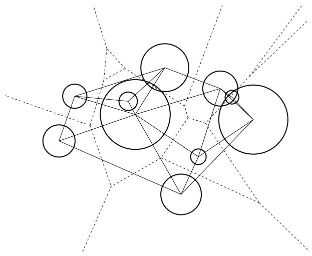


Fig. 2. Regular triangulation of a set of circles in the plane (their power diagram is shown dashed)

This definition generalizes in a natural manner to the case of circles lying on a sphere $\mathbb{S}$ in $\mathbb{R}^3$: Angles between circles are measured on the sphere, triangles are drawn on the sphere, their edges being arcs of great circles. As can be seen in the next section, the space of circles provides a geometric presentation showing without any computation that the regular triangulation on $\mathbb{S}$ is a convex hull in $\mathbb{R}^3$ [38].

In the sequel, we assume that $\mathbb{S}$ is given by its center, having rational coordinates (we take the origin O without loss of generality), and a rational squared radius R^2. This is also how spheres are represented in CGAL.[1]

[1] We mention rational numbers to simplify the presentation. CGAL allows more general number types that provide field operations: $+, -, \times, /$.

3 Space of Circles

Computational geometers are familiar with the classic idea of lifting up sites from the Euclidean plane onto the unit paraboloid Π in $\mathbb{R}^3$ [9]. We quickly recall the notion of *space of circles* here and refer to the literature for a more detailed presentation [24]. In this lifting, points of $\mathbb{R}^3$ are viewed as circles of $\mathbb{R}^2$ in the space of circles: A circle $c = (p, r^2)$ in $\mathbb{R}^2$ is mapped by π to the point $\pi(c) = (x_p, y_p, x_p^2 + y_p^2 - r^2) \in \mathbb{R}^3$. A point of $\mathbb{R}^3$ lying respectively outside, inside, or on the paraboloid Π represents a circle with respectively positive, imaginary, or null radius. The circle c in $\mathbb{R}^2$ corresponding to a point $\pi(c)$ of $\mathbb{R}^3$ outside Π is obtained as the projection onto $\mathbb{R}^2$ of the intersection between Π and the cone formed by lines through $\pi(c)$ that are tangent to Π; this intersection is also the intersection of the polar plane $P(c)$ of $\pi(c)$ with respect to the quadric Π.

Points lying respectively on, above, below $P(c)$ correspond to circles in $\mathbb{R}^2$ that are respectively orthogonal, suborthogonal, superorthogonal to c. The predicate $pow(c, c')$ introduced above is thus equivalent to the orientation predicate in $\mathbb{R}^3$ that tests whether the point $\pi(c')$ lies on, above or below the plane $P(c)$. If c is the common orthogonal circle to three input circles c_1, c_2, and c_3 (where $c_i = (p_i, r_i^2)$ for each i), then $pow(c, c')$ is the orientation predicate of the four points $\pi(c_1), \pi(c_2), \pi(c_3), \pi(c')$ of $\mathbb{R}^3$. It can be expressed as

$$\operatorname{sign} \begin{vmatrix} 1 & 1 & 1 & 1 \\ x_{p_1} & x_{p_2} & x_{p_3} & x_{p'} \\ y_{p_1} & y_{p_2} & y_{p_3} & y_{p'} \\ z_{p_1} & z_{p_2} & z_{p_3} & z_{p'} \end{vmatrix}, \tag{1}$$

where $z_{p_i} = x_{p_i}^2 + y_{p_i}^2 - r_i^2$ for each i and $z_{p'}^2 = x_{p'}^2 + y_{p'}^2 - r'^2$. It allows to relate Delaunay or regular triangulations in $\mathbb{R}^2$ and convex hulls in $\mathbb{R}^3$ [9], while Voronoi diagrams in $\mathbb{R}^2$ are related to upper envelopes of planes in $\mathbb{R}^3$.

Up to a projective transformation, a sphere in $\mathbb{R}^3$ can be used for the lifting instead of the usual paraboloid [10]. In this representation the sphere has a pole[2] and can be identified to the Euclidean plane $\mathbb{R}^2$. What we are interested in this paper is the space of circles drawn on the sphere $\mathbb{S}$ itself, without any pole. This space of circles has a nice relation to the de Sitter space in Minkowskian geometry [19].

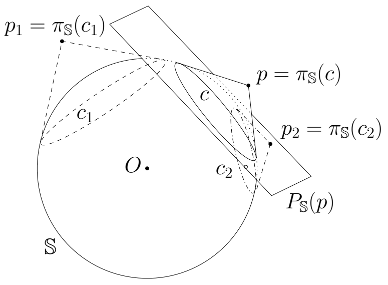

$p_1 = \pi_{\mathbb{S}}(c_1)$
$p = \pi_{\mathbb{S}}(c)$
c
c_1
$p_2 = \pi_{\mathbb{S}}(c_2)$
O
c_2
$P_{\mathbb{S}}(p)$
$\mathbb{S}$

Fig. 3. c_1 is suborthogonal to c, c_2 is superorthogonal to c

We can still construct the circle c on $\mathbb{S}$ that is associated to a point $p = \pi_{\mathbb{S}}(c)$ of $\mathbb{R}^3$ as the intersection between $\mathbb{S}$ and the polar plane $P_{\mathbb{S}}(p)$ of p with respect to the quadric $\mathbb{S}$ (Figure 3). Its center is the projection of p onto $\mathbb{S}$ and as above, imaginary radii are possible.[3] So, in the determinant in (1), x_{p_i}, y_{p_i}, and z_{p_i} (respectively $x_{p'}, y_{p'}, z_{p'}$) are precisely the

[2] See the nice treatment of infinity in [10].
[3] Remember that $\mathbb{S}$ is centered at O and has squared radius R^2.

coordinates of the points $p_i = \pi_{\mathbb{S}}(c_i)$ (respectively $p' = \pi_{\mathbb{S}}(p)$). This will be extensively used in Section 4. Again, we remark that Delaunay and regular triangulations on $\mathbb{S}$ relate to convex hulls in 3D.

Interestingly, rather than using a convex hull algorithm to obtain the Delaunay or regular triangulation on the surface as usually done for $\mathbb{R}^2$ [9], we will do the converse in the next section.

4 Algorithm

The incremental algorithm for computing a regular triangulation of circles on the sphere $\mathbb{S}$ is a direct adaptation of the algorithm in $\mathbb{R}^2$ [12]. Assume that $\mathcal{RT}_{i-1} = \mathcal{RT}(\{c_j \in \mathcal{S}, j = 1, \ldots, i-1\})$ has been computed.[4] The insertion of $c_i = (p_i, r_i^2)$ works as follows:

• locate p_i (i.e., find the triangle t containing p_i),
• `if` t is hiding p_i (i.e., if c_i and the orthogonal circle of t are suborthogonal) then `stop`; p_i is not a vertex of $\mathcal{RT}_i$. Note that this case never occurs for Delaunay triangulations.
• `else` (i) find all triangles whose orthogonal circles are superorthogonal to c_i and remove them; this forms a polygonal region that is star-shaped with respect to p_i;[5] (ii) triangulate the polygonal region just created by constructing the triangles formed by the boundary edges of the region and the point p_i.

Two main predicates are used by this algorithm:

The *orientation* predicate allows to check the orientation of three points p, q, and r on the sphere. (This predicate is used in particular to locate new points.) It is equivalent to computing the side of the plane defined by O, p, and q on which r is lying, i.e., the orientation of O, p, q, and r in $\mathbb{R}^3$.
The *power test* introduced in Section 2 boils down to an orientation predicate in $\mathbb{R}^3$, as seen in Section 3. (This predicate is used to identify the triangles whose orthogonal circles are superorthogonal to each new circle.)

The two approaches briefly presented in the introduction fall into the general framework of computing the regular triangulation of circles on the sphere. The next two sections precisely show how these predicates are evaluated in each approach.

4.1 First Approach: Using Points on the Sphere

In this approach, input points for the computation are chosen to be the projections on $\mathbb{S}$ of the rounded points of the data-set with rational coordinates. The

[4] For the sake of simplicity, we assume that the center O of $\mathbb{S}$ lies in the convex hull of the data-set. This is likely to be the case in practical applications. So, we just initialize the triangulation with four dummy points that contain O in their convex hull and can optionally be removed in the end.

[5] As previously noted for the edges of triangles, all usual terms referring to segments are transposed to arcs of great circles on the sphere.

three coordinates of an input point are thus algebraic numbers of degree two lying in the same extension field of the rationals.

In this approach weights, or equivalently radii if circles, are null. The power test consists in this case in answering whether a point s lies inside, outside,[6] or on the circle passing through p, q, and r on the sphere. Following Section 3, this is given by the orientation of p, q, r, and s, since points on the sphere are mapped to themselves by $\pi_{\mathbb{S}}$.

The difficulty comes from the fact that input points have algebraic coordinates. The coordinates of two different input points on the sphere are in general lying in different extensions. Then the 3D orientation predicate of p, q, r, and s given by (1) is the sign of an expression lying in an algebraic extension of degree 16 over the rationals, of the form $a_1\sqrt{\alpha_1} + a_2\sqrt{\alpha_2} + a_3\sqrt{\alpha_3} + a_4\sqrt{\alpha_4}$ where all a's and α's are rational. Evaluating this sign in an exact way allows to follow the exact computation framework ensuring the robustness of the algorithm.

Though software packages offer exact operations on general algebraic numbers [4,5], they are much slower than computing with rational numbers. The sign of the above simple expression can be computed as follows:

–1– evaluate the signs of $A_1 = a_1\sqrt{\alpha_1} + a_2\sqrt{\alpha_2}$ and $A_2 = a_3\sqrt{\alpha_3} + a_4\sqrt{\alpha_4}$, by comparing $a_i\sqrt{\alpha_i}$ with $a_{i+1}\sqrt{\alpha_{i+1}}$ for $i = 1, 3$, which reduces after squaring to comparing two rational numbers,
–2– the result follows if A_1 and A_2 have different signs,
–3– otherwise, compare A_1^2 with A_2^2, which is an easier instance of –1–.

To summarize, the predicate is given by the sign of polynomial expressions in the rational coordinates of the rounded-data points, which can be computed exactly using rational numbers only.

4.2 Second Approach: Using Weighted Points

In this approach, the regular triangulation of the weighted points is computed as described above. As in the previous approach, both predicates (orientation on the sphere and power test) reduce to orientation predicates on the data points in $\mathbb{R}^3$. Note that Section 3 shows that the weight of a point p is implicit, as it does not need to be explicitly computed throughout the entire algorithm.

Depending on the weights, some points can be hidden in a regular triangulation. We prove now that under some sampling conditions on the rounded data, there is actually no hidden point.

Lemma 1. *Let us assume that all data points lie within a distance δ from $\mathbb{S}$. If the distance between any two points is larger than $2\sqrt{R\delta}$, then no point is hidden.*

Proof. A point is hidden iff it is contained inside the 3D convex hull of the set of data points $\mathcal{S}$. Let p be a data point, at distance ρ from O. We have

[6] On $\mathbb{S}$, the interior (respectively exterior) of a circle c that is not a great circle of $\mathbb{S}$ corresponds to the interior (respectively exterior) of the half-cone in 3D, whose apex is the center of $\mathbb{S}$ and that intersects $\mathbb{S}$ along c.

$\rho \in [R-\delta, R+\delta]$. Denote by d_p the minimum distance between p and the other points. If $d_p > \sqrt{(R+\delta)^2 - \rho^2}$, the set $B(O, R+\delta) \setminus B(p, d_p)$ is included in the half-space $H^+ = \{q : \langle q-p, O-p\rangle > 0\}$. Under these conditions, all other points belong to H^+ and p is not inside the convex hull of the other points. It follows that if the distance between any two data points is larger than $\sup_\rho \sqrt{(R+\delta)^2 - \rho^2} = 2\sqrt{R\delta}$, no point is hidden.

Let us now assume we use double precision floating point numbers as defined in the IEEE standard 754 [7,27]. The mantissa is encoded using 52 bits. Let γ denote the worst error, for each Cartesian coordinate, done while rounding a point on $\mathbb{S}$ to the nearest point whose coordinates can be represented by double precision floating point numbers. Let us use the standard term ulp(x) denoting the gap between the two floating-point numbers closest to the real value x [32]. Assuming again that the center of $\mathbb{S}$ is O, one has $\gamma \leq \text{ulp}(R) = 2^{-52+\lfloor\log_2(R)\rfloor} \leq 2^{-52}R$. Then, δ in the previous lemma is such that $\delta \leq \sqrt{3/4}\gamma < 2^{-52}R$. Using the result of the lemma, no point is hidden in the regular triangulation as soon as the distance between any two points is greater than $2^{-25}R$, which is highly probable in practice.

Note that this approach can be used as well to compute the convex hull of points that are not close to a sphere: The center of the sphere can be chosen at any point inside a tetrahedron formed by any four non-coplanar data points.

5 Implementation and Experiments

Both approaches presented in Section 4 have been implemented in C++, based on the CGAL package that computes triangulations in $\mathbb{R}^2$. The package introduces an infinite vertex in the triangulation to compactify $\mathbb{R}^2$. Thus the underlying combinatorial triangulation is a triangulation of the topological sphere. This allows us to reuse the whole combinatorial part of CGAL 2D triangulations [36] without any modification. However, the geometric embedding itself [40], bound to $\mathbb{R}^2$, must be modified by removing any reference to the infinite vertex. A similar work was done to compute triangulations in the 3D flat torus [16,15], reusing the CGAL 3D triangulation package [34,35] as much as possible.

Also, the genericity offered in CGAL by the mechanism of traits classes, that encapsulate the geometric predicates needed by the algorithms, allows us to easily use exactly the same algorithm with two different traits classes for our two approaches.

To display the triangulation and its dual, the code is interfaced with the CGAL 3D spherical kernel [21,22], which provides primitives on circular arcs in 3D. The vertices of the triangulations shown are the projections on the sphere of the rounded-data points. The circular arcs are drawn on the surface of the sphere (see Figures 5 and 6).

We compare the running time of our approaches with several available software packages on a MacBook Pro 3,1 equipped with a 2.6 GHz Intel Core 2

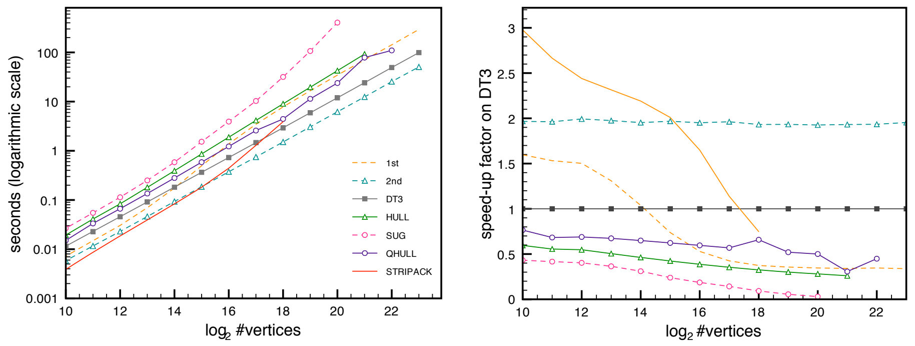


Fig. 4. Comparative benchmarks. The programs were aborted when their running time was above 10 minutes (HULL, SUG, QHULL) or in case of failure (STRIPACK)

processor and 2GB 667 MHz DDR2 SDRAM[7] (see Figure 4). We consider large sets of random data points[8] (up to 2^{23} points) on the sphere, rounded to double coordinates. Figure 5 indicates running times on some real-life data.

Graph 1st of Figure 4 shows the results of our first approach. We coded a traits class implementing the exact predicates presented in Section 4.1, together with semi-static and dynamic filtering [31]. The non-linear behavior of the running time is due to the fact that our semi-static filters hardly ever fail for less than 2^{13} points, and almost always fail for more than 2^{18} points.

Graph 2nd shows the results of the second approach. One of the predefined kernels[9] of CGAL provides us directly with an exact implementation of the predicates, filtered both semi-statically and dynamically. In our experiments we have observed that no point is hidden with such distributions, even when the data-set is large, which confirms in practice the discussion of Section 4.2.

The CGAL 3D Delaunay triangulation (graph DT3) [35], with the same CGAL kernel, also provides this convex hull as a by-product. We insert the center of the sphere to avoid penalizing this code with too many predicate calls on five cospherical points that would always cause filters to fail.

For these three approaches, 3D spatial sorting reduces the running time of the location step of point insertion [23,14].

If the data points are lying exactly on a sphere, their Delaunay Triangulation can be extracted from an arrangement of geodesic arcs as computed by the code of Fogel and Setter [25,26]. Since it is not the main purpose of their algorithm, the running times are not comparable: close to 600 seconds for 2^{12} points. Note however that the code is preliminary and has not been fully optimized yet. No graph is shown.

---

7 Further details: MAC OS X version 10.5.7, 64 bits; compiler `g++ 4.3.2` with -O3 and -DNDEBUG flags, `g77 3.4.3` with -O3 for Fortran. All running times mentioned exclude time used by input/output.

8 Generated by `CGAL::Random_points_on_sphere_3`

9 Precisely `CGAL::Exact_predicates_inexact_constructions_kernel`

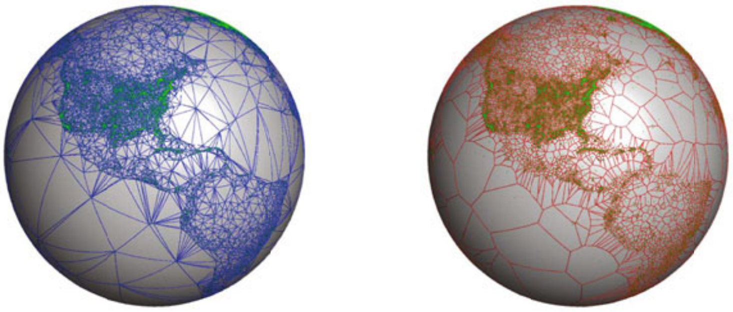


Fig. 5. Delaunay triangulation (left) and Voronoi diagram (right) of 20,950 weather stations all around the world. Data and more information can be found at `http://www.locationidentifiers.org/`. Our second approach computes the result in 0.14 seconds, while Qhull needs 0.35 seconds, and the first approach 0.57 seconds. STRIPACK fails on this data-set.

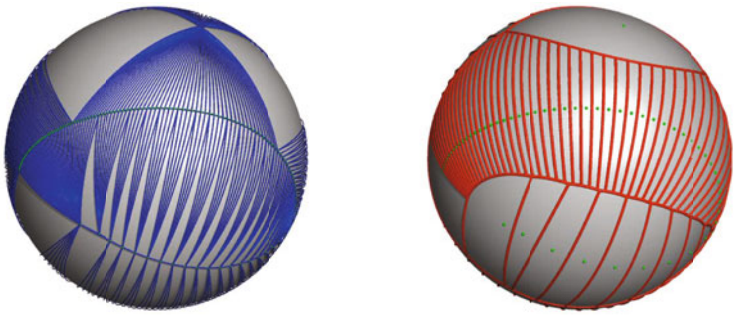


Fig. 6. Delaunay triangulation of $\mathcal{S}_{250}$ (left), Voronoi diagram of $\mathcal{S}_{100}$ (right). STRIPACK fails for e.g. $n = 1,500$.

We consider the following two software packages computing a convex hull in 3D,[10] for which the data points are first rounded to points with integer coordinates. Predicates are evaluated exactly using single precision computations.

Graph HULL corresponds to the code [1] of Clarkson, who uses a randomized incremental construction [18] with an exact arithmetic on integers [17].

Graph SUG shows the running times of Sugihara's code in Fortran [6,38].

Graph QHULL shows the performance of the famous Qhull package of Barber et al. [2] when computing the 3D convex hull of the points. The option we use handles round-off errors from floating point arithmetic by merging facets of the convex hull when necessary. The convex hull of points situated close to the sphere contains in practice all the input points (see Lemma 1). In this situation QHULL is clearly outperformed by the second approach. However, QHULL can

[10] The plot does not show the results of the CGAL 3D convex hull package [28] because it is much slower than all other methods (roughly 500 times slower than QHULL).

be about twice faster than our second approach when almost all the input points are hidden.

Renka computes the triangulation with an algorithm similar to our first approach, but his software STRIPACK, in Fortran, uses approximate computations in double [37]. Consequently, it performs quite well on random points (better than our implementations for small random data-sets), but it fails on some data-sets: Using STRIPACK, we did not manage to compute a triangulation of more than 2^{19} random points (it returns an error flag). The same occurred for the inputs used to produce Figures 5 and 6. Our implementations handle arbitrary data sets.

To test for exactness we devised a point set that is especially hard to triangulate because it yields many very flat triangles in the triangulation. This point set is defined as

$$\mathcal{S}_n = \left\{ \begin{pmatrix} \cos\theta \sin\phi \\ \sin\theta \sin\phi \\ \cos\phi \end{pmatrix} \middle| \theta \in \left\{0, \tfrac{\pi}{n}, \ldots, \tfrac{(n-1)\pi}{n}, \pi\right\}, \phi = \tfrac{(\theta^2+1)}{\pi^2} \right\} \cup \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, -\tfrac{1}{\sqrt{3}} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \right\}$$

In Figure 6 we show the Delaunay triangulation of $\mathcal{S}_{250}$ and the Voronoi diagram of $\mathcal{S}_{100}$.

In Table 1, we compare the memory usage of our two approaches, the 3D Delaunay triangulation, and Qhull.[11] The given figures given in bytes per processed vertex (bppv) and averaged over several data-sets of size larger than 2^{16}.

Table 1. Memory usage

approach	bppv
1st	113
2nd	113
DT3	174
QHULL	288

6 Conclusion

The results show that our second approach yields better timings and memory usage than all the other tested software packages for large data-sets, while being fully robust. This justifies a typical phenomenon: the well-designed specialized solution outperforms the more general one. Here the specialized one is our second approach, and the general one is the Delaunay triangulation 3D computation from which the 3D convex hull is extracted.

The first approach is slower but still one of the most scalable. It exactly computes the triangulation for input points with algebraic coordinates lying on the sphere, and thus ensures that in any case all points will appear in the triangulation. It is the only one to do so within reasonable time and thus being useful for real-world applications.

Acknowledgments

We warmly acknowledge Ophir Setter and Efi Fogel who kindly worked on their code to give us access to it. We wish to thank Jean-Marc Schlenker for very interesting discussions on the de Sitter space.

[11] Memory usage measurements are done with `CGAL::Memory_sizer` for the first approach, second approach and for the 3D Delaunay triangulation. For the Qhull package the measurement is done with the -Ts option, taking into account the memory allocated for facets and their normals, neighbor and vertex sets.

References

1. Hull, a program for convex hulls, http://www.netlib.org/voronoi/hull.html
2. Qhull, http://www.qhull.org/
3. Cgal, Computational Geometry Algorithms Library, http://www.cgal.org
4. Core number library, http://cs.nyu.edu/exact/core_pages
5. Leda, Library for efficient data types and algorithms, http://www.algorithmic-solutions.com/enleda.htm
6. Three-dimensional convex hulls, http://www.simplex.t.u-tokyo.ac.jp/~sugihara/opensoft/opensofte.html
7. IEEE standard for floating-point arithmetic. IEEE Std 754-2008, pp. 1–58 (August 2008)
8. Aurenhammer, F.: Power diagrams: properties, algorithms and applications. SIAM Journal of Computing 16, 78–96 (1987)
9. Aurenhammer, F.: Voronoi diagrams: A survey of a fundamental geometric data structure. ACM Computing Surveys 23(3), 345–405 (1991)
10. Berger, M.: The space of spheres. In: Geometry, vol. 1-2, pp. 349–361. Springer, Heidelberg (1987)
11. Boissonnat, J.D., Yvinec, M.: Algorithmic Geometry. Cambridge University Press, UK (1998); Translated by Hervé Brönnimann
12. Bowyer, A.: Computing Dirichlet tessellations. The Computer Journal 24(2), 162–166 (1981)
13. Brown, K.Q.: Geometric transforms for fast geometric algorithms. Ph.D. thesis, Dept. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Report CMU-CS-80-101 (1980)
14. Buchin, K.: Constructing Delaunay triangulations along space-filling curves. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 119–130. Springer, Heidelberg (2009)
15. Caroli, M., Teillaud, M.: 3D periodic triangulations. In: CGAL Editorial Board (ed.) CGAL User and Reference Manual, 3.5 edn. (2009)
16. Caroli, M., Teillaud, M.: Computing 3D periodic triangulations. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 37–48. Springer, Heidelberg (2009); Full version available as INRIA Reserch Report No 6823, http://hal.inria.fr/inria-00356871
17. Clarkson, K.L.: Safe and effective determinant evaluation. In: Proceedings 33rd Annual IEEE Symposium on Foundations of Computer Science, October 1992, pp. 387–395 (1992)
18. Clarkson, K.L., Mehlhorn, K., Seidel, R.: Four results on randomized incremental constructions. Computational Geometry: Theory and Applications 3(4), 185–212 (1993)
19. Coxeter, H.S.M.: A geometrical background for de Sitter's world. American Mathematical Monthly 50, 217–228 (1943)
20. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications, 2nd edn. Springer, Berlin (2000)
21. de Castro, P.M.M., Cazals, F., Loriot, S., Teillaud, M.: 3D spherical geometry kernel. In: CGAL User and Reference Manual. CGAL Editorial Board, 3.5 edn. (2009)
22. de Castro, P.M.M., Cazals, F., Loriot, S., Teillaud, M.: Design of the CGAL 3D Spherical Kernel and application to arrangements of circles on a sphere. Computational Geometry: Theory and Applications 42(6-7), 536–550 (2009)

23. Delage, C.: Spatial sorting. In: CGAL Editorial Board (ed.) CGAL User and Reference Manual, 3.5 edn. (2009)
24. Devillers, O., Meiser, S., Teillaud, M.: The space of spheres, a geometric tool to unify duality results on Voronoi diagrams. In: Proceedings 4th Canadian Conference on Computational Geometry, pp. 263–268 (1992); Full version available as INRIA Research Report No 1620, http://hal.inria.fr/inria-00074941
25. Fogel, E., Setter, O.: Software for Voronoi diagram on a sphere. Personal communication
26. Fogel, E., Setter, O., Halperin, D.: Exact implementation of arrangements of geodesic arcs on the sphere with applications. In: Abstracts of 24th European Workshop on Computational Geometry, pp. 83–86 (2008)
27. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys 23(1), 5–48 (1991)
28. Hert, S., Schirra, S.: 3D convex hulls. In: CGAL Editorial Board (ed.) CGAL User and Reference Manual, 3.5 edn. (2009)
29. Kettner, L., Mehlhorn, K., Pion, S., Schirra, S., Yap, C.: Classroom examples of robustness problems in geometric computations. Computational Geometry: Theory and Applications 40, 61–78 (2008)
30. Lawson, C.L.: Software for C^1 surface interpolation. In: Rice, J.R. (ed.) Math. Software III, pp. 161–194. Academic Press, New York (1977)
31. Li, C., Pion, S., Yap, C.K.: Recent progress in exact geometric computation. Journal of Logic and Algebraic Programming 64(1), 85–111 (2005)
32. Muller, J.M.: On the definition of ulp(x). Research Report 5504, INRIA (February 2005), http://hal.inria.fr/inria-00070503/
33. Na, H.S., Lee, C.N., Cheong, O.: Voronoi diagrams on the sphere. Computational Geometry: Theory and Applications 23, 183–194 (2002)
34. Pion, S., Teillaud, M.: 3D triangulation data structure. In: CGAL Editorial Board (ed.) CGAL User and Reference Manual, 3.5 edn. (2009)
35. Pion, S., Teillaud, M.: 3D triangulations. In: CGAL Editorial Board (ed.) CGAL User and Reference Manual, 3.5 edn. (2009)
36. Pion, S., Yvinec, M.: 2D triangulation data structure. In: CGAL Editorial Board (ed.) CGAL User and Reference Manual, 3.5 edn. (2009)
37. Renka, R.J.: Algorithm 772: STRIPACK: Delaunay triangulation and Voronoi diagram on the surface of a sphere. ACM Transactions on Mathematical Software 23(3), 416–434 (1997), Software available at http://orion.math.iastate.edu/burkardt/f_src/stripack/stripack.html
38. Sugihara, K.: Laguerre Voronoi diagram on the sphere. Journal for Geometry and Graphics 6(1), 69–81 (2002)
39. Yap, C.K., Dubé, T.: The exact computation paradigm. In: Du, D.-Z., Hwang, F.K. (eds.) Computing in Euclidean Geometry, 2nd edn. Lecture Notes Series on Computing, vol. 4, pp. 452–492. World Scientific, Singapore (1995)
40. Yvinec, M.: 2D triangulations. In: CGAL Editorial Board (ed.) CGAL User and Reference Manual, 3.5 edn. (2009)

Fault Recovery in Wireless Networks: The Geometric Recolouring Approach

Henk Meijer[1], Yurai Núñez-Rodríguez[2], and David Rappaport[2,⋆]

[1] Roosevelt Academy, Middelburg, The Netherlands
h.meijer@roac.nl
[2] Queen's University, Kingston, ON Canada
{yurai,daver}@cs.queensu.ca

Abstract. Duplication of information allows distributed systems to recover from data errors, or faults. If faults occur spontaneously, without notification, and disguised incorrect data blends in with correct data, their detection becomes non-trivial. Known solutions for fault recovery use monitoring mechanisms that compare the data in multiple nodes to infer the occurrence of faults. To this end, we propose a localized geometric approach to fault recovery in wireless networks. We compare our approach with a more traditional combinatorial approach that uses a majority rule. Our experiments show that our geometric approach is an improvement over the majority rule in some cases, whereas in the other cases a hybrid method that combines the best of both strategies is superior to each individual method.

Keywords: fault recovery, recolouring, wireless networks, computational geometry, experimental algorithms, localized algorithms.

1 Introduction

We consider fault recovery in networks consisting of disambiguating two-state variables at the nodes. This can be achieved by using information duplicates stored across the network. Our results easily generalize to multiple-state variables if applied independently to their constituent bits. We examine a geometric technique that provides autonomous detection and recovery of faults. We compare our technique with an existing non-geometric approach and demonstrate its effectiveness. The geometric method we propose may not be the only effective one; so, with this work we hope to open the discussion and make progress towards the best localized fault recovery strategy.

In what follows we refer to faulty and healthy nodes as red and blue, without defining which colour is assigned to which state. Colouring the nodes will simplify our presentation and will put our work in the same context as previous work. Thus, the change of status of a node (from faulty to healthy, or vice versa) is called a *recolouring*.

⋆ Supported by an NSERC of Canada Discovery Grant.

The rest of this document is organized as follows. Section 2 reviews previous work on fault recovery techniques for distributed systems and the origins of a technique we propose for fault recovery, namely, geometric recolouring. Section 3 presents the details of fault recovery using our geometric approach. In Section 4 we present our experimental results and demonstrate the suitability of geometric recolouring for fault recovery. Finally, we identify a set of open problems derived from our work.

2 Previous Work

We first survey previous work on fault recovery in distributed systems. Then we introduce the fundaments of a geometric recolouring technique as used for a related problem, the red-blue separation problem.

Several studies on fault recovery have focused on distributed systems with regular topology. For example, Floccini et al. [2] consider fault recovery on toroidal meshes and Luccio, Pagli, and Sanossian [5] study this problem on butterfly networks. Their work focuses on finding *monopolies*, that is, configurations of faulty nodes that can cause the entire network to fail. As will be seen later on, we are more concerned with the length of recolouring sequences. The underlying goal of both efforts remains to study systems that can autonomously recover from faults through localized algorithms.

It is often assumed that faults occur as a consequence of manufacturing defects or other random, spontaneous causes. This is the case considered by Krishnamachari and Iyengar [4]. To our knowledge, no previous work has considered the occurrence of failures in correlation with the geometric location of the nodes.

It is widely accepted that the data collected by the nodes of a network, in the case of sensor networks for instance, is correlated to their geographic location (see [11,1] for example); thus, there is no reason not to believe that errors induced on the network by the influence of the environment are also correlated to their spatial distribution. Notice also that considering faulty areas is a generalization to considering isolated errors. The latter can be seen as independent faults on areas that are small enough to contain a single node.

It is assumed that a node does not know whether it is faulty or not just by reading its data. It can however, compare its colour (state) with its neighbours' colours. All the previously cited approaches to fault recovery and other studies (see the survey by Peleg [8]) use a *majority voting* rule. The majority voting rule recolours a node if it has more neighbours of the opposite colour. This strategy can obviously turn healthy nodes to faulty as much as healing faulty ones. However, based on the fact that faulty nodes should be a "non-dominant minority", we expect that the cooperative effort makes progress towards healing the faulty nodes. On the other hand, we argue that the majority rule is not the best approach if applied to geometric graphs involving areas of faulty nodes. We propose a geometric recolouring approach to this end.

Geometric recolouring, as introduced by Reinbacher et al. [10], has been used for assisting geographic data classification. Given a planar set of red and blue points, their goal is to separate the red from the blue by polygonal curves with small perimeter. A reclassification method is performed as a preprocessing step to correct misclassified points, as the input is assumed to possibly contain errors. This reclassification technique is termed recolouring, which we call *geometric recolouring* for reasons that will become evident in what follows. Their experiments show that the use of recolouring yields separating curves with smaller perimeters.

Reinbacher et al. use a Delaunay triangulation of the point set as the underlying structure for recolouring, specifying a neighbour relation among the points. They then define a point, p, as *surrounded* when there is a contiguous set of oppositely coloured neighbours of p, in the triangulation, that span a radial angle greater than 180° (see Figure 1). Points that are surrounded are iteratively recoloured, in no particular order, until no point remains surrounded. This strategy raises an interesting question concerning the finiteness of recolouring sequences. It turns out that if one chooses the threshold angle to consider a point as surrounded as any value smaller than 180°, there may exist trivial infinite recolouring sequences [10]. The problem is far less trivial for thresholds greater than 180°.

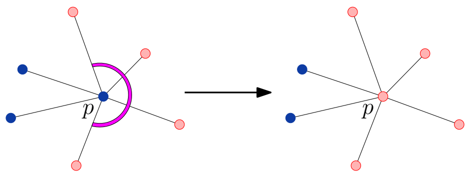


Fig. 1. Recolouring surrounded point p

Using properties of the triangulation, Reinbacher et al. show that these recolouring sequences are finite and guaranteed to terminate in at most $2^n - 1$ iterations. They also show an example triangulation that yields $O(n^2)$ recolourings. In a previous work [6], we proved that any triangulation can have at most $O(n^2)$ recolourings, closing the gap between the lower and the upper bound. Moreover, we extended our results to other geometric graphs and proved a set of bounds for the length of recolouring sequences, ranging from linear for trees to infinite for planar graphs. In the following section we use one of our polynomial bounds for a convenient construction, the NIC graph.

3 Fault Recovery in Wireless Networks

Before presenting our geometric approach to fault recovery we propose a general framework for recolouring in wireless networks. We review a more traditional combinatorial approach to recolouring. Then we present the challenges of our geometric approach, along with proposed solutions. Finally, we include a hybrid method that combines the combinatorial and geometric approaches.

A node can be considered as surrounded or dominated by neighbours of the opposite colour according to different criteria. For now, we assume that we are provided with a general function SURROUNDED that returns TRUE for a node surrounded by neighbours of the opposite colour, or FALSE otherwise. Different versions of SURROUNDED will be studied in what follows.

As pointed out earlier, a recolouring does not necessarily mean that a faulty node is being fixed; on the contrary, sometimes a healthy node can become faulty by the same mechanism. However, our experiments demonstrate that whenever the cause of the fault affects a relatively small area, the number of nodes that are recovered from faults is substantially larger than the number of nodes that turn faulty. In fact, in many cases all faulty nodes are able to recover (see Section 4 for more details).

The fault recovery algorithm simply consists of the *Recolouring Protocol*, as defined next, to be executed at all nodes. The protocol defines successive recolourings of a node, according to the function SURROUNDED, and a mechanism to notify its neighbours whenever a change of colour occurs.

Algorithm 1. Recolouring Algorithm

```
input : Network $G = (N, L)$ with bi-chromatic nodes.
output: Network $G = (N, L)$ with a different node colouring such that no more
        nodes can be recoloured.

Recolouring Protocol

Step 1. Broadcast a COLOUR message to all neighbours, with the node's colour
    information.
Step 2. If there is a COLOUR message in the node's queue, the new colour of the
    corresponding neighbour is considered for updating the node's colour according
    to the function SURROUNDED .
Step 2.1. If the node is recoloured, broadcast a message to all neighbours with the
    new colour information.
Step 3. If there is no COLOUR message in the queue and no COLOUR message
    has been received for $T$ time units, go to Step 1.
Step 4. Go to Step 2.
```

Step 3 of the recolouring protocol ensures that if a node becomes faulty, the neighbours are informed of its colour change. The parameter T can be adjusted for an optimal tradeoff between fast response to faults and low network traffic. The algorithm cycles idly (or terminates temporarily) once no more nodes can be recoloured. However, it restarts itself after T time units of inactivity, to recover from possible new faults.

Notice that the nodes do not necessarily wait until they know the colour of all neighbours. The protocol is presented in a way that no synchronization is required. In fact, asynchrony is one key aspect for the termination of the algorithm.

On the other hand, if rounds of recolourings occur synchronously for all surrounded nodes, the process may not terminate. Goles and Olivos [3] explain the behaviour of such systems and how they may fall into configurations that oscillate infinitely.

We have used the concepts of synchrony and rounds in an intuitive, informal way; the reader is referred to Peleg's book [9] for formal definitions. In the following we define other matters of time and synchrony that are relevant to our work. The *recolouring time* of a node p is the time it takes from the actual recolouring, as defined by Step 2 of the algorithm, to the realization by its neighbours that p has been recoloured. We define two recolourings to be *simultaneous* if the corresponding recolouring times overlap. The *sequential model* is defined as a hypothetical model in which no simultaneous recolourings occur, as if there was a global scheduler controlling the network's activity. Last, we define our *asynchronous model* to be one in which the recolourings of nodes occur independently from one another. Thus, simultaneous recolourings happen only as a matter of chance. Next we establish that asynchronous systems behave like the sequential model in the long run with high probability.

Lemma 1. *Let $G = (N, L)$ be a network in which nodes operate asynchronously. The probability that any two nodes $p, q \in N$ recolour simultaneously n times, tends to zero as n tends to infinity.*

Proof. As defined for our synchronous model, the chance that nodes p and q are recoloured simultaneously for the i-th time has the associated probability $P_i(p, q) < 1$. Without considering any particular probability distribution, we can bound $P_i(p, q)$ by $P(p, q)$, the highest probability of simultaneous recolourings of p and q over all colour configurations. Thus, $P_i(p, q) \leq P(p, q) < 1, \forall i$. We can conclude that the joint event consisting of an unbounded number of simultaneous recolourings of p and q has infinitesimal probability, as stated in the following.

$$\lim_{n\to\infty} \prod_{i=1}^{n} P_i(p, q) \leq \lim_{n\to\infty} \prod_{i=1}^{n} P(p, q) = 0$$

because $P(p, q) < 1$, which concludes our proof.

Note that the existence of $P(p, q) < 1$ is guaranteed by the perfect asynchrony assumption. In practical scenarios, if the physical network implementation may cause synchronized behaviour after certain colour configurations, random response times can be introduced to further reinforce the network asynchrony.

The previous lemma proves that long sequences of simultaneous recolourings are unlikely for two or more nodes. Thus, in the long run, a purely asynchronous network behaves (with high probability) like the sequential model. This implies that if the surrounded function of choice ensures finiteness for the recolouring process in the sequential model, it also produces finite recolouring (with high probability) for truly asynchronous environments. Therefore, in the sequel we limit our study of recolouring strategies to the sequential model.

The Surrounded function can be implemented based on simple combinatorial properties of a node and its neighbours. In this case a majority rule is considered. Thus, we define a function Surrounded_combinatorial that returns

TRUE for nodes with more neighbours of the opposite colour than neighbours of its colour and FALSE otherwise. The majority "voting" rule has been applied to a wide variety of dynamic systems, such as cellular automata and other distributed computing systems [2,4].

In order to prove that this simple strategy terminates, we extend our colouring convention to colour the links of the network. The links connecting either pairs of blue or red nodes are coloured blue or red, respectively. For a connected pair of differently coloured nodes, we mix the colours to obtain a *magenta link*.

Theorem 1. *Let $G = (N, L)$ be a network with bi-chromatic node set N. In the sequential model, the Recolouring Algorithm with parameter T and the* SURROUNDED_COMBINATORIAL function terminates after $O(|L|)$ recolourings from the time of the last fault plus T.

Proof. We use a simple counting argument on the number of magenta links. The number of magenta links is obviously at most $|L|$. After T units of time from the last fault, any notification of colour change (i.e., COLOUR message) is a consequence of a recolouring, as opposed to a delayed broadcast from a node that has become faulty due to environmental factors. Then, with every recolouring, the number of magenta links incident to the recoloured node decreases. Because no other recolouring occur simultaneously, according to our definition of the sequential model, the overall number of magenta links decreases with every recolouring. Thus, at most $O(|L|)$ recolourings can occur, which proves the theorem.

We assume that each node knows all its neighbours and the angle each neighbour is from it. With the angle information at hand, a new technique can be developed for implementing the SURROUNDED function of Algorithm 1. The geometric criterion we use for fault recovery is geometric recolouring as presented in the previous section. That is, the function SURROUNDED_GEOMETRIC is defined to return TRUE if the angle defined by the oppositely coloured neighbours of the node in question is greater than 180°, and FALSE otherwise. Also, nodes with one neighbour are never considered surrounded, and nodes with all (2 or more) neighbours of the opposite colour are always surrounded, as the surrounding angle is considered 360°.

In order to guarantee the termination of the geometric recolouring approach to fault recovery, some preprocessing of the network is required. In what follows we provide a set of preliminary results required to introduce the preprocessing algorithm and the geometric approach to fault recovery.

It is known that a geometric recolouring process can be infinite for general (non-planar) networks (see [6]). Thus, the recolouring strategy cannot be directly applied to any network, because it is crucial that the fault recovery process terminates. Instead, it can be applied to a network that approximates the original network as well as possible and guarantees finite recolouring. To this end we use NIC networks, as defined next.

Definition 1 ***(convex node and convex links).*** *A* convex node *p of a network is a node with two consecutive incident links, the* convex links *with respect to p, that define an angle greater than* 180°.

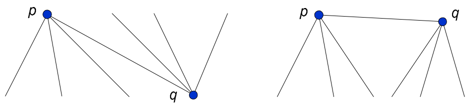


Fig. 2. Two cases of adjacent convex nodes p and q sharing a convex link

Theorem 2. *Let $G = (N, L)$ be a (not necessarily plane) network with set of bi-chromatic nodes N and set of links L, such that every node p in G satisfies one of the following three conditions:*

- *p has degree less than or equal to 1,*
- *p is not convex,*
- *p is convex and is adjacent to another convex node through a convex link (i.e., p is not an* isolated convex *node).*

The length of geometric recolouring sequences of G is $O(|N||L|)$.

This theorem is a generalization of our bound for geometric recolouring in triangulations from [6]. The full proof can be found in [7].

We define a network that satisfies the conditions of Theorem 2 to be a *NIC network*. One of the advantages of using NIC networks is that they can be described using only local properties of the nodes and therefore, can be computed in a localized manner.

Corollary 3. *Let $G = (N, L)$ be a NIC network with bi-chromatic node set N. In the sequential model, the Recolouring Algorithm with parameter T and the* SURROUNDED_GEOMETRIC *function terminates after $O(|N||L|)$ recolourings from the time of the last fault plus T.*

In what follows, we show how to compute a NIC network that represents, as well as possible, the original structure of the network in a localized manner. The idea is to start with the original network and either "add" or "remove" a small number of links such that the resulting network is a NIC network. By adding and removing links we mean that a node considers new nodes as neighbours or disregards neighbours, only for recolouring purposes.

Through edge additions one could construct a NIC network or even a (non-planar) superset of a triangulation, which is known to have at most a cubic ($O(N^3)$) number of recolourings (Theorem 13 [6]). However, adding links may involve pairs of nodes that are multiple hops away from each other in the network, which calls for non-localized algorithms and more communication intensive distributed computation. The goal then is to remove the minimum number of links so that the network satisfies the NIC conditions. This way the topology of the original network is fairly well preserved. The next theorem states that it is hard to find such optimal configuration, even if centralized computation is allowed. We formally state the problem and the complexity of its decidability version, which directly implies the hardness of its minimization version.

Problem 1. **Non-Isolated Convex (NIC)**
Instance: (G, k), where $G = (N, L)$ is a network with $|N| = n$ and k is a numeric constant.

Question: Is there a set of links $L' \subset L$ such that $|L'| \leq k$ and $G' = (V, L \setminus L')$ satisfies the NIC conditions?

Theorem 4. *The NIC problem (Problem 1) is NP-Complete.*

We omit the proof of this theorem due to limited space. For details see [7].

Because the optimal solution is hard to find, we study heuristic algorithms that eliminate a relatively small number of links. It is noteworthy that in the worst case the optimal number of link removals may be linear in the number of links and quadratic in the number of nodes. An example network that exhibits this complexity consists of a complete bipartite network, where the nodes of each partition lie on one of two parallel lines (see [7] for details).

The example we just described shows that optimal solutions, and heuristic solutions alike, cannot always eliminate a small number of links in the worst case. Therefore, we propose a simple heuristic algorithm that eliminates a small number of links according to our experiments. The heuristic algorithm for constructing the NIC network, the NIC Algorithm (Algorithm 2), is described next. The NIC Algorithm operates under the same assumptions as the geometric recolouring strategy: all nodes know their neighbours and the angle they define with respect to them. The algorithm consists of a single protocol executed at all nodes.

Link marks, as used in the protocol, are relative to the node, that is, a link can be marked differently by each incident node. Notice that Step 3 of the protocol assures that no isolated convex node will remain connected to a non-convex node. The algorithm is guaranteed to terminate because links are always removed and never replaced. Note that a network empty of links satisfies the NIC conditions so, in the worst case the algorithm terminates when all the links have been removed. Obviously, this is a very pessimistic analysis, as normally only a small fraction of the links are removed (see Section 4). Furthermore, this protocol does not fall into infinite loops because once a link is convex, it never becomes non-convex.

We also propose a simple hybrid strategy that combines the combinatorial and the geometric strategies and yields the best experimental results for certain degrees of connectivity, as will be discussed in Section 4. The combination of the combinatorial and geometric recolouring strategies requires some extra care; otherwise the resulting strategy may not terminate, despite each separate strategy does. For *hybrid recolouring* we define a SURROUNDED_HYBRID function that uses the majority rule for the most part, except that when the number of neighbours of the same and opposite colours are equal, the geometric (surrounding angle) criterion is used to break the tie.

It is easy to see that this process yields a finite recolouring sequence if the geometric component considers a NIC subnetwork: the number of magenta links always decreases or remains the same (see the proof of Theorem 1). Also, while

Algorithm 2. NIC Algorithm

```
input  : Network $G = (N, L)$ with bi-chromatic nodes.
output : Network $G' = (N, L')$ such that $L' \subseteq L$ and $G'$ satisfies the NIC
         conditions.

NIC Protocol (executed at node $p$)

Step 1. Mark all links incident to $p$ as unknown.
Step 2. If there is no message in $p$'s message queue and there are still links marked
    as unknown, then send a message to each neighbour. The type of the message
    sent is either CONVEX or NON-CONVEX, depending on the convexity of the
    link with respect to $p$.
Step 3. If there is a message in the node's message queue, process the message
    according to its type:
  CONVEX: the link through which the message was received is marked as convex. If
    the link was not marked as convex before, then a message is sent back to the
    sender indicating the convexity with respect to $p$.
  NON-CONVEX: the link through which the message was received, $l$, is marked as
    non-convex. If $l$ is convex with respect to $p$, and $p$ is an isolated convex node,
    then $p$ removes $l$, sends a REMOVE message to the corresponding neighbour,
    and sends CONVEX messages over any other link that may have become convex
    after removing $l$.
  REMOVE: the link through which the message was received, $l$, is removed.
Step 4. Go to Step 2.
```

recolourings that preserve the number of magenta links occur, geometric recolouring converges for the same reasons as Theorem 2. It then follows that the number of recolourings is at most the multiplication of the maximum possible number of recolourings for each method. This is stated in the following theorem.

Theorem 5. *Let $G = (N, L)$ be a network with bi-chromatic node set N. In the sequential model, the Recolouring Algorithm with parameter T and the* SURROUNDED_HYBRID *function terminates after $O(|N||L|^2)$ recolourings from the time of the last fault plus T.*

4 Experiments

There are aspects of recolouring in networks that make an accurate probabilistic analysis quite complicated; for example, small changes in the recolouring order may produce completely different colour configurations. Thus, we turn to experimentation for our analysis.

We coded our recolouring simulator using Java. The experimental test bed consists of a set of connected networks generated at random with $N = 100$ nodes uniformly distributed over a 100 by 100 square grid with area $A = 10^4$. Two nodes share a link if and only if the distance between them is at most a certain unit Δ, to form what is known as a unit disk graph. We generate a

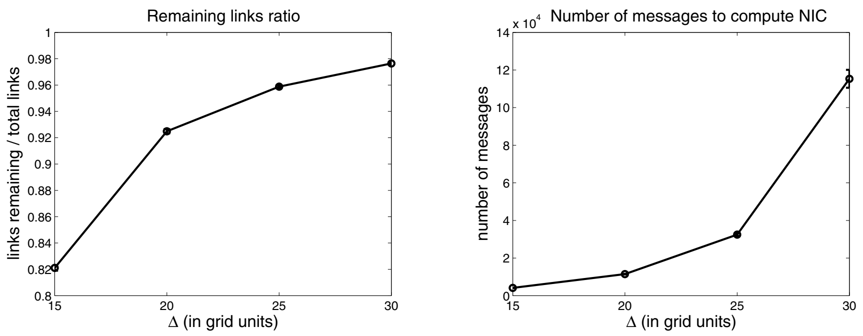


Fig. 3. Results produced by the NIC Algorithm: (left) remaining links ratio, (right) number of messages

set of 1000 random connected networks with unit distance Δ taking on values 15, 20, 25, and 30 times the width of a grid square. These distances have been chosen so that the network is k-connected with high probability for values of k ranging from 1 to 10: $\Delta = 15$ approximately corresponds to $k = 1$ and $\Delta = 30$ to $k = 10$. The results plotted below are averaged over the 1000 randomly generated networks. Notice that the network size (number of nodes) is not critical for our results, as the phenomena we study affect only localized, relatively small areas of the network. The density of the network, however, does play a crucial role. For this reason, our experiments consider different degrees of connectivity, as explained above.

We first present the experimental results for the NIC algorithm. The mean ratio between number of links remaining and total number of links is plotted in Figure 3 (left) for different transmission radii (Δ). It is noticeable that the results improve as Δ and the network connectivity increase. Obviously, for higher values of Δ the convex nodes tend to appear only at the boundary of the grid. According to the NIC Algorithm these are the only nodes from which incident links are removed. Another measure of interest is the number of messages incurred while computing the NIC network. This result is plotted in Figure 3 (right). The graph shows a super-linear growth in the number of messages with respect to Δ. This is expected, as the connectivity of the network increases, at a nearly quadratic rate with respect to the transmission radius (Δ), as presented by Wang and Yi [12] (Theorem 2). Whether more efficient heuristics can be devised for locally computing NIC networks remains an interesting question.

Next we present the results of the recolouring process for all the methods described above. For our experiments we have induced faults on the nodes that fall within a randomly chosen circular area within the grid. The circular area is defined by radii E =10, 12, 14, 16, 18, and 20 times the width of a grid square, and is placed such that it falls at least half its radius away from the border of the grid area. The latter is meant to eliminate the "border effect", that is, faulty nodes at the border are more difficult to recover because of the smaller number

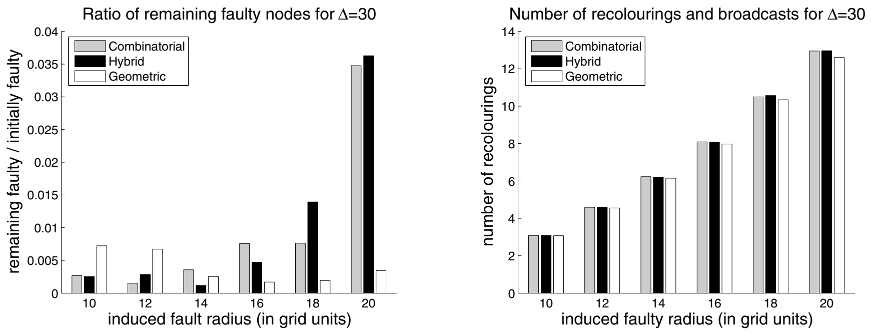


Fig. 4. Results of the recolouring algorithm: (left) fraction of the nodes that remain faulty, (right) number of recolouring and messages incurred

and angle span of neighbours around them. The expected number of faulty nodes is $\pi E^2 N/A$ in our examples.

From our experiments we conclude that for sparse graphs, $\Delta < 25$, the combinatorial and hybrid methods outperform geometric recolouring, with slight advantage to the hybrid method. This comes as no surprise, because on sparse graphs there are not enough links, or angles defined by links, to perform an accurate geometric recolouring. The most interesting results correspond to denser networks, $\Delta > 25$ (see Figure 4 (left) corresponding to $\Delta = 30$). This graph shows the mean ratio of nodes that remain (or become) faulty after the recolouring process terminates.

It is not surprising that geometric recolouring gives better results as the network gets denser and the size of the affected area increases. In such scenarios, the faulty neighbours of a node can sometimes outnumber the healthy ones, which makes the fault spread out of the affected region if using the combinatorial or hybrid methods. On the other hand, as the affected region is convex (a disk in the experiments presented above), no healthy node can be surrounded by faulty ones. This confines the faults to the region initially affected, if not heals the region altogether as it happens in most cases. Figure 4 (right) shows that, despite the smaller number of nodes healed by the combinatorial method, the number of recolourings (and broadcasts) is approximately the same for all methods. This evidences that for large error sizes on dense networks the combinatorial criterion spends many recolourings in turning healthy nodes into faulty, an obviously undesirable effect.

We also conducted experiments where the induced error was defined by a non-convex shape: we used the union of a pair of disks intersecting at a point for our experiments. The results are remarkably similar to the results previously presented for faults induced by single disks. Thus, we conclude that the effectiveness of geometric recolouring is not limited to convexly-shaped affected areas, but also to areas that are partly convex, at the very least.

5 Open Problems

In the previous sections we have mentioned some open problems and conjectures. We summarize a list of these and other problems of interest.

– Compute NIC networks as efficiently as possible, that is, using a small number of messages.
– Characterize networks that have finite geometric recolouring sequences through local properties at the nodes, other than NIC networks.
– Find other recolouring strategies suitable for fault recovery.

References

1. Adler, M., Demaine, E.D., Harvey, N.J.A., Pătraşcu, M.: Lower bounds for asymmetric communication channels and distributed source coding. In: Proc. 17th Annual ACM-SIAM Symp. on Disc. Alg. (SODA 2006), pp. 251–260 (2006)
2. Flocchini, P., Lodi, E., Luccio, F., Pagli, L., Santoro, N.: Dynamic monopolies in tori. Disc. Applied Math. 137(2), 197–212 (2004)
3. Goles, E., Olivos, J.: Periodic behaviour of generalized threshold functions. Disc. Math. 30, 187–189 (1980)
4. Krishnamachari, B., Iyengar, S.: Distributed bayesian algorithms for fault-tolerant event region detection in wireless sensor networks. IEEE Trans. on Comp. 53(3), 241–250 (2004)
5. Luccio, F., Pagli, L., Sanossian, H.: Irreversible dynamos in butterflies. In: Proc. 6th Intl. Colloq. on Structural Inf. and Comm. Complex, pp. 204–218 (1999)
6. Meijer, H., Núñez-Rodríguez, Y., Rappaport, D.: Bounds for point recolouring in geometric graphs. Comp. Geom.: Theory and Apps. 42(6-7), 690–703 (2009)
7. Núñez-Rodríguez, Y.: Problems on Geometric Graphs with Applications to Wireless Networks. PhD Thesis, School of Computing, Queen's University (2009), http://qspace.library.queensu.ca/handle/1974/5335
8. Peleg, D.: Local majority voting, small coalitions and controlling monopolies in graphs: A review. Technical Report CS96-12, Department of Mathematics & Computer Science, Weizmann Institute of Science (1996)
9. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM Monographs on Disc. Math. and Apps. (2000)
10. Reinbacher, I., Benkert, M., van Kreveld, M., Mitchell, J.S.B., Snoeyink, J., Wolff, A.: Delineating boundaries for imprecise regions. Algorithmica 50(3), 386–414 (2008)
11. Slepian, D., Wolf, J.K.: Noiseless encodings of correlated information sources. IEEE Trans. on Inf. Theory 19(4), 471–480 (1973)
12. Wan, P.J., Yi, C.W.: Asymptotic critical transmission radius and critical neighbour number for k-connectivity in wireless ad hoc networks. In: Proc. of the 5th ACM Intl. Symp. on Mobile Ad Hoc Networking and Comp., pp. 1–8 (2004)

Geometric Minimum Spanning Trees with GEOFILTERKRUSKAL[*]

Samidh Chatterjee, Michael Connor, and Piyush Kumar

Department of Computer Science, Florida State University
Tallahassee, FL 32306-4530
{chatterj,miconnor,piyush}@cs.fsu.edu

Abstract. Let P be a set of points in $\mathbb{R}^d$. We propose GEOFILTERKRUSKAL, an algorithm that computes the minimum spanning tree of P using well separated pair decomposition in combination with a simple modification of Kruskal's algorithm. When P is sampled from uniform random distribution, we show that our algorithm takes one parallel sort plus a linear number of additional steps, with high probability, to compute the minimum spanning tree. Experiments show that our algorithm works better in practice for most data distributions compared to the current state of the art [31]. Our algorithm is easy to parallelize and to our knowledge, is currently the best practical algorithm on multi-core machines for $d > 2$.

Keywords: Computational Geometry, Experimental Algorithmics, Minimum spanning tree, Well separated pair decomposition, Morton ordering, multi-core.

1 Introduction

Computing the geometric minimum spanning tree (GMST) is a classic computational geometry problem which arises in many applications including clustering and pattern classification [38], surface reconstruction [28], cosmology [4,6], TSP approximations [2] and computer graphics [26]. Given a set of n points P in $\mathbb{R}^d$, the GMST of P is defined as the minimum weight spanning tree of the complete undirected weighted graph of P, with edges weighted by the distance between their end points. In this paper, we present a practical deterministic algorithm to solve this problem efficiently, and in a manner that easily lends itself to parallelization.

It is well established that the GMST is a subset of edges in the Delaunay triangulation of a point set [33]. It is well known that that this method is inefficient for any dimension $d > 2$. It was shown by Agarwal et al. [1] that the GMST problem is related to solving bichromatic closest pairs for some subsets of the input set. The bichromatic closest pair problem is defined as follows: given two sets of points, one red and one green, find

[*] This research was partially supported by National Science Foundation through CAREER Grant CCF-0643593 and the AFOSR Young Investigator Program. Advanced Micro Devices (AMD) provided the 32-core workstation for experiments. Dr. Giri Narsimhan provided the source code for his Geometric Minimum Spanning tree algorithm [31], which was used as one of the benchmarks for the algorithm implemented in this paper. The source code associated with this paper is part of the STANN library and can be downloaded from www.compgeom.com/∼stann

the red-green pair with minimum distance between them [25]. Callahan [8] used well separated pair decomposition and bichromatic closest pairs to solve the same problem in $\mathcal{O}(T_d(n,n)\log n)$, where $T_d(n,n)$ is the time required to solve the bichromatic closest pairs problem for n red and n green points. It is also known that the GMST problem is harder than bichromatic closest pair problem, and bichromatic closest pair is probably harder than computing the GMST [17].

Clarkson [11] gave an algorithm that is particularly efficient for points that are independently and uniformly distributed in a unit d-cube. His algorithm has an expected running time of $\mathcal{O}(n\alpha(cn,n))$, where c is a constant depending on the dimension and α is an extremely slow growing inverse Ackermann function [14]. Bentley [5] also gave an expected nearly linear time algorithm for computing GMSTs in $\mathbb{R}^d$. Dwyer [16] proved that if a set of points is generated uniformly at random from the unit ball in $\mathbb{R}^d$, its Delaunay triangulation has linear expected complexity and can be computed in expected linear time. Since GMSTs are subsets of Euclidean Delaunay triangulations, one can combine this result with linear time MST algorithms [23] to get an expected $\mathcal{O}(n)$ time algorithm for GMSTs of uniformly distributed points in a unit ball. Rajasekaran [35] proposed a simple expected linear time algorithm to compute GMSTs for uniform distributions in $\mathbb{R}^d$. All these approaches use bucketing techniques to execute a spiral search procedure for finding a supergraph of the GMST with $\mathcal{O}(n)$ edges. Unfortunately, in our experiments, finding k-nearest neighbors for every point, even when $\mathsf{k} = \mathcal{O}(1)$, proved to be as expensive as finding the actual GMST. We discuss this in Section 3, and show some experimental results in Section 5.

Narasimhan et al. [31] gave a practical algorithm that solves the GMST problem. They prove that for uniformly distributed points, in fixed dimensions, an expected $\mathcal{O}(n\log n)$ steps suffice to compute the GMST using well separated pair decomposition. Their algorithm, GeoMST2, mimics Kruskal's algorithm [24] on well separated pairs and eliminates the need to compute bichromatic closest pairs for many well separated pairs. To our knowledge, this implementation is the state of the art, for practically computing GMSTs in low dimensions ($d > 2$). Although, improvements to GeoMST2 [31] have been announced [27], exhaustive experimental results are lacking in this short announcement. Another problem with this claim is that even for uniform distribution of points, there are no theoretical guarantees that the algorithm is indeed any better than $\mathcal{O}(n^2)$.

Brennan [7] presented a modification to Kruskal's classic minimum spanning tree (MST) algorithm [24] that operated in a manner similar to quicksort; splitting an edge set into "light" and "heavy" subsets. Recently, Osipov et al. [32] further expanded this idea by adding a multi-core friendly filtering step designed to eliminate edges that were obviously not in the MST (Filter-Kruskal). Currently, this algorithm seems to be the most practical algorithm for computing MSTs on multi-core machines.

The algorithm presented in this paper uses well separated pair decomposition in combination with a modified version of Filter-Kruskal for computing GMSTs. We use a compressed quad-tree to build a well separated pair decomposition, followed by a sorting based algorithm similar to Filter-Kruskal. By using a sort based approach, we eliminate the need to maintain a priority queue [31]. This opens up the possibility of filtering out well separated pairs with a large number of points, before it becomes necessary to calculate their bichromatic closest pair. Additionally, we can compute the bichromatic closest pair of well separated pairs of similar size in batches. This allows for parallel execution of large portions of the algorithm.

Since GeoMST2 is the only known practical algorithm for computing GMSTs, all our results, both experimental and theoretical, were compared against GeoMST2. The theoretical running time of GeoMST2 was analyzed only for uniform distributions. In this paper, when we talk about the theoretical running time of our algorithm, the underlying distribution should be assumed as uniform unless otherwise stated.

Our algorithm takes one parallel sort plus $\mathcal{O}(n)$ additional steps, with high probability, to compute the GMST. This result is an improvement over the original Filter-Kruskal algorithm [32]. The expected running time for constructing the MST for arbitrary graphs with random edge weights, using the original Filter-Kruskal algorithm [32] is $\mathcal{O}(m + n \log n \log m/n)$, where m and n are the number of edges and vertices of the graph respectively.

If the coordinates of the points in the input set are integers of size $\mathcal{O}(\log n)$, and the word size of the machine is greater than or equal to $\log n$, the running time of our algorithm is $\mathcal{O}(n)$ if we use radix sort [19,10][1].The running time of GeoMST2 is on the contrary, $\mathcal{O}(n \log n)$, irrespective of the data-type of the point coordinates. Thus, for uniform distributions, we achieve the same runtime complexity as those algorithms described above, but without suffering from the drawback of the bucketing technique that makes them impractical to implement for other distributions. Assuming a CRCW PRAM model with $\mathcal{O}(\log n)$ processors, if sorting is allowed to be done in parallel (mergesort), the running time is $\mathcal{O}(n)$ for both integer and floating point coordinates.

The main contributions of this paper are: ① our algorithm shows significant runtime improvements over GeoMST2 for two and higher dimensions, ② the algorithm is easy to parallelize unlike GeoMST2, ③ in contrast with GeoMST2 which is inherently $\mathcal{O}(n \log n)$, our theoretical running time is upper bounded only by one sort which improves to $\mathcal{O}(n)$ for integers under the assumptions mentioned above, ④ our algorithm is faster compared to Filter-Kruskal on geometric instances and ⑤ a parallel implementation of the well separated pair decomposition on a compressed quadtree, which can be computed in $\mathcal{O}(n)$ time [10,20]. The code is now a part of the STANN library [12] and is available for download. At the time of submission of this paper we are not aware of any other open source implementation of the well separated pair decomposition using a compressed quadtree. For comparison purposes, we worked with the distributions on which GeoMST2 was run [31].

This paper is organized as follows: In the remainder of this section, we define our notation. In Section 2 we define and elaborate on tools that we use in our algorithm. Section 3 presents our algorithm, and a theoretical analysis of the running time. Section 4 describes our experimental setup, and Section 5 compares GeoMST2 with our algorithm experimentally. Finally, Section 6 concludes the paper and presents future research directions.

Notation: Points are denoted by lower-case Roman letters. $\mathsf{Dist}(p, q)$ denotes the distance between the points p and q in L_2 metric. Upper-case Roman letters are reserved

[1] If the integer size is superlogarithmic, we can still get $o(n \log n)$ running time by applying known integer sorting algorithms on a word RAM model [10].

for sets. Scalars except for c, d, m and n are represented by lower-case Greek letters. We reserve i, j, k for indexing purposes. $\mathsf{Vol}()$ denotes the volume of an object. For a given quadtree, $\mathsf{Box}(p, q)$ denotes the smallest quadtree box containing points p and q; Fraktur letters ($\mathfrak{a}$) denote a quadtree node. $\mathsf{MinDist}(\mathfrak{a}, \mathfrak{b})$ denotes the minimum distance between the quadtree boxes of two nodes. $\mathsf{Bccp}(\mathfrak{a}, \mathfrak{b})$ computes the bichromatic closest pair of two nodes, and returns $\{u, v, \delta\}$, where (u, v) is the edge defining the Bccp and δ is the edge length. $\mathsf{Left}(\mathfrak{a})$ and $\mathsf{Right}(\mathfrak{a})$ denotes the left and right child of a node. $|.|$ denotes the cardinality of a set or the number of points in a quadtree node. $\alpha(n)$ is used to denote inverse of the Ackermann function [14]. The Cartesian product of two sets X and Y, is denoted $X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}$.

2 Preliminaries

Before we present our algorithm in detail, we need to describe a few tools which our algorithm uses extensively. These include Morton ordering, quadtrees, well separated pair decomposition, a practical algorithm for bichromatic closest pair computation and the $\mathsf{UnionFind}$ data structure. We describe these tools below.

Morton ordering and Quadtrees: Morton order is a space filling curve with good locality preserving behavior [21]. Due to this behavior, it is used in data structures for mapping multidimensional data into one dimension. The Morton order value of a point can be obtained by interleaving the binary representations of its coordinate values. If we recursively divide a d-dimensional hypercube into 2^d hypercubes until there is only one point in each, and then order those hypercubes using their Morton order value, the Morton order curve is obtained. In 2 dimensions we refer to this decomposition as a quadtree decomposition, since each square can be divided into four squares. We will explain the algorithm using quadtrees, but this can be easily extended to higher dimensions. Chan [9] showed that using a few binary operations for integer coordinates, the relative Morton ordering can be calculated by, determining the dimension in which corresponding coordinates have the first differing bit in the highest bit position. This technique can be extended to work for floating point coordinates, with only a constant amount of extra work [13]. In the next paragraph, we briefly state the quadtree construction that we use in our algorithm [10,20].

Let $p_1, p_2, ..., p_n$ be a given set of points in Morton order. Let the index j be such that $\mathsf{Vol}(\mathsf{Box}\{p_{j-1}, p_j\})$ is the largest. Then the compressed quadtree Q for the point set consists of a root holding the $\mathsf{Box}(p_{j-1}, p_j)$ and two subtrees recursively built for $p_1, ..., p_{j-1}$ and $p_j, .., p_n$. Note that this tree is simply the *Cartesian tree* [20] of $\mathsf{Vol}(\mathsf{Box}\{p_1, p_2\})$, $\mathsf{Vol}(\mathsf{Box}\{p_2, p_3\})$, $\ldots$, $\mathsf{Vol}(\mathsf{Box}\{p_{n-1}, p_n\})$. A Cartesian tree is a binary tree constructed from a sequence of values which maintains three properties. First, there is one node in the tree for each value in the sequence. Second, a symmetric, in-order traversal of the tree returns the original sequence. Finally, the tree maintains a heap property, in which a parent node has a larger value than its child nodes. The Cartesian tree can be computed using a standard incremental algorithm in linear time [20], given the ordering $p_1, p_2, ..., p_n$ [10]. Hence, for both integer as well as floating point coordinates, the compressed quadtree Q can be computed in $\mathcal{O}(n)$ excluding the time for sorting the points in Morton order which takes $\mathcal{O}(n \log n)$ [13] for floating point coordinates. We use a compressed quadtree, as opposed to the more common fair split tree [8], because the later takes $\mathcal{O}(n \log n)$ time for construction. If parallel quick-sort [14] is used for the Morton ordering, the construction is $\mathcal{O}(n)$ for both integer and floating point coordinates.

Well Separated Pair Decomposition [8]: We implemented well separated pair decomposition on a compressed quadtree, which can be computed in $\mathcal{O}(n)$ time [10,20]. Assume that we are given a compressed quadtree Q, built on a set of points P in $\mathbb{R}^d$. Two nodes $\mathfrak{a}$ and $\mathfrak{b} \in Q$ are said to be an *ϵ-well separated pair* (ϵ-WSP) if the diameter of $\mathfrak{a}$ and the diameter of $\mathfrak{b}$ are both at most $\epsilon{*}\mathsf{MinDist}(\mathfrak{a}, \mathfrak{b})$. An *$\epsilon$-well separated pair decomposition (ϵ-WSPD)*, of size m, for a quadtree Q, is a collection of well separated pairs of nodes $\{(\mathfrak{a}_1, \mathfrak{b}_1), ..., (\mathfrak{a}_m, \mathfrak{b}_m)\}$, such that every pair of points $(p, q) \in P \times P$ $(p \neq q)$ lies in exactly one pair $(\mathfrak{a}_i, \mathfrak{b}_i)$. In the following, we use WSP and WSPD to mean well separated pairs and decompositions with an ϵ value of 1.

On the compressed quadtree Q, one can execute Callahan's [8] WSPD algorithm [10]. It takes $\mathcal{O}(n)$ time to compute the WSPD, given the compressed quadtree, thus resulting in an overall theoretical running time of $\mathcal{O}(n \log n)$ for WSPD construction. The number of well separated pairs produced by this approach is more than that produced by the fair-split tree approach, although only by a constant factor. For example, for uniform distributions, on an average, we produce 27% more WSPs but are 1.33 times faster.

Bichromatic Closest Pair Algorithm: Given a set of points in $\mathbb{R}^d$, the *Bichromatic Closest Pair* (Bccp) problem asks to find the closest red-green pair [25]. The computation of the bichromatic closest pairs is necessary due to the following Lemma from [8]:

Lemma 1. *The set of all the Bccp edges of the WSPD contain the edges of the GMST.*

Clearly, $\mathsf{MinDist}(\mathfrak{a}, \mathfrak{b})$ is the lower bound on the bichromatic closest pair distance of A and B (where A and B are the point sets contained in $\mathfrak{a}$ and $\mathfrak{b}$ respectively). We use a simple Bccp algorithm proposed by Narsimhan and Zachariasen [31] .

The input to algorithm 1 are two nodes $\mathfrak{a}, \mathfrak{b} \in Q$, that contain sets $A, B \subseteq P$ and a positive real number δ, which is used by the recursive calls in the algorithm to keep track of the last closest pair distance found. The output of the algorithm is the closest pair of points (p, q), such that $p \in A$ and $q \in B$ with minimum distance $\delta = \mathsf{Dist}(p, q)$. Initially, the Bccp is equal to η, where η represents the WSP containing only the last point in Morton order of A and the first point in the Morton order of B, assuming without loss of generality, all points in A are smaller than all points in B, in Morton order. If both A and B are singleton sets, then the distance between the two points is trivially the Bccp distance. Otherwise, we compare $\mathsf{Vol}(\mathfrak{a})$ and $\mathsf{Vol}(\mathfrak{b})$ and compute the distance between the lighter node and each of the children of the heavier node. If either of these distances is less than the closest pair distance computed so far, we recurse on the corresponding pair. If both of the distances are less, we recurse on both of the pairs.

UnionFind Data Structure: The $\mathsf{UnionFind}$ data structure maintains a set of partitions indicating the connectivity of points based on the edges already inserted into the GMST. Given a $\mathsf{UnionFind}$ data structure $\mathcal{G}$, and $u, v \in P \subseteq \mathbb{R}^d$, $\mathcal{G}$ supports the following two operations: $\mathcal{G}.\mathsf{Union}(u, v)$ updates the structure to indicate the partitions containing u and v belong to the same connected component; $\mathcal{G}.\mathsf{Find}(u)$ returns

Algorithm 1. Bccp Algorithm [31]: Compute $\{p', q', \delta'\}$ =Bccp($\mathfrak{a}, \mathfrak{b}[, \{p, q, \delta\} = \eta]$)

```
Require: $\mathfrak{a}, \mathfrak{b} \in Q, A, B \subseteq P, \delta \in \mathbb{R}^+$
Require: If $\{p, q, \delta\}$ is not specified, $\{p, q, \delta\} = \eta$, an upper bound on Bccp($\mathfrak{a}, \mathfrak{b}$).
 1: procedure Bccp($\mathfrak{a}$,$\mathfrak{b}$[, $\{p, q, \delta\} = \eta$])
 2:     if ($|A| = 1$ and $|B| = 1$) then
 3:         Let $p' \in A,\ q' \in B$
 4:         if Dist($p', q'$) $< \delta$ then return $\{p', q'$, Dist($p', q'$)$\}$
 5:     else
 6:         if Vol($\mathfrak{a}$) $<$ Vol($\mathfrak{b}$) then Swap($\mathfrak{a}$,$\mathfrak{b}$)
 7:         $\gamma$ =MinDist(Left($\mathfrak{a}$), $\mathfrak{b}$)
 8:         $\zeta$ =MinDist(Right($\mathfrak{a}$), $\mathfrak{b}$)
 9:         if $\gamma < \delta$ then $\{p, q, \delta\}$ =Bccp(Left($\mathfrak{a}$), $\mathfrak{b}$, $\{p, q, \delta\}$)
10:         if $\zeta < \delta$ then $\{p, q, \delta\}$ =Bccp(Right($\mathfrak{a}$), $\mathfrak{b}$, $\{p, q, \delta\}$)
11:     return $\{p, q, \delta\}$
12: end procedure
```

the node number of the *representative* of the partition containing u. Our implementation also does *union by rank* and *path compression*. A sequence of τ $\mathcal{G}$.Union() and $\mathcal{G}$.Find() operations on n elements takes $\mathcal{O}(\tau\alpha(n))$ time in the worst case. For all practical purposes, $\alpha(n) \leq 4$ (see [14]).

3 GeoFilterKruskal Algorithm

Our GeoFilterKruskal algorithm computes a GMST for $P \subseteq \mathbb{R}^d$. Kruskal's [24] algorithm shows that given a set of edges, the MST can be constructed by considering edges in increasing order of weight. Using Lemma 1, the GMST can be computed by running Kruskal's algorithm on the Bccp edges of the WSPD of P. When Kruskal's algorithm adds a Bccp edge (u, v) to the GMST, where $u, v \in P$, it uses the UnionFind data structure to check whether u and v belong to the same connected component. If they do, that edge is discarded. Otherwise, it is added to the GMST. Hence, before testing for an edge (u, v) for inclusion into the GMST, it should always attempt to add all Bccp edges (u', v'), such that, Dist$(u', v') <$ Dist(u, v). GeoMST2 [31] avoids the computation of Bccp for many well separated pairs that already belong to the same connected component. Our algorithm uses this crucial observation as well. Algorithm 2 describes the GeoFilterKruskal algorithm in more detail.

The input to the algorithm, is a WSPD of the point set $P \subseteq \mathbb{R}^d$. All procedural variables are assumed to be passed by reference. The set of WSPs S is partitioned into set E_l that contains WSPs with cardinality less than β (initially 2), and set $E_u = S \setminus E_l$. We then compute the Bccp of all elements of set E_l, and compute ρ equal to the minimum MinDist($\mathfrak{a}, \mathfrak{b}$) for all $(\mathfrak{a}, \mathfrak{b}) \in E_u$. E_l is further partitioned into E_{l1}, containing all elements with a Bccp distance less than ρ, and $E_{l2} = E_l \setminus E_{l1}$. E_{l1} is passed to the Kruskal procedure, and $E_{l2} \cup E_u$ is passed to the Filter procedure. The Kruskal procedure adds the edges to the GMST or discards them if they are connected. Filter removes all connected WSPs. The GeoFilterKruskal procedure is recursively called, increasing the threshold value (β) by one each time, on the WSPs that survive the Filter procedure, until we have constructed the minimum spanning tree.

Algorithm 2. GEOFILTERKRUSKAL Algorithm

```
Require: $S = \{(\mathfrak{a}_1, \mathfrak{b}_1), ..., (\mathfrak{a}_m, \mathfrak{b}_m)\}$ is a WSPD, constructed from $P \subseteq \mathbb{R}^d$ ; $T = \{\}$.
Ensure: Bccp Threshold $\beta \geq 2$.
1: procedure GEOFILTERKRUSKAL(Sequence of WSPs : $S$, Sequence of Edges : $T$, UnionFind : $\mathcal{G}$, Integer : $\beta$)
2:     $E_l = E_u = E_{l1} = E_{l2} = \emptyset$
3:     for all $(\mathfrak{a}_i, \mathfrak{b}_i) \in S$ do
4:         if $(|\mathfrak{a}_i| + |\mathfrak{b}_i|) \leq \beta$ then $E_l = E_l \cup \{(\mathfrak{a}_i, \mathfrak{b}_i)\}$ else $E_u = E_u \cup \{(\mathfrak{a}_i, \mathfrak{b}_i)\}$
5:     end for
6:     $\rho = \min\{\text{MinDist}\{\mathfrak{a}_i, \mathfrak{b}_i\} : (\mathfrak{a}_i, \mathfrak{b}_i) \in E_u, i = 1, 2, ..., m\}$
7:     for all $(\mathfrak{a}_i, \mathfrak{b}_i) \in E_l$ do
8:         $\{u, v, \delta\} = \text{Bccp}(\mathfrak{a}_i, \mathfrak{b}_i)$
9:         if $(\delta \leq \rho)$ then $E_{l1} = E_{l1} \cup \{(u, v)\}$ else $E_{l2} = E_{l2} \cup \{(u, v)\}$
10:    end for
11:    KRUSKAL($E_{l1}, T, \mathcal{G}$)
12:    $E_{new} = E_{l2} \cup E_u$
13:    FILTER($E_{new}, \mathcal{G}$)
14:    if ( $|T| < (n - 1)$) then GEOFILTERKRUSKAL($E_{new}, T, \mathcal{G}, \beta + 1$)
15: end procedure
16: procedure KRUSKAL(Sequence of WSPs : $E$, Sequence of Edges : $T$, UnionFind : $\mathcal{G}$)
17:    Sort($E$): by increasing Bccp distance
18:    for all $(u, v) \in E$ do
19:        if $\mathcal{G}$.Find($u$) $\neq \mathcal{G}$.Find($v$) then $T = T \cup \{(u, v)\}$ ; $\mathcal{G}$.Union($u, v$);
20:    end for
21: end procedure
22: procedure FILTER(Sequence of WSPs : $E$, UnionFind : $\mathcal{G}$)
23:    for all $(\mathfrak{a}_i, \mathfrak{b}_i) \in E$ do
24:        if ($\mathcal{G}$.Find($u$) = $\mathcal{G}$.Find($v$) : $u \in \mathfrak{a}_i, v \in \mathfrak{b}_i$) then $E = E \setminus \{(\mathfrak{a}_i, \mathfrak{b}_i)\}$
25:    end for
26: end procedure
```

3.1 Correctness

Given previous work by Kruskal [14] as well as Callahan [8], it is sufficient to show two facts to ensure the correctness of our algorithm. First, we are considering WSPs to be added to the GMST in the order of their Bccp distance. This is obviously true considering WSPs are only passed to the KRUSKAL procedure if their Bccp distance is less than the lower bound on the Bccp distance of the remaining WSPs. Second, we must show that the FILTER procedure does not remove WSPs that should be added to the GMST. Once again, it is clear that any edge removed by the FILTER procedure would have been removed by the KRUSKAL procedure eventually, as they both use the UnionFind structure to determine connectivity.

3.2 Analysis of the Running Time

The real bottleneck of this algorithm, as well as the one proposed by Narasimhan [31], is the computation of the Bccp[2]. If $|A| = |B| = \mathcal{O}(n)$, the Bccp algorithm stated in section 3 has a worst case time complexity of $\mathcal{O}(n^2)$. Since we have to process $\mathcal{O}(n)$ edges, naively, the computation time for GMST will be $\mathcal{O}(n^3)$ in the worst case.

[2] According to the algebraic decision tree model, the lower bound of the set intersection problem can be shown to be $\Omega(n \log n)$ [18]. We can solve the set intersection problem using Bccp. If the Bccp distance between two sets is zero, we can infer that the sets intersect, otherwise they do not. Since the set intersection problem is lower bounded by $\Omega(n \log n)$, the Bccp computation is also lower bounded by $\Omega(n \log n)$.

High Probability Bound Analysis: In this section we show that algorithm 2 takes one sort plus $\mathcal{O}(n)$ additional steps, with high probability (WHP) [30], to compute the GMST. Let $Pr(\mathcal{E})$ denote the probability of occurrence of an event $\mathcal{E}$, where $\mathcal{E}$ is a function of n. An event $\mathcal{E}$ is said to occur WHP if given $\beta > 1$, $Pr(\mathcal{E}) > 1 - 1/n^{\beta}$ [30]. Let *P* be a set of *n* points chosen uniformly from a unit hypercube *H* in $\mathbb{R}^d$. Given this, we state the following lemma from [31].

Lemma 2. *Let* C_1 *and* C_2 *be convex regions in H such that* $\alpha \leq volume(C_1)/volume(C_2) \leq 1/\alpha$ *for some constant* $0 < \alpha < 1$. *If* $|C_1 \cap P|$ *is bounded by a constant, then with high probability* $|C_2 \cap P|$ *is also bounded by a constant.*

We will use the above lemma to prove the following claims.

Lemma 3. *Given a constant* $\gamma > 1$, WHP, *algorithm 2 filters out* WSPs *that have more than* γ *points.*

Proof. The proof of this lemma is similar to the one for GeoMST2. Consider a WSP $(\mathfrak{a}, \mathfrak{b})$. If both $|\mathfrak{a}|$ and $|\mathfrak{b}|$ are less than or equal to γ then the time to compute their Bccp distance is $\mathcal{O}(1)$. Let us now assume, w.lo.g., that $|\mathfrak{a}| > \gamma$. We will show that, in this case, we do not need to compute the Bccp distance of $(\mathfrak{a}, \mathfrak{b})$ WHP. Let $\overline{pq}$ be a line segment joining $\mathfrak{a}$ and $\mathfrak{b}$ such that the length of $\overline{pq}$ (let us denote this by $|\overline{pq}|$) is MinDist$(\mathfrak{a}, \mathfrak{b})$. Let C_1 be a hypersphere centered at the midpoint of $\overline{pq}$ and radius $|\overline{pq}|/4$. Let C_2 be another hypersphere with the same center but radius $3|\overline{pq}|/2$. Since $\mathfrak{a}$ and $\mathfrak{b}$ are well separated, C_2 will contain both $\mathfrak{a}$ and $\mathfrak{b}$. Now, $volume(C_1)/volume(C_2) = 6^{-d}$. Since C_1 is a convex region, if $|C_1|$ is empty, then by Lemma 2, $|C_2|$ is bounded by a constant WHP. But C_2 contains $\mathfrak{a}$ which has more than γ points. Hence C_1 cannot be empty WHP. Let $a \in \mathfrak{a}$, $b \in \mathfrak{b}$ and $c \in C_1$. Also, let the pair (a, c) and (b, c) belong to WSPs $(\mathfrak{u}_1, \mathfrak{v}_1)$ and $(\mathfrak{u}_2, \mathfrak{v}_2)$ respectively. Note that Bccp$(\mathfrak{a}, \mathfrak{b})$ must be greater than Bccp$(\mathfrak{u}_1, \mathfrak{v}_1)$ and Bccp$(\mathfrak{u}_2, \mathfrak{v}_2)$. Since our algorithm adds the Bccp edges by order of their increasing distance, c and the points in $\mathfrak{a}$ will be connected before the Bccp edge between $\mathfrak{a}$ and $\mathfrak{b}$ is examined. The same is true for c and the points in $\mathfrak{b}$. This causes $\mathfrak{a}$ and $\mathfrak{b}$ to belong to the same connected component WHP, and thus, our filtering step will get rid of the well separated pair $(\mathfrak{a}, \mathfrak{b})$ before we need to compute its Bccp edge. ◻

Lemma 4. WHP, *the total running time of the* UnionFind *operation is* $\mathcal{O}(\alpha(n)n)$.

Proof. Lemma 3 shows that, WHP, we only need to compute Bccp distances of WSPs of constant size. Since we compute Bccp distances incrementally, WHP, the number of calls to the GEOFILTERKRUSKAL procedure is also bounded above by $\mathcal{O}(1)$. In each of such calls, the FILTER function is called once, which in turn calls the Find(u) function of the UnionFind data structure $\mathcal{O}(n)$ times. Hence, there are in total $\mathcal{O}(n)$ Find(u) operations done WHP. Thus the overall running time of the Union() and Find() operations is $\mathcal{O}(\alpha(n)n)$ WHP (see the paragraph on UnionFind in Section 2). ◻

Theorem 1. *Algorithm 2 takes one sort plus* $\mathcal{O}(n)$ *additional steps,* WHP, *to compute the GMST.*

Proof. We partition the list of well separated pairs twice in the GEOFILTERKRUSKAL method. The first time we do it based on the need to compute the Bccp of the well separated pair. We have the sets E_l and E_u in the process. This takes $\mathcal{O}(n)$ time except for the Bccp computation. In $\mathcal{O}(n)$ time we can find the pivot element of E_u for the next partition. This partitioning also takes linear time. From Lemma 3, we can infer that the recursive call on GEOFILTERKRUSKAL is performed $\mathcal{O}(1)$ times WHP. Thus the total time spent in partitioning is $\mathcal{O}(n)$ WHP. Since the total number of Bccp edges required to compute the GMST is $\mathcal{O}(n)$, by Lemma 3, the time spent in computing all such edges is $\mathcal{O}(n)$ WHP. Total time spent in sorting the edges in the base case is $\mathcal{O}(n \log n)$. Including the time to compute the Morton order sort for the WSPD, the total running time of the algorithm is one sort plus $\mathcal{O}(n)$ additional steps WHP. ⊔⊓

Remark 1. As explained in Section 2, the Morton order sort required to construct the compressed quadtree for the WSPD is $\mathcal{O}(n)$ if points in P have integer coordinates. This is also applicable in case of sorting the edges inside the KRUSKAL procedure of algorithm 2. Thus the whole algorithm runs in $\mathcal{O}(n)$ time in this case, WHP. On the contrary, GeoMST2 will always take $\mathcal{O}(n \log n)$ steps even if the points in the input set have integer coordinates and the WSPD is constructed using a quadtree. This is because it adds edges to the GMST one at a time and before each addition it has to invoke an insertion and/or deletion procedure in a priority queue of WSPs [31]. Each such operation is $\mathcal{O}(\log n)$ [14]. Since there are $\mathcal{O}(n)$ WSPs, GeoMST2 will run in $\mathcal{O}(n \log n)$ steps.

Remark 2. Using a k-nearest neighbor graph, we can modify algorithm 2 such that its running time is $\mathcal{O}(n)$ WHP, if points in P have integer coordinates. One can first compute the minimum spanning forest of the k-nearest neighbor graph of the point set, for some given constant k, using a randomized linear time algorithm [22]. In this forest, let T' be the tree with the largest number of points. Rajasekaran [35] showed that there are only n^β points left to be added to T' to get the GMST, where $\beta \in (0, 1)$. In our algorithm, if the set T is initialized to T', then our analysis shows that WHP, only $\mathcal{O}(n^\beta)$ additional computations will be necessary to compute the GMST.

Remark 3. Computation of GMST from k-nearest neighbor graph can be parallelized efficiently in a CRCW PRAM. From our analysis we can infer that in case of a uniformly randomly distributed set of points P, if we extract the $\mathcal{O}(1)$ nearest neighbors for each point in the set, then these edges will contain the GMST of P WHP. Callahan [8] showed that it is possible to compute the k-nearest neighbors of all points in P in $\mathcal{O}(\log n)$ time using $\mathcal{O}(\mathsf{k}n)$ processors. Hence, using $\mathcal{O}(n)$ processors, the minimum spanning tree of P can then be computed in $\mathcal{O}(\log n)$ time [37].

Remark 4. We did not pursue implementing the algorithms described in remarks two and three because they are inherently Monte Carlo algorithms [29]. While they can achieve a solution that is correct with high probability, they do not guarantee a correct solution. One can design an exact GMST algorithm using k-nearest neighbor graphs; however preliminary experiments using the best practical parallel nearest neighbor codes [13,3] showed construction times that were slower than the GEOFILTERKRUSKAL algorithm.

3.3 Parallelization

Parallelization of the WSPD algorithm: In our sequential version of the algorithm, each node of the compressed quadtree computes whether its children are well separated or not. If the children are not well separated, we divide the larger child node, and recurse. We parallelize this loop using OpenMP [15] with a dynamic load balancing scheme. Since for each node there are $\mathcal{O}(1)$ candidates to be well separated [10], and we are using dynamic load balancing, the total time taken in CRCW PRAM, given p processors, to execute this algorithm is $\mathcal{O}(\lceil (n \log n)/\mathsf{p} \rceil + \log n)$ including the preprocessing time for the Morton order sort.

Parallelization of the GeoFilterKruskal **algorithm:** Although the whole of algorithm 2 is not parallelizable, we can parallelize most portions of the algorithm. The parallel partition algorithm [34] is used in order to divide the set S into subsets E_l and E_u (See Algorithm 2). ρ can be computed using parallel prefix computation. In our actual implementation, we found it to be more efficient to wrap it inside the parallel partition in the previous step, using the atomic compare-and-swap instruction. The further subdivision of E_l, as well as the Filter procedure, are just instances of parallel partition. The sorting step used in the Kruskal procedure as well as the Morton order sort used in the construction of the compressed quadtree can also be parallelized [34]. We use OpenMP to do this in our implementation. Our efforts to parallelize the linear time quadtree construction showed that one can not use more processors on multi-core machines to speed up this construction, because it is memory bound.

4 Experimental Setup

The GeoFilterKruskal algorithm was tested in practice against several other implementations of geometric minimum spanning tree algorithms. We chose a subset of the algorithms compared in [31], excluding some based on the availability of source code and the clear advantage shown by some algorithms in the aforementioned work. Table 1 lists the algorithms that will be referred to in the experimental section.

GeoFilterKruskal was written in C++ and compiled with g++ 4.3.2 with -O3 optimization. Parallel code was written using OpenMP [15] and the parallel mode extension to the STL [34]. C++ source code for GeoMST, GeoMST2, and Triangle were provided by Narsimhan. In addition, Triangle used Shewchuk's triangle library for Delaunay triangulation [36]. The machine used has 8 Quad-Core AMD Opteron(tm) Processor 8378 with hyperthreading enabled. Each core has a L1 cache size of 512 KB, L2 of 2MB and L3 of 6MB with 128 GB total memory. The operating system was CentOS 5.3. All data was generated and stored as 64 bit C++ doubles.

In the next section there are two distinct series of graphs. The first set displays graphs of total running time versus the number of input points, for two to five dimensional points, with uniform random distribution in a unit hypercube. The L_2 metric was used for distances in all cases, and all algorithms were run on the same random data set. Each algorithm was run on five data sets, and the results were averaged. As noted above, Triangle was not used in dimensions greater than two.

Table 1. Algorithm Descriptions

Name	Description
GeoFK#	Our implementation of Algorithm 2. There are two important differences between the implementation and the theoretical version. First, the BCCP Threshold β in section 3 is incremented in steps of size $\mathcal{O}(1)$ instead of size 1, because this change does not affect our analysis but helps in practice. Second, for small well separated pairs (less than 32 total points) the BCCP is computed by a brute force algorithm. In the experimental results, GeoFK1 refers to the algorithm running with 1 thread. GeoFK8 refers to the algorithm using 8 threads.
GeoMST	Described by Callahan and Kosaraju [8]. This algorithm computes a WSPD of the input data followed by the BCCP of every pair. It then runs Kruskal's algorithm to find the MST.
GeoMST2	Described in [31]. This algorithm improves on GeoMST by using marginal distances and a priority queue to avoid many BCCP computations.
Triangle	This algorithm first computes the Delaunay Triangulation of the input data, then applies Kruskal's algorithm. Triangle only works with two dimensional data.

Table 2. Point Distribution Info

Name	Description
unif	c_1 to c_d chosen from unit hypercube with uniform distribution (U^d)
annul	c_1 to c_2 chosen from unit circle with uniform distribution, $c_3 ... c_d$ chosen from U^d
arith	c_1 = 0, 1, 4, 9, 16... c_2 to c_d are 0
ball	c_1 to c_d chosen from unit hypersphere with uniform distribution
clus	c_1 to c_d chosen from 10 clusters of normal distribution centered at 10 points chosen from U^d
edge	c_1 chosen from U^d, c_2 to c_d equal to c_1
diam	c_1 chosen from U^d, c_2 to c_d are 0
corn	c_1 to c_d chosen from 2^d unit hypercubes, each one centered at one of the 2^d corners of a (0,2) hypercube
grid	n points chosen uniformly at random from a grid with $1.3n$ points, the grid is housed in a unit hypercube
norm	c_1 to c_d chosen from $(-1, 1)$ with normal distribution
spok	For each dimension d' in d $\frac{n}{d}$ points chosen with $c_{d'}$ chosen from U^1 and all others equal to $\frac{1}{2}$

The second set of graphs shows the mean total running times for two dimensional data of various distributions, as well as the standard deviation. The distributions were taken from [31] (given n d-dimensional points with coordinates $c_1 ... c_d$), shown in Table 2.

5 Experimental Results

As shown in Figure 1, GeoFK1 performs favorably in practice for almost all cases compared to other algorithms (see Table 1). In two dimensions, only Triangle outperforms GeoFK1. In higher dimensions, GeoFK1 is the clear winner when only one thread is used.

Our algorithm tends to slowdown as the dimension increases, primarily because of the increase in the number of well separated pairs [31]. For example, the number of well separated pairs generated for a two dimensional uniformly distributed data set of size 10^6 was approximately 10^7, whereas a five dimensional data set of the same size had 10^8 WSPs.

Figure 1, column 2, shows that in most cases, GeoFK1 performs better regardless of the distribution of the input point set. Apart from the fact that Triangle maintains its superiority in two dimensions, GeoFK1 performs better in all the distributions that we have considered, except when the points are drawn from *arith* distribution. In the data set from *arith*, the ordering of the WSPs based on the minimum distance is the same

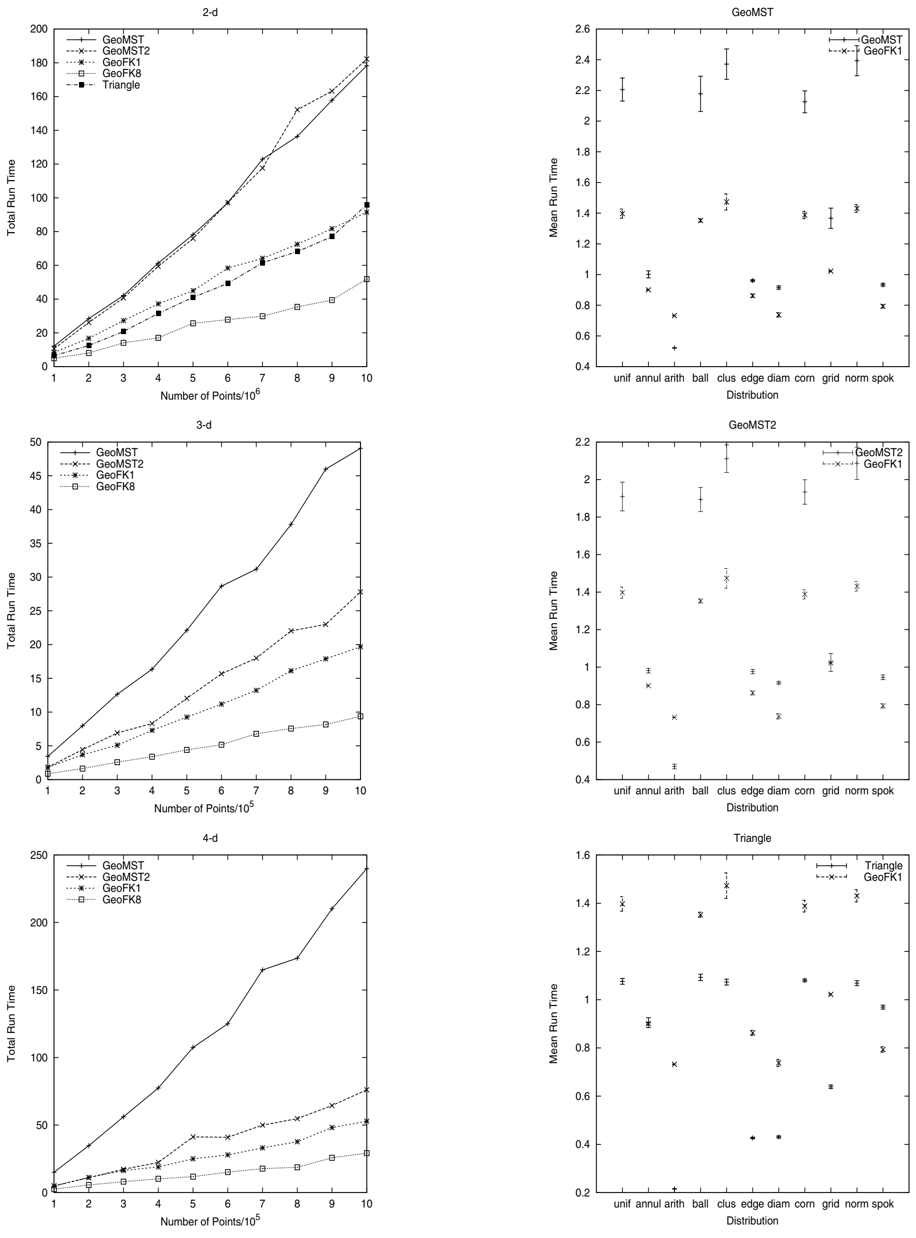


Fig. 1. The series of graphs in column 1 shows the total running time for each algorithm for varying sized data sets of uniformly random points, as the dimension increases. Data sets ranged from 10^6 to 10^7 points for 2-d data, and 10^5 to 10^6 for other dimensions. The graphs in column 2 show mean run time and standard deviation to compute the GMST on data sets of various distributions (see Table 2). The data set size was 10^6 points. For each data set size 5 tests were done and the results averaged.

as based on the Bccp distance. Hence the second partitioning step in GeoFK1 acts as an overhead. The results from Figure 1, column 2 are for two dimensional data. The same experiments for data sets of other dimensions did not give significantly different results, and so were not included.

6 Conclusions and Future Work

This paper strives to demonstrate a practical GMST algorithm, that is theoretically efficient on uniformly distributed point sets, works well on most distributions and is multi-core friendly. To that effect we introduced the GeoFilterKruskal algorithm, an efficient, parallelizable GMST algorithm that in both theory and practice accomplished our goals. We proved a running time of $\mathcal{O}(n \log n)$, as well as provided extensive experimental results.

This work raises many interesting open problems. Since the main parallel portions of the algorithm rely on partitioning and sorting, the practical impact of other parallel sort and partition algorithms should be explored. In addition, since the particular well separated pair decomposition algorithm used is not relevant to the correctness of our algorithm, the use of a tree that offers better clustering might make the algorithm more efficient. Experiments were conducted only for L_2 metric in this paper. As a part of our future work, we plan to perform experiments on other metrics. We also plan to do more extensive experiments on the k-nearest neighbor approach in higher dimensions, for example $d > 10$.

References

1. Agarwal, P.K., Edelsbrunner, H., Schwarzkopf, O., Welzl, E.: Euclidean minimum spanning trees and bichromatic closest pairs. Discrete Comput. Geom. 6(5), 407–422 (1991)
2. Arora, S.: Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. J. ACM 45(5), 753–782 (1998)
3. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. J. ACM 45(6), 891–923 (1998)
4. Barrow, J.D., Bhavsar, S.P., Sonoda, D.H.: Minimal spanning trees, filaments and galaxy clustering. MNRAS 216, 17–35 (1985)
5. Bentley, J.L., Weide, B.W., Yao, A.C.: Optimal expected-time algorithms for closest point problems. ACM Trans. Math. Softw. 6(4), 563–580 (1980)
6. Bhavsar, S.P., Splinter, R.J.: The superiority of the minimal spanning tree in percolation analyses of cosmological data sets. MNRAS 282, 1461–1466 (1996)
7. Brennan, J.J.: Minimal spanning trees and partial sorting. Operations Research Letters 1(3), 138–141 (1982)
8. Paul, B.: Callahan. Dealing with higher dimensions: the well-separated pair decomposition and its applications. PhD thesis, Johns Hopkins University, Baltimore, MD, USA (1995)
9. Chan, T.M.: Manuscript: A minimalist's implementation of an approximate nearest n eighbor algorithm in fixed dimensions (2006)
10. Chan, T.M.: Well-separated pair decomposition in linear time? Inf. Process. Lett. 107(5), 138–141 (2008)
11. Clarkson, K.L.: An algorithm for geometric minimum spanning trees requiring nearly linear expected time. Algorithmica 4, 461–469 (1989); Included in PhD Thesis

12. Connor, M., Kumar, P.: Stann library, http://www.compgeom.com/~stann/
13. Connor, M., Kumar, P.: Parallel construction of k-nearest neighbor graphs for point clouds. In: Proceedings of Volume and Point-Based Graphics, August 2008, pp. 25–32. IEEE VGTC (2008); Accepted to IEEE Transactions on Visualization and Computer Graphics (2009)
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press/McGraw-Hill (2001)
15. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE Computational Science and Engineering 5(1), 46–55 (1998)
16. Dwyer, R.A.: Higher-dimensional voronoi diagrams in linear expected time. In: SCG 1989: Proceedings of the fifth annual symposium on Computational geometry, pp. 326–333. ACM, New York (1989)
17. Erickson, J.: On the relative complexities of some geometric problems. In: Proc. 7th Canad. Conf. Comput. Geom., pp. 85–90 (1995)
18. Erickson, J.G.: Lower bounds for fundamental geometric problems. PhD thesis, University of California, Berkeley, Chair-Seidel, Raimund (1996)
19. Franceschini, G., Muthukrishnan, S.M., Pătraşcu, M.: Radix sorting with no extra space. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 194–205. Springer, Heidelberg (2007)
20. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: STOC 1984: Proceedings of the sixteenth annual ACM symposium on Theory of computing, pp. 135–143. ACM, New York (1984)
21. Morton, G.M.: A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada (1966)
22. Karger, D.R., Klein, P.N., Tarjan, R.E.: A randomized linear-time algorithm to find minimum spanning trees. Journal of the ACM 42, 321–328 (1995)
23. Klein, P.N., Tarjan, R.E.: A randomized linear-time algorithm for finding minimum spanning trees. In: STOC 1994: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, pp. 9–15. ACM, New York (1994)
24. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. In: Proc. American Math. Society, pp. 7–48 (1956)
25. Krznaric, D., Levcopoulos, C., Nilsson, B.J.: Minimum spanning trees in d dimensions. Nordic J. of Computing 6(4), 446–461 (1999)
26. Langetepe, E., Zachmann, G.: Geometric Data Structures for Computer Graphics. A. K. Peters, Ltd., Natick (2006)
27. March, W., Gray, A.: Large-scale euclidean mst and hierarchical clustering. In: Workshop on Efficient Machine Learning (2007)
28. Mencl, R.: A graph based approach to surface reconstruction. Computer Graphics Forum 14, 445–456 (2008)
29. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York (2005)
30. Motwani, R., Raghavan, P.: Randomized algorithms. ACM Comput. Surv. 28(1), 33–37 (1996)
31. Narasimhan, G., Zachariasen, M.: Geometric minimum spanning trees via well-separated pair decompositions. J. Exp. Algorithmics 6, 6 (2001)
32. Osipov, V., Sanders, P., Singler, J.: The filter-kruskal minimum spanning tree algorithm. In: Finocchi, I., Hershberger, J. (eds.) ALENEX, pp. 52–61. SIAM, Philadelphia (2009)
33. Preparata, F.P., Shamos, M.I.: Computational geometry: an introduction. Springer, New York (1985)
34. Putze, F., Sanders, P., Singler, J.: Mcstl: the multi-core standard template library. In: PPoPP 2007: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 144–145. ACM, New York (2007)

35. Rajasekaran, S.: On the euclidean minimum spanning tree problem. Computing Letters 1(1) (2004)
36. Shewchuk, J.R.: Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In: Lin, M.C., Manocha, D. (eds.) FCRC-WS 1996 and WACG 1996. LNCS, vol. 1148, pp. 203–222. Springer, Heidelberg (1996); From the First ACM Workshop on Applied Computational Geometry
37. Suraweera, F., Bhattacharya, P.: An o(log m) parallel algorithm for the minimum spanning tree problem. Inf. Process. Lett. 45(3), 159–163 (1993)
38. Zahn, C.T.: Graph-theoretical methods for detecting and describing gestalt clusters. Transactions on Computers C-20(1), 68–86 (1971)

Practical Nearest Neighbor Search in the Plane

Michael Connor and Piyush Kumar

Department of Computer Science, Florida State University
Tallahassee, FL 32306-4530
{miconnor,piyush}@cs.fsu.edu

Abstract. This paper shows that using some very simple practical assumptions, one can design an algorithm that finds the nearest neighbor of a given query point in $\mathcal{O}(\log n)$ time in theory and faster than the state of the art in practice. The algorithm and proof are both simple and the experimental results clearly show that we can beat the state of the art on most distributions in two dimensions.

Keywords: Nearest Neighbor Search, Delaunay Triangulation, Morton ordering, Randomized algorithms.

1 Introduction

Nearest neighbor search is a fundamental geometric problem important in a variety of applications including data mining, machine learning, pattern recognition, computer vision, graphics, statistics, bioinformatics, and data compression [7, 1]. Applications of the nearest neighbor problem in the plane are particularly motivated by problems in Geographic Information Systems [17].

Linear space, $\mathcal{O}(n \log n)$ pre-processing and $\mathcal{O}(\log n)$ query time algorithms are known but seem to have large constants [13, 9]. $(1+\epsilon)$-approximation algorithms seem to be faster than the exact algorithms in practice [16]. When one requires exact answers, one of the most practical algorithms available for this problem with provable guarantees is due to Devillers [10, 4]. Very recently, Birn et al. [3] have announced a very practical algorithm for this problem which beats the state of the art [16, 10] in query times but unfortunately does not have any worst case query time guarantees better than linear. In this paper, we present an algorithm which is faster in both pre-processing as well as query times compared to [3] on most distributions. We also show that if the query comes from a doubling metric, our queries are bounded by $\mathcal{O}(\log n)$ query time in expectation.

For the nearest neighbor search problem, the seminal work of Arya et al. [2] is the de facto standard for distribution independent approximate nearest neighbor search problems in fixed dimensions. Their C++ library ANN [16] is very carefully optimized both for memory access and speed, and hence has been the choice of practitioners for many years (in various areas). ANN's optimized kd-tree implementation has only recently been beaten by Birn et al. [3] in two dimensions. Birn et al. [3] also report that CGAL's recent kd-tree implementation is competitive with ANN's.

Our work is strongly related to some previous algorithms on Morton ordering [5, 6, 8], compass routing [14] and Delaunay triangulations [3].

One of the first properties we exploit in this paper is the following: We have shown in previous work [8] that if a point cloud P is shifted using a random shift, one can achieve an expected constant factor approximation for the nearest neighbor of a query point q by just locating q using a Morton order binary search in P and checking $\mathcal{O}(1)$ points around the location returned by the binary search.

Another interesting feature that we use is the fact that compass routing is supported by Delaunay Triangulations [14, 3]. It was shown by Kranakis et al. [14] that if one wants to travel between two vertices s and t of a Delaunay triangulation, one can only use local information on the current node and the coordinates of t to reach t, starting from s. The routing algorithm is simple, the next vertex visited is the one whose distance to t is minimum amongst the vertices connected to the current vertex. This type of local greedy routing has also been studied as the 'small-world phenomenon' or the 'six degrees of separation' [15]. Recently, this routing algorithm was successfully used for nearest neighbor searching [3] and we base our algorithm on this observation as well.

The key contributions of this paper are:

1. Compared to previous compass routing nearest neighbor techniques, our preprocessing speed on data sets is faster [3].
2. Compared to the state of the art, our query times are faster on most data sets [3, 16].
3. Our implementation has provable expected logarithmic query time.

The main drawbacks of our approach are:

1. The algorithm is randomized and the bounds are expected. To bound the query time, we assume that the query point has an expansion constant $\gamma = \mathcal{O}(1)$. This is described in depth in Section 3.
2. The algorithm is only suited for two dimensions because of its dependence on Delaunay Triangulations which have quadratic space complexity in dimensions greater than two.
3. We assume that each coordinate of the input points fits in a word, and operations on words like `XOR` and `MSB` can be performed in constant time [6, 11, 8].

The paper is organized as follows: In the remainder of this section, we define our notation. The next section gives the outline and description of our algorithm. It also briefly describes some of the tools that we use in the algorithm. Section 3 analyzes the computational complexity for our algorithm. Section 4 describes the experimental setup we use. Section 5 presents our experimental results. Section 6 concludes the paper.

Notation: Points are denoted by lower-case Roman letters. $\mathsf{Dist}(p, q)$ denotes the distance between the points p and q in L_2 metric. P is reserved to refer to a point set. n is reserved to refer to the number of points in P.

We write $p < q$ iff p precedes q in Morton order ($>$ is used similarly). We use p^s to denote the shifted point $p + (s, s, \ldots, s)$. $P^s = \{p^s | p \in P\}$. p_i is the i-th point in the sorted Morton ordering of the point set.

Upper-case Roman letters are reserved for sets. Scalars except for c, d, m and n are represented by lower-case Greek letters. We reserve i, j, k for indexing purposes.

$\mathsf{Ball}(c, r)$ is used to denote a ball with center c and radius r. We also use $\mathsf{Ball}(p, q)$ to represent the diametral ball of p and q. For point q, let $\mathcal{N}_q^k$ be the k points in P, closest to q. $|.|$ denotes the cardinality of a set or the number of points in P inside the geometric entity enclosed in $||$. Let, $\mathsf{nn}(p, \{\})$ return the nearest neighbor of p in a given set. Finally, $\mathsf{rad}(p, \{\})$ returns the distance from point p to the farthest point in a set.

2 The Algorithm

Algorithm 1 describes both our nearest neighbor pre-processing as well as the query algorithm. Our pre-processing algorithm essentially splits the input point set P into three layers. First the Delaunay triangulation, G, of P is computed and a maximal independent set computed on this graph. P' is the set $P \setminus \{$Maximal Independent set of $G\}$. We construct Layer 1, with points P' sorted in Morton order, Layer 2, with the Delaunay triangulation of points P', and Layer 3, with edges connecting the points in Layer 2 to a maximal independant set of points. (see Figure 1).

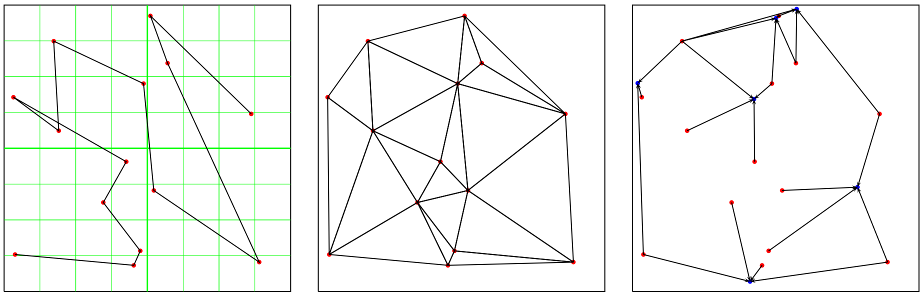
Algorithm 1. Nearest Neighbor Algorithm

Require: Randomly shifted point set P of size n. Morton order compare operator $<$.

```
 1: procedure PREPROCESS($P$)
 2:     $G = (P, E) \leftarrow$ Delaunay Triangulation of $P$
 3:     $P' = P \setminus \{$Maximal Independent set of $G\}$
 4:     $P' \leftarrow$ Sort$(P', <)$
 5:     $G' = (P', E') \leftarrow$ Delaunay Triangulation of $P'$
 6:     for all $p \in P'$ do:
 7:         $H(p) \leftarrow \{q | e = (p, q) \in G$ where $q \in P \setminus P'\}$.
 8:     for all $F$ in $\{G', H\}$
 9:         for all $v \in F$ and degree$(v) = \Omega(1)$ do:
10:             Pre-process VoronoiCell$(v, F)$ for fast lookups and jumps.
11: end procedure

12: procedure COMPASSROUTING(point $v$, point $q$, Graph $G = (P, E)$)
13:     Require: $v \in G$.
14:     repeat
15:             If degree$(v) = \Omega(1)$ then
16:                 If $q \in$VoronoiCell$(v, G)$ then return $v$
17:                 else: Update $v$ using preprocessed VoronoiCell$(v, G)$.
18:             else:
19:                 for all $v' \in G$ incident on $v$ do:
20:                     If $\mathsf{Dist}(v', q) < \mathsf{Dist}(v, q)$ then:
21:                         $v \leftarrow v'$ ; break
22:     until No improvement found
23: end procedure

24: procedure QUERY(point $q$)
25:     $i' \leftarrow$BINARYSEARCH$(P', q, <)$
26:     $p'_i \leftarrow \mathsf{nn}(q, \{p_{i'-\eta}, \ldots, p_{i'+\eta}\})$ where $\eta = \mathcal{O}(1)$
27:     $p'_j \leftarrow$COMPASSROUTING$(p'_i, q, G')$
28:     return $\mathsf{nn}(p'_j, H(p'_j))$     // Uses preprocessed VoronoiCell$(p'_j, H)$ if $|H(p'_j)| = \Omega(1)$
29: end procedure
```



(a) The first layer, consisting of the non-maximal independent set vertices sorted in Morton order. Queries are processed using a binary search to find an approximate nearest neighbor ball.

(b) The second layer, consisting of the points in the first layer, and the edges of their Delaunay triangulation. The queries are processed by using compass routing, starting at the nearest point found in the previous layer, and ending at the nearest neighbor in this layer.

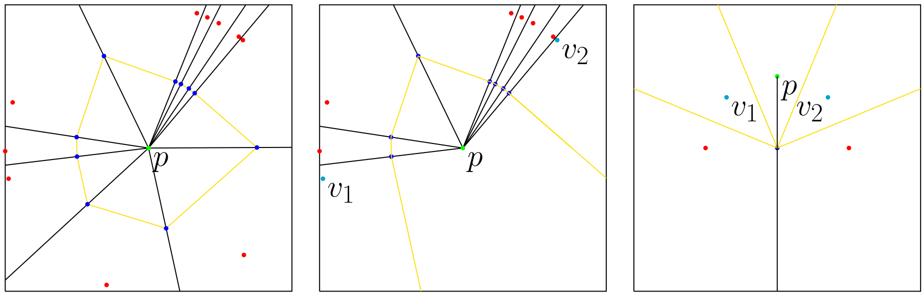
(c) The final layer, consisting of edges that connect the points in the second layer to the points in the maximal independent set. In this layer, we refine the nearest neighbor found in the previous step by scanning those points adjacent to it in this graph. This results in the final answer.

Fig. 1. The three layers of the query algorithm

Our pre-processing is complete unless there is a point p in Layer 2 or 3 with large degree ($\Omega(1)$). In these cases, we compute the vertices of the Voronoi cell of the point p along with the rays emanating from p and going through these vertices, partitioning the space around p into sectors. In order to locate a point closer to our query, we locate it in a sector, and compare the distance to a vertex found there (Figure 2). Note that any point in the plane can be located in these sectors in $\mathcal{O}(\log n)$ time using a binary search.

Our query algorithm starts by locating the query point q in the Morton ordering of points in Layer 1 (P'). Then it scans $\eta = \mathcal{O}(1)$ points around the location returned using binary search and finds the closest point p'_i to q among those points. COMPASSROUTING, which we describe next is used to find the nearest neighbor of q in Layer 2 starting from p'_i. Let this point in Layer 2 be called $p'_j \in P'$. We then find the nearest neighbor of q in Layer 3 and return the answer. We now describe how we do COMPASSROUTING and give the details of handing large degree vertices in Layers 2 and 3.

Our COMPASSROUTING algorithm is simple. Assuming there are no high degree vertices in Layer 2, it starts with a point v and selects the closest point to q that is incident on v. If there are no such vertices, it declares v as the solution, or else it jumps to the next point and repeats. In case it hits a point p that has large degree, it has access to the Voronoi rays emanating from p. It first locates q in the ray system of p in $\mathcal{O}(\log n)$ time using a binary search and then tests whether q lies in the Voronoi cell of p. If this is the case, p is returned as the answer. Otherwise the point opposite to p in the sector containing q is nearer to q compared to p and hence we jump to that point and continue routing.



(a) Here we see the center vertex p, and the sectors defined by rays passing through the Voronoi vertices. To find a nearer neighbor, we locate which sector the query lies in (via a binary search), then check the distance to the adjacent point that lies in that sector.

(b) In the first degenerate case, the center vertex has an open Voronoi cell. If the query point lies in this sector, we check the distance to the two points in the open sector $(v1, v2)$.

(c) In the second degenerate case, the center vertex is co-circular with several adjacent vertices, and there is only one Voronoi vertex. In this case, we always check the nearest co-circular points in the clockwise and counter-clockwise directions for a nearer neighbor $(v1, v2)$, and ignore the other co-circular points.

Fig. 2. The three cases for linear degree vertices

One optimization we implemented in COMPASSROUTING was to jump to any neighbor of v closer to q instead of jumping to the neighbor of v closest to q (also used by Birn et al. [3]). This fortunately does not have any effect on either the correctness or the running time analysis, but got us a slight improvement in the overall running time of the query.

In case of a vertex in Layer 3 with large degree, we again jump to the preprocessed Voronoi cell of p'_j with respect to the points in $H(p'_j)$ and use the same trick as in Layer 2. The nice property that we use in this case is that, if q does not lie in the Voronoi cell of p'_j, then the point opposite to p'_j in the sector containing q is the nearest neighbor of q, as opposed to just being a nearer neighbor. Note that using the maximal independent set does not actually improve the running time of the algorithm, it just reduces the total number of distance calculations we must do in practice.

It should be noted that there are two degenerate cases when considering the Voronoi cell of p, both of which can be resolved in constant time. The first case occurs when a sector contains an open face of the Voronoi cell (Figure 2b). This sector will contain two adjacent vertices, not one, and if q lies in such a sector, we simply compute the distance to both.

The second degenerate case occurs when an edge of the Voronoi cell of p has length 0 (Figure 2c), implying that some or all of the points bordering p are co-circular with p. In this case, we can identify in pre-processing the points immediately clockwise and counter-clockwise to p on the circle. If q is determined to lie in a sector bordering an edge with 0 length, one of those two found points must be nearer to the actual nearest neighbor than p.

3 Analysis of the Algorithm

Let P be a finite set of points in $\mathbb{R}^d$ such that $|P| = n$. For the purpose of this section, we will assume that both the query point as well as P are randomly shifted using a random point $(s, s, \ldots, s)$. Let μ be a counting measure on P. Let the measure of a ball, $\mu(\mathsf{Ball}(c, r))$ be defined as the number of points in $\mathsf{Ball}(c, r) \cap P$. A point q is said to have expansion constant γ if for all $k \in (1, n)$:

$$\mu(\mathsf{Ball}(q, 2 \times \mathsf{rad}(q, \mathcal{N}_q^k))) \leq \gamma k$$

This is a similar restriction to the doubling metric restriction on metric spaces [7, 12, 8]. Throughout the analysis, we will assume that our query point q has an expansion constant $\gamma = \mathcal{O}(1)$. Note that for finding exact nearest neighbors in $\mathcal{O}(\log n)$ time, the queries with high γ are precisely the queries which drive provable $(1+\epsilon)$-approximate nearest neighbor data structures to spend more time in computing the solution when ϵ is close to zero [16].

We first begin by defining **Compass Routing** formally (our definition is slightly different compared to [14]): Given a geometric graph $G = (P, E)$, an initial vertex $s \in G$ and a destination q (may not be in the graph), let v_i be the closest vertex in G to q. We want to travel from s to v_i, when the only information available to us at any point in time is the coordinates of our destination, our current position, and the edges incident at the vertex we are located at. Starting at s, we will traverse the edge $(s, s') \in E$ incident on s that leads us closest to q. We assign $s = s'$ and repeat this procedure till we can no longer continue decreasing the distance to q. In the next Lemma, we prove a simple property of Compass Routing on Delaunay Graphs in d-dimensions.

Lemma 3.1. *Let $P \subset \mathbb{R}^d$ and $G = (P, E)$ be the graph output from its Delaunay triangulation. Let q be a query point for which we want to compute the nearest neighbor in P. Compass routing on G yields nearest neighbor of q in P.*

Proof. Let the compass routing begin with a vertex $v_0 \in P$. Let v_i be the vertex on which compass routing stops and can not improve the distance to q. Let $\mathsf{Nbr}(v_i)$ be the set of all vertices having an edge with v_i in G.

This implies that $\mathsf{Ball}(q, \mathsf{Dist}(q, v_i))$ is empty of vertices in $\mathsf{Nbr}(v_i)$. For a contradiction, let $v^* \neq v_i$ in P be the nearest neighbor of q. Then $v^* \in \mathsf{Ball}(q, \mathsf{Dist}(q, v_i))$ and there is no edge between v^* and v_i in G.

We will now draw a ball with v_i and v^* on its boundary such that it lies inside $\mathsf{Ball}(q, \mathsf{Dist}(q, v_i))$. If this ball is empty, then $v^* \in \mathsf{Nbr}(v_i)$ which is a contradiction of the Delaunay property of the graph. Otherwise, we shrink the ball keeping it hinged on v_i and inside $\mathsf{Ball}(q, \mathsf{Dist}(q, v_i))$, till it contains only one point $v_j \in P$. This again is a contradiction since v_j is closer to q than v_i and $(v_i, v_j) \in G$ (Compass routing should not have terminated at v_i). See Figure 3. □

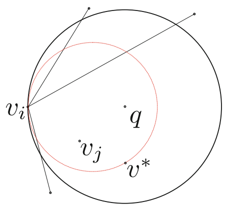


Fig. 3. Proof of Lemma 3.1

The next Lemma proves that our query Algorithm returns correct answers.

Lemma 3.2. *The* Query *function in Algorithm 1 returns the correct nearest neighbor of* q *in* P.

Proof. We mainly need to prove that the correctness of our query algorithm in not affected by separating P into three layers. Lemma 3.1 ensures that we find the nearest neighbor in Layer 2. Let this neighbor be p'_j. Hence $\mathsf{Ball}(q, \mathsf{Dist}(q, p'_j))$ is empty of points in Layer 2. If $\mathsf{Ball}(q, \mathsf{Dist}(q, p'_j))$ is empty, then p'_j will be returned as the nearest neighbor of q correctly by the Query function. Otherwise, $|\mathsf{Ball}(q, \mathsf{Dist}(q, p'_j))| = 1$ because if there were more points than 1 in this ball, there would exist a Delaunay edge between two of these points contradicting the fact that they are a maximal independent set in the Delaunay triangulation of P. Let v^* in Layer 3 be inside $\mathsf{Ball}(q, \mathsf{Dist}(q, p'_j))$ in this case. Then we can draw an empty ball passing through p'_j and v^* keeping it inside $\mathsf{Ball}(q, \mathsf{Dist}(q, p'_j))$. This means there must be a Delaunay edge connecting p'_j and v^* implying that $v^* \in H(p'_j)$ and hence the Query function must return v^* correctly. □

The following two observations help us bound the running time of our query, assuming that q has expansion constant $\gamma = \mathcal{O}(1)$:

Lemma 3.3. *In* $\mathcal{O}(\log n)$ *time,* $\mathsf{Ball}(q, r)$ *can be computed such that, in expecation,* $|\mathsf{Ball}(q, r)| = \mathcal{O}(1)$.

Proof. This follows from a lemma by Chan [6](and is explicitly proved in Lemmas 2.1-2.3 of [8]) that shows the nearest neighbor to q chosen from $\mathcal{O}(1)$ points in P adjacent to q in Morton order is contained in a box that has, in expectation, only a constant factor more points than the box containing $\mathsf{nn}(q, P)$. □

Theorem 3.4. *Given* q*, with expansion constant* $\gamma = \mathcal{O}(1)$*,* $\mathsf{nn}(q,P)$ *can be found in* $\mathcal{O}(\log n)$ *time in expectation.*

Proof. Given that P is sorted in Morton order, a binary search for q obviously takes only $\mathcal{O}(\log n)$ time. This yields a ball to be refined with only expected $\mathcal{O}(1)$ vertices of the Delaunay triangulation of P. Compass routing can therefore give us a path containing only $\mathcal{O}(1)$ vertices. Given that any vertex can be processed in $\mathcal{O}(\log n)$ time to find a nearer neighbor by using the Voronoi cell, $\mathsf{nn}(q, P)$ can be found in $\mathcal{O}(\log n)$ time in expectation. Note that splitting P in two layers, does not increase the running time because the number of points visited is still expected $\mathcal{O}(1)$ (By Lemma 3.3). □

The construction time of the algorithm is $\mathcal{O}(n \log n)$, bounded by the sorting of the input set in Morton order, as well as constructing the Delaunay graph and Voronoi graph, all of which have $\mathcal{O}(n \log n)$ running times. The maximal independent set is found in $\mathcal{O}(n)$ time.

While this proof is independent of dimension, the practicality of the algorithm is questionable in dimensions higher than two, where the Delaunay graph is not constrained to have a linear number of edges. It remains to be seen if there is a practical solution to the nearest neighbor problem using this approach in dimensions higher than two.

4 Experimental Setup

Our Delaunay nearest neighbor algorithm (DelaunayNN) was tested in practice against two algorithms. The first was ANN, the kd-tree nearest neighbor implementation from David Mount [16]. The second was our implementation of the full Delaunay hierarchy (FDH) algorithm presented by Birn et al. [3].

Experiments were conducted on a machine with dual 2.66 GHz Quad-core Intel Xeon CPUs, using a total of 4 GB DDR memory. Each core had 2 MB of total cache. The operating system used was SUSE Linux version 11.2, kernel 2.6.31.8-0.1. All source code was compiled using g++ version 4.4.1, with `-O3` enabled.

DelaunayNN was written using C++. It used the Triangle library by Shewchuk [18] to construct the Delaunay triangulation in pre-processing. In our implementation the constant η was set to the value 4, which we determined empirically to be a good value. It should also be noted that the maximum degree of any vertex for all of the tested data sets was 64, which was small enough that the Voronoi preprocessing of points was not needed for any of the experiments (in line 8 and 27 of Algorithm 1, the lower bound $\Omega(1)$ was replaced by 64). FDH was implemented using C++. It used the CGAL library [4] to construct the Delaunay hierarchy in pre-processing. In both cases, exact predicates were used to construct the Delaunay graphs. To keep a fair comparison with ANN, however, both used inexact floating point arithmetic when computing distances for queries. In all experimental cases this had no impact on the solution. For both DelaunayNN and FDH, points were stored along with edges of the graph in order to take advantage of spatial locality in the cache, at the cost of some storage efficiency. In all experiments, the nearest neighbor to the query point was found exactly. ANN used $\epsilon = 0$.

For comparison purposes, point distributions for the experiments were chosen to be the same as those used by Birn et al. [3, 10]. To recap, there are four distributions used:

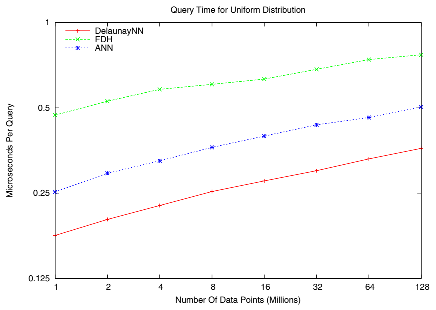
1. Data points chosen uniformly at random from the unit square. Query points chosen uniformly at random from a square 5% larger than the unit square.
2. Data points chosen uniformly at random from the unit circle. Query points chosen uniformly at random from the smallest containing square around the unit circle.
3. Data points chosen with 95% from the unit circle, 5% from the smallest square containing the unit circle. Query points chosen at random from the unit circle.
4. Data points chosen with $x = [-1000, 1000]$ and $y = x^2$. Query points chosen uniformly at random from the rectangle containing the parabola.

We chose to conduct our experiments using large data sets, to better understand the asymptotic behavior of the algorithms. For each experiment, point sets were created ranging in size from one million to 128 million points. 100,000 queries were used in each experiment. To account for randomness in the algorithms and the system, each experiment was run five times (with unique data and query sets for each), and the results were averaged. The next section describes the results and shows graphs of the run time. *Note that all graphs use a base 2 logarithmic scale*. At the end of the paper, we also include tables with the discrete timing results.

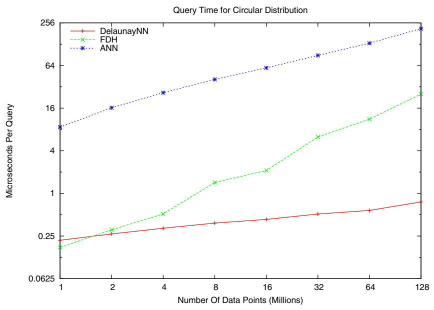
5 Experimental Results

As shown in Figure 4, our algorithm behaves very well in practice on point sets from various distributions. For data sets of sufficient size, the DelaunayNN implementation proves faster than FDH in all cases, and faster than ANN in almost all cases.

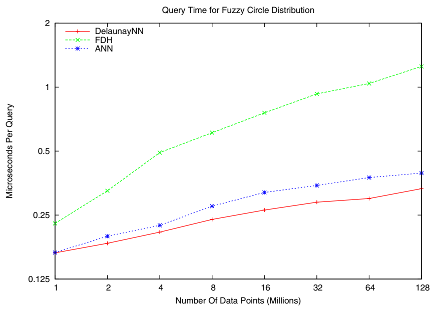
Figure 4(a) shows the results for uniform distribution, which most closely follows the bounded expansion constant considered in the analysis. All implementations performed very well, with low average query times even for very large data sets. Overall, the increase in average query time was significantly less than $\log n$ for all three implementations.



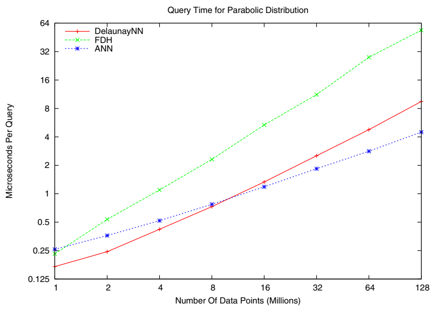
(a) Showing average time per query versus data set size for points taken from uniform distribution.



(b) Showing average time per query versus data set size for points taken from a unit circle. Query points were taken from the smallest square enclosing the circle.



(c) Showing average query time versus data set size for points taken from the unit circle, plus some points chosen uniformly at random from the square containing the circle. Query points taken from the circle.



(d) Showing average time per query versus data set size for points taken from a parabola. Query points were taken from the smallest rectangle enclosing the parabola.

Fig. 4. Average time per query on various distributions

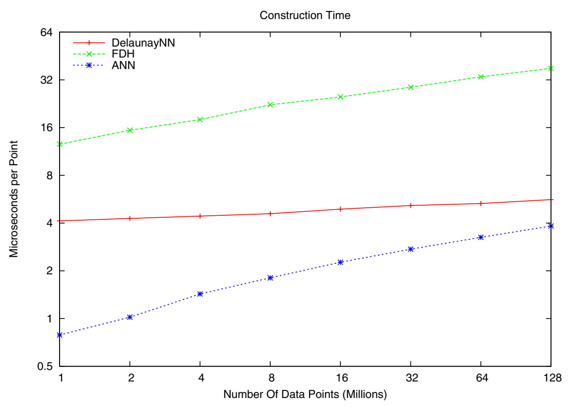


Fig. 5. Showing average time per point to pre-process data sets for queries. Data was taken uniformly at random from the unit square.

Table 1. Query Time for Uniform Dist

Data Size (millions)	ANN (μsecs)	FDH (μsecs)	DelaunayNN (μsecs)
1	0.25270	0.47128	0.17729
2	0.29390	0.52770	0.20186
4	0.32515	0.58120	0.22614
8	0.36260	0.60524	0.25335
16	0.39758	0.63178	0.27608
32	0.43566	0.68395	0.30011
64	0.46234	0.74034	0.33027
128	0.50375	0.76978	0.35981

Table 2. Query Time for Circle Dist

Data Size (millions)	ANN (μsecs)	FDH (μsecs)	DelaunayNN (μsecs)
1	8.63243	0.17260	0.21796
2	16.18790	0.30660	0.26801
4	26.42330	0.51370	0.32227
8	40.67710	1.42964	0.38098
16	59.25260	2.10402	0.42979
32	88.78520	6.26529	0.51216
64	132.12500	11.19470	0.57460
128	212.07100	25.28850	0.75866

Table 3. Query Time for Fuzzy Circle Dist

Data Size (millions)	ANN (μsecs)	FDH (μsecs)	DelaunayNN (μsecs)
1	0.16654	0.22836	0.16604
2	0.19873	0.32517	0.18411
4	0.22376	0.49217	0.20783
8	0.27521	0.61018	0.23853
16	0.31957	0.75705	0.26399
32	0.34461	0.92965	0.28739
64	0.37554	1.04076	0.29913
128	0.39424	1.25171	0.33284

Table 4. Query Time for Parabola Dist

Data Size (millions)	ANN (μsecs)	FDH (μsecs)	DelaunayNN (μsecs)
1	0.25818	0.23096	0.16939
2	0.36093	0.53571	0.24350
4	0.51931	1.09705	0.42040
8	0.77374	2.31555	0.73104
16	1.18471	5.35778	1.33284
32	1.84100	11.21760	2.52317
64	2.82095	27.85140	4.77147
128	4.49590	53.95780	9.48711

In Figure 4(b) we see that for data sets where points are distributed on a circle, the DelaunayNN displays timing results that are very similar to it's performance on uniformly distributed data, where both ANN and FDH perform substantially worse than on uniform data. This trend continues in Figure 4(d), with the exception of ANN's performance, which is again closer to its performance on uniform data.

Table 5. Pre-Processing Time for Uniform Dist

Data Set Size (millions)	ANN (μsecs)	FDH (μsecs)	DelaunayNN (μsecs)
1	0.788120	12.569800	4.126410
2	1.019810	15.401650	4.278760
4	1.428278	17.975875	4.426575
8	1.804425	22.293875	4.581163
16	2.264475	24.980688	4.900006
32	2.736934	28.750000	5.159531
64	3.250344	33.437500	5.312781
128	3.829984	37.812500	5.625000

The one anomalous case we had is documented in Figure 4(c). In this case, ANN had a marked edge in performance for larger point sets over DelaunayNN and FDH. It is also worth noting that for this type of distribution, all implementations had significantly worse scaling than on other distributions.

Figure 5 shows the difference in pre-processing time for the various implementations on uniform data. While ANN maintains a distinct advantage, DelaunayNN scales much better as the data set size increases. It is also clear that using divide and conquer approach allows for Delaunay triangulation with much more reasonable construction times, whereas FDH is forced to use the practically less efficient, incremental construction. Tables 1 to 5 show the data corresponding to the graphs.

6 Conclusions and Future Work

We have presented an algorithm for finding the nearest neighbor for a query in two dimensions that has both an expected run time bound of $\mathcal{O}(\log n)$ and strong experimental performance when compared to existing, state of the art implementations. It remains to be seen if this approach can be applied in a reasonable manner to dimensions higher than two, and if it can be extended to allow for efficient solutions to the k-nearest neighbor problem.

Acknowledgements. We would like to thank Samidh Chatterjee for discussion and suggestions regarding this paper. This research was partially supported by National Science Foundation through CAREER Grant CCF-0643593, Florida State University Committee on Faculty Research Support (COFRS) Summer Award and the Air Force Young Investigator Program.

References

[1] Arya, S., Mount, D.: Computational geometry: Proximity and location. In: Mehta, D., Sahni, S. (eds.) Handbook of Data Structures and Applications, ch. 3, pp. 63–1, 63–22. CRC Press, Boca Raton (2005)

[2] Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.: An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. J. ACM 45, 891–923 (1998)
[3] Birn, M., Holtgrewe, M., Sanders, P., Singler, J.: Simple and Fast Nearest Neighbor Search. In: 2010 Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, January 16, pp. 43–54 (2010)
[4] Boissonnat, J.-D., Devillers, O., Teillaud, M., Yvinec, M.: Triangulations in cgal (extended abstract). In: SCG 2000: Proceedings of the sixteenth annual symposium on Computational geometry, pp. 11–18. ACM, New York (2000)
[5] Chan, T.M.: Closest-point problems simplified on the ram. In: SODA 2002: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 472–473. Society for Industrial and Applied Mathematics, Philadelphia (2002)
[6] Chan, T.M.: Manuscript: A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions (2006)
[7] Clarkson, K.L.: Nearest-neighbor searching and metric space dimensions. In: Shakhnarovich, G., Darrell, T., Indyk, P. (eds.) Nearest-Neighbor Methods for Learning and Vision: Theory and Practice, pp. 15–59. MIT Press, Cambridge (2006)
[8] Connor, M., Kumar, P.: Fast construction of k-nearest neighbor graphs for point clouds. IEEE Transactions on Visualization and Computer Graphics 99 (PrePrints) (2010)
[9] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications, 2nd edn. Springer, Heidelberg (2000)
[10] Devillers, O.: The Delaunay Hierarchy. International Journal of Foundations of Computer Science 13, 163–180 (2002)
[11] Eppstein, D., Goodrich, M.T., Sun, J.Z.: The skip quadtree: a simple dynamic data structure for multidimensional data. In: Proc. of the twenty-first annual symposium on Computational geometry, pp. 296–305. ACM Press, New York (2005)
[12] Karger, D.R., Ruhl, M.: Finding nearest neighbors in growth-restricted metrics. In: STOC 2002: Proceedings of the thirty-fourth annual ACM symposium on Theory of computing, pp. 741–750. ACM, New York (2002)
[13] Kirkpatrick, D.G.: Optimal search in planar subdivisions. SIAM Journal on Computing 12(1), 28–35 (1983)
[14] Kranakis, E., Singh, H., Urrutia, J.: Compass routing on geometric networks. In: Proc. of 11th Canadian Conference on Computational Geometry, pp. 51–54 (1999)
[15] Milgram, S.: The small world problem. Psychology Today 1(1), 60–67 (1967)
[16] Mount, D.: ANN: Library for Approximate Nearest Neighbor Searching (1998), http://www.cs.umd.edu/~mount/ANN/
[17] Samet, H.: Applications of spatial data structures: Computer graphics, image processing, and GIS. Addison-Wesley Longman Publishing Co., Inc., Boston (1990)
[18] Shewchuk, J.R.: Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In: Lin, M.C., Manocha, D. (eds.) FCRC-WS 1996 and WACG 1996. LNCS, vol. 1148, pp. 203–222. Springer, Heidelberg (1996); From the First ACM Workshop on Applied Computational Geometry

Author Index

Abeledo, Hernán 202
Abraham, Ittai 23
Amaldi, Edoardo 266
Ambrosino, Daniela 314
Andonov, Rumen 106
Anghinolfi, Davide 314
Aslanyan, Hakob 386

Bar-Noy, Amotz 361
Bar-Yanai, Roni 373
Batz, Gernot Veit 166
Baumann, Frank 118
Beiko, Robert G. 141
Berger, Annabell 35
Blair, Jean 411
Boccia, Maurizio 288
Boyar, Joan 178
Brunel, Edith 47
Bruun, Asger 424
Buchheim, Christoph 118

Cánovas, Rodrigo 94
Caroli, Manuel 462
Chatterjee, Samidh 486
Cheilaris, Panagiotis 361
Chiarandini, Marco 326
Cicerone, Serafino 59
Clautiaux, François 302
Coniglio, Stefano 266
Connor, Michael 486, 501
Coughlan, Eamonn T. 226
Crainic, Teodor G. 288

D'Ambrosio, Claudia 350
D'Angelo, Gianlorenzo 59
de Castro, Pedro M.M. 462
Delling, Daniel 23, 47
Delort, Charles 253
Di Stefano, Gabriele 59
Doerr, Benjamin 190
Ďurian, Branislav 129

Edelkamp, Stefan 424

Feng, Yi 361
Ferraro-Petrillo, Umberto 1, 398
Festa, Paola 338
Finocchi, Irene 1
Frangioni, Antonio 350
Frigioni, Daniele 59
Fukasawa, Ricardo 202

Gamrath, Gerald 239
Geisberger, Robert 71, 166
Gemsa, Andreas 47
Goldberg, Andrew V. 23
Gonçalves, José F. 338
Görke, Robert 436
Grandinetti, Lucio 214
Grandoni, Fabrizio 398
Grimmer, Martin 35
Gualandi, Stefano 266
Guerriero, Francesca 214

Ha, Minh Hoang 302
Hanafi, Saïd 302
Hauser, Alain 154

Italiano, Giuseppe F. 1, 398

Katajainen, Jyrki 424
Katz, Bastian 449
Kieritz, Tim 83
Krug, Marcus 449
Kumar, Piyush 486, 501
Künnemann, Marvin 190

Laganà, Demetrio 214
Langberg, Michael 373
Leone, Pierre 386
Liberti, Leo 350
Liers, Frauke 118
Lochbihler, Andreas 449
Lodi, Andrea 350
Loriot, Sébastien 462
Lübbecke, Marco E. 226, 239
Luxen, Dennis 83

Maillard, Pascal 436
Malod-Dognin, Noël 106

Manne, Fredrik 411
Maue, Jens 154
Maurizio, Vinicio 59
Meijer, Henk 474
Müller-Hannemann, Matthias 35

Navarro, Gonzalo 94
Neubauer, Sabine 166
Núñez-Rodríguez, Yurai 474

Ochi, Luiz Satoru 276

Paolucci, Massimo 314
Pardalos, Panos M. 13
Patwary, Md. Mostofa Ali 411
Peleg, David 373
Peltola, Hannu 129
Peralta, René 178
Pessoa, Artur 202
Pisacane, Ornella 214

Rappaport, David 474
Rasmussen, Jens 424
Rebennack, Steffen 13
Resende, Mauricio G.C. 338
Roditty, Liam 373
Rolim, Jose 386
Rouiller, Olivier 462
Rutter, Ignaz 449

Salmela, Leena 129
Sanders, Peter 83, 166
Schulz, Jens 226
Sciomachen, Anna 314
Sforza, Antonio 288
Silva, Ricardo M.A. 338
Snelting, Gregor 449
Spanjaard, Olivier 253
Staudt, Christian 436
Sterle, Claudio 288
Stützle, Thomas 326
Subramanian, Anand 276

Tarhio, Jorma 129
Teillaud, Monique 462

Uchoa, Eduardo 202, 276

Vetter, Christian 83

Wagner, Dorothea 47, 436, 449
Wahlström, Magnus 190
Werneck, Renato F. 23
Whidden, Chris 141
Wilbaut, Christophe 302
Wormser, Camille 462

Yanev, Nicola 106

Zeh, Norbert 141