Carme Àlvarez
Maria Serna (Eds.)

LNCS 4007

Experimental Algorithms

5th International Workshop, WEA 2006
Cala Galdana, Menorca, Spain, May 2006
Proceedings

Lecture Notes in Computer Science 4007

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

Editorial Board

Carme Àlvarez Maria Serna (Eds.)

Experimental Algorithms

5th International Workshop, WEA 2006
Cala Galdana, Menorca, Spain, May 24-27, 2006
Proceedings

Springer

Volume Editors

Carme Àlvarez
Maria Serna
Universitat Politècnica de Catalunya
Software Department, Edifici Omega, Campus Nord
Jordi Girona 1-3, 08034 Barcelona, Spain
E-mail: {alvarez, mjserna}@lsi.upc.edu

Library of Congress Control Number: 2006926223

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

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

ISSN 0302-9743
ISBN-10 3-540-34597-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-34597-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11764298 06/3142 5 4 3 2 1 0

Preface

The Workshop on Experimental Algorithms, WEA 2006, is intended to be an international forum for research on the design, analysis and especially the implementation, evaluation and engineering of algorithms, as well as on combinatorial optimization and its applications. WEA 2006, held at Hotel Cala Galdana on Menorca, Spain, May 24–27, is the fifth of the series after Riga (2001), Monte Verita (2003), Rio de Janeiro (2004), and Santorini (2005).

This volume contains all contributed papers accepted for presentation, together with invited lectures by Ricardo Baeza-Yates (Yahoo! Research), Jon Bentley (Avaya Labs Research), and Sotiris Nikoletseas (University of Patras and Computer Technology Institute). The 26 contributed papers were selected out of 92 submissions received in response to the call for papers. All the papers published in the proceedings were selected by the Program Committee on the basis of at least three referee reports, with the help of trusted external referees.

We would like to thank all of the authors who responded to the call for papers, our invited speakers, and the members of the Program Committee, as well as the external referees, and the Organizing Committee members.

We gratefully acknowledge support from the Ministry of Education of Spain and the Technical University of Catalonia.

March 2006

Carme Àlvarez
Maria Serna

Organization

Program Committee

Maria Serna (Chair)	T.U. of Catalonia (Spain)
Mark de Berg	T.U. Eindhoven (The Netherlands)
Christian Blum	T.U. of Catalonia (Spain)
Aaron Clauset	University of New Mexico (USA)
Camil Demetrescu	University of Rome "La Sapienza" (Italy)
Thomas Erlebach	University of Leicester (UK)
Catherine McGeoch	Amherst College (USA)
Ulrich Meyer	Max-Plack-Institut fr Informatik (Germany)
Ian Munro	University of Waterloo (Canada)
Stefan Nher	Universitt Trier (Germany)
Gonzalo Navarro	Universidad de Chile (Chile)
Panos M. Pardalos	University of Florida (USA)
Jose Rolim	University of Geneva (Switzerland)
Adi Rosn	Technion (Israel)
Peter Sanders	Universitt Karlsruhe (Germany)
Guilhem Semerjian	University of Rome "La Sapienza" (Italy)
Paul Spirakis	University of Patras and CTI (Greece)
Osamu Watanabe	Tokyo Institute of Technology (Japan)
Peter Widmayer	ETH Zrich (Switzerland)

Steering Committee

Edoardo Amaldi	Politecnico di Milano (Italy)
David A. Bader	Georgia Institute of Technology (USA)
Josep Diaz	T.U. of Catalonia (Spain)
Giuseppe Italiano	Universit di Roma "Tor Vergata" (Italy)
David Johnson	AT&T Labs (USA)
Klaus Jansen	Universitt Kiel (Germany)
Kurt Mehlhorn	Max-Plack-Institut fr Informatik (Germany)
Ian Munro	University of Waterloo (Canada)
Sotiris Nikoletseas	University of Patras and CTI (Greece)
Jos Rolim (Chair)	University of Geneva (Switzerland)
Paul Spirakis	University of Patras and CTI (Greece)

Organizing Committee

Maria Serna (Chair)	T.U. of Catalonia (Spain)
Carme lvarez	T.U. of Catalonia (Spain)
Maria Blesa	T. U. of Catalonia (Spain)

Amalia Duch	T. U. of Catalonia (Spain)
Leonor Frias	T. U. of Catalonia (Spain)
Helena Nešetřilov	Czech U. of Agriculture (Czech Republic)

Referees

L. Anderegg
S. Angelopoulos
R. Aringhieri
D. Arroyuelo
R. Baeza-Yates
M. Baur
C. Becker
J.M. Bilbao Arrese
M. Bouget
S. Busygin
I. Caragiannis
I. Chatzigiannakis
A. Chinculuun
Y. Chiricota
R.A. Chowdhury
C. Commander
A. Clementi
F. Ducatelle
P. Fatourou
H. Fernau
K. Ferreira
K. Figueroa
I. Finocchi
D. Fotakis
G. Franceschini
P. Franciosa
K. Fredriksson
M. Gastner
S. Gomez
R. Gonzlez
R. Grossi
C. Gutwenger
D. Hay
M.J. Hirsch
P. Holme
T. Holotyak
J. Horey
J. Iacono
A. Jarry
A. Kammerdiner
J. Karkkainen
J.F. Karlin
A. Kinalis
S. Kontogiannis
P. Kumar
S. Langerman
K. Lansey
P. Leone
Z. Lotker
M. Luebbecke
M. Lukac
H. Lundgren
H. Maier
V. Makinen
M. Min
R. Monasson
L. Moraru
K. Nakata
R. Naujoks
D. Niculescu
S. Nikoletseas
K. Paluch
A. Panconesi
R. Paredes
M. Patella
S. Pettie
D. Phillips
A. Plastino
O. Powell
O. Prokopyev
K. Pruhs
M. Ragle
D. Raz
A. Ribichini
F. Ricci-Tersenghi
M. Ruhl
C. Santos Badue
E. Schiller
R. Schultz
O. Seref
O. Shylo
I. Stamatiu
J.C. Tournier
M. Varvarigos
A. Vitaletti
S. Voss
I. Weber
M. Weigt
S. Winkel
P. Woelfel
E. Wolf-Chambers
K. Yamaoka
M. Yamashita
M. Young
N. Zeh
G. Zhang

Table of Contents

Session 1

Session 2

Session 3

Session 4

Session 5

Session 6

Session 7

Session 8

Session 9

Session 10

Session 11

Session 12

Algorithms for Wireless Sensor Networks: Design, Analysis and Experimental Evaluation*

Sotiris Nikoletseas

Department of Computer Engineering and Informatics,
University of Patras, and CTI, Greece
nikole@cti.gr

Abstract. The efficient and robust realization of wireless sensor networks is a challenging technological and algorithmic task, because of the unique characteristics and severe limitations of these devices. This talk presents representative algorithms for important problems in wireless sensor networks, such as data propagation and energy balance. The protocol design uses key algorithmic techniques like randomization and local optimization. Crucial performance properties of the protocols (correctness, fault-tolerance, scalability) and their trade-offs are investigated through both analytic means and large scale simulation. The experimental evaluation of algorithms for such networks is very beneficial, not only towards validating and fine-tuning algorithmic design and analysis, but also because of the ability to study the accurate impact of several important network parameters and technological details.

1 Introduction

Recent dramatic developments in micro-electro-mechanical (MEMS) systems, wireless communications and digital electronics have already led to the development of small in size, low-power, low-cost sensor devices. Such extremely small devices integrate sensing, data processing and wireless communication capabilities. Current devices have a size at the cubic centimeter scale, a CPU running at 4 MHz, some memory and a wireless communication capability at a 4Kbps rate. Also, they are equipped with a small but effective operating system and are able to switch between "sleeping" and "awake" modes to save energy.

Their wide range of applications is based on the possible use of various sensor types (i.e. thermal, visual, seismic, acoustic, radar, magnetic, etc.) to monitor a wide variety of conditions (e.g. temperature, object presence and movement, humidity, pressure, noise levels etc.). Thus, sensor networks can be used for continuous sensing, event detection, location sensing as well as micro-sensing. Hence, sensor networks have important applications, including (a) environmental (such as fire detection, flood detection, precision agriculture), (b) health applications

* This work was partially supported by the IST Programme of the European Union under contact number IST-2005-15964 (AEOLUS) and the Programme PENED of GSRT under contact number 03ED568.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 1–12, 2006.

(like telemonitoring of human physiological data), (c) home applications (e.g. smart environments and home automation) and (d) military/security applications. For a survey of wireless sensor networks see [1].

Because of their rather unique characteristics, efficient and robust distributed protocols and algorithms should exhibit the following critical properties: **a) Scalability.** Distributed protocols for sensor networks should be highly scalable, in the sense that they should operate efficiently in extremely large networks composed of huge numbers of nodes. **b) Efficiency.** Because of the severe energy limitations of sensor networks and also because of their time-critical application scenaria, protocols for sensor networks should be efficient, with respect to both energy and time. **c) Fault-tolerance.** Sensor particles are prone to several types of faults and unavailabilities, and may become inoperative (permanently or temporarily). The sensor network should be able to continue its proper operation for as long as possible despite the fact that certain nodes in it may fail.

Since one of the most severe limitations of sensor devices is their limited energy supply, one of the most crucial goals in designing efficient protocols for wireless sensor networks is minimizing the energy consumption in the network. This goal has various aspects, including: (a) minimizing the total energy spent in the network (b) minimizing the number (or the range) of data transmissions (c) combining energy efficiency and fault-tolerance, by allowing redundant data transmissions which however should be optimized to not spend too much energy (d) maximizing the number of "alive" particles over time, thus prolonging the system's lifetime and (e) balancing the energy dissipation among the sensors in the network, in order to avoid the early depletion of certain sensors and thus the breakdown of the network.

We note that it is very difficult to achieve all the above goals at the same time. There even exist trade-offs between some of the goals above. Furthermore, the importance and priority of each of these goals may depend on the particular application. Thus, it is important to have a variety of protocols (and hybrid combinations of protocols), each of which may possibly focus at some of the energy efficiency goals above (while still performing well with respect to the rest goals). Furthermore, there exist fundamental, inherent trade-offs between important performance measures, most notably between energy dissipation and latency (i.e. time for information to get to the control center).

In the light of the above, we present and evaluate several data propagation protocols: a) *The Directed Diffusion (DD) Protocol*, that creates and maintains some global structure (e.g. a set of paths) to collect data. b) *The Low Energy Adaptive Clustering Hierarchy (LEACH) Protocol*, that uses clustering to handle data collectively and reduce energy. c) *The Local Target Protocol (LTP)*, that performs a local optimization trying to minimize the number of data transmissions. d) *The Probabilistic Forwarding Protocol (PFR)*, that creates redundant data transmissions that are probabilistically optimized, to trade-off energy efficiency with fault-tolerance. e) *The Energy Balanced Protocol (EBP)*, that focuses on guaranteeing the same per sensor energy dissipation, in order to prolong the lifetime of the network.

Through both rigorous mathematical means and performance evaluation of implemented protocols, we demonstrate the strengths, weaknesses and trade-offs of the protocols and indicate the network conditions and dynamics for which each protocol is best suitable. We believe that a complementary use of rigorous analysis and large scale simulations is needed to fully investigate the performance of protocols in wireless sensor networks. In particular, asymptotic analysis may lead to provable efficiency and robustness guarantees towards the desired scalability of protocols for sensor networks that have extremely large size. On the other hand, protocol implementation allows to investigate the detailed effect of a great number of technical specifications of real devices, a task that is difficult (if possible at all) for analytic techniques which, by their nature, use abstraction and model simplicity.

The definition of abstract (yet realistic) models for wireless sensor networks is very important, since it enables rigorous mathematical analysis of protocol performance. Such models include: a) random geometric graphs [8, 17], where a random plane network is constructed by picking points (that abstract sensors) in the plane by a Poisson process, with density d points per unit area, and then joining each pair of points by a line if they are at distance less than r (this captures transmission range). Interesting properties under this model are investigated in [5]. b) Another interesting model is that of random sector graphs, where each randomly chosen point (sensor) in the plane chooses an angle and a euclidean distance (that together define a cyclic sector corresponding to the sensor's transmission area [6]). Interesting properties (connectivity, chromatic number) are investigated in [18]. c) Stochastic models (such as Markov Chains, dynamic systems) like the ones in [12, 13] are particularly useful for capturing energy dissipation and data propagation. A new relevant model is that of random intersection graphs, where each vertex randomly picks elements from a universe, and two vertices are adjacent when they pick at least one element in common ([11]). Independence properties and algorithms are proposed in [15].

2 Representative Protocols

Directed Diffusion: Maintaining Sets of Paths. Directed Diffusion (DD) [10] is a data-centric communication paradigm, a suite of several protocols. In general, it requires some coordination between sensors to create and maintain a somewhat global structure (e.g. a set of paths) for propagating data. DD uses four elements: a) interest messages, issued by the control center, containing attribute-value pairs, specifying data matching the attributes. b) Gradients towards the control center, created when receiving interest messages, storing a direction towards the sink and a value (data rate) for "pulling down" data. c) Data messages, created by the relevant sensors to the task description (as contained in the interest messages). d) Reinforcements of gradients (i.e. favoring one or more neighbors at each level of the tree) to select "best" paths (wrt some criteria) for "drawing down" real data. Reinforcement can be positive (i.e. reinforce the neighbor that first reported a new event or the one with the higher

data rate) or negative (i.e. when a gradient does not deliver any new messages for some time or when its data rate is low).

Since DD reinforces certain "good" paths for getting data, it improves over flooding a lot. Especially in the case where the network conditions do not change a lot, it incurs significant energy savings. In networks of high dynamics however (where many changes happen in the network) its performance drops, since established paths may now become inefficient (or even break down), since path maintenance and update may be too slow wrt the changes in the network.

LEACH: A Clustering Protocol. LEACH [9] partitions the network into clusters of sensors, with a single sensor in each cluster being a cluster-head. Non cluster-heads transmit data to their cluster-head; cluster-heads gather received data, compress it and send it directly to the sink. To avoid energy depletion of cluster-head sensors, clusters are created in a dynamic way over time, and cluster-heads rotate in a randomized way.

Because of compression and aggregation of data at cluster-heads and collective transmission to the sink, LEACH manages to reduce energy dissipation, especially in small area networks. In large networks however, directed transmissions are distant and expensive. Also when the network traffic is high (i.e. many agents are sensed and reported) the performance of LEACH drops, since rotation of cluster-heads may be slow and not avoid energy depletion of cluster-heads.

LTP: A Hop-by-Hop Data Propagation Protocol. The LTP Protocol was introduced in [2]. The authors adopt a two-dimensional (plane) framework: A *smart dust cloud* (a set of particles) is spread in an area. Let d (usually measured in numbers of $particles/m^2$) be the *density* of particles in the area. Let $\mathcal{R}$ be the maximum (radio/laser) transmission range of each grain particle. A *receiving wall* $\mathcal{W}$ is defined to be an infinite line in the smart-dust plane. Any particle transmission within range $\mathcal{R}$ from the wall $\mathcal{W}$ is received by $\mathcal{W}$. The wall represents in fact the authorities (the fixed control center) who the realization of a crucial event should be reported to. The wall notion generalizes that of the sink and may correspond to multiple (and/or moving) sinks. Each smart-dust particle is aware of the general location of $\mathcal{W}$.

Let $d(p_i, p_j)$ the distance (along the corresponding vertical lines towards $\mathcal{W}$) of particles p_i, p_j and $d(p_i, \mathcal{W})$ the (vertical) distance of p_i from $\mathcal{W}$. Let *info($\mathcal{E}$)* the information about the realization of the crucial event $\mathcal{E}$ to be propagated. Let p the particle sensing the event and starting the execution of the protocol. In this protocol, each particle p' that has received *info($\mathcal{E}$)*, does the following:

- *Search Phase*: It uses a periodic low energy directional broadcast in order to discover a particle nearer to $\mathcal{W}$ than itself. (i.e. a particle p'' where $d(p'', \mathcal{W}) < d(p', \mathcal{W})$).
- *Direct Transmission Phase*: Then, p' sends *info($\mathcal{E}$)* to p''.
- *Backtrack Phase*: If consecutive repetitions of the *search phase* fail to discover a particle nearer to $\mathcal{W}$, then p' sends *info($\mathcal{E}$)* to the particle that it originally received the information from.

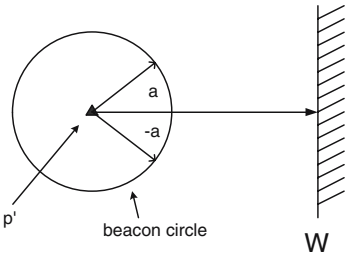


Fig. 1. Example of the Search Phase

Definition 1. Let h_{opt} be the (optimal) number of "hops" (direct, vertical to $\mathcal{W}$ transmissions) needed to reach the wall, in the *ideal* case in which particles always exist in pair-wise distances $\mathcal{R}$ on the vertical line from p to $\mathcal{W}$. Let h be the actual number of hops (transmissions) taken to reach $\mathcal{W}$. The *"hops" efficiency* of the protocol is the ratio $C_h = \frac{h}{h_{opt}}$.

Clearly, the number of hops (transmissions) needed characterizes the energy consumption and the time needed to propagate the information $\mathcal{E}$ to the wall. Remark that $h_{opt} = \left\lceil \frac{d(p,\mathcal{W})}{\mathcal{R}} \right\rceil$, where $d(p, \mathcal{W})$ is the (vertical) distance of p from the wall $\mathcal{W}$. In the case where the protocol is randomized, or in the case where the distribution of the particles in the cloud is a random distribution, the number of hops h and the efficiency ratio C_h are random variables and one wishes to study their expected values.

Towards a rigorous analysis of the protocol, [2] makes the following simplifying assumption: *The search phase always finds a* p'' (of sufficiently high battery) in the semicircle of center the particle p' currently possessing the information about the event and radius R, in the direction towards $\mathcal{W}$. Note that this assumption on always finding a particle can be relaxed in many ways. [2] also assumes that the position of p'' is uniform in the arc of angle $2a$ around the direct line from p' vertical to $\mathcal{W}$. It is also assumed that each target selection is stochastically *independent* of the others, in the sense that it is always drawn uniformly randomly in the arc $(-\alpha, \alpha)$. By analysing the stochastic process of data propagation, the following can be obtained:

Lemma 1 ([2]). *The expected "hops efficiency" of the local target protocol in the a-uniform case is*

$$E(C_h) \simeq \frac{\alpha}{\sin \alpha}$$

for large h_{opt}. *Also, for* $0 \leq \alpha \leq \frac{\pi}{2}$, *it is* $1 \leq E(C_h) \leq \frac{\pi}{2} \simeq 1.57$.

PFR: A Probabilistic Forwarding Protocol. To combine energy efficiency and fault-tolerance, the Probabilistic Forwarding Protocol (PFR) has been introduced in [3]. The modeling assumptions made can be found in [3]. Notice that GPS information is not needed for this protocol. Also, there is no need to know the global structure of the network.

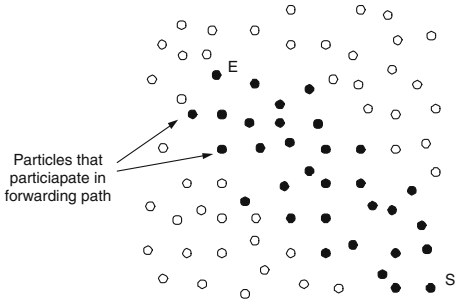


Fig. 2. Thin Zone of particles

The basic idea of the protocol lies in probabilistically favoring transmissions towards the sink within a *thin zone* of particles around the line connecting the particle sensing the event $\mathcal{E}$ and the sink (see Figure 2). Note that transmission along this line is energy optimal. The protocol evolves in two phases:

Phase 1: The "Front" Creation Phase. Initially the protocol builds (by using a limited, in terms of rounds, flooding) a sufficiently large "front" of particles, to guarantee the survivability of the data propagation process. During this, each particle having received the data to be propagated, deterministically forwards them towards the sink.
Phase 2: The Probabilistic Forwarding Phase. Each particle P possessing the information under propagation, calculates an angle ϕ and broadcasts $info(\mathcal{E})$ to all its neighbors with probability $\mathbb{P}_{fwd}$ (or it does not propagate any data with probability $1 - \mathbb{P}_{fwd}$) defined as follows:

$$\mathbb{P}_{fwd} = \begin{cases} 1 & \text{if } \phi \geq \phi_{threshold} \\ \frac{\phi}{\pi} & \text{otherwise} \end{cases}$$

where ϕ is the angle defined by the line EP and the line PS and $\phi_{threshold} = 134^o$.

The closer to the optimal line a particle lies, the larger is its angle and thus the higher is its probability to forward data. By occupancy arguments and geometry, correctness is shown:

Lemma 2 ([3]). PFR *succeeds with probability 1* in sending the information from E to S given a sufficiently high density of sensors in the network.

By estimating (using stochastic processes) the size of the area of particles activated by the protocol, the following energy efficiency result is obtained:

Theorem 1 ([3]). The energy efficiency of the PFR protocol is $\Theta\left(\left(\frac{n_0}{n}\right)^2\right)$ where $n_0 = |ES|$ and $n = \sqrt{N}$, where N is the number of particles in the network. For $n_0 = |ES| = o(n)$, this is $o(1)$.

Finally, using geometry, it is shown that PFR is quite fault tolerant:

Lemma 3 ([3]). PFR manages to propagate data across lines parallel to ES, and of constant distance, with *fixed* nonzero probability.

EBP: An Energy Balance Protocol. Most data propagation techniques do not *explicitly* take care of the possible overuse of certain sensors in the network. As an example, remark that in hop-by-hop transmissions towards the sink, the sensors lying closer to the sink tend to be utilized exhaustively (since all data passes through them). Thus, these sensors may die out very early, thus resulting to network collapse although there may be still significant amounts of energy in the other sensors of the network. Similarly, in clustering techniques the cluster-heads that are located far away with respect to the sink, tend to spend a lot of energy.

A protocol trying to balance energy dissipation among the sensors in the network; the EBP (Energy Balance) protocol, introduced in [7], probabilistically chooses between either propagating data one hop towards the sink or sending directly to the sink. The first choice is more energy efficient, while the latter bypasses the critical (close to the sink) sectors. The appropriate probability for each choice in order to achieve energy balance is calculated in [7]. Due to lack of space, we will present EBP during the talk.

3 Experimental Framework

For the purpose of the comparative study of protocols, we have designed a new simulator, which we named **simDust**, that was implemented in `Linux` using `C++` and the `LEDA` [14] algorithmic and data structures library.

An interesting feature of our simulator, is the ability to experiment with very large networks. **simDust** enables protocol implementation using just `C++`. Additionally, it generates all necessary statistics based on a large variety of metrics that are implemented (such as delivery percentage, energy consumption, delivery delay, longlivety, etc.). **simDust** operates in descrete time rounds and measures energy dissipation (in various operation modes) in detail.

On each execution of the experiment, let K be the total number of crucial events $(\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_K)$ and k the number of events that were *successfully* reported to the sink $\mathcal{S}$. We below provide some definitions.

Definition 2. The success rate, $\mathbb{P}_s$, is the fraction of the number of events *successfully* propagated to the sink over the *total number of events*, i.e. $\mathbb{P}_s = \frac{k}{K}$. Let E_i be the available energy for the particle i. Then $E_{avg} = \frac{\sum_i^n E_i}{n}$ is the average energy per particle in the smart dust network, where n is the number of the total particles dropped. Let h_A (for "active") be the *number of "active" sensor particles* participating in the sensor network. Let I_s be the *injection rate*, measured as the probability of occurrence of a crucial event during a round.

4 An LTP vs PFR Comparison

Due to lack of space, we have present only a few comparison results. We compare three versions of LTP and PFR. We generate a variety of sensor fields in a 100m

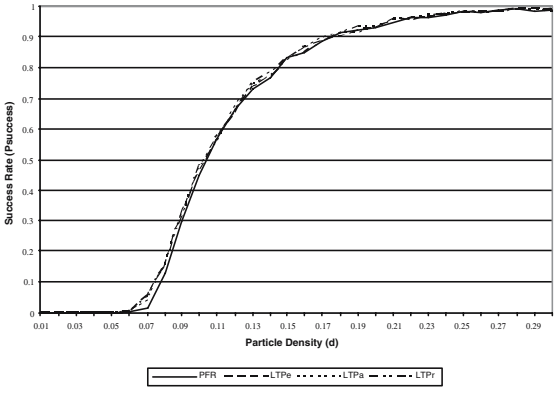


Fig. 3. Success Probability ($\mathbb{P}_s$) over particle density $d = [0.01, 0.3]$

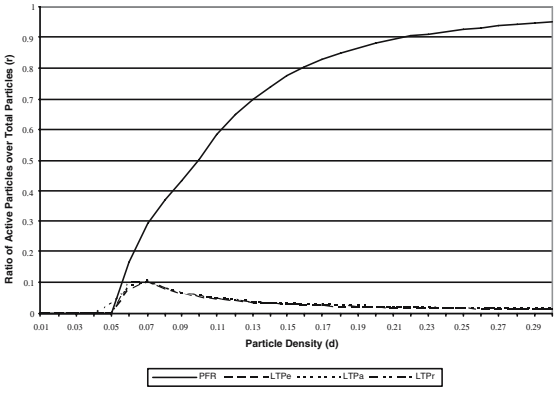


Fig. 4. Ratio of Active Particles over Total Particles (r) over particle density $d = [0.01, 0.3]$

by 100m square. In these fields, we drop $n \in [100, 3000]$ particles randomly uniformly distributed on the smart-dust plane, i.e. for densities $0.01 \leq d \leq 0.3$. Each smart dust particle has a fixed radio range of $\mathcal{R} = 5m$ and $\alpha = 90^o$. The particle p that initially senses the crucial event is always explicitly positioned at $(x, y) = (0, 0)$ and the sink is located at $(x, y) = (100, 100)$. We repeated each experiment for more than 5,000 times in order to achieve good average results.

We start by examining the success rate of the four protocols (see Fig. 3), for different densities. Initially, when the density is low (i.e. $d \leq 0.06$), the protocols fail to propagate the data to the sink. However as the density increases, the success rate increases quite fast and for high densities, all four protocols almost always succeed in propagating the data.

Figure 4 depicts the ratio of active particles over the total number of particles ($r = \frac{h_A}{n}$). In this figure we clearly see that PFR, for low densities (i.e. $d \leq 0.07$), indeed activates a small number of particles (i.e. $r \leq 0.3$) while the ratio (r)

increases as the density of the particles increases. The LTP based protocols seem more efficient and the ratio r seems to be independent of the total number of particles (because only one particle in each hop becomes active).

5 A LEACH vs PFR Comparison

Again, due to lack of space, we only present a few comparison results. We evaluate the performance of **H-TEEN** (a hierarchical extension of LEACH) and **SW-PFR** (an extension where sensors alternate between sleep and awake modes of operation to save energy). In our experiments, we generate a variety of sensor fields. The field size ranges from 200m by 200m to 1500m by 1500m. In these fields, we drop $n \in [500, 3000]$ particles randomly uniformly distributed on the smart-dust plane.

We start by examining the success rate of the protocols wrt the network size (Figure 5), for two different injection rates I_s (0.05 and 0.8). We focus on *extreme* values to investigate the divergent behavior in extreme settings. Initially, for low injection rates, in small networks (500mx500m), both protocols behave almost optimally achieving a success rate close to one and decrease as injection rate increases. However, for larger network size the impact of the injection rate seems to be more significant. In particular, **H-TEEN**'s success rate drops from 85% to almost 30%, while **PFR**, even though its initial success rate is about 70%, seems to be less affected by the increase in injection rate.

The apparent dependence of **TEEN** protocol's performance from the injection rate is due to its clustering scheme. As injection rate increases a cluster head is responsible for delivering more events, thus it consumes more energy during its leadership. If injection rate becomes too high, cluster heads are more likely to exhaust their energy supplies before a new cluster head is elected. When a cluster head "dies", the cluster ceases to function and all events that occur on that cluster are lost until a new cluster head is elected. Furthermore large

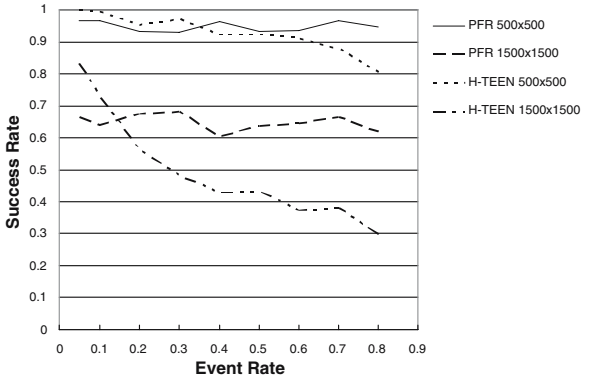


Fig. 5. Success rate for various injection rates and network area 500x500, 1500x1500

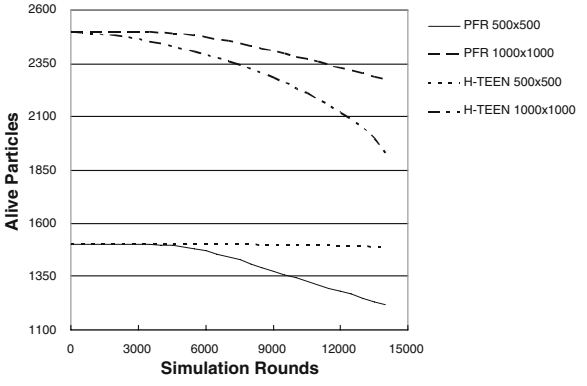


Fig. 6. Number of alive particles for $I_s = 0.05$ and network size 500x500, 1000x1000

network sizes worsen this phenomenon because more energy is required from the cluster head to propagate an event to the sink (since the energy spent in one hop is of the order of the square of the transmission distance). On the other hand the **SW-PFR** protocol is mostly unaffected by high injection rates but influenced by larger network sizes. This is due to its multihop nature, since more hops are required for the propagation of an event.

We move on by examining the average energy consumed per particle in time (see full paper). Finally, we examine the way the number of alive particles varies with time (see figure 6).

We notice that for the **H-TEEN** protocol for both network sizes the number of alive nodes decreases at a constant rate, whilst for the **SW-PFR** protocol there is a sudden decrease in the number of alive particles. This observation depicts the property of the **H-TEEN** protocol to evenly distribute the energy dissipation to all the particles in the network. On the contrary, the **SW-PFR** protocol stresses more the particles which are placed closer to the sink, so at a point in time these particles start to "die" rapidly.

On the other hand, in the larger network area (1000m x 1000m) the particles in **H-TEEN** protocol "die" more rapidly than those of **SW-PFR**. This was expected because in **H-TEEN** protocol particles are forced to transmit in larger distances than those in **SW-PFR**, so they consume more energy and "die" faster. We should also notice the same behavior of **SW-PFR** as in fig. 6, from a point in time and on, particles start to die faster than before and this is because of the fact that **SW-PFR** stresses more the particles that lie near the sink forcing them to propagate the majority of the messages that occur in the smart-dust network.

6 Future Directions

Wireless sensor networks constitute a new fascinating field, where complementary approaches (algorithms, systems, applications and technology) are needed.

From an algorithmic perspective, new abstract models and model extensions are needed (hiding details but still being realistic) to enable the necessary performance analysis of distributed algorithms – occasionally even asymptotic analysis. The interplay of geometry, graph theory and randomness create many challenging problems for rigorous treatment ([16]). Inherent trade-offs, lower bounds and impossibility results should be further investigated towards properly guiding technological efforts by pointing out inherent limitations. New efficient but simple algorithms should be designed and analysed and paradigms should be established. At the system level, versatile network stacks and lightweighted operating systems and middleware are needed. A complementary use of rigorous means and performance evaluation using experimental implementation of algorithms is highly beneficial. Experimental algorithmics are especially useful to precisely evaluate the crucial impact of several network parameters and technological details.

References

1. I.F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci: Wireless sensor networks: a survey. In the Journal of Computer Networks, Volume 38, pp. 393-422, 2002.
2. I. Chatzigiannakis, S. Nikoletseas and P. Spirakis: Smart Dust Protocols for Local Detection and Propagation. Distinguished Paper. In *Proc. 2nd ACM Workshop on Principles of Mobile Computing* – POMC'2002, pp. 9-16. Also, accepted in the *ACM Mobile Networks (MONET) Journal, Special Issue on Algorithmic Solutions for Wireless, Mobile, Adhoc and Sensor Networks*, 10:1, February 2005.
3. I. Chatzigiannakis, T. Dimitriou, S. Nikoletseas and P. Spirakis: A Probabilistic Algorithm for Efficient and Robust Data Propagation in Smart Dust Networks. In the Proceedings of the 5th European Wireless Conference on Mobile and Wireless Systems beyond 3G (EW 2004), 2004. Also, in the Journal of Ad-Hoc Networks, 2005.
4. I. Chatzigiannakis, T. Dimitriou, M. Mavronicolas, S. Nikoletseas and P. Spirakis: A Comparative Study of Protocols for Efficient Data Propagation in Smart Dust Networks. In *Proc. International Conference on Parallel and Distributed Computing* – EUPOPAR 2003. Also in the *Parallel Processing Letters (PPL) Journal*, 2004.
5. J. Diaz, M. Penrose, J. Petit and M. Serna: Approximation Layout Problems on Random Geometric Graphs. J. of Algorithms, 39:78-116, 2001.
6. J. Diaz, J. Petit and M. Serna: A Random Graph Model for Optical Networks of Sensors. In J. of IEEE Transactions on Mobile Computing, Vol. 2, Nr. 3, 2003.
7. H. Euthimiou, S. Nikoletseas and J. Rolim: Energy Balanced Data Propagation in Wireless Sensor Networks. In Proc. 4th International Workshop on Algorithms for Wireless, Mobile, Ad-Hoc and Sensor Networks (WMAN '04), IPDPS 2004, 2004. Also, in the Journal of Wireless Networks (WINET), 2005.
8. E. N. Gilbert: Random Plane Networks. J. Soc. Ind. Appl. Math 9 (4) 533-543, 1961.
9. W. R. Heinzelman, A. Chandrakasan and H. Balakrishnan: Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proc. 33rd Hawaii International Conference on System Sciences* – HICSS'2000.

10. C. Intanagonwiwat, R. Govindan and D. Estrin: Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proc. 6th ACM/IEEE International Conference on Mobile Computing* – MOBICOM'2000.
11. M. Karoński, E.R. Scheinerman and K.B. Singer-Cohen: On Random Intersection Graphs: The Subgraph Problem. Combinatorics, Probability and Computing journal (1999) 8, 131-159.
12. P. Leone and J. Rolim: Towards a Dynamical Model for Wireless Sensor Networks. *In Proceedings of ALGOSENSORS 06*, 2006.
13. P. Leone, S. Nikoletseas and J. Rolim: An Adaptive Blind Algorithm for Energy Balanced Data Propagation. *In Proceedings of DCOSS 05*, 2005.
14. K. Mehlhorn and S. Näher: LEDA: A Platform for Combinatorial and Geometric Computing. *Cambridge University Press*, 1999.
15. S. Nikoletseas, C. Raptopoulos and P. Spirakis: The Existence and Efficient Construction of Large Independent Sets in General Random Intersection Graphs. *In ICALP 2004. Also in TCS Journal.*
16. C. Papadimitriou: Algorithmic Problems in Ad Hoc Networks. In IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 05), Springer/LNCS 3560, 2005.
17. M. Penrose: Random Geometric Graphs. Oxford University Press, 2003.
18. V. Sanwalani, M. Serna and P. Spirakis: Chromatic Number of Random Scaled Sector Graphs. In the Theoretical Computer Science (TCS) Journal, to appear in 2006.

Numerical Estimation of the Impact of Interferences on the Localization Problem in Sensor Networks*

Matthieu Bouget, Pierre Leone, and Jose Rolim

Computer Science Department,
University of Geneva,
1211 Geneva 4,
Switzerland

Abstract. In this paper we numerically analyze the impact of interferences on the probability of success of a localization algorithm. This problem is particularly relevant in the context of sensor networks. Actually, our numerical results are relevant even when we do not consider interferences. Moreover, our numerical computations show that the main harmful interferences are the ones occurring between sensors which get localized at the same time and send simultaneously their own location. This is demonstrated by varying the time span of the random waiting time before the emissions. We then observe that the longer the waiting time the closer the curves are to the ones obtained without interferences. Hence, this proves to be an efficient way of reducing the impact of interferences. Moreover, our numerical experiments demonstrate that among the sectors of disk with same area, the one with the smaller radius of emission and larger angle of emission is the more appropriate to the localization algorithm.

1 Introduction

Although the numerical tools involved in our analysis can be tailored to deal with different protocols we proceed to the analysis of a simple protocol of transmission which is relevant in the field of sensor networks [1]. The impact of the interferences depends on the statistical occupation of the channels of transmission. Hence, we particularize our analysis to a probabilistic localization algorithm (discussed below) and actually deal with the impact of interferences on the performances of this algorithm.

Two important characteristics of sensor networks are the large number of nodes involved in the composition of the networks and that sensors are usually battery powered, hence limiting the energy consumption is a key issue [27, 28]. Also, the protocols involved in the establishment and use of the networks have to be as simple as possible to limit the energy consumption due to the exchange of synchronization messages. These requirements make relevant to consider *random access* channel introduced in [4]. Besides its simplicity random access is relevant

* This research was supported in part by Swiss SER Contract No. C05.0030.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 13–23, 2006.

in some situations to optimize the transmission delay [7] and hence, can be a better alternative of the different protocols.

Another important characteristic of sensor networks is that they are *data-centric*, meaning that sensors are less important than the data they convey. Typically, sensors are used to proceed to some measurements and convey the measured values towards one (or more) particular stations which is able to collect and process the data's. However, for the data to be meaningful one should attach to it the location where the measurement was made. Then sensors are to be localized. This can be done by adding hardware resources, for instance GPS electronic devices, to sensors. But this would conflict with the requirements of minimizing the energy consumption as well as lowering the price of the entire system. The probabilistic localization algorithm studied in this paper assumes that a few anchors are equipped with electronic devices to ensure their localization. The others sensors compute their own position by 3-lateration given the position of their localized neighbors. This procedure requires the estimate of the distance between sensors. This can be achieved for instance, by Time of Arrival (ToA) or Received Signal Strength Indicator (RSSI) techniques.

2 Related Work

Localization algorithms are widely analyzed in the literature. General considerations and various strategies can be found in [26, 28, 10]. In [29] the accuracy of *range free* localizations algorithms are analyzed. The main interest of these protocols as opposed to *range based* protocols is that they minimize the required hardware. Particularly, there is no need to estimate the distances between sensors. In [8] numerical evaluation of various protocols is done in the context of optical sensor networks. In [13, 14] the authors consider a particular technique for estimating the distances between sensors and proceed to real experiments based on Motes[1] sensor systems. From a complexity point of view, in [3, 6] NP-hardness results are provided for the localization and connex problems. These results support the application of approximation algorithms and numerical investigations. Concerning the analysis of the impact of interferences, we mention [9, 16] and references therein. These papers are based on a model of interferences called the capture model which assumes that a communication can be established given that the ratio of the signal to noise is large enough. In this paper, we consider a different model called the collision model, at once two sensors emits towards a third same sensor there is collision and the data is lost. Moreover, we keep fixed the number of sensors and look for the impact of the networks parameters. The numerical methods we use for the numerical experiments are particular stochastic estimation methods [24, 22]. The general frameworks as well as some applications of the methods are discussed in [19]. The numerical experiments presented in this paper are different than the ones suggested in [19] and actually both papers are complementary. As far as we know, no previous similar works are present in the literature. However, stochastic estimation methods seems to

[1] http://www.xbow.com

appear sporadically for dynamic control of communications in wireless networks, see for instance[17, 20, 25].

3 Sensor Networks Characteristics and the Localization Protocol

Sensors composing the networks establish wireless communication through directional antenna. The parameters of the communications are the range (radius) of communication r which is the maximal distance a sensor can send a data, the angle of emission α and the direction of the emission β. The former is chosen randomly and independently by each sensor with uniform distribution on the circle and the others two parameters r, α are the same for all the sensors. We denote by p the area of the region covered by the emission, see Figure1. The model of interferences is the collision model where a collision occurs at once two stations emit at the same time in the same region. This is illustrated in Figure 1: no data from x nor y can be received in the hatched region.

The sensors are assumed to be randomly and independently scattered in the unit square region $[0, 1] \times [0, 1]$ with uniform distribution. Given the relative positions of sensors directional communication can be established, see Figure 1, and this defines the directional graph of communication see Figure 2. Up to our knowledge results concerning the connectivity of such graphs are only of asymptotic character, see [23]. The choice of the uniform distribution is arbitrary. The localization protocol we wish to numerically analyze the performances works as follows. We consider a given sensor network composed of n sensors. We choose randomly a fixed number l among them and assume that they are localized sensors (assumed to be equipped with electronic devices to ensure the localization). Each localized sensor choose randomly and independently a waiting time w in $0, \ldots, \log(n)$ and wait for w clock. Then, it sends to all its neighbors its location and we assume that the receivers are able to estimate the distance from them to the sender. Once a sensor receives the coordinates of three localized neighbors it computes its own location by 3-lateration. Hence, it chooses a waiting time and send its coordinate in turn. The algorithm is **successful** if more than 90% of the sensors manage to compute their location. Notice that 90% is arbitrarily chosen. However, it is important not to consider the situation where all the sensors have to be localized for the success of the localizations algorithm. Indeed, if the

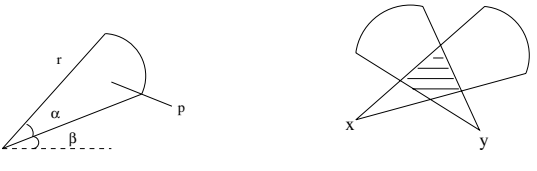


Fig. 1. To the left: Sensor with radius of emission r, angle of emission α and orientation of the emission β. To thre right: Collision region (hatched), x and y are the emitters.

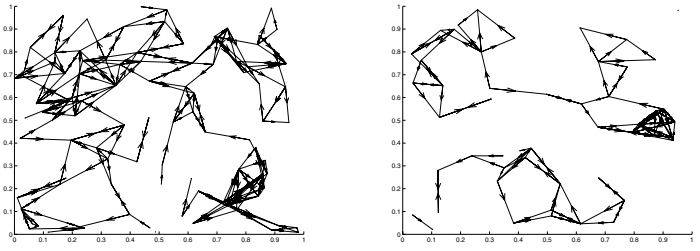


Fig. 2. n=100 r=0.2 a=1.5; n=50 r=0.2 R=3.14

number of sensors is reasonable $(50, 100)$ some of them can even be isolated. The process is probabilistic and hence, it does make sense to look for the probability of success of the algorithm. Our interest here is to numerically investigate the probability of success of the algorithm with respect to the network parameters. It is important to notice that the main impact of the waiting time is to reduce the collisions occurring between sensors which are close from each others. The choice of the waiting time belonging to $[0, \log(n)]$ is motivated in [2].

4 Numerical Methods

The numerical methods we use to compute the chance of success with respect of the networks parameters of the localization algorithm are particularization of stochastic estimation procedures first introduced in [24, 22]. We denote by Adj the set of adjacency matrix which corresponds to communication graphs. This set is a probabilistic space and there exists a probability measure on it which does not need to be explicited. We denote by Ω the set of subset of sensors which are initially localized. This set is embodied with a probability measure corresponding to choosing l of them among the set of sensors. In a given set of computation, all the network parameters are kept constant except the angle on transmission α. Then, the localization algorithm can be seen as a map

$$L : Adj \times \Omega \times [0, 2\pi] \to \{0, 1\} \tag{1}$$

where $L(a, \omega, \alpha)$ is 0 or 1 depending on the success or failure of the localization algorithm. The fixed network parameters are omitted to simplify the notation. Given a particular value $p \in [0, 1]$, our problem is to find the particular value of α such that $\text{Prob}(L(\cdot, \cdot, \alpha) = 1) = p$. The probability is taken with respect to the $Adj \times \Omega$ space and α is a definite numerical value. This is why we use the 'dot' notation $L(\cdot, \cdot, \alpha)$ simplified in $L(\alpha)$. The computations introduced below are based on simulations of the localization algorithm. At each step of the computations a communication graph is generated, a fixed number of initially localized sensors are determined and the process is simulated. This leads to an observed success or failure of the localization algorithm. This observation is denoted by $L(\alpha)$.

To solve this problem we use stochastic estimation methods in the region where $\frac{d}{d\alpha}\text{Prob}(L(\alpha))$ is large since the convergence speed is proportional to this term (see later). In the regions where the above mentioned derivative is small, hence slowing the convergence rate of the stochastic estimation method, we use the typical estimator which consists in averaging the results of a large number of observations. In both cases, we check that the results belong to a confidence interval of 3 degrees with probability 95% [15]. To construct the confidence interval we basically use 20 estimates and the mean value as the result. Basically, the number of iterations of the stochastic estimation methods to compute one estimate is about 30'000. In the region where the method becomes not efficient enough we use the mean of about 100′000 observations to reach the fixed confidence interval.

To proceed to the computation, we fix n the number of sensors and r the radius of emission. Then, we choose a value p of the probability of success and look for the corresponding value of the angle of emission α. To avoid the choice of a value of p too big and hence not corresponding to any α, we compute $\tilde{\alpha}$ such that $m(\tilde{\alpha})$ is maximal with a Kiefer-Wolfovitz algorithm [18] and estimate its value by averaging about 100′000 observations (the procedure is done about 20 times to construct the confidence interval). The stochastic estimation algorithm is based on the hypothesis that there exist a value α^* of the parameter such that

$$m(\alpha^*) = \text{Prob}\big(L(\alpha^*) = 1\big) = p, \text{ and } m'(\alpha^*) > 0. \tag{2}$$

Then, it can be proven [24, 31] that the sequence $(\alpha_n)_{n\geq 0}$ recursively defined by

$$\alpha_{n+1} = \alpha_n + \frac{1}{n}\big(p - L(\alpha_n)\big) \tag{3}$$

converges to α^*, i.e. $(\alpha_n \to \alpha^*)$. Moreover, $\sqrt{n}(\alpha_n - \alpha^*) \to N\left(0, \frac{\sigma^2}{(2m'(\alpha^*)-1)}\right)$.

At each step of the computation a random communication graph has to be generated with the corresponding value of the parameter α_n. The localization algorithm is then simulated and the result is denoted by $L(\alpha_n)$ which is 0 or 1 accordingly to the success or failure of the algorithm. The fact that the function $m(\alpha)$ is differentiable is discussed in [19]. It is due to the fact that the state of outcomes is finite. Actually, the same argument is applicable to the situation where collisions are taken into account. By a suitable change of sign in the formula (3), one can cope with the situation where $m'(\alpha^*) < 0$.

5 Numerical Results

We provide two set of experiments with the same parameters: one assumes that the communications are not altered by interferences and the other takes into account interferences with the collisions model.

If interferences are not taken into account the set of random communication graphs on which the localization algorithm succeeds is monotone. Broadly speaking, this means that by adding new edges to a (successful) communication graph

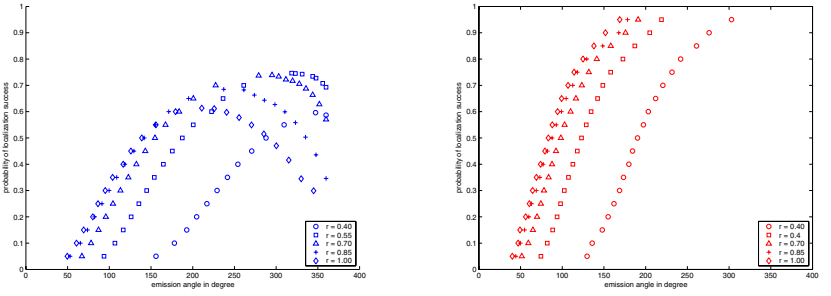


Fig. 3. $n = 50$ sensors, 5 anchors, with collisions (left) and without (right). The curves depicted from right to left are going in increasing the radius of emission. Waiting time in $[0, 4]$.

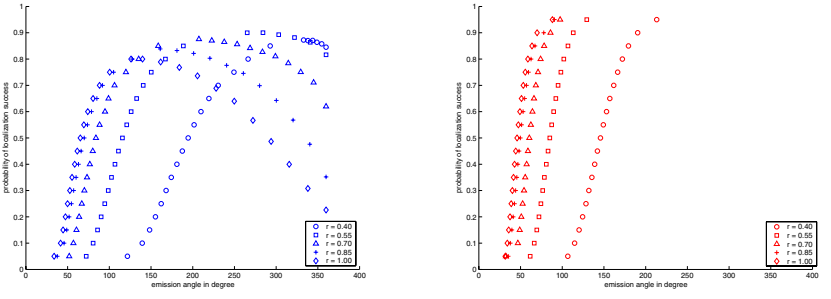


Fig. 4. $n = 50$ sensors, 10 anchors, with collisions (left) and without (right). The curves depicted from right to left are going in increasing the radius of emission. Waiting time in $[0, 4]$.

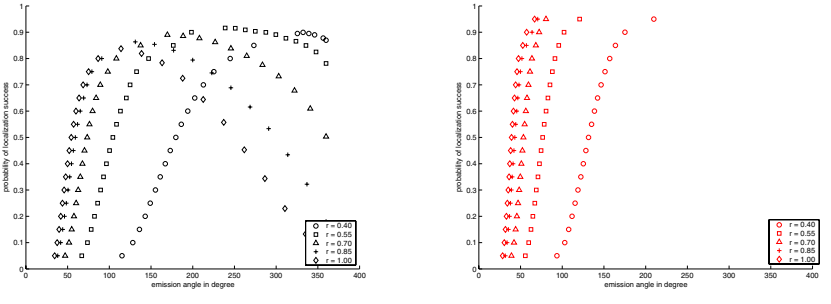


Fig. 5. $n = 50$ sensors, 15 anchors, with collisions (left) and without (right). The curves depicted from right to left are going in increasing the radius of emission. Waiting time in $[0, 4]$.

leads again to a communication graph on which the localization algorithm succeeds. In our setting of experiments, edges are added by increasing the angle of emission α. This is a particular situation for which there are general applicable theoretical results. In particular, sharp threshold is expected as the number of sensors increases [11, 12, 5, 30].

To conduct the numerical experiments, we fix n the number of sensors as well as the radius of emission r and the number of initially localized sensors (anchors). Then, we look for the curve describing the probability of success of the localization protocol with respect to the angle of emission α, i.e. $\text{Prob}(L(\alpha) = 1)$. The numerical experiments are conducted once considering the interferences between simultaneous emissions and once without. This allows measuring the impact of the interferences on the localization algorithm. On a same figure, the experiments are repeated with different values of r, keeping n constant, to measure the impact on increasing the radius on emission. Proceeding this way, we have to keep in mind that increasing the radius of emission increases the energy consumption. Indeed, the energy consumed increases as r^γ, where $2 \leq \gamma \leq 5$ depending on the environment conditions. This point is very important to be considered since minimizing the energy consumption is a key issue in wireless sensor networks.

The numerical experiments introduced above are repeated with different number of sensors n and different number of anchors. In Figures 3, 4, 5, the probability of success of the algorithm is plotted with $n = 50$ sensors with respectively 5, 10 and 15 anchors. The right picture shows the numerical results without collisions. It is observed, as theoretically expected, that as the radius of emission increases the angle of emission necessary to ensure the same probability of success decreases. As colisions are taken into account one observe on Figures 3,4,5 that increasing the angle of emissions increases initially the chance of success up to a maximal value and afterwards the chance of succes decreases. On these figures one can also observe that for large value of the emission angle, the performances are better with small radius of emission. The qualitative behavior does not change as the number of anchors changes.

It is worth to stress the importance of the waiting time, compare Figures 3 and 6 as well as the results plotted in Figures 7,8. Sensors which receive a location data at a same time wait for a random time in $0, \ldots, \log_e(n)$, in order to reduce the interferences between such stations [2]. Our numerical computations confirm that this waiting time is efficient in reducing the impact of the interferences on the localization algorithm. Indeed, for large values of n, we observe, see in particular Figure 8, that the performances of the algorithm with and without interferences are very close from each other.

Although this observation is intuitively clear when assuming no interferences, it is not at all evident that this can still be observed when considering the interferences between simultaneous emissions. However, it is a general observation (see Figures 3,7,8,4,5,6) that increasing the angle of emission increases the chance of success of the localization algorithm.

To reinforce this observation we run simulations of the localization algorithm without waiting time and observe that in the same condition the localization algorithm fails nearly all the time to locate a significant set of sensors. Moreover, the bound $\log_e(n) \approx 4$ for the waiting time is obtained with an asymptotic analysis and hence, is valid as the number of sensors is large enough. Actually, we observe as the number of sensors is smaller, Figure 3, that the impact of the interferences on the performance of the localization algorithm is much more important. We ran

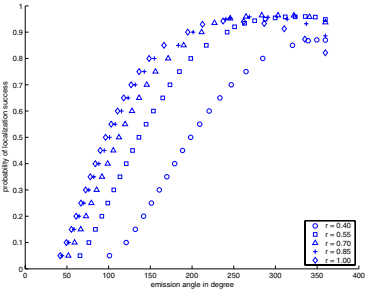


Fig. 6. $n = 50$ sensors, 5 anchors with collisions. The curves depicted from right to left are going in increasing the radius of emission. Waiting time in $[0, 10]$.

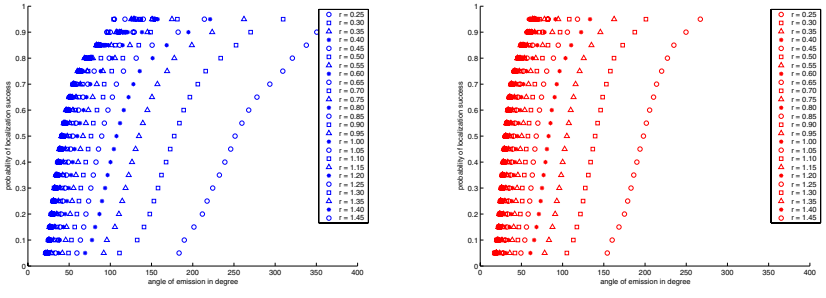


Fig. 7. $n = 100$ sensors, 10 anchors, with collisions (left) and without (right). The curves depicted from right to left are going in increasing the radius of emission. Waiting time in $[0, 5]$.

a new set of simulations with $n = 50$ sensors and a waiting time bounded by 10. The results are plotted in Figure 6 and comparing the results obtained without interferences (right of Figure 3) it is clear that the impact of a longer waiting time on improving the performances of the algorithm is important. With the numerical experiments of Figures 3, 7, 8 it is possible to estimate the impact of the shape of the emission pattern on the success of the localization algorithm. For this purpose, we consider a value of $p = 0.45$ choosen arbitrarily. For each couple (r_i, α_i) leading to a probability of success of $p = 0.45$ we compute $\alpha_i \times r_i^2$ which is proportional to the area of the emission pattern. The results are plotted in Figure 9 corresponding from left to right to $n = 50, 100, 1000$ sensors, i.e. couples (r_i, α_i) are measured on the Figures 3, 7, 8, respectively. The numerical results show that the probability of success depends on the product $\alpha \times r^2$. Actually, the probability depends on the shape of the emission pattern and not uniquely on the area covered. This leads to the question whether there exists an optimal shape of the emission pattern? The numerical results in Figure 9 show that to minimize the energy consumption, keeping the probability of success constant, the radius of emission has to be chosen as small as possible. Hence, the observed optimal shape of the emission pattern is

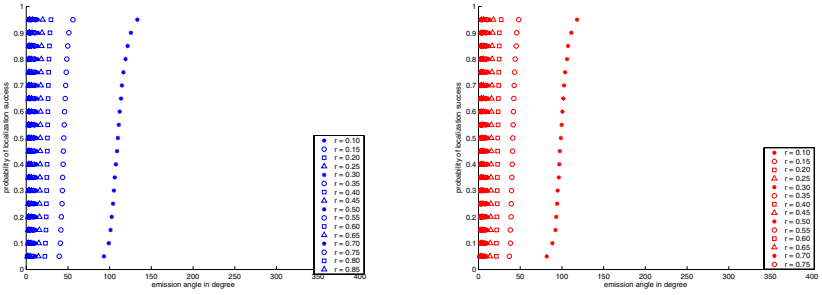


Fig. 8. $n = 1000$ sensors, 100 anchors, with collisions (left) and without (right). The curves depicted from right to left are going in increasing the radius of emission. Waiting time in $[0, 7]$.

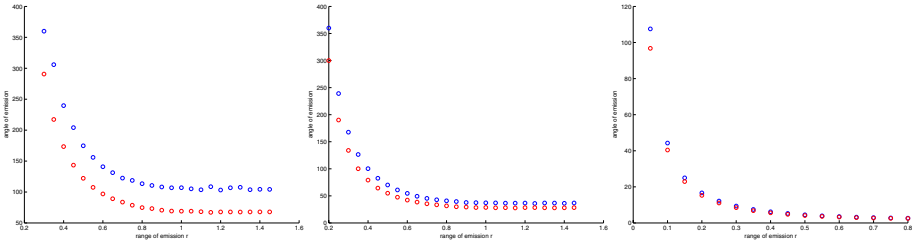


Fig. 9. $n = 50$, $n = 100$ and $n = 1000$ sensors repectively, 5 anchors, with collisions (blue) and without (red). Variation of the area of the emission pattern (with $p = 0.45$ constant) as a function of r. Plot of $\alpha(x) \times x^2$, $x = 0.30, 0.35, \ldots$ with respect to x.

the circle. More generally, we can postulate that the symmetric radiation pattern is much more appropriate to the localization algorithm.

We should mention that the numerical methods used in this paper prove to work well and lead to some useful numerical results. The computations show clearly the relevance of the waiting time before emitting. This is also relevant to model dynamical behavior of sensor networks.

References

1. I.F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, Vol. 38, Issue 4, pp. 393–422 (2002).
2. G. Arzhantseva, J. Diaz, J. Petit, J. Rolim, M.J. Serna. Broadcasting on Networks of Sensors Communicating through Directional Antennas. International Workshop on Ambient Intelligence Computing, CTI Press, pp. 1-12, 2003. Ellinika Grammata.
3. J. Aspnes, D. Goldenberg and Y.R. Yang, On the Computational Complexity of Sensor Network Localization. In Proceedings of the 1st International Workshop ALGOSENSORS 2004, Turku, Finland, LNCS 3121, Springer Verlag, 2004.

4. N. Abramson, The aloha system - another alternative for computer communications. In Proceedings Fall Joint Comput. Conf., AFIPS Conf., Montvale, NJ, vol. 44, pp. 281-285, 1970.
5. B. Bollobás, A. Thomason, Threshold functions. Combinatorica 7, pp. 35-38, 1987.
6. H. Breu, D.G. Kirpatrick, Unit disk graph recognition is NP-hard. Journal of Computational Geometry 9, pp. 3-24, 1998.
7. A.B. Carleial, M.E. Hellman, Bistable behavior of ALOHA-type systems. IEEE Trans. Commun., vol. COM-23, pp. 401-410, Apr. 1975.
8. J. Diaz, J. Petit and M. Serna. Evaluation of Basic Protocols for Optical Smart Dust Networks. *In Proc. of the 2nd Int'l Workshop on Experimental and Efficient Algorithms (WEA 2003)*, Ascona, Switzerland, LNCS 2647, Springer Verlag, May 2003.
9. O. Dousse, F. Baccelli, P. Thiran, Impact of Interferences on Connectivity in Ad Hoc Networks. In Proceedings of Infocom, San Francisco, April 2003.
10. N. Bulusu, J. Heidmann and D. Estrin. GPS-less low-cost outdoor localization for very small devices. *IEEE Personal Communications, Special Issue on Smart Spaces and Environments*, Vol. 7, No. 5, pp. 28–34, October 2000.
11. E. Friedgut, G. Kalai, Every monotone graph property has a sharp threshold. Proc. Amer. Math. Soc. 124, pp. 2993-3002, 1996.
12. E. Friedgut, Hunting for Sharp Threshold. Random Structures Algorithms 26, no. 1-2, 37–51, 2005.
13. L. Girod, D. Estrin, Robust range estimation using accoustic and multimodal sensing. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001), March 2001.
14. L. Girod, V. Bychkovskiy, J. Elson, D. Estrin, Locating Tiny Sensors in Time and Space: A Case Study. In Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02), p. 214, Freiburg, Germany, september 2002.
15. P. W. Glynn, M. Hsieh Confidence Regions for Stochastic Approximation Algorithms.*Proceedings of the 2002 Winter Simulation Conference*, pp. 370-376 (2002).
16. P. Gupta, P.R. Kumar, The capacity of wireless networks. IEEE Trans. Inform. Theory, vol. 46(2), pp. 288-404, March 2000.
17. B. Hajek, Stochastic Approximation Methods for Decentralized Control of Multiaccess Communications. IEEE Transactions on Information Theory, vol. IT-31, no. 2, March 1985.
18. J. Kiefer, J. Wolfowitz Stochastic Estimation of the Maximum of a Regression Function, *Ann. Math. Statist.*, 23, pp. 462-466 (1952).
19. P. Leone, P. Albuquerque, C. Mazza, J. Rolim, A Framework for Probabilistic Numerical Evaluation of Sensor Networks. In Proceedings of the 4th International Workshop (WEA 2005), pp. 341-353, Santorini Island, Greece, May 2005.
20. P. Leone, S. Nikoletseas and J. Rolim. An adaptative Blind Algorithm for Energy Balanced Data Propagation in Wireless Sensor Networks. To appear in Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS'05), California, Marina del Rey, June 30 - July 1 2005.
21. M. Leopold, M. B. Dydensborg, and P. Bonnet, Bluetooth and Sensor Networks: A Reality Check. In Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03), pages 103–113, Los Angeles, Clifornia, USA, November 2003.
22. M.B. Nevel'son and R.Z. Has'minskii. Stochastic Approximation and Recursive Estimation. Translation of Mathematical Monographs, Vol. 47, American Math. Soc. (1976).

23. M. Penrose, Random Geometric Graphs, Oxford Studies in Probability 5, 2003.
24. H. Robbins and S. Monro. A stochastic approximation method. Ann. Math. Stat., Vol. 22, pp. 400–407 (1951).
25. A. Segal, Recursive estimation from discrete-time point process. IEEE transactions on Information Theory, vol. IT-22, pp. 421-431, July 1976.
26. A. Savvides, C. Han, M. B. Strivastava, Dynamic fine-grained localization in Ad-Hoc networks of sensors. In Proceedings of MOBICOM 2001, pp. 166-179.
27. M. Srivastava. Part II: Sensor Node Platforms & Energy Issues. *Tutorial on Wireless Sensor Networks, Mobicom'02* (2002). Avalaible at http://nesl.ee.ucla.edu/tutorials/mobicom02.
28. M. Srivastava. Part III: Time & Space Problems in Sensor Networks. *Tutorial on Wireless Sensor Networks, Mobicom'02* 2002. Avalaible at http://nesl.ee.ucla.edu/tutorials/mobicom02.
29. G. Stupp, M. Sidi, the Expected Uncertainty of Range Free Localization Ptotocols in Sensors Networks. In Proceedings of the 1st International Workshop, ALGOSENSORS 2004, Turku, Finland, LNCS 3121, Springer Verlag 2004.
30. M. Talagrand, On Russo's Approximate Zero-One Law. The Annals of Probability, vol. 22, no. 3, pp. 1576-1587, 1994.
31. J.H. Venter. An extension of the Robbins-Monro procedure. *Ann. Math. Stat.*, Vol. 38, pp. 181–190 (1967).

An Efficient Heuristic for the Ring Star Problem

Thayse Christine S. Dias[1], Gilberto F. de Sousa Filho[2], Elder M. Macambira[3], Lucidio dos Anjos F. Cabral[3], and Marcia Helena C. Fampa[1]

[1] Universidade Federal do Rio de Janeiro,
Programa de Engenharia de Sistemas e Computação,
Rio de Janeiro - RJ - Brasil
{thayse, fampa}@cos.ufrj.br
[2] Universidade Federal da Paraíba,
Departamento de Informática,
João Pessoa - PB - Brasil
gilberto@lavid.ufpb.br
[3] Universidade Federal da Paraíba,
Departamento de Estatística,
João Pessoa - PB - Brasil
{elder, lucidio}@de.ufpb.br

Abstract. In this paper, we consider a combinatorial optimization problem that arises in the design of telecommunications network. It is known as the Ring Star Problem. In this problem the aim is to locate a simple cycle through a subset of vertices of a graph with the objective of minimizing the sum of two costs: a routing cost proportional to the length of the cycle, and an assignment cost from the vertices not in the cycle to their closest vertex on the cycle. We propose a new hybrid metaheuristic approach to solve the Ring Star Problem. In the hybrid metaheuristic, we use a General Variable Neighborhood Search (GVNS) to improve the quality of the solution obtained with a Greedy Randomized Adaptive Search Procedure (GRASP). A set of extensive computational experiments on instances from the classical TSP library and randomly generated are reported, comparing the GRASP/GVNS heuristic with other heuristic found in the literature. These results indicate that the proposed hybrid metaheuristic is highly efficient and superior to the other available method proposed for the Ring Star Problem.

Keywords: ring star problem, network design, GRASP, VNS, heuristics.

1 Introduction

New technology in telecommunications leads to challenging network design problems. The problem considered in this article arises in the design of telecommunications network in which terminals (user nodes) lying on a access network are connected to concentrators (switches or multiplexers) lying on a backbone network linked to a central unit (root) (see, e.g., [3]). Roughly speaking, the problem then consists of selecting a subset of user nodes where concentrators

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 24–35, 2006.

will be installed, and interconnect them by a ring network, which results in a ring topology, and assign the other user nodes to those concentrators by point-to-point links, which results in a star topology. The goal is to find a feasible solution that minimize the cost of ring and the cost of assigning vertices on the star to the nearest vertex on the ring, i.e., the total cost of all connections must be minimized.

The combinatorial optimization problem above is also known as the Ring Star Problem, or RSP for short. The RSP can be formally defined as follows. Let $G = (V, E \cup A)$ be a complete mixed graph where $V = \{v_1, v_2, \ldots, v_n\}$ is the vertex set, $E = \{(v_i, v_j) : v_i, v_j \in V, i < j\}$ is the edge set and $A = \{[v_i, v_j] : v_i, v_j \in V\}$ is the arc set (loops $[v_i, v_i]$ are included in A). The vertex v_1 is referred to as the root, each edge (v_i, v_j) is associated with a non-negative *routing cost* c_{v_i,v_j} and each arc $[v_i, v_j]$ is associated with a non-negative *assignment cost* d_{v_i,v_j}. The routing cost of a solution is the sum of all edge costs on the cycle. The assignment cost of a solution is computed as the sum of the minimum assignment costs from non-visited to visited vertices, i.e.,

$$\sum_{v_i \in V \setminus V'} \min_{v_j \in V'} d_{v_i,v_j}. \tag{1}$$

The Ring Star Problem consists of determining a Hamiltonian cycle through a subset V' of V including v_1 and at least two other vertices that minimizes the sum of the routing cost of the cycle and the assignment cost of the vertices not on the cycle to their nearest vertex on the cycle.

The Ring Star Problem is known to be NP-hard since the special case in which the assignment cost are very large compared to the routing cost is the classical Traveling Salesman Problem (TSP) [6]. In [6, 7], the authors provide integer linear programming formulations and develop branch-and-cut algorithms for the Ring Star Problem. In [9], the RSP is solved by an hybrid algorithm called Variable Neighborhood Tabu Search (VNTS).

Metaheuristics such as Simulated Annealing (SA), Greedy Randomized Adaptive Search Procedure (GRASP), Tabu Search (TS) and Variable Neighborhood Search (VNS) have been used successfully for solving combinatorial optimization problem in practice and they have been applied to a very large variety of hard optimization problems in telecommunications (see, e.g., [8]).

The aim of this paper is to propose a new hybrid approach for the Ring Star Problem, consisting of a combination of the Greedy Randomized Adaptive Search Procedure and General Variable Neighborhood Search (GVNS). We intend show that metaheuristic based on GRASP and GVNS can give good results for the RSP.

In this hybrid approach, several well known methods are used to execute local search. Furthermore, our metaheuristic makes use a GVNS to improve the solutions obtained with GRASP. An important characteristic of our proposal is that GVNS operates as perturbation procedure of the solution given by GRASP. We evaluate experimentally this metaheuristic and compare the solutions obtained by the new heuristic with the solutions obtained by Variable Neighborhood Tabu Search proposed in [9].

The remainder of the paper is organized as follows. In Section 2, we describe GRASP metaheuristic and his adaptation to the Ring Star Problem. GVNS metaheuristic is presented in Section 3. Section 4 gives a description of how GRASP and GVNS hybrid metaheuristics are combined to solve the Ring Star Problem. Experimental results, with benchmark instances, are presented in Section 5. Finally, concluding remarks are made in Section 6.

2 Greedy Randomized Adaptive Search Procedure

GRASP (Greedy Randomized Adaptive Search Procedure) is a multistart or iterative process, where different points in the search space are probed with local search for high-quality solutions [1]. It has been used with success to provide solutions for several difficult combinatorial optimization problems [2]. Each iteration of GRASP consists of the construction of a randomized greedy solution, followed by a local search, starting from the constructed solution. The best solution from all iterations is returned as result.

The elements which completely determine a GRASP are: the heuristic function, the way in which the Restricted Candidate List (RCL) is built, the improved method and the stopping rule. The pseudo-code in Figure 1 illustrates the main blocks of a GRASP procedure for minimization, in which `MaxIter` iterations are performed and `RandomSeed` is used as the initial seed for the pseudorandom number generator.

```
procedure GRASP(MaxIter, RandomSeed)
1.   s* ← ∅; // best solution
2.   f(s*) ← ∞;
3.   for i ← 1 to MaxIter do
4.        s ← ConstructGreedyRandomizedSolution(RandomSeed);
5.        s' ← LocalSearch(s);
6.        Update_Solution(s',s*);
7.   end-for
8.   return(s*).
end GRASP.
```

Fig. 1. GRASP pseudo-code for minimization

In the remainder of this section, we describe in detail the phases of the GRASP for the Ring Star Problem, in order to facilitate the discussion of the hybrid approach that will follow in the next sections.

2.1 Construction Phase

The construction phase is iterative, greedy and adaptive, in the sense that the element chosen at each iteration during the construction phase is dependent on those previously chosen.

The GRASP construction phase, shown in Figure 2, builds a feasible solution by selecting one vertex, one at a time. A solution is a tour or cycle S including

```
procedure ConstructGreedyRandomizedSolution(RandomSeed)
1.  s* ← ∅;   f(s*) ← ∞;
2.  for IterFilter ← 1 to MaxIterFilter do // number of solutions of the pool
3.      s ← {v1}; // current solution
4.      while the stopping condition is met do
5.          g_min = min{g(v_i) | v_i ∈ V \ s};   g_max = max{g(v_i) | v_i ∈ V \ s};
6.          L = {v_i ∈ V \ s | g(v_i) ≤ g_min + α(g_max − g_min)};
7.          Select v_i, at random, from L;
8.          s = s ∪ {v_i};   V = V \ {v_i};
9.      end-while
10.     if (f(s) < f(s*)) then
11.         s* ← s;   f(s*) ← f(s);
12.     end-if
13. end-for
14. return(s*).
end ConstructGreedyRandomizedSolution.
```

Fig. 2. The GRASP construction phase

the root, denoted by v_1. Every non visited vertex is allocated to its nearest vertex in the cycle, being the allocation cost equal to the distance between them. The length of the cycle is given by

$$LC(S) = \sum_{(v_i,v_j)\in S} c_{v_i,v_j}, \tag{2}$$

and its allocation cost is given by

$$AC(S) = \sum_{v_i\in S} \min_{v_j\in V(S)} d_{v_i,v_j}, \tag{3}$$

where $V(S)$ is the set of vertices of the cycle S. The RSP consists of finding the cycle S visiting the depot that minimizes $LC(S) + AC(S)$.

In the following, we describe an adaptive greedy function $g : L \rightarrow \Re$, a construction mechanism for the RCL, and a probabilistic selection criterion for the GRASP construction phase proposed.

The initial solution is obtained as follows. To obtain an initial cycle S we start only with the root v_1. The greedy function g takes into account the contribution to the objective function achieved by selecting a particular element. In the case of the Ring Star Problem, it is intuitive to relate the greedy function to the smallest increment in the value of the length of cycle S, i.e., to insert the vertex in the best position, plus the sum of the cost of connectivity average of vertices its outgoing (allocation). More formally, for each $v_i \in V \setminus S$ we define the following

$$g(v_i) = \texttt{SmallestIncrement}(v_i) + \frac{\sum_{j\in V\setminus S} d_{v_i,v_j}}{|V \setminus S|}. \tag{4}$$

The greedy choice is to select the vertex v_i with smallest $g(v_i)$.

The probabilistic component of a GRASP is characterized by randomly choosing one of the best candidates in the list, but not necessarily the top candidate. The list of best candidates corresponds to the Restricted Candidate List. To define the construction mechanism proposed for the RCL, let

$$g_{min} = \min\{g(v_i) \mid v_i \in V \setminus S\} \text{ and } g_{max} = \max\{g(v_i) \mid v_i \in V \setminus S\}. \quad (5)$$

To select the vertex to be added to the solution, a Restricted Candidate List is defined to include all vertices v_i in the candidate set L having cost $g(v_i) \leq g_{min} + \alpha(g_{max} - g_{min})$ where $\alpha \in [0, 1]$. In the following, one vertex $v_i \in L$ is chosen at random and it is added to the solution, i.e., $S = S \cup \{v_i\}$ and $V = V \setminus \{v_i\}$.

Furthermore, the GRASP construction phase uses short-term memory scheme. Their scheme maintains a pool of solutions to be used in the construction phase. So, the initial solution returned by construction phase corresponds to the best solution of the pool.

The GRASP construction phase stops when there are at least three vertices in the cycle and the objective function cannot be decreased anymore.

2.2 Local Search Phase

It is almost always beneficial to apply a local search to attempt to improve each constructed solution. A local search algorithm works in an iterative fashion by successively replacing the current solution S by a better solution in the neighborhood of the current solution and a new neighborhood search is initialized. Otherwise, the solution is locally optimal with respect to the neighborhood and the local search ends.

The definition of the neighborhood $N(S)$ is crucial for the performance of the local search. The local search procedure here, is based on a neighborhood that seek to improve the length of the cycle and the total allocation cost in the current solution as defined in (2) and (3).

Let v_i and v_j two vertices in S. Four simple and well known types of moves are defined to obtain the neighborhood $N(S)$:

- add: at each iteration a new vertex v_j that does not belong to the solution is inserted;
- drop: at each iteration the vertex v_j that belongs to the solution is removed;
- add/drop: at each iteration the interchange of two vertices v_i and v_j consists of a combination of an add and a drop move. The vertex v_i is removed from solution, and the vertex v_j is inserted in the best possible position in the solution;
- ℓ-opt: at each iteration the length of the cycle is improved by performing classical edge interchange moves such as 2-opt and 3-opt by means of a TSP.

Figure 3 gives the pseudo-code of the local search using these types of moves. Lines 1-5 correspond to moves used in which each neighbor solution is obtained through a move involving add, drop, add/drop or ℓ-opt, respectively. The best

procedure `LocalSearch`(s)
1. $H_{add} \leftarrow \{s' \in N_{add}(s) \mid f(s') < f(s)\}$;
2. $H_{drop} \leftarrow \{s' \in N_{drop}(s) \mid f(s') < f(s)\}$;
3. $H_{add/drop} \leftarrow \{s' \in N_{add/drop}(s) \mid f(s') < f(s)\}$;
4. $H_{\ell-opt} \leftarrow \{s' \in N_{\ell-opt}(s) \mid f(s') < f(s)\}$;
5. $H \leftarrow H_{add} \cup H_{drop} \cup H_{add/drop} \cup H_{\ell-opt}$;
6. **while** $|H| > 0$ **do**
7. Select $s' \in H$;
8. $s \leftarrow s'$;
9. $H_{add} \leftarrow \{s' \in N_{add}(s) \mid f(s') < f(s)\}$;
10. $H_{drop} \leftarrow \{s' \in N_{drop}(s) \mid f(s') < f(s)\}$;
11. $H_{add/drop} \leftarrow \{s' \in N_{add/drop}(s) \mid f(s') < f(s)\}$;
12. $H_{\ell-opt} \leftarrow \{s' \in N_{\ell-opt}(s) \mid f(s') < f(s)\}$;
13. $H \leftarrow H_{add} \cup H_{drop} \cup H_{add/drop} \cup H_{\ell-opt}$;
14. **end-while**
15. return(s).

end `LocalSearch`.

Fig. 3. The GRASP local search phase

neighbor s' of the current solution s is selected in line 7, i.e., an improving move is found. So, the neighbor solutions of the current solution s are computed in lines 9-13. The local search ends when no further such improvement is possible.

3 General Variable Neighborhood Search

Variable Neighborhood Search (VNS) is a relatively new technique based on systematic increase the size of the neighborhood in a heuristic search [4].

In general, heuristic searches proceed by performing a sequence of local changes in a initial solution which improve each time the value of the objective function until a local optimum is found. VNS algorithms overcome this situation changing the neighborhood structure always a local search is trapped in a local minimum.

In its basic form, VNS explores a set of neighborhoods, denoted by N_k, ($k = 1, ..., k_{max}$), of the current solution, makes a local search from a neighbor solution to a local optimum, and moves to it if there has been an improvement. A basic local search, i.e. $k_{max} = 1$, consists of applying an improving move while such move exists, and a VNS algorithm can be implemented by the combination of series of random and improving (local) searches. This approach offers different degrees of flexibility: the order in the search can be changed, the choice of N_k to be used and how many of them, and the search strategy chosen to be used in changing neighborhoods.

The General Variable Neighborhood Search (GVNS) method applies two (possible different) series of neighborhoods; one for the shaking and one for the descent. The GNVS led to the most successful applications of VNS metaheuristic recently reported. Several of those applications use the same set of neighborhoods structures for shaking and descent. The shake procedure consists in obtaining a random point from the current neighborhood $N_k(s)$. This procedure applies

```
procedure GVNS(s, k_max, MaxIterGVNS)
1.  Select the set of neighborhood structures N_k, for each k = 1, ..., k_max;
2.  s* ← s;   f(s*) ← f(s);
3.  for Iter ← 1 to MaxIterGVNS do
4.      k ← random();
5.      Apply k random moves to the solution s to get s';
6.      Apply some local search to the s' until a local minimum s'' is found;
7.      if f(s'') < f(s*) then
8.          s* ← s'';   s ← s'';
9.      end-if
10. end-for
11. return(s*).
end GVNS.
```

Fig. 4. Basic GVNS pseudo-code for minimization

a number k of random base moves. The Figure 4 describes the idea of a basic GVNS algorithm.

In the remainder of this section, we describe in detail the basic GVNS proposed for the Ring Star Problem, in order to facilitate the discussion of the hybrid approach that will follow in the next sections.

The basic GVNS proposed for the RSP uses one shake procedure very simple. Given a size k for the shake procedure, we choose k times two vertices at random, v_i and v_j. If the vertex v_i is in the cycle and v_j is outside the cycle, we do the corresponding add/drop move, .i.e., we drop v_i from solution and add v_j to solution. If both vertices are in the cycle, we drop v_i from solution. If both vertices are outside the cycle, we add v_j to solution.

Furthermore, local search add, drop, add/drop and ℓ-opt moves defined for GRASP are used in basic GVNS with neighborhoods $N_k(k = 1, \ldots, k_{max})$. So, the loop in lines 3-10 investigates one neighborhood at a time, until a local optimum with respect to neighborhoods add, drop, add/drop and ℓ-opt is found.

4 A GRASP/GVNS Heuristic

A natural way of hybridizing GRASP and GVNS is to use GVNS in the second phase or in the phase after the local search method of the GRASP (c.f., [5]). In this section, we describe the new GRASP heuristic combined with GVNS for the the Ring Star Problem using the second way.

The pseudo-code in Figure 5 illustrates the main blocks of a heuristic based on the GRASP and GVNS algorithms described in Sections 2 and 3.

The heuristic takes as parameters the number `MaxIter` of iterations, the value `RandomSeed` used as the initial seed for the pseudo-random number generator, the value k_{max} and the number `MaxIterGVNS` of iterations of the GVNS. The loop in lines 2-9 performs `MaxIter` iterations. In lines 3-7 is used the algorithm GRASP described in Section 2. The VNS strategy, using neighborhoods add, drop, swap and ℓ-opt, is implemented in line 8, as described in Section 3.

```
procedure GRASP+GVNS(MaxIter, RandomSeed, k_max, MaxIterGVNS)
1.  f(s*) ← ∞;
2.  for i ← 1 to MaxIter do
3.      s ← ConstructGreedyRandomizedSolution(RandomSeed);
4.      s' ← LocalSearch(s);
5.      if (f(s') < f(s*)) then
6.          s* ← s';   f(s*) ← f(s');
7.      end-if
8.      GVNS(s', k_max, MaxIterGVNS);
9.  end-for
10. return(s*).
end GRASP+GVNS.
```

Fig. 5. Pseudo-code of the GRASP+GVNS heuristic

5 Computational Results

In this section, we reports some results obtained with preliminary experiments for Ring Star Problem using the GRASP+GVNS heuristic on randomly generated test problems and on a set of instances from the TSPLIB. All computational experiments have been performed on a SEMPRON 2.6 GHz AMD processor with 512 Mbytes of RAM memory under Linux operating system. The GRASP+GVNS heuristic was implemented in C ++ language.

5.1 Test Problems

Before we describe the experimental results, we must comment about benchmark instances used. The set of instances is divided into two categories. The first set of test problems we considered in our computational experiments is called category C1 and the test problems are taken from [6, 7] involving 10, 20, 30, 40, 50, 75 and 100 vertices and having EUC2D format (Euclidean distances). To define the routing and assignment costs, we have proceeded as in [6, 7], i.e., we have set $c_{ij} = \beta \times l_{ij}$ and $d_{ij} = (10 - \beta) \times l_{ij}$, where $\beta \in \{5, 7, 9\}$ and l_{ij} correspond to the Euclidean distance between vertices v_i and v_j.

The second set is called category C2. The test problems of C2 are taken on a subset of instances from the TSPLIB involving between 50 and 100 vertices and having EUC2D format. We also have the same definition for c_{ij} and d_{ij}. We have restricted our computational experiments to the 210 instances of category C1, and the 33 instances of category C2.

5.2 Experiments

Our objective with the experimental part of this paper is to evaluate the effectiveness of hybridizing GRASP and GVNS when used in solving of the Ring Star Problem. The number of GRASP iterations were fixed at MaxIter $= n$, the value of MaxIterFilter was fixed at 50 and the values for α were taken on $\{0.5, 0.6, 0.7, 0.8, 0.9\}$. The number of GVNS iterations were fixed at

Table 1. Results obtained by the GRASP+GVNS heuristic for C1 instances

Instances C1	$\alpha = 0.5$	$\alpha = 0.6$	$\alpha = 0.7$	$\alpha = 0.8$	$\alpha = 0.9$
standard deviation (GAP)	0.00332	0.00305	0.00287	0.01490	0.00232
best solutions found	40	41	39	43	40

Table 2. Comparison between algorithms for instances with 75 vertices in C1

Instance	GRASP + GVNS				VNTS				GAP	β
	f_min	f_avg	f_max	t_avg	f_min	f_avg	f_max	t_avg		
f75_1.rcp	33520	33545	33565	104,61	33730	34148,4	35076	3,85	0,006265	5
	37920	37945,8	37963	84,87	37920	37969,6	38129	4,79	0	7
	27479	27498,2	27575	28,88	27479	27479	27479	5,44	0	9
f75_2.rcp	30745	30775	30795	98,26	31100	31565,4	32530	3,91	0,011547	5
	35032	35032,6	35035	89,07	35032	35185	35287	3,85	0	7
	26972	26972	26972	39,78	26972	26997,8	27101	4,08	0	9
f75_3.rcp	33100	33100	33100	89,53	33705	34128	34766	4,09	0,018278	5
	37098	37124,6	37150	84,00	37096	37100,4	37110	4,32	-0,000054	7
	27388	27497,6	27545	36,16	27388	27417,2	27534	3,93	0	9
f75_4.rcp	31285	31285	31285	92,48	31517	31621,6	31997	4,03	0,007416	5
	34583	34659,4	34767	77,79	34583	35192,4	36271	6,38	0	7
	25946	25952,4	25978	33,84	25855	25973	26063	3,79	-0,003507	9
f75_5.rcp	29840	29840	29840	104,71	29906	30483,6	31207	3,94	0,002212	5
	33734	33783	33838	116,07	33726	33726	33726	3,98	-0,000237	7
	28783	28795	28803	38,57	28770	28770	28770	6,35	-0,000452	9
f75_6.rcp	32420	32448	32465	87,96	32696	33270	33550	3,91	0,008513	5
	37122	37178,8	37406	105,29	37188	37434,6	37894	4,67	0,001778	7
	26034	26062,8	26098	40,84	25962	25984,8	26019	4,01	-0,002766	9
f75_7.rcp	31900	31940	32040	134,34	32067	32390,8	32826	3,93	0,005235	5
	35381	35453	35381	82,33	35389	35493,2	35560	4,13	0,000226	7
	28178	28178	28178	53,45	28178	28178	28178	9,45	0	9
f75_8.rcp	30900	30939	30950	100,67	31271	31935,2	32462	4,64	0,012006	5
	35356	35399,4	35431	81,16	35356	35356	35356	5,65	0	7
	26575	26688,2	26792	55,09	26543	26543	26543	5,97	-0,001204	9
f75_9.rcp	30825	30833	30865	147,92	30825	31373,6	31641	4,10	0	5
	34028	34028	34028	106,91	34028	34049,6	34055	6,00	0	7
	24957	25083	25135	41,09	24957	24957	24957	7,45	0	9
f75_10.rcp	30930	30948	30975	94,22	31250	31645,6	32010	4,59	0,010346	5
	34728	34772,6	34852	81,77	34753	35244,2	35849	4,37	0,000720	7
	28382	28416,4	28425	49,43	28286	28286	28286	8,0	-0,003382	9

`MaxIterGVNS` = 10. Each one of the 243 instances was executed five times with different seeds.

Category C1

In order to evaluate the performance of our approach, first we run the algorithm presented in [9], kindly sent to us by authors, and after we run the GRASP+GVNS heuristic. We conducted our computational study with five parameters for α. From the results of the experiments we conclude that the appropriated value for α is 0.8. Table 1 summarizes the results obtained for the 210 instances of C1. For each parameter α, we reports the standard deviation (GAP) of the heuristic solution value with respect to the best solution found by [9] and the number of instances for which one best solution was found (number of improvements). The Table 1 shows that the GRASP+GVNS heuristic significantly improved the solutions obtained by VNTS [9]. The new heuristic found best solutions for 43 out of the 210 instances of C1 for $\alpha = 0.8$. Furthermore,

Table 3. Comparison between algorithms for instances with 100 vertices in C1

Instance	GRASP + GVNS				VNTS				GAP	β
	f_min	f_avg	f_max	t_avg	f_min	f_avg	f_max	t_avg		
f100_1.rcp	36665	36790	36915	340,9	37250	37696,4	38991	10,2	0,015955	5
	41279	41349,4	41427	363,9	41279	41573,2	41952	11,3	0	7
	29543	29560,6	29565	123,7	29543	29547,4	29565	12,5	0	9
f100_2.rcp	37500	37594	37710	359,7	37664	38383,8	39204	9,5	0,004373	5
	41983	42064,8	42264	321,6	41971	42182,4	42506	12,8	-0,00029	7
	31166	31197,2	31227	129,2	31115	31115	31115	7,8	-0,00164	9
f100_3.rcp	34750	34831	34855	389,2	34890	35664	36654	9,7	0,004029	5
	37912	37935,8	38011	325,3	38133	38133	38133	10,7	0,005829	7
	28180	28180	28180	150,9	28180	28180	28180	9,5	0	9
f7100_4.rcp	37345	37428	37660	430,8	37762	38021,4	38425	11	0,011166	5
	41907	41984,6	42039	364,8	41752	41821	42085	10,1	-0,0037	7
	33452	33479	33505	122,2	33452	33452	33452	14	0	9
f100_5.rcp	35730	35730	35730	447,5	35958	36743,8	37552	10,6	0,006381	5
	40338	40349,2	40378	354,7	40802	41024	41779	11,3	0,011503	7
	33451	33451	33451	146,4	33451	33451	33451	25,1	0	9
f100_6.rcp	36355	36359	36365	418,2	36823	37235,2	37631	9,4	0,012873	5
	41743	41823,6	41915	319,2	41772	42063,6	42751	10,3	0,000695	7
	31677	31850,2	31998	158,9	31612	31612	31612	14,5	-0,00205	9
f100_7.rcp	35840	35948	36095	392,9	35984	36591	37109	9,8	0,004018	5
	38950	39016,8	39176	310,3	38910	38912,8	38924	9,9	-0,00103	7
	29911	29923	29926	134	29911	29914	29926	7,5	0	9
f100_8.rcp	37575	37666	37740	396,2	37885	38421	39343	10,5	0,00825	5
	42463	42741	42930	319,4	42553	42652,6	42885	11	0,002119	7
	30789	30789,4	30791	188,3	30779	30779	30779	25,6	-0,00032	9
f100_9.rcp	36780	36890	37005	427,6	37242	37600,6	38074	10,5	0,012561	5
	40372	40430,6	40493	324,1	40372	40773,6	42311	11,3	0	7
	31828	31828	31828	168,1	31801	31801	31801	9,3	-0,00085	9
f100_10.rcp	38355	38436	38585	409,5	38509	39634	40465	9,8	0,004015	5
	43130	43152,8	43175	316,6	43074	43335,4	43551	10,4	-0,0013	7
	31425	31425	31425	147	31425	31425	31425	13,6	0	9

the GRASP+GVNS is robust for all values of α because the standard deviations yielded not exceeding 0.0149 with a small increase in computational time.

In Tables 2 and 3, we compare the results obtained by GRASP+GVNS heuristic using $\alpha = 0.8$ with VNTS for the instances with 75 and 100 vertices. We report the problem characteristic, the minimum, average and maximum objectives values, the average computational times in seconds by each algorithm and the value of β. The GRASP+GVNS heuristic, as illustrated by the column GAP, found best solutions for 12 out of the 30 and 14 out of the 30 instances with 75 and 100 vertices, respectively.

Category C2

In this category were submitted 33 instances to both algorithms. For all instances tested, the GRASP+GVNS found best solutions 13 out of the 33 instances and found the same solutions in 13 others, as illustrated by the column GAP. The standard deviation is low when analyzing the value of GAP of different executions for the same instance and β, reinforced the robustness of the GRASP+GVNS approach. The Table 4 shows the comparison between GRASP+GVNS heuristic using $\alpha = 0.9$ and VNTS heuristic for the modified instances from TSP.

Table 4. Comparison between algorithms for instances in C2

Instance	GRASP + GVNS				VNTS				GAP	β
	f_min	f_avg	f_max	t_avg	f_min	f_avg	f_max	t_avg		
eil51.rcp	1995	1998	2005	12,35	2025	2040,2	2059	1,27	0,015038	5
	2113	2117,4	2123	10,11	2113	2113	2113	1,76	0	7
	1244	1244	1244	4,98	1244	1244	1244	0,98	0	9
berlin52.rcp	36115	36152	36260	16,80	36720	37326,8	38259	1,40	0,016752	5
	37376	37447,2	37548	16,26	37376	37675	38446	2,05	0	7
	20361	20361	20361	7,41	20361	20361	20361	1,02	0	9
eil76.rcp	2460	2461	2465	91,10	2507	2511,2	2515	4,11	0,019106	5
	2504	2504	2504	83,70	2504	2504	2504	5,68	0	7
	1710	1710	1710	30,14	1710	1710	1710	6,17	0	9
pr76.rcp	500395	500403	500405	133,81	502225	509954	513713	4,21	0,003657	5
	556939	557755,8	558555	74,643	555858	557953,4	559760	4,67	-0,001941	7
	424359	424359	424359	31,22	424359	424661	425114	3,94	0	9
rat99.rcp	5885	5898	5915	331,90	5997	6094,2	6160	9,92	0,019031	5
	6447	6460,8	6476	229,20	6436	6442	6451	8,92	-0,001706	7
	5150	5158,8	5164	140,50	5150	5151,6	5158	8,25	0	9
rd100.rcp	37975	38006	38060	330,45	38447	38872,6	39175	10,58	0,012429	5
	40952	41033	41131	237,33	40971	41046,2	41347	10,18	0,000464	7
	31776	31780,8	31784	120,95	31776	31776	31776	9,49	0	9
kroA100.rcp	100785	100797	100845	321,652	101230	103126,6	105305	10,25	0,004415	5
	115438	115519,4	115607	282,35	115388	117095,4	120601	11,82	-0,000433	7
	94467	94574,4	94852	101,00	94265	94652,6	95697	8,03	-0,002138	9
kroB100.rcp	104575	104758	104880	350,46	105073	107006	108159	10,75	0,004762	5
	118112	118481,4	118661	240,84	118183	119392,8	122302	9,43	0,000601	7
	94018	94047,6	94055	87,48	93938	93965,6	94026	8,04	-0,000851	9
kroC100.rcp	99570	99588	99600	318,58	99940	100747,6	102000	10,25	0,003716	5
	113533	113566	113698	266,27	113533	113533	113533	11,57	0	7
	92894	92894	92894	169,62	92894	92894	92894	25,92	0	9
kroD100.rcp	101795	101896	102115	337,98	103998	104948,8	106275	9,97	0,021543	5
	117297	117525,8	117814	304,49	116924	118105,2	121016	9,84	-0,003180	7
	92225	92249,8	92349	129,51	92102	92102	92102	18,95	-0,001334	9
kroE100.rcp	104915	105079	105220	341,77	105003	105693,4	106547	10,67	0,000839	5
	116471	116471	116471	253,31	116471	117562	119027	10,91	0	7
	96116	96281,2	96346	116,10	96116	96119,6	96122	107,07	0	9

6 Concluding Remarks

In this work, we proposed a new heuristic based on Greedy Randomized Adaptive Search Procedure (GRASP) and General Variable Neighborhood Search (GVNS) applied to the Ring Star Problem. The new heuristic produces within reasonable computing times highly accurate solutions and compares well with a more elaborate heuristic developed in [9]. In the majority of the test problems, GRASP+GVNS heuristic gives a better minimum objective value than VNTS. Also, it must be emphasized that GRASP+VNS is more robust than VNTS because the difference between the maximum and minimum objective values is smaller in GRASP+GVNS than in VNTS.

Future research will be focus on applying this hybrid approach to larger instances of the categories tested.

References

1. T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.
2. P. Festa and M. G. C. Resende. An annotated bibliography of GRASP. Technical Report TD-5WYSEW, AT&T Labs Research, 2004.

3. E. Gourdin, M. Labbé, and H. Yaman. Telecommunication and location. In Z. Drezner and H. Hamacher, editors, *Facility Location: applications and theory*, pages 275– 305. Springer, 2002.
4. P. Hansen and N. Mladenović. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.
5. P. Hansen and N. Mladenović. Variable neighbourhood search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, page 145184. Kluwer, 2003.
6. M. Labbé, G. Laporte, I. R. Martín, and J.J. S. González. The ring star problem: Polyhedral analysis and exact algorithm. *Networks*, 43(3):177–189, 2004.
7. M. Labbé, G. Laporte, I. R. Martín, and J.J. S. González. Locating median cycles in networks. *European Journal of Operational Research*, 160(2):457–470, 2005.
8. S.L. Martins and C.C. Ribeiro. Metaheuristics and applications to optimization problems in telecommunications. In M.G.C. Resende and P. M. Pardalos, editors, *Handbook of Optimization in Telecommunications*, pages 103–128. Springer-Verlag, 2006.
9. J. M. Pérez, M. Moreno Vega, and I. R. Martín. Variable neighbourhood tabu search and its application to the median cycle problem. *European Journal of Operational Research*, 151:365–378, 2003.

An Incremental Model for Combinatorial Maximization Problems

Jeff Hartline and Alexa Sharp

Cornell University, Ithaca, NY 14853
{jhartlin, asharp}@cs.cornell.edu

Abstract. Many combinatorial optimization problems aim to select a subset of elements of maximum value subject to certain constraints. We consider an incremental version of such problems, in which some of the constraints rise over time. A solution is a sequence of feasible solutions, one for each time step, such that later solutions build on earlier solutions incrementally. We introduce a general model for such problems, and define incremental versions of maximum flow, bipartite matching, and knapsack. We find that imposing an incremental structure on a problem can drastically change its complexity. With this in mind, we give general yet simple techniques to adapt algorithms for optimization problems to their respective incremental versions, and discuss tightness of these adaptations with respect to the three aforementioned problems.

Keywords: analysis of algorithms, approximation techniques, combinatorial problems, network analysis, online problems.

1 Introduction

There has been recent interest in incremental versions of classic problems such as facility location [1], k-median [2], maximum flow [3], and k-centre [4]. These problems model situations in which there is a natural hierarchy of levels with different characteristics, such as local vs. wide-area networks or multilevel memory caches. Incremental variations of NP-hard problems contain their non-incremental versions as special cases and therefore remain NP-hard. It is interesting to ask whether incremental versions of polytime problems remain polytime, or whether the incremental structure alters the problem enough to increase its complexity.

Traditional algorithms require all input at the outset and then determine a single solution. In practice, however, many problems require solutions to be built up over time due to limited resources and rising constraints. In this scenario, one or more of the inputs to the problem increases at discrete time intervals. One wants the best solution at each step, subject to the constraint that elements in the current solution cannot be removed. We would like a good solution at all stages; however, a commitment at some stage may limit the options at later stages. It is this tension between local and global optimality that makes these incremental problems challenging.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 36–48, 2006.

For example, consider the routing of a message from a source to a destination in an untrusted network. To increase security and preserve power at the internal router nodes, it is desirable to divide the message among node-disjoint paths, as proposed by Lou and Fang [5]. Adversaries within the network must intercept all components of a well-divided message before they are able to decipher it. Also, transmitting a message along many node-disjoint paths prevents individual routers from depleting their battery faster than their neighbors. Thus more node-disjoint paths available in the network results in an increase in data confidentiality and energy efficiency.

Now suppose the underlying network changes over time: transmission distances increase and links appear between existing routers. We still want as many node-disjoint paths as possible, but we do not want to reprogram routers. More specifically, we would prefer not to change any of the existing node-disjoint paths.

The above example is an application of *incremental maximum flow*, defined on a directed network with source s, sink t, and a non-decreasing sequence of capacity functions, one for each time step. The goal is to find a sequence of s-t flows such that each flow satisfies its corresponding capacity constraints but does not remove flow from any prior solution. For each time step ℓ, we compare the value of the ℓ^{th} flow to the value of the optimal flow satisfying the current capacity constraints. We want this ratio to be close to 1, but because we are constrained to keep all flow from previous time steps, this may not be possible. Therefore one goal is to find a sequence of flows that maximizes the minimum of this ratio over all values of ℓ. An algorithm for this problem is said to be r-*competitive* if the minimum of the ratio over all ℓ is no less than r. This value of r is called the *competitive ratio* of the algorithm. Alternatively, if we are more interested in overall performance rather than the performance at each time step, our goal would be to maximize the sum of the solutions over all time steps. In our routing example, this objective would correspond to maximizing the throughput over all time.

Following this framework, we can define incremental versions of other combinatorial maximization problems. *Incremental bipartite matching* is defined on a bipartite graph where edges appear over time. A solution for this problem is a sequence of matchings, one for each time step, such that each matching contains all previous matchings. This model can be applied to job scheduling, as it can be costly or disruptive to reassign jobs. In *incremental knapsack* we are given a set of items with sizes and a sequence of increasing knapsack capacities. We want a sequence of knapsack solutions such that each solution contains all the items in the previous solution. Memory allocation is one application of incremental knapsack.

Our Results. We introduce a general incremental model and analyze the complexity of the three incremental problems defined above with respect to two different objective functions: maximum ratio and maximum sum. We find that incremental bipartite matching remains polytime in many cases and becomes

Table 1. Best known approximation factors for some maximization problems and their incremental variants. $*$: n is the number of nodes in the flow network. $**$: fully polytime approximation scheme from [19, 20]. $\mathcal{H}_k$ is the k^{th} harmonic number, which is $\Theta(\log(k))$.

Problem	Single-Level	k-Level Max Sum	k-Level Max Ratio
Bipartite Matching	1	1	1 $(k = 2)$ 1/2 $(k > 2)$
Weighted Bipartite Matching	1	1	NP-hard $(k \geq 2)$
Max Flow	1	$1/\mathcal{H}_k$ (tight)	$O(1/n)^*$ (tight)
Knapsack	$1 - \epsilon^{**}$	$1/\mathcal{H}_k$	$(1-\epsilon)^2/2$

NP-hard in others, whereas incremental max flow is NP-hard even for very basic models. Our central contribution is a general technique to translate exact or approximate algorithms for non-incremental optimization problems into approximation algorithms for the corresponding incremental versions. We find that these techniques yield tight algorithms in the case of max flow, but can be improved for bipartite matching and knapsack. The best known approximation bounds are given in Table 1.

The incremental model is laid out in Section 2. We present complexity results for the max sum objective in Section 3 and analogous results for the max ratio objective in Section 4. We conclude with some extensions in Section 5.

Related Work. Several incremental problems have been studied recently; Mettu and Plaxton [2] study incremental versions of uncapacitated k-median and give a 29.86-competitive algorithm. Plaxton [1] introduces incremental facility location and gives a $(1 + \epsilon)\alpha$-competitive algorithm, given an α-approximation to uncapacitated facility location. This results in a 12.16-competitive algorithm. Gonzales [6] gives a 2-approximation algorithm for k-center, which is also a 2-competitive algorithm for the incremental k-center problem studied by [2, 1, 4, 7]. Lin et al. [4] present a general framework for cardinality constrained minimization problems, resulting in approximation algorithms for k-vertex cover and k-set cover, an improved approximation algorithm for incremental k-median, and alternative approximation algorithms for incremental k-MST and incremental facility location.

In the area of polynomial time problems, Hartline and Sharp [3] introduce incremental versions of max flow, and find the first instance of a polynomial-time problem whose incremental version is NP-hard. In [8] we present a general model and algorithmic framework for combinatorial covering problems, achieving a 4α-approximation for any incremental covering problem with an α-approximation for its offline version.

Online problems share many similarities with the incremental model. For instance, their input changes with time and their solutions build on each other incrementally. However, online algorithms act with no knowledge of future input and are evaluated only on their final output [9, 10]. Online algorithms have been

studied in many contexts, including bin packing [11], graph coloring [12], and bipartite matching [13].

Stochastic optimization also resembles our incremental framework, in that instances have multi-stage input and incremental solutions. However, the problem instance is not fully known at the outset, and the goal is to find a single solution of minimum cost. We motivate our general model by those developed for stochastic problems [14, 15, 16]. General models for single-level optimization problems are available in [17, 18].

2 Preliminaries

Single-Level Problems. We define a single-level abstract combinatorial optimization problem that we adapt to the incremental setting. Such a problem Π consists of a ground set from which we select a subset of elements of optimal value that satisfy input constraints. In particular, let X be the ground set, $\mathcal{F} \subseteq 2^{\mathtt{X}}$ the set of *feasible solutions* as defined by problem constraints, and $v : 2^{\mathtt{X}} \to \mathbb{R}$ a valuation function on element sets. The goal is to return an $S \in \mathcal{F}$ optimizing $v(S)$. Let $\mathsf{OPT}(X, \mathcal{F}, v)$, or $\mathsf{OPT}(\mathcal{F})$ when X and v are understood, denote such a solution.

This notation is adapted from the general minimization models of [14, 15], however it is general enough to represent both maximization and minimization problems. This paper considers packing problems, a subclass of maximization problems that are "monotone," in the sense that any subset of a feasible solution is also feasible. In particular, if $\mathcal{F}$ is nonempty then the empty set is a feasible solution: $\emptyset \in \mathcal{F}$. We further assume that $v(\emptyset) = 0$.

Incremental Problems. Given any maximization problem Π, we define its *incremental version* Π^k as follows. There will be k *levels*. Each level ℓ has its own feasible set $\mathcal{F}_\ell$. A feasible solution is a tuple $\mathbf{S} = (S_1, S_2, \ldots, S_k)$ such that $S_\ell \in \mathcal{F}_\ell$ and $S_1 \subseteq S_2 \subseteq \cdots \subseteq S_k$. Although we do not explicitly assume that $\mathcal{F}_\ell \subseteq \mathcal{F}_{\ell+1}$, we may do so without loss of generality.

In contrast to the single-level problem, where the goal is to find a solution of maximum value, there are several possible objective functions in the incremental variation. For the *maximum ratio* problem, the objective is to satisfy the maximum possible proportion of each level's optimal solution: find $\mathbf{S}$ maximizing $\mathcal{R}(\mathbf{S}) = \min_\ell \frac{v(S_\ell)}{v(\mathsf{OPT}(\mathcal{F}_\ell))}$. This is the same as the competitive ratio of an online problem, and is a standard metric for incremental problems [1, 2]. If one is less concerned about fairness over all levels, and is more concerned about overall performance, then the maximum sum objective is more appropriate: for the *maximum sum* problem, the objective is to maximize the sum of the solutions over all levels: find $\mathbf{S}$ maximizing $\mathcal{V}(\mathbf{S}) = \sum_\ell v(S_\ell)$.

We now consider three well-known problems, and demonstrate how they fit into this framework. There are multiple ways to define incremental versions of these problems, but we introduce only those subject to discussion in this paper.

2.1 Bipartite Matching

A bipartite matching instance consists of a graph $G = (U \cup V, E)$; the elements are the edges of the graph, and the feasible solutions are matchings contained in E. The value of a matching M is $v(M) = |M|$.

Incremental bipartite matching is defined on a sequence of k bipartite graphs $G_\ell = (U \cup V, E_\ell)$, where $E_\ell \subseteq E_{\ell+1}$. The elements are the edges of E_k, and the feasible set at level ℓ is just the matchings of E_k contained in E_ℓ. Therefore a solution is a sequence of matchings $(M_1, M_2, \ldots, M_k)$ such that M_ℓ is a matching in the graph G_ℓ, and $M_\ell \subseteq M_{\ell+1}$. The maximum single-level matching for level ℓ is denoted by M^*_ℓ. The weighted case is defined analogously, except each edge $e \in E_\ell$ has a fixed weight $w_e \geq 0$, and $v(M_\ell) = \sum_{e \in M_\ell} w_e$.

2.2 Maximum Flow

A maximum flow instance consists of a directed graph $G = (V, E)$ with source s, sink t, and a capacity function c; the elements are unit s-t flow paths, and the feasible solutions are the flows satisfying the given capacity function. The value of a flow is the number of unit s-t flow paths it contains.

Incremental maximum flow is defined on a directed graph $G = (V, E)$ with source s, sink t, and a non-decreasing sequence of k capacity functions $c_\ell : E \to \mathbb{Q}, 1 \leq \ell \leq k$, that define k feasible sets. A solution is a sequence of s-t flows $(f_1, f_2, \ldots, f_k)$ such that the flow f_ℓ on any edge e does not exceed the capacity $c_\ell(e)$ but is at least $f_{\ell-1}(e)$, the amount sent along e by the previous flow. We denote the value of a flow f_ℓ by $|f_\ell|$, and the maximum single-level flow at level ℓ by f^*_ℓ.

For other possible interpretations of incremental max flow, see [3].

2.3 Knapsack

A knapsack instance consists of capacity B and a set of items U, item $u \in U$ with size $|u|$ and value v_u; the elements are the items we could place in the knapsack, while the feasible solutions are subsets of items that fit in the knapsack. In this paper we only consider the case where $v_u = |u|$; the value of a set of items U' is therefore the combined size $|U'| = \sum_{u \in U'} |u|$. This special case is sometimes called the maximum subset sum problem.

Incremental knapsack is defined on a set of items U, item $u \in U$ with size $|u|$, but instead of a single capacity B we have a sequence of k capacities $B_1 \leq B_2 \leq \cdots \leq B_k$ that define k feasible sets. A solution is a sequence of subsets $(U_1, U_2, \ldots, U_k)$ of U such that $|U_\ell| \leq B_\ell$, and $U_\ell \subseteq U_{\ell+1}$. We denote the value of the maximum single-level solution at level ℓ by B^*_ℓ.

3 The Maximum Sum Objective Function

In this section we discuss how the max sum incremental structure affects the complexity of the problems introduced in Section 2. We give a general technique

to convert an algorithm for a problem Π into an approximation algorithm for its incremental variant Π^k, and analyze its tightness with respect to these problems.

Theorem 1. *Max sum weighted incremental bipartite matching is in P.*

Proof. We transform our incremental instance $(G_1, G_2, \ldots, G_k, w)$ into a single instance (G, w') of the max weight matching problem, which can then be solved in polytime [21]. We create a graph $G = (V, E)$ where $E = E_k$. For each edge e, we assign it weight $w'_e = w_e \cdot (k - \ell + 1)$ if e first appears in the edge set E_ℓ, i.e. if $e \in E_\ell \backslash E_{\ell-1}$. This is the amount e would contribute to the sum if we were to add it to our solution at level ℓ. For a matching M returned by the max weight matching algorithm, we define an incremental solution $M_\ell = M \cap E_\ell$. We argue that M is a maximum weight matching if and only if $(M_1, M_2, \ldots, M_k)$ is the optimal weighted incremental max sum solution. This follows from the one-to-one correspondence between the value of the maximum weight matching and the value of our incremental solution:

$$w(M) = \sum_{e \in M} w'_e \;=\; \sum_{\ell=1}^{k} \sum_{e \in M_\ell \backslash M_{\ell-1}} w'_e \;=\; \sum_{\ell=1}^{k} \sum_{e \in M_\ell \backslash M_{\ell-1}} w_e \cdot (k - \ell + 1)$$

$$= \sum_{\ell=1}^{k} w(M_\ell \backslash M_{\ell-1})(k - \ell + 1) \;=\; \sum_{\ell=1}^{k} w(M_\ell) \;=\; \mathcal{V}(M_1, M_2, \ldots, M_k). \;\square$$

Theorem 1 shows that the max sum incremental structure does not affect the complexity of bipartite matching, suggesting that incremental versions of polytime problems may remain polytime. However, Theorem 3.1 of [3] can be extended to show that adding an incremental structure to max flow alters it enough to significantly change its complexity. This illustrates a dichotomy between the closely related problems of bipartite matching and max flow.

Theorem 2. *[3] Max sum incremental flow is NP-hard.*

As there are many incremental problems for which no polytime algorithm exists, we turn our attention to approximation algorithms.

Theorem 3. *Given an α-approximation algorithm* ALG *for a problem Π, we obtain an $O(\frac{\alpha}{\log k})$-approximation for its max sum incremental version Π^k.*

Proof. We first run the approximation algorithm for each single-level input to obtain $\mathsf{ALG}(\mathcal{F}_\ell)$ with $v(\mathsf{ALG}(\mathcal{F}_\ell)) \geq \alpha \cdot v(\mathsf{OPT}(\mathcal{F}_\ell))$. We then consider the k incremental solutions

$$\mathbf{H}_\ell = (\underbrace{\emptyset, \emptyset, \ldots, \emptyset}_{\ell-1}, \mathsf{ALG}(\mathcal{F}_\ell), \ldots, \mathsf{ALG}(\mathcal{F}_\ell))$$

for which $\mathcal{V}(\mathbf{H}_\ell) = (k - \ell + 1) \cdot v(\mathsf{ALG}(\mathcal{F}_\ell))$. Out of these k solutions, return one of maximum value. Denote this solution by $\mathbf{H}^*$ such that for all ℓ

$$\mathcal{V}(\mathbf{H}^*) \geq (k - \ell + 1) \cdot \alpha \cdot v(\mathsf{OPT}(\mathcal{F}_\ell)), \text{ and therefore}$$

$$v(\mathsf{OPT}(\mathcal{F}_\ell)) \leq \frac{1}{\alpha} \cdot \frac{1}{k - \ell + 1} \cdot \mathcal{V}(\mathbf{H}^*).$$

If $\mathbf{O}^*$ is an optimal incremental solution, then

$$\mathcal{V}(\mathbf{O}^*) \leq \sum_{\ell=1}^{k} v(\mathsf{OPT}(\mathcal{F}_\ell)) \;\leq\; \mathcal{V}(\mathbf{H}^*) \cdot \frac{1}{\alpha} \cdot \sum_{\ell=1}^{k} \frac{1}{k-\ell+1}$$
$$= \mathcal{V}(\mathbf{H}^*) \cdot \frac{\mathcal{H}_k}{\alpha} \quad = \quad \mathcal{V}(\mathbf{H}^*) \cdot O\left(\frac{\log k}{\alpha}\right),$$

where $\mathcal{H}_k$ is the k^{th} harmonic number. □

While this algorithm is not tight for incremental bipartite matching, Theorem 4 shows it is tight for incremental max flow. The proof relies heavily on gadgetry described in [3], in which we show that any 3-SAT instance can be converted into a two-level unit-capacity directed flow network. This network has two linked components: a clause component c consisting of level 1 edges and a variable component v consisting of level 2 edges. If the clause component appears[1] at level ℓ_c and its corresponding variable component appears at level $\ell_v > \ell_c$, then the results of [3] can easily be extended into the following lemma:

Lemma 1. *Let $\ell'_c \geq \ell_c$ denote the earliest level in which clause component c carries flow. If $\ell'_c < \ell_v$, then any flow through variable component v determines a satisfying assignment. Also, any satisfying assignment can be used to achieve a flow with separate flow paths through components c and v.*

Theorem 4. *Max sum incremental flow is NP-hard to β-approximate, $\beta > \frac{1}{\mathcal{H}_k}$.*

Proof. Suppose we have a $1/(\mathcal{H}_k - \epsilon)$-approximation for max sum on k-level networks. We solve any instance of 3-SAT by constructing an incremental flow network and using the approximation algorithm to identify satisfiable formulas.

First, let $b = \frac{1}{\epsilon}$. Define $a_0^* = 0$, and $a_\ell^* = \lfloor \frac{bk}{1+k-\ell} \rfloor$ for integers $1 \leq \ell \leq k$. Observe that $\sum_{\ell=1}^{k} a_\ell^* > bk(\mathcal{H}_k - \epsilon)$ because $\lfloor \frac{bk}{1+k-\ell} \rfloor > \frac{bk}{1+k-\ell} - 1$. Given an instance ϕ of 3-SAT, we build a k-level flow network using $O(b^2k^2)$ copies of the clause-variable component pairs constructed from ϕ. We create a $b(k-1) \times bk$ matrix of components as shown in Figure 1. Each level ℓ is assigned columns $a_{\ell-1}^* + 1$ through a_ℓ^*. Each such column j contains variable components $v_{1j}, v_{2j}, \ldots, v_{a_{\ell-1}^* j}$ and clause components $c_{j(a_\ell^*+1)}, \ldots, c_{j(bk-1)}, c_{j(bk)}$, all linked in series between the source and the sink. Components in these columns contain only level ℓ edges. Variable component v_{ab} is linked to clause component c_{ab}.

In this construction, the maximum flow possible at level ℓ has value a_ℓ^*, thus $UB = \sum_{\ell=1}^{k} a_\ell^*$ is an upper bound on the flow sum. This is strictly larger than $bk(\mathcal{H}_k - \epsilon)$ as noted earlier. Observe that any level ℓ flow must pass through clause components $c_{j(a_\ell^*+1)}, \ldots, c_{j(bk)}$ for some column $j \leq a_\ell^*$. If we ever send more than a_ℓ^* units of flow then this extra flow must pass through variable component $v_{jj'}$ for some $a_\ell^* < j' \leq bk$. Thus by Lemma 1 any flow strictly larger than a_ℓ^* that contains positive flow at level ℓ yields a satisfying assignment for ϕ.

[1] A component is said to *appear* at level ℓ if, in the incremental flow network, all of its edges have capacity 0 prior to level ℓ and capacity 1 for all subsequent levels.

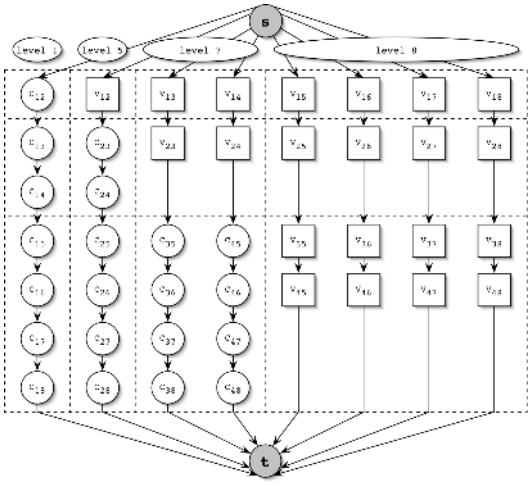


Fig. 1. Circles denote clause components and squares denote variable components. Clause-variable tuple (c_{ij}, v_{ij}) consists of clause c_{ij} component in column i and variable v_{ij} in row j. $k = 8$ and $\epsilon = b = 1$.

If such an assignment exists, we can achieve the flow sum upper bound UB by applying Lemma 1 to send flow through all clause-variable pairs. If no such assignment exists, consider incremental solution $(f_1, f_2, \ldots, f_k)$ and take the smallest ℓ such that $|f_k| \leq a^*_\ell$. Because there is no assignment, $|f_1| = \ldots = |f_{\ell-1}| = 0$. Also, $|f_\ell| \leq \ldots \leq |f_k| \leq a^*_\ell$, and therefore our flow sum $\sum_\ell |f_\ell| \leq (1+k-\ell)a^*_\ell = (1+k-\ell)(\lfloor \frac{bk}{1+k-\ell} \rfloor) \leq bk$. We use our $1/(\mathcal{H}_k - \epsilon)$-approximation to distinguish between these cases, and therefore determine whether or not ϕ has a satisfying assignment. □

The standard pseudo-polynomial dynamic programming algorithm for knapsack can be extended to a $O((B_k)^k)$ algorithm for max sum incremental knapsack. We suspect that similar techniques to those of Section 4 will give a max sum approximation polynomial in k with a better ratio than that established in Theorem 3, however this has yet to be proven.

4 The Maximum Ratio Objective Function

In this section, we give analogous results for the max ratio objective function.

Theorem 5. *Max ratio 2-level incremental bipartite matching is in P.*

Proof. We transform an incremental instance G_1, G_2 into a single instance (G, w) of the maximum weight matching problem. We create a graph $G = (V, E)$ with $E = E_2$. For each edge e, we assign it weight 1 if $e \in E_1$ and 0 otherwise. For each $1 \leq m \leq |M^*_2|$ we find the max weight matching M^m of size m. From each such matching we define an incremental solution $M^m_\ell = M^m \cap E_\ell$. We return a solution (M^m_1, M^m_2) of maximum ratio. By the nature of the weights given

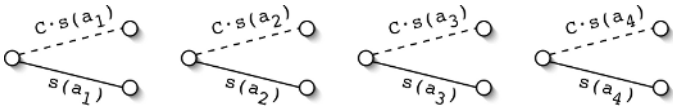


Fig. 2. A weighted incremental bipartite matching instance constructed from an instance of partition with $A = \{a_1, a_2, a_3, a_4\}$. Solid and dashed lines represent level 1 and level 2 edges, respectively. Edges are labeled with their weights.

to level 1 edges, if an (m', m) matching exists then $|M_1^m| \geq m'$. Therefore our solution must have a ratio no worse than that of the (m', m) matching. □

This technique can be generalized for arbitrary k when the optimal ratio r^* is 1. However, the following theorem shows that adding weights makes the problem intractable, and distinguishes it from the polytime max sum version.

Theorem 6. *Max ratio 2-level weighted incremental matching is NP-hard.*

The proof of Theorem 6 follows from a reduction from *partition*, known to be NP-complete [22]. Given a finite set A and sizes $s(a) \in \mathbb{Z}^+$ for all a in A, the partition problem finds a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)$. We construct a 2-level instance of weighted incremental bipartite matching such that a ratio of $\frac{1}{2}$ is achievable if and only if the desired partition of A exists.

For each element $a \in A$, we create a 2-level gadget consisting of two incident edges e_a^1 and e_a^2. Edge e_a^1 is a level 1 edge of weight $s(a)$, and edge e_a^2 is a level 2 edge of weight $C \cdot s(a)$, for some $C > 1$. This construction is shown in Figure 2.

Let $S = \sum_{a \in A} s(a)$. Then the optimal level 1 matching M_1^* selects all level 1 edges, with total weight S. The optimal level 2 matching M_2^* selects all level 2 edges, with total weight $C \cdot S$. Let us define $C = S + 1$.

Lemma 2. *There is a partition of A if and only if there exists an incremental matching achieving ratio $\frac{1}{2}$.*

[⇒] Suppose we have a partition $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a) = \frac{S}{2}.$$

We create an incremental matching (M_1, M_2) by selecting $M_1 = \{e_a^1 \mid a \in A'\}$ and $M_2 = M_1 \cup \{e_a^2 \mid a \in A \setminus A'\}$. This is a feasible solution, as we use exactly one gadget edge for each element $a \in A$ in our incremental matching. Furthermore,

$$\begin{aligned} r_1 &= \frac{w(M_1)}{w(M_1^*)} \\ &= \frac{\sum_{a \in A'} s(a)}{S} \\ &= \frac{1}{2} \end{aligned} \qquad \begin{aligned} r_2 &= \frac{w(M_2)}{w(M_2^*)} \\ &= \frac{w(M_1) + C \sum_{a \in A \setminus A'} s(a)}{C \cdot S} \\ &= \frac{\frac{S}{2} + C \cdot \frac{S}{2}}{C \cdot S} \quad \geq \quad \frac{1}{2} \end{aligned}$$

[⇐] Now suppose we have an incremental matching (M_1, M_2) achieving ratio $\frac{1}{2}$. First we claim that $w(M_1) = \frac{S}{2}$: If not, then in order to achieve the stated ratio, we have $w(M_1) \geq \frac{S}{2} + 1$, and hence $w(M_2) \leq \frac{S}{2} + 1 + C \cdot (\frac{S}{2} - 1)$. But then

$$\begin{array}{rcccl} r_2 \leq \frac{\frac{S}{2}+1+C\cdot(\frac{S}{2}-1)}{C\cdot S} & = & \frac{1}{2} + \frac{S+2-2C}{2\cdot C\cdot S} & & \\ = \frac{1}{2} + \frac{S+2-2\cdot(S+1)}{2\cdot(S+1)\cdot S} & = & \frac{1}{2} - \frac{1}{2\cdot(S+1)} & < & \frac{1}{2} \end{array}$$

which contradicts our ratio of $\frac{1}{2}$. Therefore we define A' to be the elements a such that $e_a^1 \in M_1$ so that $\sum_{a \in A'} s(a) = w(M_1) = \frac{S}{2}$. □

Theorem 6 follows directly from Lemma 2. However, the hardness of incremental bipartite matching with polysize weights is unknown, as there is a pseudo-polynomial time dynamic programming algorithm for partition.

Despite the hardness of weighted incremental bipartite matching, Theorem 5 still manages to distinguish incremental matching from incremental max flow, which is NP-hard for $r^* = 1$ and unit capacities. This follows from an extension of the following theorem from [3].

Theorem 7. *[3] Max ratio incremental flow is NP-hard.*

Furthermore, [3] proves that the greedy algorithm, which repeatedly sends the maximum level ℓ flow given the incremental constraints imposed by previous levels, is a $\frac{1}{n}$-approximation algorithm, and this algorithm is tight.

Theorem 8. *[3] Max ratio incremental flow is NP-hard to g(n)-approximate for* $g \in \omega(\frac{1}{n})$.

In summary, we have a complete picture of hardness for max ratio incremental flow, and results for many cases of incremental bipartite matching. However, we have yet to discover the hardness of 3-level incremental bipartite matching with unit weights and $r < 1$, or for weighted bipartite matching with polynomial weights. Now we turn our eye to our final result: we present a constant-factor approximation algorithm for max ratio incremental knapsack.

We introduce some assumptions and notation before we present Lemmas 3-5 and the resulting algorithm. Let r^* denote the optimal max ratio. We assume items $U = \{u_1, u_2, \ldots, u_n\}$ are ordered by non-decreasing size, and we define σ_j to be the sum of the sizes of the first j items in this ordering. We say that level ℓ is *σ-good* if $\frac{r^*}{2} B_\ell^* \leq \sigma_j \leq B_\ell$ for some j, i.e. if the j smallest items are an $\frac{r^*}{2}$-approximation to B_ℓ^*. Level ℓ is *σ-bad* if it is not σ-good. If level ℓ is σ-bad then there is some j such that $\sigma_j < \frac{r^*}{2} B_\ell^*$ but $\sigma_{j+1} > B_\ell$. The following lemma, stated without proof, implies that the optimal incremental solution for this level contains an item at least as big as u_{j+1}. We call this item level ℓ's *required item.*

Lemma 3. *Given knapsack size B and solution $\hat{U}$, if $U' \subseteq U$ is a maximal solution but $|U'| < |\hat{U}|/2$, then $\hat{U}$ contains exactly one item of size greater than $|\hat{U}|/2$.*

Lemma 4. *If u_j is the required item of the last σ-bad level ℓ, then any $\frac{r^*}{2}$-approximation for levels $1..\ell$ with $u_j \in U_\ell$ can be extended to an $\frac{r^*}{2}$-approximation for levels $1..k$.*

Proof. By definition of ℓ and u_j we have $|u_j| > (1 - \frac{r^*}{2})B_\ell > \frac{1}{2}B_\ell$. Therefore any solution requiring $u_j \in U_\ell$ cannot contain items of size greater than $|u_j|$ in any of the first ℓ levels. Each level $h > \ell$ is σ-good, thereby having some $\sigma_{i_h} \geq \frac{r^*}{2}B_h^* \geq \frac{r^*}{2}B_\ell^*$. As any solution with $u_j \in U_\ell$ only contains items also in σ_{i_h} for all $h > \ell$, and all such levels h are σ-good, we can extend any such solution to all k levels by using the σ_{i_h} solution on levels $h > \ell$. □

Lemma 5. *If u_j is the required item of the last σ-bad level ℓ, then there exists some level $\ell' \leq \ell$ where we can place u_j such that an r^* solution still exists for levels $1..\ell' - 1$.*

Proof. Consider some optimal incremental solution, and let ℓ' be the earliest level that uses some item $u_{j'}$ at least as big as u_j. Replacing $u_{j'}$ with u_j in this solution does not affect the ratio achieved for levels 1 through $\ell' - 1$. □

The dynamic programming solution presented below assumes we know the optimal ratio r^* as well as the optimal single-level solutions $B_1^*, B_2^*, \ldots, B_k^*$. Under these assumptions, the algorithm achieves a $\frac{1}{2}$-approximation for max ratio knapsack. We then remove these assumptions at the expense of adding a $(1-\epsilon)^2$-factor to the approximation bound.

Knapsack Algorithm. Add dummy level B_{k+1} and item u_{n+1} with $B_{k+1} >> B_k$ and $|u_{n+1}| = |u_n|$. We build a table $M[1..k, 1..n]$. Entry $M[\ell, j]$ is an $\frac{r^*}{2}$-solution for levels $1..\ell$, items $\{u_1, u_2 \ldots, u_j\}$, and modified capacities $B_\ell = \min\{B_\ell, B_{\ell+1} - |u_{j+1}|\}$ if we find a solution and $\emptyset$ otherwise. If an r^* solution exists for this modified problem, we guarantee $M[\ell, j] \neq \emptyset$. We return $M[k, n]$.

$M[0, j]$ is the empty tuple as there are no levels. To compute $M[\ell, j]$ we assume that subproblem $[\ell, j]$ has an r^* solution. If this is not the case then the value of the entry does not matter, and we can set $M[\ell, j] = \emptyset$ if we ever have trouble executing the following procedure.

We first consider the smallest item first solution. If all levels are σ-good we return this greedy $\frac{r^*}{2}$-solution. Otherwise, there is at least one σ-bad level. Let $u_{j'}$ be the required item of the last σ-bad level y, which must exist assuming an r^* solution is feasible.

We pick the first $1 \leq \ell' \leq y$ such that $B_{\ell'} > u_{j'}$ and $M[\ell' - 1, j' - 1] \neq \emptyset$. We solve levels $1..\ell' - 1$ using $M[\ell' - 1, j' - 1]$, levels $\ell'..y$ by adding $u_{j'}$ at level ℓ', and levels $y + 1..\ell$ with the smallest item first algorithm. Levels $1..\ell' - 1$ are satisfied by definition of $M[\ell' - 1, j' - 1]$, levels $\ell'..y$ are satisfied by Lemma 3, and levels $y + 1..\ell$ are satisfied by Lemma 4. Moreover, because an r^* solution is feasible, Lemma 5 guarantees that such an ℓ' exists. If no such ℓ' exists, it must have been because no r^* solution was possible, and we set $M[\ell, j] = \emptyset$.

The running time of this algorithm is dominated by the computation of $M[\ell, j]$ for all nk entries. Each entry requires $O(n)$ time and therefore the running time is $O(kn^2)$.

Theorem 9. *For incremental knapsack, there is a* $\frac{(1-\epsilon)^2}{2}$*-approximation to the max ratio objective function that runs in time* $O(\frac{k^2n^5}{\epsilon}\frac{\log(u_n)}{-\log(1-\epsilon)})$.

Proof. Given r^* and B^*_ℓ for all levels ℓ, the above algorithm $\frac{1}{2}$-approximates max ratio knapsack. Although determining B^*_ℓ is NP-complete, we can use an FPTAS to find a solution $\hat{U}_\ell$ with $|\hat{U}_\ell| \geq (1-\epsilon)B^*_\ell$ in time $O(n^2\lfloor\frac{n}{\epsilon}\rfloor)$ [19, 20].

The greedy algorithm (smallest item first) gives us a lower bound on r^* of $1/u_n$. We run our algorithm with $r^* = 1/u_n$. If we find a ratio $\frac{r^*}{2}$ solution then multiply r^* by $1/(1-\epsilon)$ and try again. We continue until the algorithm fails to find a 1/2-approximation for some $r^* = \frac{1}{u_n}(\frac{1}{1-\epsilon})^q$. At this point, $\frac{1}{u_n}(\frac{1}{1-\epsilon})^{q-1} \leq r^* < \frac{1}{u_n}(\frac{1}{1-\epsilon})^q$, so if we take $\hat{r} = \frac{1}{u_n}(\frac{1}{1-\epsilon})^{q-1}$ then $\hat{r} \geq (1-\epsilon)r^*$. This may take $\frac{\log(u_n)}{-\log(1-\epsilon)}$ iterations, but can be accelerated by binary search.

With $\hat{r}$ and $\hat{U}_\ell$, the algorithm finds a solution $(U_1, U_2, \ldots, U_k)$ such that for each level ℓ

$$|U_\ell| \;\geq\; \frac{\hat{r}}{2}|\hat{U}_\ell| \;\geq\; \frac{(1-\epsilon)^{2\cdot}}{2} \cdot r^* B^*_\ell.$$

The time needed to compute $\hat{r}$, $\hat{U}_\ell$, and run the algorithm is $O(k^2n^4\lfloor\frac{n}{\epsilon}\rfloor \cdot \frac{\log(u_n)}{-\log(1-\epsilon)})$. □

5 Extensions

The large field of related work discussed in Section 1 motivates many interesting extensions to the results presented in this paper. The competitive ratio of an online algorithm is a comparison between the algorithm's output and the best offline non-incremental solution. Online solutions, however, are inherently incremental, hence analysis of these online algorithms could benefit from theoretical results on the corresponding offline incremental problem. This issue is further addressed in [8].

Our incremental model can be extended to handle incomplete knowledge of future constraints, such as with online and stochastic problems. It is worth investigating a model that relaxes the incremental constraint but charges some price for every violation, as seen in on-line bipartite matching [13]. Alternatively, one could relax the packing constraint but charge some price for each broken constraint. Lastly, any given optimization problem has many potential incremental variants, only a few of which were discussed in this paper.

References

1. Plaxton, C.G.: Approximation algorithms for hierarchical location problems. In: Proceedings of the 35th Annual ACM Symposium on Theory of Computing, ACM Press (2003) 40–49
2. Mettu, R.R., Plaxton, C.G.: The online median problem. In: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, Washington, DC, USA, IEEE Computer Society (2000) 339

3. Hartline, J., Sharp, A.: Hierarchical flow. In: Proceedings of the International Network Optimization Conference. (2005) 681–687
4. Lin, G., Nagarajan, C., Rajaraman, R., Williamson, D.P.: A general approach for incremental approximation and hierarchical clustering. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms. (2006)
5. Fang, Y., Lou, W.: A multipath routing approach for secured data delivery. In: Proceedings of IEEE Micron 2001, IEEE Computer Society (2001) 1467–1473
6. Gonzalez, T.: Clustering to minimize the maximum intercluster distance. Theoretical Computer Science **38** (1985) 293–306
7. Dasgupta, S.: Performance guarantees for hierarchical clustering. In: Proceedings of the 15th Annual Conference on Computational Learning Theory, Springer-Verlag (2002) 351–363
8. Hartline, J., Sharp, A.: An incremental model for combinatorial minimization. Submitted for publication. Available at `www.cs.cornell.edu/~asharp`. (2006)
9. Borodin, A., El-Yaniv, R.: Online computation and competitive analysis. Cambridge University Press, New York, NY, USA (1998)
10. Fiat, A., Woeginger, G.J., eds.: Online Algorithms, The State of the Art. In Fiat, A., Woeginger, G.J., eds.: Online Algorithms. Volume 1442 of Lecture Notes in Computer Science., Springer (1998)
11. Coffman, E.G., Garey, M.R., Johnson, D.S.: Dynamic bin packing. SIAM Journal on Computing **12** (1983) 227–258
12. Gyárfás, A., Lehel, J.: Online and first-fit colorings of graphs. J. Graph Th. **12** (1988) 217–227
13. Karp, R.M., Vazirani, U.V., Vazirani, V.V.: An optimal algorithm for on-line bipartite matching. In: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, ACM Press (1990) 352–358
14. Gupta, A., Pál, M., Ravi, R., Sinha, A.: Boosted sampling: approximation algorithms for stochastic optimization. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing, ACM Press (2004) 417–426
15. Immorlica, N., Karger, D., Minkoff, M., Mirrokni, V.S.: On the costs and benefits of procrastination: approximation algorithms for stochastic combinatorial optimization problems. In: Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics (2004) 691–700
16. Dean, B.C., Goemans, M.X., Vondrak, J.: Adaptivity and approximation for stochastic packing problems. In: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2005) 395–404
17. Plotkin, S.A., Shmoys, D.B., Tardos, É.: Fast approximation algorithms for fractional packing and covering problems. In: Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press (1991) 495–504
18. Garg, N., Koenemann, J.: Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In: Proceedings of the 39th Annual Symposium on Foundations of Computer Science, IEEE Computer Society (1998) 300
19. Ibarra, O.H., Kim, C.E.: Fast approximation algorithms for the knapsack and sum of subset problems. J. ACM **22** (1975) 463–468
20. Lawler, E.L.: Fast approximation algorithms for knapsack problems. Mathematics of Operations Research **4** (1979) 339–356
21. Kuhn, H.: The hungarian method for the assignment problem. Naval Research Logistics Quarterly **2** (1955) 83–97
22. Karp, R.M.: Reducibility among combinatorial problems. In Miller, R.E., Thatcher, J.W., eds.: Complexity of Computer Computations. Plenum Press (1972) 85–103

Workload Balancing in Multi-stage Production Processes

Siamak Tazari*, Matthias Müller-Hannemann, and Karsten Weihe

Darmstadt University of Technology, Dept. of Computer Science,
Hochschulstraße 10, 64289 Darmstadt, Germany
{tazari, muellerh, weihe}@algo.informatik.tu-darmstadt.de

Abstract. We consider a variant on the general workload balancing problem, which arises naturally in automated manufacturing and throughput optimization of assembly-lines. The problem is to distribute the tasks over compatible machines and phases of the process simultaneously. The total duration of all phases is to be minimized. We have proved that this variant is NP-hard (even for uniform task lengths), and we propose a novel algorithmic approach. Our approach includes an exact solver for the case of uniform task lengths, which is based on network-flow techniques and runs in polynomial time for a fixed number of phases (the number of phases is indeed very small in practice). To solve the general case with arbitrary real task lengths, we combine our solver for uniform task lengths with a shortest-path based multi-exchange local search.

We present results of an extensive computational study on real-world examples from printed circuit board manufacturing. This study demonstrates that our approach is very promising. The solution quality obtained by our approach is compared with lower bounds from an integer linear programming model. It turns out that our approach is faster than CPLEX by orders of magnitude, and the optimality gap is quite small.

Keywords: workload balancing, printed circuit board assembly, network flows, multi-exchange local search, integer linear programming.

1 Introduction

In this paper, we study the following general workload balancing problem.

Problem 1.1 (Main Problem). We are given tasks $T = \{t_1, \ldots, t_n\}$, machines $P = \{p_1, \ldots, p_m\}$, phases $Q = \{q_1, \ldots, q_s\}$, a reset time R, and a set $A \subseteq T \times Q \times P$ of feasible assignments of tasks to machines and phases. Throughout the paper, a pair $(q, p) \in Q \times P$ will be called a *bucket*.

A *schedule* is an assignment $f : T \mapsto Q \times P$ of every task to a bucket. A schedule is *feasible* if $(t, f(t)) \in A$ for all $t \in T$.

* The author was partially supported by the Deutsche Forschungsgemeinschaft (DFG), Focus Program 1126 "Algorithmic Aspects of Large and Complex Networks", grant Mu1482/2-2.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 49–60, 2006.

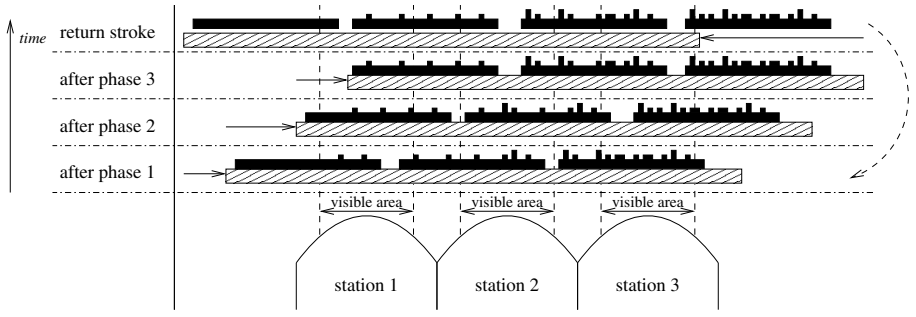


Fig. 1. Four snapshots of a board cycle with $s = 3$: the successive phases and the return stroke of the belt are shown. The arrows on each line indicate the moves of the belt from the previous phase to the current one, respectively.

For $t \in T$, let $\ell(t)$ denote the *length* of task t and $r_f(q,p)$ the number of times a reset is needed in bucket (q,p) given the schedule f. The *workload* of a machine p in a phase q is defined as $wl_f(q,p) = \sum_{t \in T, f(t)=(q,p)} \ell(t) + R \cdot r_f(q,p)$ for a given schedule f. The duration of a phase is the maximum workload of a machine in that phase: $d_f(q) = \max_{j=1,\ldots,m}\{wl_f(q,p_j)\}$. The *makespan* of the schedule is the sum of the durations of the phases: $c(f) = \sum_{i=1}^{s} d_f(q_i)$. The goal is to find a schedule that minimizes the makespan.

Exemplary illustration. For example, this problem arises in throughput optimization of assembly lines. In fact, our original motivation for this work is a particular application of assembly-line balancing: printed circuit board (PCB) manufacturing on certain modular machines such as Philips/Assembléon's AX[1]. See Figure 1. The circuit boards are the workpieces, the stations of the assembly line are the machines. The workpieces are moved forward in steps. In every phase between two moving steps, the machines perform tasks on the workpieces. The moving steps are periodic: after s steps, each workpiece has exactly assumed the position of its predecessor in the line. The actions of a machine are identical on all workpieces, so this is a periodic scheme with period s, too.

Every machine has a robot arm, which performs the tasks. In a task, a component is picked from a particular position (*feeder*) inside this machine and mounted at a prescribed position on the board. A machine comes with feeders for different component types. The feasibility relation A is induced by the following restrictions. A task may be performed by a machine if the machine has a feeder for the required component type. On the other hand, the task may be processed only in those phases in which the mounting position is in the visibility range of the machine.

The robot arm needs a *toolbit* for picking and holding a component. Different types of components require different toolbits. Each machine has a *toolbit repository*, and the reset time is induced by the necessary exchange of the toolbit at

[1] AX is a fast component mounter for printed circuit boards, Philips/Assembléon B.V., Eindhoven, The Netherlands.

that repository. A phase is finished once all machines have accomplished all of their tasks that are assigned to this phase. The time to produce one board is the sum of the durations of the s steps (up to a constant value). This time is to be minimized.

Related Work. Workload balancing (makespan minimization) of parallel machines for a single production phase is a prototypal scheduling problem which has been widely considered [1]. A number of overview papers on printed circuit board optimization and assembly lines has appeared quite recently. Scholl [2] surveys general problems and approaches to the balancing and sequencing of assembly lines. An overview on printed circuit board assembly problems has been given by Crama et al. [3].

Ayob et. al. [4] compare different models and assembly machine technologies for surface mount placement machines. Multi-station parallel placement machines like the Fuji QP-122 or Assembléon's AX found only little attention. For this type of machine, evolutionary (genetic) algorithms have been proposed in [5]. These authors consider a greatly simplified model in comparison to the one studied in this paper. Their computational results lack a comparison with optimal solutions or lower bounds. Müller-Hannemann and Weihe [6] recently studied a different variant of the workload balancing problem, where for each task exactly one machine is compatible (instead of a subset of the machines as in Prob. 1.1). However, in that variant also the movement scheme of the assembly line was part of the optimization. It is an empirical observation that task lengths differ not by too much in typical applications. Therefore, it is interesting to study the case of uniform task lengths. For example, this has been done by Grunow et al. [7]. Very large-scale neighborhood search has been applied quite successfully to several hard combinatorial optimization problems [8], in particular to the classical machine scheduling problems [9].

Our contribution and overview

Theorem 1.2. *Problem 1.1 is NP-hard in the strong sense, even if all tasks are of unit length and the reset time is zero.*

In Section 2, we will present a novel algorithmic approach for this type of problem. In the first stage of our algorithm, we will compute an optimal solution to the special case of unit task-lengths and zero reset time. This solution will be used in the second stage as a start solution to compute a good solution to the original problem. As noted above, this special case is of interest in its own right. Our research on this special case led to the following nice byproduct.

Theorem 1.3. *For a fixed number of phases, Prob. 1.1 can be solved in polynomial time if all tasks are of unit length and there is no reset time.*

The first stage of the algorithm is based on a reduction to a certain max-flow problem. Basically, the second stage may be characterized as sort of a shortest-path based multi-exchange local search where we generalize and extend the work of Frangioni et al. [9].

It might be worth mentioning that our handling of the reset times is quite generic and may be easily adapted to other types of complications. This observation demonstrates that our algorithm is quite robust in the sense that various types of complications may be naturally incorporated.

A major part of this paper is an extensive computational study on all real-world examples that are available to us (Sect. 3). We compare our solution to an integer linear programming (ILP) based approach using the ILP solver of CPLEX. Typically, our algorithm comes close to the lower bound computed by CPLEX or even hits the optimal value although it is faster than CPLEX by orders of magnitude. The comparatively small run time of our algorithm is particularly interesting for the complementary problem, to determine a favorable setup of the assembly line. In fact, here our problem naturally occurs as a subproblem, in which the quality of a trial setup is evaluated by the quality of the induced optimal task distribution. In this context, the algorithm is to be applied repeatedly a large number of times, so a small run time is mandatory.

2 Our Approach

In Section 2.1, we will first give an overview of our approach. In particular, we will formally state the subproblems that constitute the ingredients of our approach. In Sections 2.2-2.4, we will consider all of these ingredients individually and show how they build up on each other to produce a solution to our main problem.

2.1 The Big Picture

Our algorithm consists of two stages: in the first stage, we will focus on the special case of unit task-lengths and zero reset time:

Problem 2.1 (Uniform Case). Solve Prob. 1.1 with the assumption that $\ell(t) = 1$ for all $t \in T$ and $R = 0$.

As a first step, we will even consider a different objective function:

Problem 2.2 (MinMax Variant). Solve Prob. 2.1 replacing the objective function with $c_{\max}(f) = \max_{i=1,\ldots,s}\{d_f(q_i)\}$.

In Sect. 2.3, we will show that this variant can be solved efficiently. Our approach is based on a certain network-flow model, which is introduced in Section 2.2. Note that a feasible solution to the minMax variant is also a feasible solution to the uniform case itself. In fact, we will use the solution from Sect. 2.3 as a start solution for the algorithm in Sect. 2.4, which solves the uniform case optimally.

In the last stage of our algorithm (Sect. 2.5), we will eventually address Prob. 1.1 in full generality. Basically, this is a local search, for which we take the solution from Sect. 2.4 as the start solution. Note that the check whether an instance is feasible is trivial: there is a feasible schedule if, and only if, every task may be assigned to *some* bucket. We assume that this is always given.

Throughout our work, we make use of a function to control the workload in every phase by an upper bound. We call such a function $\kappa : Q \mapsto \mathbb{Z}$ a *capacity function*, and the upper bound it imposes on a phase, the *capacity* of that phase. A capacity function is called *feasible* if, and only if, there exists a schedule f such that $d_f(q) \leq \kappa(q)$ for all $q \in Q$. In this case, we also say that f is feasible for κ. The following problem plays a central role in our approach:

Problem 2.3 (Capacity Decision Variant). Given the setting in Prob. 2.1 and a capacity function κ, decide whether κ is feasible.

Let $c(\kappa) = \sum_{i=1}^{s} \kappa(q_i)$ and $c_{\max}(\kappa) = \max_{i=1,\ldots,s} \kappa(q_i)$. Note that if f is a feasible schedule for κ we have $c(f) \leq c(\kappa)$ and $c_{\max}(f) \leq c_{\max}(\kappa)$.

2.2 The Network Flow Model and the Capacity Decision Variant

Given an instance of the capacity decision variant, we construct a directed network flow graph $G_\kappa = (V, E)$ as follows. We introduce one vertex for every task, one vertex for every bucket, and in addition one source vertex and one sink vertex. Next we add one edge with capacity 1 from the source to every task vertex, one edge with capacity 1 from every task vertex to every bucket vertex in which this task may be performed, and one edge with capacity $\kappa(q)$ from every bucket vertex with phase q to the sink. We denote the value of a flow $\overline{f}$ in G with $v(\overline{f})$. It is easy to see that the following lemma holds.

Lemma 2.4. *There exists a feasible flow $\overline{f}$ in G_κ with $v(\overline{f}) = n$ if and only if κ is feasible. If such a flow exists, a feasible schedule f for the capacity decision variant can be derived as follows: for $t \in T$, set $f(t) = (q, p)$ for the unique pair (q, p) such that $\overline{f}((v, w)) = 1$, where node v represents the task t and node w represents the bucket (q, p).*

This results in an algorithm whose runtime, using the Ford-Fulkerson method [10], is $O(n \cdot |E|) = O(n^2ms)$. This is because the value of the flow is at most n, so it can be constructed by at most n augmentations.

2.3 The MinMax Variant

We can now easily derive an algorithm for the minMax variant. As said in the big picture (Sect. 2.1), this is the first step of our overall algorithm and its result will be used as a starting point for solving the uniform case optimally (Sect. 2.4).

The minMax problem itself is equivalent to finding the minimal value k such that the uniform capacity function $\kappa(\cdot) \equiv k$ is feasible. For that, we may simply test all values $k = 0, \ldots, n$ and take the smallest one for which κ becomes feasible. Alternatively, we may use binary search on $0, \ldots, n$ and obtain a complexity of $O(n^2ms \log n)$.

In view of Sect. 2.4, it is heuristically reasonable to construct a tighter capacity function κ, which is not necessarily uniform, in the hope to also minimize the sum. We can continue decreasing the capacities of individual phases as long

as it is possible. This idea is carried out in Algorithm 2.5. For this purpose, we make use of a helper-function: Given G_κ, one can easily write a function `decreasePhaseCapacity(`q`)` that tries to decrease the capacities of all the sink-edges belonging to phase q by one unit while preserving the value of the flow; it should change the function κ accordingly and return `true` if successful, otherwise do nothing and return `false`. A straightforward implementation runs in $O(m \cdot |E|) = O(nm^2s)$-time. The algorithm tries to decrease the capacities of the phases at most n times, i.e. `decreasePhaseCapacity` is called at most ns times. So the overall runtime is $O(n^2m^2s^2)$. This is very fast if m and s are small; and this is the case in some practical situations like in our application.

2.4 Solving the Uniform Case Optimally

By Theorem 1.2, we know that Prob. 2.1 is NP-hard. We found out that the hardness of the problem lies in the parameter s, the number of phases. The algorithm we present now runs in $O(u^{s-1}nm^2s)$–time (where $u \leq n$ is a parameter we will introduce shortly). If we take s to be a fixed constant, this is polynomial. Indeed, in our application we have $s \leq 7$ and $m \leq 20$ and hence our algorithm is fast enough for our purpose. Also, it is possible to stop it at any time and take the best solution it has found so far.

Let us denote $\sum_{i=1}^{j} \kappa(q_i)$ by $c_j(\kappa)$. Suppose a fixed j and a (not necessarily feasible) κ are given. We would like to find a κ' with $\kappa'(q_i) = \kappa(q_i)$ for $i > j$, such that $c_j(\kappa')$ has the minimum possible value and κ' is feasible. Let us denote this value by $\min_j(\kappa)$. Our method also takes a third parameter, u, that indicates the maximum value we expect for $\min_j(\kappa)$. If we determine that $\min_j(\kappa) > u$, our algorithm will return ∞. To obtain an initial value for u, we can take any feasible capacity function κ' and set $u := c_j(\kappa') - 1$. Specifically, we use the solution delivered by our minMax algorithm. The smaller the value of u, the faster will be our method; hence, it is fortunate that Alg. 2.5, when used as a heuristic for Prob. 2.1, supplies near-optimum solutions.

Our algorithm is a branch-and-bound method that given j, κ and u, tries to modify κ, so that it becomes (or remains) feasible and $c_j(\kappa)$ becomes $\leq u$. If successful, it decreases u step-by-step and continues the search until this is not further possible. In every function call, several values for the capacity of step q_j are probed and for each of them, the method is called recursively for $j - 1$. In order to cut down the search, we utilize the following observation.

Algorithm 2.5. *minMax algorithm; solves Prob. 2.2*

```
create a set F = ∅ of phases with fixed capacity
create a capacity function κ and set it constant equal to n
solve Prob. 2.3 for κ to find an initial feasible solution, G_κ and the flow f̄
repeat
  for all q ∈ Q \ F do
    if not decreasePhaseCapacity(q) then
      F = F ∪ {q}
until F = Q
use Lemma 2.4 to derive a feasible assignment f from f̄ and return f
```

Observation 2.6. *Let* $\kappa(q_j) = t$ *and* $\min_{j-1}(\kappa) = r$. *If we decrease* $\kappa(q_j)$ *to any value* $\leq t$, *we will have* $\min_{j-1}(\kappa) \geq r$.

This is because any κ with $\kappa(q_j) \leq t$ remains feasible if we set $\kappa(q_j) = t$. Our algorithm works as follows: we set $\kappa(q_j) = u$ and find $r := \min_{j-1}(\kappa)$. By the observation above, we know that in any solution with $c_j(\kappa) \leq u$, $\kappa(q_j)$ can be at most $u - r$. So, we assign this value to $\kappa(q_j)$, calculate $r = \min_{j-1}(\kappa)$ with this new fixation and repeat this procedure until either a solution with $c_j(\kappa) \leq u$ is found, u becomes less than r or no feasible κ with the given fixations exists anymore. If a solution is found, we decrease the value of u and continue the search. Note that there is no need to restart the search in this case.

We call our method `minimizeSum(`j`, `κ`, `u`)`. We assume that G_κ and a maximum flow $\overline{f}$ in G are also given. Our method minimizes $c_j(\kappa)$ and returns it, adjusting κ, G and $\overline{f}$ accordingly. But only if such a feasible solution exists and is at most u. Otherwise the method returns ∞. We need two additional simple methods: `increasePhaseCapacity(`q`)` that increases the capacity of phase q by one and `setPhaseCapacity(`q`, `t`)` that sets the capacity of phase q to the value t; in contrast to `decreasePhaseCapacity`, neither of these methods need to preserve the value of the flow but only have to re-maximize it.

Algorithm 2.7. `minimizeSum(`j`, `κ`, `u`)` *; solves Prob. 2.1*

(1) **if** $j = 2$ **then** return `minmizeSum2(`κ`, `u`)` {described below}
(2) create new capacity function $\kappa^\star$ and set it equal to `null`
(3) `setPhaseCapacity(`q_j`, `u`)`
(4) $r =$ `minimizeSum(`$j-1$`, `κ`, `u`)`
(5) **while** $r \leq u$ **do**
  (6) **if** $\kappa(q_j) + r \leq u$ **then** {a solution is found}
    (7) $\kappa^\star = \kappa$
    (8) $u = \kappa(q_j) + r - 1$ {reduce u}
  (9) **else**
    (10) `setPhaseCapacity(`q_j`, `$u - r$`)`
    (11) $r =$ `minimizeSum(`$j-1$`, `κ`, `u`)`
(12) **if** $\kappa^\star = null$ **then** return ∞ {no solution found}
(13) $\kappa = \kappa^\star$; adjust G_κ and $\overline{f}$ accordingly
(14) return $u + 1$

Lemma 2.8. *Algorithm 2.7 is correct and its complexity is* $O(u^{j-1}nm^2s)$.

In the beginning, $j = s$ and hence, the total runtime of our method is $O(u^{s-1}nm^2s)$. This proves Theorem 1.3.

The case of 2 phases. The recursion above reaches its base when j becomes 2. In this case, the calls to `minimizeSum(1, `κ`, `u`)` can be replaced in the following way: when it is called the first time in line (4), we can, instead, set the capacity of q_1 equal to 0 and run

while $v(\overline{f}) < n$ and $\kappa(q_1) \leq u$ **do** `increasePhaseCapacity(`q_1`)`
$r = \kappa(q_1)$.

When called in the loop in line (11), we can do the same *but* we do not need to reset its capacity back to 0; we can just continue increasing it. This is justified by Observation 2.6 above. We call the resulting method `minimizeSum2(`κ`, `u`)`. The proof of the lemma below can be found in the journal version.

Lemma 2.9. *`minimizeSum2` is correct and its runtime is* $O(unm^2 s)$.

Lower bounds. Having good lower bounds can be useful to cut down the search in the branch-and-bound algorithm described above. An easy global lower bound for Prob. 2.1, also mentioned in [7], can be given by $c(f) \geq lb_1 := \lceil \frac{n}{m} \rceil$. Using our minMax algorithm, we can find additional lower bounds: suppose we have a capacity function κ and would like to minimize $c_j(\kappa)$ while keeping the capacities of $q_{j+1}, \ldots, q_s$ fixed. We can find a lower bound lb_2^j for $c_j(\kappa)$ by merging the phases $q_1, \ldots, q_j$ into one phase q'. We set $Q' = \{q', q_{j+1}, \ldots, q_s\}$ and $A' = \{(t, q_i, p) \in A : i > j\} \cup \{(t, q', p) : (\exists 1 \leq i \leq j)\,(t, q_i, p) \in A\}$. Let κ' be the corresponding capacity function. We can minimize $\kappa'(q')$ – while keeping $\kappa'(q_{j+1}) = \kappa(q_{j+1}), \ldots, \kappa'(q_s) = \kappa(q_s)$ fixed – using a simplified version of Alg. 2.5. Let $f^\star$ be the assignment resulting from this minimization. For arbitrary values of $\kappa(q_1), \ldots, \kappa(q_j)$, let f be some feasible assignment for κ. We can construct an assignment f' for the merged problem by setting

$$(\forall t \in T) f'(t) = \begin{cases} f(t) & \text{if } f(t) = (q_i, p) \text{ and } i > j \\ (q', p) & \text{if } f(t) = (q_i, p) \text{ and } i \leq j \end{cases} . \tag{1}$$

Then we have

$$\begin{aligned} lb_2^j = d_{f^\star}(q') \leq d_{f'}(q') = \max_{i=1,\ldots,m} \{wl_{f'}(q', p_i)\} \\ = \max_{i=1,\ldots,m} \{\sum_{k=1}^{j} wl_f(q_k, p_i)\} \leq \max_{i=1,\ldots,m} \{\sum_{k=1}^{j} d_f(q_k)\} = c_j(f) \leq c_j(\kappa) \ . \end{aligned} \tag{2}$$

In addition to the local lower bounds for any j, that can be used in Alg. 2.7, we also get a global lower bound $c(f) = c_s(f) \geq lb_2 := lb_2^s = c(f^\star)$.

2.5 Shortest-Path Based Local Search and Reset Times

In order to solve our main problem, Prob. 1.1, we take the optimal solution of the uniform case, substitute the actual task-lengths and perform local search on it. We used the idea of multi-exchange neighborhoods presented by Frangioni et al. in [9] and incorporated the existence of multiple phases and reset times into it. Specifically, we do the following: Let a feasible assignment f be given. We call a bucket (q, p) *loaded* if the workload of bucket (q, p) is equal to the duration of phase q. Let $r_f(t, q, p) \geq 0$ denote the reset time needed, if task t is added to bucket (q, p). We create an improvement graph by introducing one vertex for every task and one vertex for each bucket. We connect a task t_1 to a bucket (q, p) with a directed edge if t_1 can be added to bucket (q, p) without making it loaded, i.e. if $wl_f(q, p) + \ell(t_1) + r_f(t_1, q, p) < d_f(q)$. We also connect a task t_1 to a

task t_2 with a directed edge if it is possible to remove t_2 from its bucket and add t_1 instead, so that $wl_f(q,p) + \ell(t_1) - \ell(t_2) + r_f(t_1,q,p) < d_f(q)$. By the *bucket of a vertex* v, we mean the bucket it represents or the bucket where the task it represents is assigned to. We call a path or a cycle in this graph *disjoint* if the buckets of its vertices are all different. Also, in case of a path, it must end with a bucket vertex. Now every disjoint path or cycle in this graph that includes at least one vertex whose bucket is loaded can be used to reduce the total number of loaded buckets and thus, most probably, also reduce the makespan: simply by performing the changes that every edge on that path or cycle represents.

In our specific application, reset times are caused by toolbit exchanges. We have that $r_f(t,q,p) > 0$ if a new toolbit exchange is needed to perform task t in bucket (q,p). When updating along a disjoint path or cycle that contains toolbit exchanges, we need to add the occurring toolbit exchanges to the corresponding buckets. Adding this feature improved our results considerably in some cases, see Sect. 3.

In order to find disjoint paths and cycles we chose to implement the 1–SPT heuristic described in [9], adapted to our case, using a priority queue. In this heuristic, a cost function is introduced and a shortest path algorithm is used. For details we refer to [9]. We chose not to build the graph explicitly since by sorting the tasks in each bucket in descending order according to their lengths, it is possible to decide about the existence of edges in constant time. This eliminates the need for an expensive update. We start the search once from every vertex and then repeat this procedure until no further improvement can be found.

3 Computational Results

Test instances. In this study, we used 20 widely different test jobs. Each job represents a PC board type with a standard machine setup for Assembléon's AX-5 from which we obtained an instance of our problem. All test jobs stem from customers of Assembléon and have been kindly provided by Assembléon to us. Table 1 shows some of the characteristics of these jobs. The number of tasks varies between 190 and 940 tasks, the number of placement phases s between 3 and 7. Since the AX-5 has 20 parallel robots (stations), the number of buckets varies between 60 and 140. The task lengths are quite similar, they differ by at most a factor of 4 from each other.

Testing environment. All computations are executed on a standard Intel P4 processor with 3.2 GHz and 4 GB main memory running under Suse Linux 9.2. Our algorithms are implemented in Java, we used JDK 5.0.

The uniform case. Table 1 displays the results for our instances under unit-length assumption. Columns labeled lb_1 and lb_2 give the values obtained for the two lower bounds introduced in Subsection 2.4, respectively. It turns out that lb_1 is a rather weak bound, but lower bound lb_2 is much closer to the optimum.

Algorithm 2.5 (minMax) performs surprisingly well as a heuristic for Prob. 2.1. Its running time is negligible (and therefore not reported). It actually hits the

Table 1. Results for the uniform case. The first four columns contain job characteristics. Columns labeled with lb_1, lb_2 give lower bound values, "minMax" reports the outcome of Algorithm 2.5, "opt" give the value of optimal solution (computed by Algorithm 2.7), "opt(100)" denotes the solution value, if Algorithm 2.7 is stopped after 100 recursive function calls. The two last columns report the CPU time in seconds for the "opt(100)" and "opt" versions, respectively.

instance	# tasks	# phases	# buckets	lb_1	lb_2	minMax	opt(100)	opt	time(100)	time
j1p1	304	4	80	16	18	18	18	18	0.00	0.00
j1p2	190	3	60	10	12	16	16	16	0.08	0.08
j1p3	138	4	80	7	9	11	11	11	0.11	0.10
j2p1	940	5	100	47	284	284	284	284	0.00	0.00
j2p2	804	5	100	41	68	79	78	78	1.20	7.77
j2p3	792	5	100	40	74	83	83	81	1.12	5.81
j2p4	786	5	100	40	73	82	82	81	1.11	5.14
j2p5	750	5	100	38	98	117	117	109	1.20	89.19
j2p6	634	5	100	32	120	120	120	120	0.00	0.00
j2p7	532	5	100	27	46	59	59	59	0.55	7.82
j3p1	912	5	100	46	148	148	148	148	0.00	0.00
j3p2	660	5	100	33	120	120	120	120	0.00	0.00
j3p3	376	6	120	19	44	64	64	64	0.33	2.52
j4p1	312	7	140	16	18	18	18	18	0.00	0.00
j4p2	300	7	140	15	17	17	17	17	0.00	0.00
j4p3	288	7	140	15	16	16	16	16	0.00	0.00
j4p4	288	7	140	15	18	20	20	20	0.46	0.80
j4p5	212	5	100	11	24	24	24	24	0.00	0.00
j5p1	362	6	120	19	26	42	42	42	0.58	0.98
j5p2	345	6	120	18	23	46	46	46	0.63	1.10

optimal value quite often as can be seen by comparing the columns labeled "minMax" and "opt". Intuitively, the reason might rely on the following observation: if the capacity of a phase can not be decreased in the algorithm, it will also be impossible to decrease it later on. That is, the only way to arrive at a better sum would be to *increase* the capacities of some phases and see if one can in return, decrease the capacities of other phases more than that. And this seems not to be much too likely. Except for one extremely hard case (j2p5), the running time of Algorithm 2.7 is below 8 seconds. In the exceptional case, the absolute difference of the result provided by the minMax-heuristic is quite large. This results in a runtime of about 89 seconds. Usually, Algorithm 2.7 requires most of its time to prove optimality. Since this algorithm is used as a starting solution for the local search, we also experimented with a "fast version" called opt(100) which terminates after 100 recursive function calls (and so terminates after about one second). In all but three cases this heuristic already found the optimum solution.

Impact of local search and toolbit exchanges. Table 2 shows our computational results for the scenario with real times and toolbit exchanges. The second column shows the makespan (cycle time) in seconds after running the

Table 2. Results for the real-time scenario including toolbit exchanges. Comparison with CPLEX 9.1.

instance	make-span	LS impact	#TE	cplex 60s	cplex 300s	cplex lb	our gap	gap cplex60	gap cplex300
j1p1	9.00	11.24%	0	-	8.62	7.87	14.43%	-	9.56%
j1p2	8.12	3.24%	0	8.64	8.47	8.12	0.00%	6.29%	4.22%
j1p3	5.64	4.89%	0	5.6	5.51	5.27	7.00%	6.18%	4.47%
j2p1	85.83	44.38%	8	82.55	82.46	80.84	6.18%	2.12%	2.00%
j2p2	38.33	5.17%	0	38.27	38.26	38.26	0.18%	0.03%	0.01%
j2p3	40.45	3.96%	0	40.19	40.19	40.18	0.67%	0.01%	0.01%
j2p4	40.37	3.96%	0	39.91	39.91	39.86	1.28%	0.12%	0.12%
j2p5	49.04	13.12%	6	44.36	44.36	44.36	10.56%	0.00%	0.00%
j2p6	39.74	36.36%	6	36.98	36.98	36.98	7.49%	0.00%	0.00%
j2p7	30.42	4.38%	0	30.42	30.42	30.42	0.00%	0.00%	0.00%
j3p1	82.98	0.00%	0	82.98	82.98	82.98	0.00%	0.00%	0.00%
j3p2	63.23	0.69%	0	63.23	63.23	63.23	0.00%	0.00%	0.00%
j3p3	47.19	2.42%	18	42.14	42.14	42.14	12.00%	0.00%	0.00%
j4p1	8.98	3.46%	0	-	9.57	8.13	10.53%	-	17.70%
j4p2	8.29	4.49%	0	-	8.38	7.72	7.37%	-	8.54%
j4p3	8.09	2.35%	0	-	8.58	7.48	8.21%	-	14.67%
j4p4	9.47	4.83%	0	11.26	10.83	8.99	5.24%	25.24%	20.40%
j4p5	11.42	3.73%	0	11.91	11.9	11.42	0.00%	4.28%	4.20%
j5p1	25.47	11.14%	10	23.33	23.33	23.29	9.35%	0.18%	0.18%
j5p2	24.81	8.94%	17	24.12	24.12	24.12	2.86%	0.00%	0.00%

local search procedure. The local search was started with the result of an optimal unit-length solution. The third column gives the percentage reduction obtained in comparison with this starting solution. We observe that local search helps a lot to reduce the cycle time if the addition of toolbit exchanges allows a better workload balancing.

Lower bounds and comparison with CPLEX. We compared the quality of the solutions obtained by our approach with a lower bound on the solution value (in several cases with the exact optimum). To this end, our problem including toolbit exchanges has been modeled as an integer linear program (ILP). Formulating the constraints for toolbit exchanges is a bit tricky. Due to the space restrictions, we deferred the complete ILP problem formulation to the journal version.

To solve the ILP problems, we used ILOG CPLEX 9.1 with standard settings. In Table 2, we display the solutions values obtained by CPLEX after 60 and 300 seconds CPU time, the lower bound value obtained after 300s, as well as the optimality gap after 60 and 300 seconds, respectively. The given time limits to CPLEX are relatively small, but in comparison to the running time of our method considerably longer. Recall that short time limits are justified by the fact that instances of this type have to be solved several hundreds of times for different setups. Thus we can afford in practice only a few seconds per instance.

The computational results are quite interesting. In 7 out of 20 cases, CPLEX managed to find even the optimal solution within 60 seconds. In contrast, CPLEX failed to find even some feasible solution in 4 cases in the same time limit. Within $300s$, there was at least one feasible solution in all but one case. For one hard case (instance j1p1) we had to use a different CPLEX option (MIP Integer Feasibility) to find a feasible solution within $300s$. The final gap of our solution to the lower bound provided by CPLEX is relatively small, in 5 cases we provably found the optimum solution, and in 4 further cases (j4p1 - j4p4) our gap is considerably smaller than the CPLEX gap after $300s$. On the other hand, there are several instances which are seemingly easier to handle for CPLEX than for our approach. Thus there is no clear winner, but our approach is much faster and always guarantees at least a feasible solution after a few seconds.

4 Conclusion

We have presented a novel algorithm for a practically relevant workload-balancing problem. Our computational results demonstrate that this approach is a good choice in practice. The run time is well under control, so the algorithm may be used as a subroutine in solvers for optimal machine setups. This will be the next step in our future work.

References

1. Brucker, P.: Scheduling Algorithms. 4th edn. Springer-Verlag (2004)
2. Scholl, A.: Balancing and Sequencing of Assembly Lines. 2nd edn. Physica-Verlag, Heidelberg (1999)
3. Crama, Y., Klundert, J., Spieksma, F.C.R.: Production planning problems in printed circuit board assembly. Discrete Applied Mathematics **123** (2002) 339–361
4. Ayob, M., Cowling, P., Kendall, G.: Optimisation of surface mount placement machines. In: Proceedings of IEEE International Conference on Industrial Technology. (2002) 486–491
5. Wang, W., Nelson, P.C., Tirpak, T.M.: Optimization of high-speed multistation SMT placement machines using evolutionary algorithms. IEEE Transactions on Electronic Packaging and Manufacturing **22** (1999) 137–146
6. Müller-Hannemann, M., Weihe, K.: Moving policies in cyclic assembly-line scheduling. Theoretical Computer Science **351** (2006) 425–436
7. Grunow, M., Günther, H.O., Schleusener, M.: Component allocation for printed circuit board assembly using modular placement machines. International Journal of Production Research **41** (2003) 1311–1331
8. Ahuja, R.K., Ergun, O., Orlin, J.B., Punnen, A.P.: A survey of very large-scale neighborhood search techniques. Discrete Applied Mathematics **123** (2002) 75–102
9. Frangioni, A., Necciari, E., Scutellà, M.G.: A multi-exchange neighborhood for minimum makespan parallel machine scheduling problems. J. Comb. Optim. **8** (2004) 195–220
10. Ford, L.R., Fulkerson, D.R.: Flows in Networks. Princeton University Press, Princeton, NJ (1962)

Fault Cryptanalysis and the Shrinking Generator*

Marcin Gomułkiewicz[1], Mirosław Kutyłowski[1], and Paweł Wlaź[2]

[1] Wrocław University of Technology
[2] Lublin University of Technology

Abstract. We present two efficient and simple fault attacks on the shrinking generator. In a first case if the attacker can stop control generator for some small number of steps and observe the output, then with high probability he can deduce the full control sequence, and so the other input bitstream. The second method assumes that the attacker can disturb the control sequence (in an unpredictable and random way) and observe many samples of such experiments. Then he can reconstruct a certain sequence that agrees with the input sequence of the generator on a large fraction of bits.

1 Introduction

This paper presents two new fault attacks (see [2]) against the shrinking generator by Coppersmith *et al.* [3], one of the major designs of efficient and secure pseudorandom generators. Our attacks are unique in many ways. To our best knowledge these are the first fault attacks on the shrinking generator; they also seem to be among the most efficient attacks. On top of that, it is generally agreed that the shrinking generator is composed out of LFSRs. However, we require no specific design of the generator (it can consist of any bit generators), apart from the possibilities regarding injecting faults of the assumed kind.

The paper is organized as follows: first we give a very brief overview of the shrinking generator. Then the Section 2 describes our first attack, based on synchronization fault assumption (the assumption is similar to those from [9]). The Section 3 presents another attack, based on weaker assumptions. Finally the Section 4 briefly summarizes our results and states open problems.

The Shrinking Generator in a Pill. The shrinking generator [3] is an attempt to create cryptographically strong pseudorandom bitstream generator out of relatively weak components. Many other solutions of this kind [7, 1, 4] were proven to be weak [13, 14]. The shrinking generator successfully faces the trial of time: the best known attacks against it are exponential in the LFSR's length [5, 8, 10, 11, 12], or based on the assumption that the feedback is known [6].

Amazingly, the construction of the shrinking generator is very simple. It consists of two LFSRs we shall call the base (or input) generator A and the control

* Partially supported by KBN project no. 0T00A 003 23.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 61–72, 2006.

generator C; their output is denoted as $a_1, a_2, a_3, \ldots$ and $c_1, c_2, c_3, \ldots$, respectively. The output $z_1, z_2, z_3, \ldots$ is composed of those and only those of a_i for which $c_i = 1$. Formally: $z_t = a_i$ for i so that:

$$t = \sum_{j=1}^{i} c_j \qquad \text{and} \qquad c_i = 1 \ .$$

2 An Attack with Synchronization Faults

Assumptions. The attacker has a device implementing the shrinking generator and can use it freely. Assume that the base and control generators of the shrinking generator output bits according to the uniform distribution over $\{0, 1\}$. Also assume, that an attacker can disturb clocking of the device, namely we assume that he can stop the control sequence for a couple of steps, and observe the generator's output.

2.1 Basic Attack

Assume we have number of output sequences related to holding the control generator for 0 (i.e. with no fault), $1, 2, \ldots, n-1$ steps. That is, we stop the control generator for a number of steps, the base generator moves on, so we have sequence of outputs $Z^k = z_1^k z_2^k \ldots z_N^k$ related somehow to the base bitstreams $a_{k+1}, a_{k+2}, a_{k+3}, \ldots$ and the control bitstream $c_1, c_2, c_3, \ldots$.

Let us take a closer look at the data – see Table 1. Our algorithm shall guess the length of blocks of zeroes that separate consecutive ones in the control bitstream. Note that if two ones are consecutive, the 2nd bit of the bitstream i is the same as the 1st bit of the bitstream $i+1$, for each i. Analogously, if the ones are separated by a single zero, then the 2nd bit of bitstream i is the same as the 1st bit of bitstream $i+2$, for each i. If so, it is easy to construct an algorithm that shall guess the number of zeroes separating the ones. It returns a set S_l of all non-contradictory solutions – number of zeroes in the control stream between the two ones corresponding to output positions l and $l+1$. However, we have to assume that the number of zeroes between two ones does not exceed a certain parameter *maxzeroes*. To obtain the whole control sequence we have to execute the algorithm for $l = 1, 2, \ldots, N-1$ and consider the set of all possible solutions $\mathbf{S} = S_1 \times S_2 \times \ldots \times S_{N-1}$ Then $\mathbf{s} = (s_1, s_2, \ldots, s_{N-1})$ corresponds to a possible control sequence:

$$\underbrace{00\ldots0}_{?}1\underbrace{00\ldots0}_{s_1}1\underbrace{00\ldots0}_{s_2}1\ldots1\underbrace{00\ldots0}_{s_{N-1}}1 \ .$$

Of course, it is impossible to recover the number of zeroes preceding the first one in the control sequence.

Probability of a False Solution

Lemma 1. *Let n be the number output sequences analyzed and the correct number of zeroes between consecutive ones considered is l. Then the output set S contains l with probability 1, and contains an m, $m \neq l$, with probability $2^{-(n-m-1)}$.*

Table 1. Idea of the basic attack

A	a_i	a_{i+1}	a_{i+2}	a_{i+3}	a_{i+4}	a_{i+5}	a_{i+6}
S	1	1	0	1	0	0	1
Z^0	a_i	$\dagger a_{i+1}$	a_{i+2}	$*a_{i+3}$	a_{i+4}	a_{i+5}	$\ddagger a_{i+6}$
Z^1	$\dagger a_{i+1}$	a_{i+2}	a_{i+3}	a_{i+4}	a_{i+5}	a_{i+6}	a_{i+7}
Z^2	a_{i+2}	$*a_{i+3}$	a_{i+4}	a_{i+5}	a_{i+6}	a_{i+7}	a_{i+8}
Z^3	a_{i+3}	a_{i+4}	a_{i+5}	$\ddagger a_{i+6}$	a_{i+7}	a_{i+8}	a_{i+9}
...				...			

Proof. We are checking the hypothesis that there are m consecutive zeroes separating the two ones. If $m = l$, then all checked equations are identities. Assume that $m \neq l$. Let k be the number of pair checked: $k = n - m - 1$. So the following equations are checked:

$$\begin{aligned} a_i &= a_{i+(l-m)} \\ a_{i+1} &= a_{i+(l-m)+1} \\ &\dots \\ a_{i+k-1} &= a_{i+(l-m)+k-1} \end{aligned}$$

Let us split those equations into separate "chains":

$$\begin{aligned} a_j &= a_{j+(l-m)} = a_{j+2(l-m)} = \dots \\ a_{j+1} &= a_{j+(l-m)+1} = a_{j+2(l-m)+1} = \dots \\ &\dots \\ a_{j+(l-m)-1} &= a_{j+2(l-m)-1} = a_{j+3(l-m)-1} = \dots \end{aligned}$$

Let us consider a single chain $a_i = a_{i+(l-m)} = a_{i+2(l-m)} = \dots = a_{i+p(l-m)}$. According to our assumption a_i are random independent bits, so $\Pr\{a_i = 1\} = \frac{1}{2}$. Hence, the distribution of the vector $(a_i, a_{i+(l-m)}, a_{i+2(l-m)}, \dots, a_{i+p(l-m)})$ is also uniform. Then the number of its' all possible states is 2^{p+1}, while only two of them yield a vector of all-equal elements. Then the probability of fulfilling the equations building the chain equals 2^{-p}.

Since all chains together contain k equations and the chains are disjoint, the events of fulfilling all chains' equations are independent, which concludes the proof. □

Lemma 1 shows that the probability of a false result grows rapidly with the assumed length of the gap between the ones. That is why we assume that the control sequence does not contain a block of more than *maxzeroes* zeroes. Of course, if this assumption is false, the algorithm will fail.

2.2 Full Attack

Now we assume that we still have all Z^k, but permuted according to an unknown, random permutation π. Such a situation can happen when we can stop the

clocking for a couple of steps (no more than $n-1$), but we cannot control the exact number of these steps. In such a scenario we would simply perform the experiments until we collect n different outputs, which would give us the complete set of outputs and no information about their ordering.

To perform an attack we first deduce the unknown permutation π, shuffle the outputs according to π^{-1}, and then perform the basic attack.

Assume for a moment that the control sequence C is of the form $00\ldots011\ldots$. So if the Z^k sequences are in the correct order, the following equations hold:

$$z_2^1 = z_1^2, \quad z_2^2 = z_1^3, \quad \ldots \quad z_2^{n-1} = z_1^{n-2}$$

(see Table 1); conversely, for Z^k and Z^l ($l \neq k+1$) we have $z_2^k = z_1^l$ with probability $\frac{1}{2}$. Therefore we shall think of $Z^{\pi(i)}$'s as of vertices in a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{0, 1, 2, \ldots, n-1\}$ and there is an edge from vertex l to vertex k if and only if $z_2^k = z_1^l$. Similarly, if

$$C = 00\ldots0\underbrace{1\ldots1}_{m}\ldots\ , \tag{1}$$

then

$$\mathcal{E} = \left\{ (p,q) \in \mathcal{V}\times\mathcal{V} \colon p \neq q \wedge \bigwedge_{j=1}^{m-1} \left(z_{j+1}^p = z_j^q\right) \right\}. \tag{2}$$

Let $\left\{\left(v_0^i, v_1^i, \ldots, v_{n-1}^i\right)\right\}_{i\in I}$ denote all Hamiltonian paths in $\mathcal{G}$. Then, for any given set of output sequences $Z^{\pi(j)}$ there exists $i \in I$ so that $v_j^i = \pi(j)$ holds for all $j \in \{0, 1, \ldots, n-1\}$. Of course, assumption (1) might be false, so it may be desirable to relax it: assume that among the first l control bits there are exactly k ones, and then there is continuous block of m ones. Then we could slightly modify the condition (2):

$$\mathcal{E} = \left\{ (p,q) \in \mathcal{V}\times\mathcal{V} \colon p \neq q \wedge \bigwedge_{j=1}^{m-1} \left(z_{j+1+k}^p = z_{j+k}^q\right) \right\}. \tag{3}$$

Of course, finding all Hamiltonian paths in a graph is a hard problem. However, in our restricted case it can be solved with a straightforward approach (since due to condition (3) the graph can be made almost arbitrarily sparse) and so we may use the following procedure:

- choose $m \in \{2, 3, 4, \ldots\}$;
- for $k = 0, 1, 2, \ldots$ construct the graph $\mathcal{G}$,
- find all possible Hamilton paths in $\mathcal{G}$; if none found, then k was incorrect; each path in $\mathcal{G}$ corresponds to some $\widehat{\pi}(\cdot)$;
- for each $\widehat{\pi}$ sort outputs $Z^{\pi(i)}$ according to $\widehat{\pi}^{-1}$; if $\widehat{\pi}$ was guessed right, then for all i holds $i = \widehat{\pi}^{-1}(\pi(i))$, and so the output sequences are in the correct order;
- for every sorted set of output sequences perform the basic attack.

2.3 Simulation Results

The algorithm described was implemented and run on a low-end home PC, with an AMD Athlon XP1800+ CPU and 512MB of RAM. Their results for 100 runs for each choice of parameters are summarized by Table 2.

Table 2. Statistics of the attack; the columns have the following meaning: n – the number of output sequences, N – the length of output sequences, mz – the *maxzeroes* parameter, "c. p." – the number of column pairs checked during construction of $\mathcal{G}$, "fails" – the number of fails (i.e. no solutions found), "avg. sol. count" – mean number of solutions (if found any), "avg. time" – mean calculations' time (in seconds), "avg. time 90%" – mean calculations' time after rejecting 10% of the worst cases

n	N	mz	c. p.	fails	avg. sol. count	avg. time	avg. time 90%
15	50	10	2	0	6086.89	4.5286	1.8920
15	50	10	4	11	4.56	0.0378	0.0047778
20	100	15	4	1	14.58	0.6402	0.071333
20	200	15	4	0	260.84	0.8431	0.10578
25	100	15	5	20	1.09	1.3391	0.048889
25	500	15	5	1	2.34	1.3790	0.10844
25	100	15	7	64	1.44	0.0178	0.0086667
25	500	15	7	12	1.69	0.0278	0.021667
30	100	15	6	44	1.00	5.4973	0.015778
30	500	15	6	1	1.00	1.0959	0.036111
35	100	20	6	39	1.00	0.7908	0.054111
35	500	20	6	1	1.00	1.0873	0.13156
35	100	20	8	86	1.00	0.0177	0.016667
35	500	20	8	44	1.04	0.0655	0.061444
35	100	25	6	40	2.02	0.5346	0.059444
35	500	25	6	1	1.31	9.7138	0.32267
35	100	25	8	83	1.12	0.0178	0.017444
35	500	25	8	37	2.13	0.3143	0.057444
40	100	25	7	69	1.00	0.0883	0.023000
40	1000	25	7	2	1.00	0.8059	0.099000
40	100	25	9	93	1.00	0.0253	0.023333
40	1000	25	9	35	1.02	0.1619	0.14811
45	100	25	8	78	1.00	0.0935	0.027778
45	1000	25	8	9	1.00	39.284	0.12200
45	100	25	9	86	1.00	0.0272	0.026111
45	1000	25	9	32	1.00	0.1899	0.17100
50	2000	30	10	32	1.00	9.2640	0.45800
50	2000	30	12	82	1.00	0.7974	0.78867
n	N	mz	c. p.	fails	avg. sol. count	avg. time	avg. time 90%

3 An Attack with Random Faults

Now we assume that the attacker can disturb somehow the control sequence. It may be, for example, some additional cycles before the start of the system. Then he can observe only the output of the generator. The second assumption is that the procedure can be repeated with the same input sequence and with different faults on control sequence. Our goal is to obtain bits of the input sequence.

3.1 General Idea of the Attack

We assume a probabilistic model: we have an input sequence $A = a_1a_2a_3\ldots$ and a set of control sequences $C^{(i)} = c_1^{(i)}c_2^{(i)}c_3^{(i)}\ldots$, $i = 1, 2, \ldots, n$ where all $A, C^{(1)}, C^{(2)}, \ldots, C^{(n)}$ are independent and truly random. For each $i = 1, 2, \ldots, n$ a sequence $Z^{(i)}$ is produced from A and $C^{(i)}$ by the shrinking generator. The attacker knows only $Z^{(i)}$ for $i = 1, 2, \ldots, n$ and tries to recover the sequence A.

Consider $z_1^{(i)}$, $i = 1, 2, \ldots, n$. If most of them are equal to, say, 0, then it is natural to assume that a_1 also equals zero. This is concluded from the fact that about half of $z_1^{(i)}$, $i = 1, 2, \ldots, n$, are a_1. Similar considerations are to be carried on for a_2, since about $\frac{1}{4}$ of $z_2^{(i)}$ are equal to a_2, about $\frac{1}{4}$ of $z_2^{(i)}$ are equal to a_3 (control sequence is then of the form $011*$ or $101*$) etc. Details of these fractions are given later. Of course situation is that clear if almost all of $z_j^{(i)}$ for a given j are equal to zero (or almost all are equal to one), but of course there are many intermediate situations which shall be discussed further.

Linear Equations. Let us count how many times one of a_j is taken as one of $z_k^{(1)}, z_k^{(2)}, \ldots, z_k^{(n)}$ and denote this number by $d(k, j)$. Now we can write the following system of equations:

$$\begin{cases} d(1,1)a_1 + d(1,2)a_2 + d(1,3)a_3 + \cdots = \sum_{j=1}^{n} z_1^{(j)}, \\ d(2,2)a_2 + d(2,3)a_3 + \cdots = \sum_{j=1}^{n} z_2^{(j)}, \\ d(3,3)a_3 + \cdots = \sum_{j=1}^{n} z_3^{(j)}, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \end{cases} \tag{4}$$

Of course the attacker does not know the numbers $d(k, j)$, but for large n he can approximate fractions $d(k, j)/n$ with appropriate probabilities. By $p_{k,j}$ denote the probability that jth element of a sequence A is taken as kth element of output sequence. More formally:

$$p_{k,j} = \Pr\left\{\sum_{i=1}^{j-1} c_i = k - 1 \quad \wedge \quad c_j = 1\right\} \tag{5}$$

where $c_1, c_2, \ldots$ is a random control sequence. So, $d(k, j)$ can be approximated by $n \cdot p_{k,j}$.

It is interesting now to investigate the distribution of random variable X_k such that $X_k = j$ if and only if jth element of the sequence A is taken as kth element of the output sequence Z. By definition $\Pr\{X_k = j\} = p_{k,j}$ and it follows from (5) that

$$p_{k,j} = \binom{j-1}{k-1} 2^{-j}, \quad \text{for } j \geq k, \quad \text{and} \quad p_{k,j} = 0, \quad \text{for } j < k. \tag{6}$$

Theorem 1

$$EX_k = 2k \quad \textit{and} \quad \operatorname{Var} X_k = 2k.$$

Proof. By the definition of X_k and from (6) it is obvious that

$$\sum_{j=k}^{\infty} \binom{j-1}{k-1} 2^{-j} = 1, \quad \text{for } k \geq 1. \tag{7}$$

In order to prove the equation $EX_k = 2k$, we write

$$\begin{aligned} EX_k &= \sum_{j=k}^{\infty} j \cdot p_{k,j} = \sum_{j=k}^{\infty} j \binom{j-1}{k-1} 2^{-j} = \sum_{j=k}^{\infty} j \frac{(j-1)!}{(k-1)!(j-k)!} 2^{-j} \\ &= \sum_{j=k}^{\infty} k \frac{j!}{k!(j-k)!} 2^{-j} = k \sum_{j=k}^{\infty} \binom{j}{k} 2^{-j} = 2k \sum_{j=k+1}^{\infty} \binom{j-1}{(k+1)-1} 2^{-j} \end{aligned}$$

The last expression equals $2k$ due to equation (7). To prove that $\operatorname{Var} X_k$, we first calculate EX_k^2.

$$\begin{aligned} EX_k^2 &= \textstyle\sum_{j=k}^{\infty} j^2 \cdot p_{k,j} = \sum_{j=k}^{\infty} j^2 \binom{j-1}{k-1} 2^{-j} \\ &= \textstyle\sum_{j=k}^{\infty} j^2 \frac{(j-1)!}{(k-1)!(j-k)!} 2^{-j} = \sum_{j=k}^{\infty} k \cdot j \frac{j!}{k!(j-k)!} 2^{-j} \\ &= k \textstyle\sum_{j=k}^{\infty} j \binom{j}{k} 2^{-j} = 2k \sum_{j=k+1}^{\infty} (j-1) \binom{j-1}{(k+1)-1} 2^{-j} \\ &= 2k \textstyle\sum_{j=k+1}^{\infty} j \binom{j-1}{(k+1)-1} 2^{-j} - 2k \sum_{j=k+1}^{\infty} \binom{j-1}{(k+1)-1} 2^{-j} \\ &= 2kEX_{k+1} - 2k = 4k^2 + 2k. \end{aligned}$$

Now we obtain $\operatorname{Var} X_k$ as $EX_k^2 - (EX_k)^2 = 2k$. □

Knowing EX_k and $\operatorname{Var} X_k$, we can approximate the number of variables that with high probability will be sufficient to build a finite version of system (4). Clearly, the exact probability $\Pr(X_k \leq x)$ is expressed by the formula

$$\Pr(X_k \leq x) = \textstyle\sum_{j=k}^{\lfloor x \rfloor} \binom{j-1}{k-1} 2^{-j}. \tag{8}$$

The results obtained according to (8) are presented as Table 3.

Table 3. Maximum number v of variable that will suffice to build an equation number k (with probability 0.99), according to formula (8)

k	1	10	20	30	40	50	100	1000
v	7	33	57	80	103	125	235	2106

The formula (8) is rather complicated and that makes it hard to observe the overall tendency. So we proceed in order to find a compact formula which would approximate (8) and would enable us to conclude about properties of defined random variables.

Theorem 2. *For random variables X_k defined earlier, for all integers $k \geq 1$ and for any real $p \in (0,1)$ we have*

$$\Pr\left(X_k \leq 2k \cdot \exp\sqrt{-2(\ln p)/k}\right) \geq 1 - p \tag{9}$$

Lemma 2. *For random variables X_k defined above, and for all integers $k \geq 1$, $r \geq 1$ we have*

$$EX_k^r \leq 2^r \frac{(k+r-1)!}{(k-1)!}. \tag{10}$$

Proof (of Lemma 2). If $r = 1$, then (10) is implied by Theorem 1. So assume that (10) is fulfilled for some r (and for all k) and we will prove it for $r+1$ and for all k. We pick an arbitrary k and write

$$\begin{aligned}
EX_k^{r+1} &= \textstyle\sum_{j=k}^{\infty} j^{r+1} \cdot p_{k,j} = \sum_{j=k}^{\infty} j^{r+1} \binom{j-1}{k-1} 2^{-j} \\
&= \textstyle\sum_{j=k}^{\infty} j^{r+1} \frac{(j-1)!}{(k-1)!(j-k)!} 2^{-j} = \sum_{j=k}^{\infty} k \cdot j^r \frac{j!}{k!(j-k)!} 2^{-j} \\
&= k \textstyle\sum_{j=k}^{\infty} j^r \binom{j}{k} 2^{-j} = 2k \sum_{j=k+1}^{\infty} (j-1)^r \binom{j-1}{(k+1)-1} 2^{-j} \\
&\leq 2k \textstyle\sum_{j=k+1}^{\infty} j^r \binom{j-1}{(k+1)-1} 2^{-j} = 2kEX_{k+1}^r \\
&\leq 2k \cdot 2^r \frac{((k+1)+r-1)!}{((k+1)-1)!} = 2^{r+1} \frac{(k+(r+1)-1)!}{(k-1)!} \ .
\end{aligned}$$

□

Proof (of Theorem 2). Let $m > 1$ be some real number, we will approximate probability $\Pr(X_k > 2km)$. Pick any integer r. Since for all positive valued random variables Y we have $EY > w \cdot \Pr(Y > w)$, we can write

$$\Pr(X_k > 2km) = \Pr(X_k^r > 2^r k^r m^r) < \frac{EX_k^r}{2^r k^r m^r}.$$

Now due to Lemma 2 we have

$$\begin{aligned}
\Pr(X_k > 2km) &\leq \tfrac{(k+r-1)!}{(k-1)! k^r m^r} = \tfrac{k(k+1)\ldots(k+r-1)}{k^r} m^{-r} \\
&= \left(1 + \tfrac{1}{k}\right)\left(1 + \tfrac{2}{k}\right) \ldots \left(1 + \tfrac{r-1}{k}\right) m^{-r} \\
&< \exp\left(\tfrac{1}{k}\right) \exp\left(\tfrac{2}{k}\right) \ldots \exp\left(\tfrac{r-1}{k}\right) m^{-r} \\
&= \exp\left(\frac{r(r-1)}{2k}\right) m^{-r} < e^{\frac{r^2}{2k}} m^{-r}.
\end{aligned}$$

Function $f(x) = e^{\frac{x^2}{2k}} m^{-x}$, $x \in \mathbf{R}_+$ reaches its absolute minimum for $x = k \ln m$, therefore we have

$$\Pr(X_k > 2km) < e^{\frac{(k \ln m)^2}{2k}} m^{-k \ln m} = e^{\frac{k(\ln m)^2}{2}} e^{-k(\ln m)^2} = e^{-\frac{k(\ln m)^2}{2}}.$$

Now let $p = e^{-k(\ln m)^2/2}$ then $m = \exp\left(\sqrt{\frac{-2 \ln p}{k}}\right)$ and for such m we have of course

$$\Pr\left(X_k > 2k \cdot \exp\left(\sqrt{\tfrac{-2 \ln p}{k}}\right)\right) < p.$$

Since there is a 1-to-1 correspondence between all $p \in (0, 1)$ and all reals $m > 1$, the theorem follows. □

Theorem 2 shows that for a fixed small p, the ratio between the number of needed variables and expected value of X_k with high probability approaches 1 as k goes to ∞. We can also see that for large enough k, going from equation number k to equation number $k + 1$ we will need only about 2 more variables.

3.2 The Algorithm and the Simulations

Input: n sequences $Z^{(1)}, Z^{(2)}, \ldots, Z^{(n)}$, as described in 3.1, a number of equations w, a real number $p \in (0, 1)$. The closer p to zero, the better solution.
Output: $x_1, x_2, \ldots, x_{w_v}$ – a candidate for the input sequence of the generator.
Method:

1. For each $k = 1, 2, \ldots, w$, calculate the number v_k which guarantees inequality

$$\Pr(X_k \le v_k) \ge 1 - p$$

2. Write the following approximation of the system (4):

$$\begin{cases} p_{1,1}x_1 + p_{1,2}x_2 + p_{1,3}x_3 + \cdots + p_{1,v_1}x_{v_1} = \frac{1}{n}\sum_{j=1}^{n} z_1^{(j)}, \\ p_{2,2}x_2 + p_{2,3}x_3 + \cdots + p_{2,v_2}x_{v_2} = \frac{1}{n}\sum_{j=1}^{n} z_2^{(j)}, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ p_{w,w}x_w + \cdots + p_{w,v_w}x_{v_w} = \frac{1}{n}\sum_{j=1}^{n} z_w^{(j)}, \end{cases} \tag{11}$$

3. Let $i := 1$, $v_0 := 0$ and $S_{\text{prev}} := \emptyset$.
   i numbers the equation considered in this step, S_{prev} is the set of solutions found after dealing with equations $1, 2, \ldots, i-1$.
4. Let $S := \emptyset$.
   S is the set solutions considered after regarding equation i.
5. Take some $s \in S_{\text{prev}}$. Let $S_{\text{prev}} := S_{\text{prev}} \setminus \{s\}$.
6. Consider all $2^{v_i - v_{i-1}}$ possible values of $x_{v_{i-1}+1}, \ldots, x_{v_i}$. For each of them concatenate it with s, name the concatenation s' and let $S := S \cup \{s'\}$. If $|S| = M + 1$ remove the element of S with the highest rank. The choice of the rank function and M is a matter of discussion further on.
7. If $S_{\text{prev}} \neq \emptyset$, then go to Step 5.
8. $i := i + 1$.
9. If $i \le w$, then put $S_{\text{prev}} := S$ and go to Step 4.
10. Take $s \in S$ with the lowest rank, and output v_w of its variables.

M, the maximum number of partial solutions kept while going from one equation to another, should be chosen carefully. If M is high (say, 2^{v_w}), then the algorithm would be equivalent to evaluating all possible guesses for v_w and therefore not practical even for $w = 20$ (see table 3). Too low values of M decrease chances to get a good solution. We have experimented with different values of M. Results for $M = 1000$ and $M = 10000$ are summarized in Table 4.

Now let us determine the rank function. Let the variables in a partial solution be $x_1, x_2, \ldots x_{v_i}$. Then

$$\text{rank}(s, i) = \sum_{m=1}^{i} \left(\frac{1}{n}\sum_{j=1}^{n} z_m^{(j)} - \sum_{j=m}^{v_m} p_{m,j}x_j \right)^2$$

Note that this formula lets us to use ranks calculated for equation $i - 1$, as our solutions are obtained from concatenation of partial solutions previously found with new guesses for a few new variables ($v_i - v_{i-1}$ is usually 2 or 3).

Although the algorithm outputs v_w bits of our guess of the input sequence, we have to stress that not all of the bits are of equal quality. The less is the number, the higher is the probability that the algorithm provides the right bit. Some experimental data is provided by Table 5.

One may ask how our assumption about independence of bits of control sequences stands in real situation. The Table 6 was calculated for control sequences

Table 4. Average percentage of the first 20 bits of our solution that fit the real input sequence (standard deviation in parentheses) for different values of n and M. 120 experiments with $w = 40$ executed for each choice of parameters.

n	M	hits
100	1000	67.25% (13.16%)
100	10000	70.88% (12.28%)
1000	1000	78.00% (11.87%)
1000	10000	84.08% (11.14%)
10000	1000	93.29% (8.96%)
10000	10000	95.00% (8.78%)

Table 5. Average percentage of correct values of bits (standard deviation in parentheses) for $M = 10000$ and $w = 40$. The results are computed for the first f bits of the algorithm output. Each row is a result of 120 trials.

n	f	hits
100	10	79.08% (17.70%)
100	20	70.88% (12.28%)
100	40	61.56% (8.59%)
100	60	58.51% (6.41%)
10000	10	100.00% (0.00%)
10000	20	95.00% (8.78%)
10000	40	77.96% (9.20%)
10000	60	69.50% (7.15%)

generated by good quality LFSRs of length 32 with 4 taps. The fault are generated as one additional shift. One might expect much worse behavior, but the experiments show that the difference is insignificant.

So far we can find most of few tens of bits, on average, of the input sequence. Finding sufficiently long sequence of bits is, in case of LFSR, equivalent to getting to know the whole sequences (it requires k bits in the case of known taps, and $2k$ bits in the case of unknown taps, where k is the number of bits of LFSR). If the attacker had some extreme power – like the possibility of obtaining, say, million different outputs for the same input sequence (and disturbed control sequences), then he could obtain more than 80% of 64 bits of the key. If one can have no more than 100 such sequences, then we expect to get about 60% of 64 bits. Observe (see the Table 5), that bits at the beginning of the sequence are more "reliable" than others, so if one wants to perform exhaustive check, the changing of bits should begin from "the right end" of bits being guessed. This way we will minimize searching, and in an average case most of guessing will be not necessary and the attack on 64 bits becomes realistic.

Table 6. The situation for faults generated by additional shifts in the control LFSR of length 32 and 4 taps. The table contains the average percentage of bits in the first 20 output bits that are reconstructed correctly (standard deviation in parentheses) for different values of n and M, and for $w = 40$. Each row is a result of 120 trials.

n	M	hits
100	1000	65.50% (10.17%)
100	10000	67.79% (10.76%)
1000	1000	76.33% (12.63%)
1000	10000	79.83% (11.42%)
10000	1000	93.96% (8.49%)
10000	10000	95.63% (8.00%)

4 Conclusions and Open Problems

Our attacks can be easily adapted against the weaker generators [7, 1, 4] related to the shrinking generator. In those cases the attacks are even easier.

Our attacks require injecting specific faults and restarting the device with partially the same internal state. While injecting such faults is potentially possible (especially for the second attack) it may require some design faults (so that potentially vulnerable parts of the device were placed on external layers). It shows at least that careful examining of a chip design might be necessary.

Potentially, such attacks can be adapted against the other algorithms like stream ciphers, hash functions, and even some symmetric block ciphers (for instance injecting synchronization faults into DES key schedule seems to be beneficial due to its' "shift-and-permute" design).

References

1. T. Beth, F. C. Piper, **The Stop-and-Go Generator**, EUROCRYPT'84, LNCS 209, pp. 88–92, Springer-Verlag, 1985.
2. D. Boneh, R. A. DeMillo, R. J. Lipton, **On the Importance of Checking Cryptographic Protocols for Faults**, EUROCRYPT '97, LNCS 1233, pp. 37–51, Springer-Verlag, 1997.
3. D. Coppersmith, H. Krawczyk, Y. Mansour, **The Shrinking Generator**, CRYPTO'93, LNCS 773, pp. 22–39, Springer-Verlag, 1993.
4. W. Chambers, D. Gollmann, **Clock-Controlled Shift Registers: A Review**, IEEE J. Selected Areas Comm., 7(4): 525–533, May 1989.
5. E. Dawson, J. Dj. Golič, L. Simpson, **A Probabilistic Correlation Attack on the Shrinking Generator**, Information Security and Privacy – ACISP '98, LNCS 1438, pp. 147–158, Springer-Verlag, 1998.
6. P. Ekdahl, Th. Johansson, W. Meier, **Predicting the Shrinking Generator with Fixed Connections**, EUROCRYPT 2003, LNCS 2656, pp. 330–344, Springer-Verlag, 2003.

7. P. R. Geffe, **How to Protect Data with Ciphers That Are Really Hard to Break**, Electronics, Jan. 4, pp. 99–101, 1973.
8. J. Dj. Golič, L. O'Connor, **Embedding and Probabilistic Correlation Attacks on Clock-Controlled Shift Registers**, EUROCRYPT '94, LNCS 950, pp. 230–243, Springer-Verlag, 1995.
9. M. Gomułkiewicz, M. Kutyłowski, Th. H. Vierhaus, P. Wlaź, **Synchronization Fault Cryptanalysis for Breaking A5/1**, International Workshop on Efficient and Experimental Algorithms – WEA'05, LNCS 3503, pp. 415–427, Springer Verlag, 2005.
10. M. Krause, S. Lucks, E. Zenner, **Improved Cryptanalysis of the Self-Shrinking Generator**, Information Security and Privacy – ACISP 2001, LNCS 2119, pp. 21–35, Springer-Verlag, 2001.
11. W. Meier, O. Staffelbach, **The Self-shrinking Generator**, EUROCRYPT'94, LNCS 950, pp. 205–214, Springer-Verlag, 1995.
12. M. Mihaljevic, **A Faster Cryptanalysis of the Self-shrinking Generator**, Information Security and Privacy – ACISP '96, LNCS 1172, pp. 182–188, Springer-Verlag, 1996.
13. T. R. N. Rao, Chung-Huang Yang, Kencheng Zeng, **An Improved Linear Syndrome Algorithm in Cryptanalysis With Applications**, CRYPTO'90, LNCS 537, pp. 34–47, Springer-Verlag, 1991.
14. E. Zenner, **On the Efficiency of the Clock Control Guessing Attack**, Information Security and Cryptology – ICISC 2002, LNCS 2587, pp. 200–212, Springer-Verlag, 2003.

Some Advances in the Theory of Voting Systems Based on Experimental Algorithms*

Josep Freixas[1] and Xavier Molinero[2]

[1] Dept. de Matemàtica Aplicada 3
Universitat Politècnica de Catalunya,
Escola Politècnica Superior d'Enginyeria de Manresa,
E-08242 Manresa, Spain
josep.freixas@upc.edu
[2] Dept. de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya,
Escola Politècnica Superior d'Enginyeria de Manresa,
E-08242 Manresa, Spain
molinero@lsi.upc.edu

Abstract. In voting systems, game theory, switching functions, threshold logic, hypergraphs or coherent structures there is an important problem that consists in determining the weightedness of a voting system by means of trades among voters in sets of coalitions. The fundamental theorem by Taylor and Zwicker [8] establishes the equivalence between weighted voting games and k-trade robust games for each positive integer k. Moreover, they also construct, in [9], a succession of games G_k based on magic squares which are $(k-1)$-trade robust but not k-trade robust, each one of these games G_k has k^2 players.

The goal of this paper is to provide improvements by means of different experiments to the problem described above. In particular, we will classify all complete games (a basic class of games) of less than eight players according to whether they are: a weighted voting game or a game which is $(k-1)$-trade robust but not k-trade robust for all values of k. As a consequence it will we showed the existence of games with less than k^2 players which are $(k-1)$-trade robust but not k-trade robust. We want to point out that the classifications obtained in this paper by means of experiments are new in the mentioned fields.

1 Introduction

Simple games can be viewed as models of voting systems in which a single alternative, such as a bill or an amendment, is pitted against the status quo.

* This research was supported by the Spanish *Ministerio de Ciencia y Tecnología* programme TIC2002-00190 (AEDRI II) and TIN2005-05446 (ALINEX), and Grant BFM 2003–01314 of the *Ministerio de Ciencia y Tecnología* and the European Regional Development Fund.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 73–84, 2006.

Definition 1. *A simple game G is a pair $(N, \mathcal{W})$ in which $N = \{1, 2, \ldots, n\}$ and $\mathcal{W}$ is a collection of subsets of N that satisfies: $N \in \mathcal{W}$, $\emptyset \notin \mathcal{W}$ and (monotonicity) $S \in \mathcal{W}$ and $S \subseteq T \subseteq N$ then $T \in \mathcal{W}$.*

Any set of voters is called a *coalition*, and the set N is called the *grand coalition*. Members of N are called *players* or *voters*, and the subsets of N that are in $\mathcal{W}$ are called *winning coalitions*. The intuition here is that a set S is a winning coalition iff the bill or amendment passes when the players in S are precisely the ones who voted for it. A subset of N that is not in $\mathcal{W}$ is called a *losing coalition*. A *minimal winning coalition* is a winning coalition all of whose proper subsets are losing. Because of monotonicity, any simple game is completely determined by its set of minimal winning coalitions. Before proceeding, we introduce a real–world example (see Taylor [7] for an extensive illustration of real–world examples modeled as simple games).

Example 1. The System to Amend the Canadian Constitution. Since 1982, an amendment to the Canadian Constitution can become law only if it is approved by at least seven of the ten Canadian provinces, subject to the proviso that the approving provinces have, among them, at least half of Canada's population. It was first studied in Kilgour [6]. A census (in percentages) for the Canadian provinces was: Prince Edward Island (1%), Newfoundland (3%), New Brunswick (3%), Nova Scotia (4%), Manitoba (5%), Saskatchewan (5%), Alberta (7%), British Columbia (9%), Quebec (29%) and Ontario (34%).

For example observe that coalitions (from now on we make use of abridgements to denote the province) $S_1 = \{PEI, New, Man, Sas, Alb, BC, Que\}$ and $S_2 = \{NB, NS, Man, Sas, Alb, BC, Ont\}$ are minimal winning coalitions because they both have exactly 7 provinces and their total population surpasses the 50%. Instead, coalitions $T_1 = \{Man, Sas, Alb, BC, Que, Ont\}$ and $T_2 = \{PEI, New, NB, NS, Man, Sas, Alb, BC\}$ are both losing because T_1 does not have 7 members and T_2 does not reach the 50% of the total Canada's population.

2 Preliminaries

2.1 Weighted Simple Games

Of fundamental importance to simple games are the subclasses of weighted simple games and complete simple games.

Definition 2. *A simple game $G = (N, \mathcal{W})$ is said to be weighted if there exists a "weight function" $w : N \to \mathbb{R}$ and a real number "quota" $q \in \mathbb{R}$ such that a coalition S is winning precisely when the sum of the weights of the players in S meets or exceeds the quota.*

The *associated weight vector* is $(w_1, \ldots, w_n)$. Any specific example of such a weight function $w : N \to \mathbb{R}$ and quota q as in Definition 2 are said to *realize* G as a weighted game. A particular realization of a weighted simple game is denoted as $[q; w_1, \ldots, w_n]$.

Simple games and in particular weighted games and complete games, which we will introduce later on, have been studied in a variety of different mathematical contexts: Boolean or switching functions, threshold logic, hypergraphs, coherent structures, Sperner systems, and clutters. One of the most important problems for all those fields is determining whether a simple game can be realized as a weighted simple game. The only results giving necessary and sufficient conditions can be found under one of the next three topics: *geometric approach based on separating hyperplanes*; *algebraic approach based on systems of linear inequalities*; *approach based on trading transforms*. The geometric approach requires translating the question of weightedness into one of separability via a hyperplane of two convex subsets of $\mathbb{R}^n$. The key idea in the algebraic approach involves translating weightedness via vector sums into conditions equivalent to the solvability of systems of linear inequalities. The approach based on trades is the most natural and suggests several interpretations that will be tackled here from a computational viewpoint.

Definition 3. *Suppose $G = (N, \mathcal{W})$ is a simple game. Then a trading transform (for G) is a coalition sequence $\mathcal{J} = \langle S_1, \ldots, S_j, T_1, \ldots, T_j \rangle$ (from G) of even length satisfying the following condition: $|\{i : p \in S_i\}| = |\{i : p \in T_i\}|$ for all $p \in N$. S_i are called the pre–trade coalitions and the T_i the post–trade coalitions, and we will say that $\langle S_1, \ldots, S_j \rangle$ has been converted by a trade to $\langle T_1, \ldots, T_j \rangle$.*

Definition 4. *A k-trade for a simple game G is a trading transform $\mathcal{J} = \langle S_1, \ldots, S_j, T_1, \ldots, T_j \rangle$ in which $j \leq k$. The simple game G is k-trade robust if there is no such $\mathcal{J}$ for which all the Ss are winning in G and all the Ts are losing in G. If G is k-trade robust for all k, then G is said to be trade robust.*

Loosely speaking, G is k-trade robust if a sequence of k or fewer (not necessarily distinct) winning coalitions can never be rendered losing by a trade.

Theorem 1. *(Taylor and Zwicker, [8]). For a simple game $G = (N, \mathcal{W})$, the following are equivalent:*

(i) G is weighted. (ii) G is trade robust. (iii) G is $2^{2^{|N|}}$-trade robust.

Notice that a naive checking of *(iii)* is a finite (albeit lengthy) process, whereas a naive checking of weightedness directly is an infinite process. Moreover, Theorem 1 actually provides a fairly simple and uniform procedure for showing that certain games are not weighted: one produces a sequence of winning coalitions and indicates trades among these winning coalitions that convert all of them to losing coalitions.

In Example 1 we have seen that the trading transform $\mathcal{J} = \langle S_1, S_2, T_1, T_2 \rangle$ converts the winning coalitions S_1 and S_2 to the losing ones T_1 and T_2. Therefore, by Theorem 1, the system is not weighted. Another complex voting system is the current European Economic Community. The countries are: Germany, United Kingdom, France, Italy, Spain, Poland, Romania, The Netherlands, Greece, Czech Republic, Belgium, Hungary, Portugal, Sweden, Bulgaria,

Austria, Slovak Republic, Denmark, Finland, Ireland, Lithuania, Latvia, Slovenia, Estonia, Cyprus, Luxemburg and Malta. They are represented by the set $\{1, 2, \ldots, 27\}$ where Germany = 1, United Kingdom = 2, and so on.

The first decision rule is the simple game given by $v_1 \cap v_2 \cap v_3$, where the three weighted voting games corresponding to votes, countries and population, are the following:

$v_1 = [255;\ 29, 29, 29, 29, 27, 27, 14, 13, 12, 12, 12, 12, 12, 10, 10, 10, 7, 7, 7, 7, 7, 4, 4, 4, 4, 4, 3]$,
$v_2 = [14;\ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$,
$v_3 = [620;\ 170, 123, 122, 120, 82, 80, 47, 33, 22, 21, 21, 21, 21, 18, 17, 17, 11, 11, 11, 8, 8, 5, 4, 3, 2, 1, 1]$.

Freixas [2] proves that this system cannot be expressed as intersection of only one or two voting systems. To see that it fails to be 2-trade robust, we may consider coalitions $S_1 = T_1 \setminus \{14, 15, 17\} \cup \{3\}$ and $S_2 = T_2 \setminus \{3\} \cup \{14, 15, 17\}$ where $T_1 = [1, 2] \cup [5, 19] \cup \{26\}$ and $T_2 = [1, 13]$ (if $i \leq j$ we write $[i, j] = \{k \in N\ :\ i \leq k \leq j\}$). The corresponding weights are:

	v_1	v_2	v_3
S_1	256	16	803
S_2	255	15	897

	v_1	v_2	v_3
T_1	254	18	727
T_2	257	13	883

So after trades, the losing coalitions T_1 and T_2 in v convert to the winning coalitions S_1 and S_2; consequently, game v cannot be weighted because, after the trade, S_1 and S_2 cannot simultaneously gain weight.

The question of whether any bounded amount of trade robustness implies weightedness was settled by Taylor and Zwicker.

Theorem 2. *(Taylor and Zwicker, [9]) For each integer $k \geq 3$, there exists a simple game G_k with k^2 players, that is $(k-1)$-trade robust, but not k-trade robust.*

It will be of interest checking whether it exists a game with less than 9 players being 2-trade but not 3-trade, or a game with less than 16 players being 2 and 3 trade robust but not 4-robust. If the answer to these questions were affirmative then it would be of interest determining the minimum number of voters needed to reach games within these categories. In this paper we will make experiments in order to solve this problem for some values. Unfortunately the number of simple games is too large to be tackled straightforwardly. We introduce another significant class of simple games that will help us to face our problem.

2.2 Complete Simple Games

Definition 5. *Suppose $(N, \mathcal{W})$ is a simple game. Then G is said to be swap robust if a one–for–one exchange between two winning coalitions can never render both losing.*

Thus, swap robustness differs from trade robustness in two ways: the trades involve only two coalitions, and the exchanges are one for one. It is fairly easy to generate simple games that are not swap robust. Let us consider the following relations.

Definition 6. *Let $(N, \mathcal{W})$ be a simple game, i and j be two voters. Players i and j are said to be equally desirable, denoted by $i \sim j$ if: for any coalition S such that $i \notin S$ and $j \notin S$, $S \cup \{i\} \in \mathcal{W} \Leftrightarrow S \cup \{j\} \in \mathcal{W}$.*

Definition 7. *(Isbell, [5]) Let $(N, \mathcal{W})$ be a simple game, i and j be two voters. Player i is said to be more desirable than j, denoted by $i \succ j$ if the following two conditions are fulfilled:*

1. *For every coalition S such that $i \notin S$ and $j \notin S$, $S \cup \{j\} \in \mathcal{W} \Rightarrow S \cup \{i\} \in \mathcal{W}$.*
2. *There exists a coalition T such that $i \notin T$ and $j \notin T$, $T \cup \{i\} \in \mathcal{W}$ and $T \cup \{j\} \notin \mathcal{W}$.*

The *desirability relation* denoted by $\succeq$ is defined in N as follows: $i \succeq j$ if $i \succ j$ or $i \sim j$, we say that i is at least as desirable as j as coalitional partner. It is straightforward to check that $\sim$ is an equivalence relation, and that $\succeq$ is a partial ordering of the resulting equivalence classes.

Definition 8. *A simple game $(N, \mathcal{W})$ is complete or linear if the desirability relation is a complete preordering.*

In a complete simple game we may decompose N in a collection of subsets, called classes, $N_1 > N_2 > \cdots > N_t$ forming a partition of N and understanding that if $i \in N_p$ and $j \in N_q$ then: $p = q$ iff $i \sim j$ and $p < q$ if $i \succ j$. The following is a characterization of complete simple games.

Theorem 3. *(Taylor and Zwicker, [10]) G is a complete simple game iff G is swap robust.*

Because trade robustness implies swap robustness it may be concluded that if a simple game is weighted then it is complete. Carreras and Freixas [1] provide a classification theorem for complete simple games that allow to enumerate all these games up to isomorphism by listing the possible values of certain invariants. Previously to state it we need some preliminaries.

If $\overline{n} = (n_1, \ldots, n_t) \in \mathbb{N}^t$, we define $\Lambda(\overline{n}) = \{\overline{m} \in (\mathbb{N} \cup \{0\})^t : \overline{m} \leq \overline{n}\}$ the set of all vectors $\overline{m} = (m_1, \ldots, m_t)$ whose components satisfy $0 \leq m_k \leq n_k$ for all $k = 1, \ldots, t$ with the ordering δ given by the comparison of partial sums; that is,

$$\overline{m}\,\delta\,\overline{p} \text{ iff } \sum_{i=1}^{k} m_i \geq \sum_{i=1}^{k} p_i \quad \text{for } k = 1, 2, \ldots, t.$$

If $\overline{m}\,\delta\,\overline{p}$ we will say that $\overline{m}$ *δ-dominates* $\overline{p}$. If $\overline{m} \not\delta\ \overline{p}$ and $\overline{p} \not\delta\ \overline{m}$ we will say that $\overline{m}$ and $\overline{p}$ are *not δ-comparable*. From now on, we shall write $\Sigma_k(\overline{m}) = \sum_{i=1}^{k} m_i$ for $k = 1, 2, \ldots, t$ and $\Sigma(\overline{m}) = (\Sigma_1(\overline{m}), \ldots, \Sigma_t(\overline{m}))$ so that $\overline{m}\,\delta\,\overline{p}$ iff $\Sigma(\overline{m}) \geq \Sigma(\overline{p})$. It is not difficult to check that the couple $(\Lambda(\overline{n}), \delta)$ is a distributive lattice. Finally, two simple games $(N, \mathcal{W})$ and $(N', \mathcal{W}')$ are said to be isomorphic if there is a bijective map $f : N \to N'$ such that $S \in \mathcal{W}$ iff $f(S) \in \mathcal{W}'$; f is called an *isomorphism* of simple games.

To make understandable the following theorem we need to introduce the lexicographical ordering by partial sums. If $h < h'$, then there exists some l such that $\Sigma_k(\overline{m}_h) = \Sigma_k(\overline{m}_{h'})$ for $k < l$ and $\Sigma_l(\overline{m}_h) > \Sigma_l(\overline{m}_{h'})$.

Theorem 4. *(Carreras and Freixas, [1]).*

Part A *Let $G = (N, \mathcal{W})$ be a complete simple game with nonempty classes $N_1 > N_2 > \cdots > N_t$, let $\overline{n}$ be the vector defined by their cardinalities, and let $\mathcal{M} = (m_{i,j})$, with $1 \leq i \leq r$ and $1 \leq j \leq t$, be the matrix satisfying the four conditions below:*
(1) $m_{i,j} \in \mathbb{N} \cup \{0\}$ and $0 \leq m_{i,j} \leq n_i$ for all i, j with $1 \leq i \leq r$ and $1 \leq j \leq t$;
(2) every pair of rows of $\mathcal{M}$, $\overline{m}_h$ and $\overline{m}_{h'}$ are not δ-comparable if $h \neq h'$;
(3) if $t = 1$ then $m_{1,1} > 0$; if $t > 1$ then for every $k < t$ there exists some h such that $m_{h,k} > 0$ and $m_{h,(k+1)} < n_{k+1}$; and
(4) the rows of $\mathcal{M}$ are lexicographically ordered by partial sums.

Part B *(Uniqueness) Two complete simple games $(N, \mathcal{W})$ and $(N', \mathcal{W}')$ are isomorphic iff $\overline{n} = \overline{n}'$ and $\mathcal{M} = \mathcal{M}'$.*

Part C *(Existence) Given a vector $\overline{n}$ and a matrix $\mathcal{M}$ satisfying the conditions of part A, there exists a complete simple game $(N, \mathcal{W})$ the characteristic invariants of which are $\overline{n}$ and $\mathcal{M}$.*

We need now to describe how to get $(\overline{n}, \mathcal{M})$ from $(N, \mathcal{W})$ and reciprocally. As $(N, \mathcal{W})$ is a complete simple game either $i \succ j$, or $i \sim j$, or $j \succ i$ for all pair of voters. Voters being equally desirable are grouped in classes N_k, and the notation $N_p > N_q$ means that $i \succ j$ for each $i \in N_p$ and $j \in N_q$, Let $N_1 > N_2 > \cdots > N_t$. Components of vector $\overline{n}$ are defined by $n_k = |N_k|$. Rows of $\mathcal{M}$ are obtained in the following way, for each $S \in \mathcal{W}$ we consider the associated vector $\overline{s} \in \Lambda(\overline{n})$ with components $s_k = |S \cap N_k|$, $\overline{s}$ is a row of $\mathcal{M}$ if it is not dominated for any other vector associated to a winning coalition. Once the collection of non-dominated vectors corresponding to winning coalitions is determined we need to order them lexicographically. Reciprocally, given $\overline{n}$ and $\mathcal{M}$ let $n = \Sigma_t(\overline{n}) = n_1 + n_2 + \cdots + n_t$ be the number of players, $N = \{1, 2, \ldots, n\}$ be the set of players, and $N_1, N_2, \ldots, N_t$ be subsets of N formed, respectively, by $n_1, n_2, \ldots, n_t$ elements (which may be chosen following the natural ordering). By Theorem 4(A), none of these subsets is empty. For each $S \subseteq N$ we consider $\overline{s} = (s_1, s_2, \ldots, s_t)$, where $s_k = |S \cap N_k|$ for $k = 1, 2, \ldots, t$. Then, the set of winning coalitions is $\mathcal{W} = \{S \subseteq N \ : \ \overline{s} \ \delta \ \overline{m}_h \text{ for some row } \overline{m}_h \text{ of } \mathcal{M}\}$.

Theorem 4 is a parametrization theorem, because it allows one to enumerate all complete games up to isomorphism by listing the possible values of certain invariants. For Example 1, the characteristic invariants are $\overline{n} = (2, 8)$, $\mathcal{M} = (1\ 6)$. Although the voting system of the current European Economic Community is complete its representation using characteristic invariants is a bit complex.

3 Experiments on the Complete Simple Games

3.1 A New Theoretical Approach to Performance Experiments

In this subsection we reformulate the developed theory in the preceding section in order to deal with it from a more efficient computational viewpoint. For lack of space, we just give here the main ideas of some new theoretical results.

In general, to describe a simple game it is enough giving the list of minimal winning coalitions. If moreover, the game is complete and the ordering induced by the desirability relation among components is known, then we can use a subset of minimal winning coalitions to entirely describe the game. Indeed, a coalition S such that its associated vector $\overline{s}$ is a row of $\mathcal{M}$ is called a *δ-minimal winning* coalition. Notice that each δ-minimal winning coalition is a minimal winning coalition but the reciprocal is not true. For instance, in Example 1 a coalition formed by Ontario, Quebec and 5 more provinces is minimal winning but not δ-minimal winning since its associated vector $(2, 5)$ is not a row of matrix $\mathcal{M}$.

For the purpose of studying when a game is weighted in terms of trade robustness we may confine to study only complete simple games, because the remaining simple games are not swap robust which is the simplest case of not being 2-trade robust. Within the framework of complete simple games we can take advantage of using the equivalent representation $(\overline{n}, \mathcal{M})$, which allows *using models of coalitions instead of coalitions* and *considering only models which are rows of matrix $\mathcal{M}$*. These latter properties become essential in this section devoted to algorithms and experiments. The basic idea is that it simplifies the description of the algorithms as well as it meaningfully improves the performance of the experiments.

Definition 9. (cf. Definition 3) *Suppose $G = (N, \mathcal{W})$ is a simple game. Then a δ-trading transform (for G) is a coalition sequence $\mathcal{J} = \langle S_1, \ldots, S_j, T_1, \ldots, T_j \rangle$ (from G) of even length satisfying condition $|\{i : p \in S_i\}| = |\{i : p \in T_i\}|$ for all $p \in N$, where $S_1, \ldots, S_j$ are δ-minimal winning coalitions.*

Definition 10. (cf. Definition 4) *A k-δ-trade for a simple game G is a δ-trading transform $\mathcal{J} = \langle S_1, \ldots, S_j, T_1, \ldots, T_j \rangle$ in which $j \leq k$. The simple game G is k-δ-trade robust if there is no such $\mathcal{J}$ for which all the Ss are δ-minimal winning coalitions in G and all the Ts are losing in G. If G is k-δ-trade robust for all k, then G is said to be δ-trade robust.*

Proposition 1. *Let $G = (N, \mathcal{W})$ be a complete game. Then, G is k-trade robust iff G is k-δ-trade robust.*

In the following, we provide a trading version applied to indices of columns of $\mathcal{M}$ and vectors instead of players and coalitions.

Definition 11. *Let $G = (N, \mathcal{W})$ be a complete simple game with characteristic invariants $(\overline{n}, \mathcal{M})$. A vectorial trading transform for G is a vectorial sequence $\mathcal{J}' = \langle \overline{x}_1, \ldots, \overline{x}_j, \overline{y}_1, \ldots, \overline{y}_j \rangle$ of even length satisfying the following conditions:*

$$\sum_{i=1}^{j} x_{i,k} = \sum_{i=1}^{j} y_{i,k} \quad \forall\, k \in [1, t] \tag{1}$$

where $\overline{x}_1, \ldots, \overline{x}_j$ are rows of $\mathcal{M}$ with repetitions allowed, and $\overline{y}_1, \ldots, \overline{y}_j$ belong to $\Lambda(\overline{n})$.

Definition 12. *Let $G = (N, \mathcal{W})$ be a complete simple game with characteristic invariants $(\overline{n}, \mathcal{M})$. Then, G is k-invariant-trade robust (k-I-T-R, for short) if there is no a vectorial trading transform $\mathcal{J}' = \langle \overline{x}_1, \ldots, \overline{x}_j, \overline{y}_1, \ldots, \overline{y}_j \rangle$ such that each $\overline{x}_i$ is a row of $\mathcal{M}$ and each $\overline{y}_k \in \Lambda(\overline{n})$ for $1 \leq k \leq j$ satisfies $\overline{y}_k \not\oint \overline{m}_i$ for every row $\overline{m}_i$ of $\mathcal{M}$. If G is k-I-T-R for all positive integer k, then $(N, \mathcal{W})$ is invariant-trade robust (I-T-R, for short).*

The following proposition states that Definitions 11 and 12 merely correspond to Definitions 9 and 10 if we consider the context $(\overline{n}, \mathcal{M})$ instead of $(N, \mathcal{W})$. We omit the proof because applying Proposition 1 it follows that k-trade robust is equivalent to k-δ-trade robust and word–by–word this is equivalent to k-I-T-R.

Proposition 2. *Let $G = (N, \mathcal{W})$ be a simple game with characteristic invariants $(\overline{n}, \mathcal{M})$. Then, $(N, \mathcal{W})$ is k-trade robust iff $(\overline{n}, \mathcal{M})$ is k-I-T-R.*

Theorem 1 by Taylor and Zwicker for simple games can be adapted to complete simple games.

Theorem 5. *Let G be a complete simple game $(N, \mathcal{W})$ with characteristic invariants $(\overline{n}, \mathcal{M})$ and t being the number of columns of $\mathcal{M}$. Then the following are equivalent:*

(i) G is weighted. (ii) G is invariant-trade robust. (iii) G is 2^{2^t}-I-T-R.

3.2 A Full Classification for Complete Simple Games with Less Than Eight Voters

All experiments we have made are based on Theorem 5. It gives a new viewpoint to determine if a complete simple game is trade robust. Our programs[1] have been written for C++ and run under Linux in Pentium 4 at 1.7 GHz with 512 Mb of RAM. To set an example, we just sketch here one of the implemented algorithms. So, Algorithm 1 sketches the used recursive function (based on backtracking method) to determine if a given simple game with characteristic invariants $(\overline{n}, \mathcal{M})$ is k-I-T-R or not. The parameters of function *DoingForkTrade* mean the following: $\mathcal{M}$ and $\overline{n}$ are the given matrix and vector, r and t are the number of computed rows and columns for the current matrix $\mathcal{Y} = (y_{i,j})$,[2] and finally, the auxiliary parameter *aux* is used to improve the algorithm. This function calls three additional functions:

- Function *CanonicalMatrix*($\mathcal{Y}$) returns **true** if the rows of $\mathcal{Y}$ are in lexicographic order (by rows and by columns), **false** otherwise.
- Function *AnyRowOfYDominatesM*($\mathcal{Y}, \mathcal{M}$) returns **true** if any row of $\mathcal{Y}$ dominates at least one row of $\mathcal{M}$, **false** otherwise.
- Function *PartialSums*($\mathcal{M}, i_r, e_r, i_t, e_t$) adds up the integers of $\mathcal{M}$ from row i_r to row e_r and from column i_t to column e_t.

[1] They are available on request from the authors.

[2] We denote by $\mathcal{Y}$ any of the matrices with rows $\overline{y}_1, \ldots, \overline{y}_j$ that are solution of equation in Definition 11, showing a failure of j-trade robust.

Algorithm 1. Recursive function that returns **true** iff $(\overline{n}, \mathcal{M})$ is k-I-T-R following the criteria of Proposition 2

```
1.  procedure DoingForkTrade(M, Y, n, r, t, aux)
2.      if (r = k) then  /* k-th row of Y */
3.          y_{r,t} := aux;
4.          if y_{r,t} > n_t or aux < 0 then return true fi
5.          if t = Colums_Of_Matrix(M) then  /* t-th column of Y */
6.              if CanonicalMatrix(Y) and not AnyRowOfYDominatesM(Y, M) then
7.                  Print("It is not trade robust."); return false
8.              fi
9.          else r := 1; t := t + 1; aux := PartialSums(M, 1, k, t, t)
10.                 if not DoingForkTrade2(M, Y, n, r, t, aux) then return false fi
11.         fi
12.     else Maux := PartialSums(M, 1, k, t, t); Yaux := PartialSums(Y, 1, r, t, t)
13.         for y_{r,t} from 0 to min(n_t, Maux − Yaux) do
14.             if not DoingForkTrade(M, Y, n, r + 1, t, aux − y_{r,t}) then
15.                 return false
16.             fi
17.         end
18.     fi
19.     return true
20. end
```

In general, given $(\overline{n}, \mathcal{M})$ and a positive integer k, calling the recursive function *DoingForkTrade*$(\mathcal{M}, \mathcal{Y}, \overline{n}, 1, 1,$ *PartialSums*$(\mathcal{M}, 1, k, 1, 1))$, it returns **true** (Line 19) if it is k-I-T-R, and **false** (Line 7) otherwise.

Unfortunately, the number of matrices associated to a fixed number n of voters is huge for $n > 8$. However, the experiments are successful for a small fixed number of columns (two or three) and for small numbers of voters ($n < 9$).

Table 1 provides a detailed classification of all complete simple games: the number of complete games (briefly CG), the number of weighted games (briefly WG), and the number of non k-invariant-trade robust but $(k-1)$-invariant-trade robust games (non k-I-T-R, for short). Finally, the number of complete games being non weighted is gathered in non invariant-trade robust games (I-T-R, for short).

Table 1. Full classification of simple games for $n < 8$

n	1	2	3	4	5	6	7
CG	1	3	8	25	117	1171	44313
WG	1	3	8	25	117	1111	29373
non I-T-R	0	0	0	0	0	60	14940
non 2-*I-T-R*	0	0	0	0	0	57	13915
non 3-*I-T-R*	0	0	0	0	0	3	1011
non 4-*I-T-R*	0	0	0	0	0	0	14

In particular, $n = 6$ is the minimum number of voters required to achieve simple games which are 2-I-T-R but not 3-I-T-R; $n = 7$ is the minimum number of voters required to achieve simple games which are 3-I-T-R but not 4-I-T-R. Tables 4 and 5 in Appendix enumerate all these extreme cases giving vector $\overline{n}$, matrix $\mathcal{M}$ and a matrix $\mathcal{Y}$ which fulfills equation in Definition 11 and shows a failure to be k-I-T-R: $k = 3$ for $n = 6$, and $k = 4$ for $n = 7$.

3.3 A Detailed Analysis for Simple Games with Two and Three Columns

The huge number of simple games for more than 8 voters does not allow to consider all simple games in a reasonable time. However, fixing a small number of columns (two or three) we can do an exhaustive experimental study even for 10 voters. In terms of simple games (voting systems) few columns mean the existence of many equally desirable voters which is highly frequent.

For two columns, we have checked that all simple games with $n \leq 10$ are either I-T-R or non 2-I-T-R (see Table 2). For three columns we also have found the full classification (see Table 3). We do not show particular examples because of the lack of space.

Table 2. Number of non k-I-T-R simple games with just two columns

n	1	2	3	4	5	6	7	8	9	10
CG	0	1	5	15	36	76	148	273	485	839
non 2-I-T-R	0	0	0	0	0	2	10	34	94	229
non 3-I-T-R	0	0	0	0	0	0	0	0	0	0

Table 3. Number of non k-I-T-R simple games with just three columns

n	1	2	3	4	5	6	7	8	9	10
CG	0	0	0	6	50	262	1114	4278	15769	58147
non 2-I-T-R	0	0	0	0	0	6	130	1116	6858	35431
non 3-I-T-R	0	0	0	0	0	0	6	39	160	506
non 4-I-T-R	0	0	0	0	0	0	2	11	39	115
non 5-I-T-R	0	0	0	0	0	0	0	0	2	—

4 Conclusion and Future Work

In this paper we have made experiments that allow:

(*i*) To classify all complete simple games, CG, for $n < 8$ according whether to they are: WG, non 2-I-T-R, non 3-I-T-R and non 4-I-T-R.
(*ii*) To check if a particular complete simple game with n voters is k-I-T-R for each positive integer k.
(*iii*) To study important subclasses, those with either one, two or three columns, whenever $n < 11$.

The given results in (i) for $n = 6, 7$ are new as well as the results obtained in (iii) for $n > 5$.

Our experiments suggest three important theoretical conjectures which we are developing.

Conjecture 1. Any simple game with just two types of voters (two classes) is either `I-T-R` or non 2-`I-T-R`.

Conjecture 2. Three columns are enough to find games which are $(k-1)$-`I-T-R` but non k-`I-T-R` for any positive integer k.

Conjecture 3. It is possible to find a game which is $(k-1)$-`I-T-R` but non k-`I-T-R` for any positive integer k, where the number of voters is $\mathrm{O}(k)$ instead of k^2 (cf. Theorem 2).

It is also interesting to study the required CPU time depending on the number of voters, the number of columns. Even, to do an exhaustive analysis depending on the number of rows (games with a single row are called *complete simple games with minimum* and have been studied in [3]).

Another future work is, for a fixed number of voters n, to generate a random game $(\overline{n}, \mathcal{M})$ and study trade robustness.

Finding appropriate weights for voters and a quota (a realization), for the class of *WG*.

References

1. F. Carreras and J. Freixas. Complete simple games. *Mathematical Social Sciences*, 32:139–155, 1996.
2. J. Freixas. The dimension for the European Union Council under the Nice rules. *European Journal of Operational Research*, 156(2):415–419, 2004.
3. J. Freixas and M.A. Puente. *Complete games with minimum.* Annals of Operations Research, 84:97–109, 1998.
4. J. Freixas and W.S. Zwicker. Weighted voting, abstention, and multiple levels of approval. *Social Choice and Welfare*, 21:399–431, 2003.
5. J.R. Isbell. A class of simple games. *Duke Mathematics Journal*, 25:423–439, 1958.
6. D.M. Kilgour. A formal analysis of the amending formula of Canada's Constitution. *Act. Canadian Journal of Political Science*, 16:771–777, 1983.
7. A.D. Taylor. *Mathematics and Politics.* Springer Verlag, New York, USA, 1995.
8. A.D. Taylor and W.S. Zwicker. A characterization of weighted voting. *Proceedings of the American mathematical society*, 115:1089–1094, 1992.
9. A.D. Taylor and W.S. Zwicker. Simple games and magic squares. *Journal of combinatorial theory, ser. A*, 71:67–88, 1995.
10. A.D. Taylor and W.S. Zwicker. *Simple games: desirability relations, trading, and pseudoweightings.* Princeton University Press, New Jersey, USA, 1999.

Appendix

This appendix shows the specific examples which are $(k-1)$-`I-T-R` but non k-`I-T-R` for the highest value of k and for $n = 6, 7$.

Table 4. Non 3-I-T-R simple games for $n = 6$

Vector $\overline{n}$	*Matrix* $\mathcal{M}$	*Matrix* $\mathcal{Y}$	*Vector* $\overline{n}$	*Matrix* $\mathcal{M}$	*Matrix* $\mathcal{Y}$
(1, 1, 1, 1, 1, 1)	$\begin{pmatrix} 1&0&0&1&0&1 \\ 0&1&1&0&0&1 \\ 0&1&0&1&1&0 \end{pmatrix}$	$\begin{pmatrix} 0&0&1&1&1&1 \\ 0&1&0&1&0&1 \\ 1&1&0&0&0&0 \end{pmatrix}$			
(1, 1, 1, 2, 1)	$\begin{pmatrix} 1&0&0&1&1 \\ 0&1&1&0&1 \\ 0&1&0&2&0 \end{pmatrix}$	$\begin{pmatrix} 0&0&1&2&1 \\ 0&1&0&1&1 \\ 1&1&0&0&0 \end{pmatrix}$	(1, 1, 2, 2)	$\begin{pmatrix} 1&0&0&2 \\ 0&1&1&1 \end{pmatrix}$	$\begin{pmatrix} 0&0&2&2 \\ 0&1&0&2 \\ 1&1&0&0 \end{pmatrix}$

Table 5. Non 4-I-T-R simple games for $n = 7$

Vector $\overline{n}$	*Matrix* $\mathcal{M}$	*Matrix* $\mathcal{Y}$	*Matrix* $\mathcal{M}$	*Matrix* $\mathcal{Y}$	*Matrix* $\mathcal{M}$	*Matrix* $\mathcal{Y}$	*Matrix* $\mathcal{M}$	*Matrix* $\mathcal{Y}$
(1, 1, 1, 1, 1, 1, 1)	$\begin{pmatrix} 1010001 \\ 1001010 \\ 0101011 \\ 0011101 \end{pmatrix}$	$\begin{pmatrix} 0011011 \\ 0111000 \\ 1000111 \\ 1001001 \end{pmatrix}$						
(1, 1, 1, 1, 1, 2)	$\begin{pmatrix} 110002 \\ 101011 \\ 011102 \end{pmatrix}$	$\begin{pmatrix} 011012 \\ 100112 \\ 101002 \\ 111000 \end{pmatrix}$						
(1, 1, 1, 2, 1, 1)	$\begin{pmatrix} 100010 \\ 010011 \\ 001101 \end{pmatrix}$	$\begin{pmatrix} 000211 \\ 001011 \\ 011000 \end{pmatrix}$						
(1, 1, 1, 2, 2)	$\begin{pmatrix} 10002 \\ 01102 \\ 01021 \end{pmatrix}$	$\begin{pmatrix} 00122 \\ 01012 \\ 01012 \\ 11000 \end{pmatrix}$						
(1, 1, 2, 1, 2)	$\begin{pmatrix} 10011 \\ 01102 \end{pmatrix}$	$\begin{pmatrix} 00212 \\ 01012 \\ 10002 \\ 11000 \end{pmatrix}$	$\begin{pmatrix} 10002 \\ 01011 \\ 00202 \end{pmatrix}$	$\begin{pmatrix} 00112 \\ 00112 \\ 01002 \\ 11000 \end{pmatrix}$	$\begin{pmatrix} 11010 \\ 10012 \\ 01102 \end{pmatrix}$	$\begin{pmatrix} 00212 \\ 01012 \\ 11001 \\ 11001 \end{pmatrix}$	$\begin{pmatrix} 11010 \\ 10200 \\ 01102 \end{pmatrix}$	$\begin{pmatrix} 00212 \\ 01200 \\ 11001 \\ 11001 \end{pmatrix}$
(1, 1, 2, 2, 1)	$\begin{pmatrix} 10021 \\ 01101 \\ 00220 \end{pmatrix}$	$\begin{pmatrix} 00211 \\ 00211 \\ 01021 \\ 11000 \end{pmatrix}$	$\begin{pmatrix} 10101 \\ 10020 \\ 01021 \end{pmatrix}$	$\begin{pmatrix} 00221 \\ 10011 \\ 10011 \\ 11000 \end{pmatrix}$				
(1, 2, 2, 2)	$\begin{pmatrix} 1010 \\ 0102 \end{pmatrix}$	$\begin{pmatrix} 0022 \\ 0200 \\ 1001 \\ 1001 \end{pmatrix}$	$\begin{pmatrix} 1102 \\ 0221 \end{pmatrix}$	$\begin{pmatrix} 0212 \\ 0212 \\ 1022 \\ 1200 \end{pmatrix}$				
(2, 2, 3)	$\begin{pmatrix} 210 \\ 103 \end{pmatrix}$	$\begin{pmatrix} 023 \\ 201 \\ 201 \\ 201 \end{pmatrix}$						
(2, 3, 2)	$\begin{pmatrix} 102 \\ 031 \end{pmatrix}$	$\begin{pmatrix} 022 \\ 022 \\ 022 \\ 200 \end{pmatrix}$						

Practical Construction of k-Nearest Neighbor Graphs in Metric Spaces*

Rodrigo Paredes[1], Edgar Chávez[2], Karina Figueroa[1,2], and Gonzalo Navarro[1]

[1] Center for Web Research, Dept. of Computer Science, University of Chile
[2] Escuela de Ciencias Físico-Matemáticas, Univ. Michoacana, Mexico
{raparede, kfiguero, gnavarro}@dcc.uchile.cl, elchavez@umich.mx

Abstract. Let $\mathbb{U}$ be a set of elements and d a distance function defined among them. Let $NN_k(u)$ be the k elements in $\mathbb{U}-\{u\}$ having the smallest distance to u. The k-nearest neighbor graph (kNNG) is a weighted directed graph $G(\mathbb{U}, E)$ such that $E = \{(u, v), v \in NN_k(u)\}$. Several kNNG construction algorithms are known, but they are not suitable to general metric spaces. We present a general methodology to construct kNNGs that exploits several features of metric spaces. Experiments suggest that it yields costs of the form $c_1 n^{1.27}$ distance computations for low and medium dimensional spaces, and $c_2 n^{1.90}$ for high dimensional ones.

1 Introduction

Let $\mathbb{U}$ be a set of elements and d a distance function defined among them. Let $NN_k(u)$ be the k elements in $\mathbb{U} - \{u\}$ having the smallest distance to u according to the function d. The *k-nearest neighbor graph* (kNNG) is a weighted directed graph $G(\mathbb{U}, E)$ connecting each element to its k-nearest neighbors, thus $E = \{(u, v), v \in NN_k(u)\}$. Building the kNNG is a direct generalization of the *all-nearest-neighbor* (ANN) problem, so ANN corresponds to the 1NNG construction problem. kNNGs are central in many applications: cluster and outlier detection [14, 4], VLSI design, spin glass and other physical process simulations [6], pattern recognition [12], query or document recommendation systems [3], and others.

There are many kNNG construction algorithms which assume that nodes are points in $\mathbb{R}^D$ and d is the Euclidean or some L_p Minkowski distance. However, this is not the case in several kNNG applications. An example is collaborative filters for Web searching, such as query or document recommendation systems, where kNNGs are used to find clusters of similar queries, to later improve the quality of the results shown to the final user by exploiting cluster properties [3].

To handle this problem one must resort to a more general model called *metric spaces*. A metric space is a pair $(\mathbb{X}, d)$, where $\mathbb{X}$ is the universe of objects and d is a distance function among them that satisfies the triangle inequality.

Another appealing problem in metric spaces is similarity searching [8]. Given a finite metric database $\mathbb{U} \subseteq \mathbb{X}$, the goal is to build an index for $\mathbb{U}$ such that

* Supported in part by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile; and CONACyT, Mexico.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 85–97, 2006.

later, given a query object $q \in \mathbb{X}$, one can find elements of $\mathbb{U}$ close to q using as few distance computations as possible. See [8] for a comprehensive survey.

We have already demonstrated kNNG searching capabilities in general metric spaces [20], where we give kNNG-based search algorithms with practical applicability in low-memory scenarios, or metric spaces of medium or high dimensionality. Hence, in this paper we focus on a metric kNNG construction methodology, and propose two algorithms based on such methodology. According to our experimental results, they have costs of the form $c_1 n^{1.27}$ distance computations for low and medium dimensionality spaces, and $c_2 n^{1.90}$ for high dimensionality ones. Note that a naive construction requires $O(n^2)$ distance evaluations.

1.1 A Summary of Metric Space Searching

Given the universe of objects $\mathbb{X}$, a metric space is a pair $(\mathbb{X}, d)$, where $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$ is any distance function in $\mathbb{X}$ that is symmetric and satisfies the triangle inequality. Some examples are $(\mathbb{R}^D, L_p)$, the space of strings under the edit distance, or the space of documents under the cosine distances.

The metric database is a finite set $\mathbb{U} \subseteq \mathbb{X}, n = |\mathbb{U}|$. A similarity query is an object $q \in \mathbb{X}$, and allows two basic types: the *Range query* (q, r) retrieves all objects $u \in \mathbb{U}$ such that $d(u, q) \leq r$; and the *k-Nearest neighbor query* $NN_k(q)$ retrieves the k objects in $\mathbb{U}$ closest to q according to the distance d. A $NN_k(q)$ algorithm is called *range-optimal* [16] if it uses the same number of distance evaluations as the equivalent range query whose radius retrieves exactly k objects. We call this radius *covering radius*.

An *index* $\mathcal{I}$ is a data structure built over $\mathbb{U}$ using some cells from the whole $\mathbb{U} \times \mathbb{U}$ distance matrix. $\mathcal{I}$ permits solving the above queries without comparing q with each element in $\mathbb{U}$. There are two kinds of indices: pivot based and compact partition based. Search algorithms use $\mathcal{I}$ and some distance evaluations to discard – using the triangle inequality – as many objects as they can, to produce a small candidate set $\mathcal{C}$ that could be relevant to q. Later, they exhaustively check $\mathcal{C}$ by computing distances from q to each candidate to obtain the query result.

As the distance is considered expensive to compute, it is customary to use the number of distance evaluations as the complexity measure both for index construction and object retrieving. For instance, each computation of the cosine distance takes 1.4 msecs in our machine (Pentium IV of 2 GHz). This is really costly even compared with the operations introduced by the graph, such as the shortest path computation using Dijkstra's algorithm.

Many authors agree that the proximity query cost worsens quickly as the *intrinsic dimensionality* of the space grows. This is known as the *curse of dimensionality*. Although there is not and accepted criterion to define the intrinsic dimensionality in a metric space, a general agreement is that spaces with low variance and large mean in their distance histograms have a large intrinsic dimension.

1.2 Related Work on kNNG Construction

The naive approach to construct kNNGs uses $\frac{n(n-1)}{2} = O(n^2)$ distance computations and $O(kn)$ memory. For each $u \in \mathbb{U}$ we compute the distance to all the

others, selecting the k lowest-distance objects. However, there are alternatives to speed up the procedure. The proximity properties of the Voronoi diagram [2] or its dual, the Delaunay triangulation, allow solving the problem more efficiently. The ANN problem can be optimally solved in $O(n \log n)$ time in the plane [13] and in $\mathbb{R}^D$ for any fixed D [9, 22], but the constant depends exponentially on D. In $\mathbb{R}^D$, kNNGs can be built in $O(nk \log n)$ time [22] and even in $O(kn + n \log n)$ time [5, 6, 11]. Approximation algorithms have also been proposed [1]. However, these alternatives, except the naive one, are unsuitable for metric spaces, as they use coordinate information that is not necessarily available in general metric spaces.

Clarkson states the first generalization of ANN to metric spaces [10], where the problem is solved using randomization in $O(n \log^2 n \log^2 \Gamma(\mathbb{U}))$ expected time, where $\Gamma(\mathbb{U})$ is the distance ratio between the farthest and closest pairs of points in $\mathbb{U}$. The author argues that in practice $\Gamma(\mathbb{U}) = n^{O(1)}$, in which case the approach is $O(n \log^4 n)$ time. However, the analysis needs a sphere packing bound in the metric space. Otherwise the cost must be multiplied by "sphere volumes", that are also exponential on the dimensionality. Moreover, the algorithm needs $\Omega(n^2)$ space for high dimensions, which is too much for practical applications.

In [15], another technique for general metric spaces is given. It solves n range queries of decreasing radius by using a pivot-based index. As it is well known, the performance of pivot-based algorithms worsens quickly as the dimension of the space grows, limiting the applicability of this technique. Our pivot based algorithm (Section 2.4) can be seen as an improvement over this technique.

Recently, Karger and Ruhl present the *metric skip list* [18], an index that uses $O(n \log n)$ space and can be constructed with $O(n \log n)$ distance computations. The index answers $NN_1(q)$ queries using $O(\log n)$ distance evaluations with high probability. Later, Krauthgamer and Lee introduce *navigating nets* [19], another index that can be constructed also with $O(n \log n)$ distance computations, yet using $O(n)$ space, and which gives an $(1 + \epsilon)$-approximation algorithm to solve $NN_1(q)$ queries in time $O(\log n) + (1/\epsilon)^{O(1)}$. Both of them could serve to solve the ANN problem with $O(n \log n)$ distance computations but not to build kNNGs. In addition, the hidden constants are exponential on the intrinsic dimension, which makes these approaches useful only in low dimensional metric spaces.

2 Our Methodology

We are interested in practical kNNG construction algorithms for general metric spaces. This problem is equivalent to solve n $NN_k(u)$ queries for all $u \in \mathbb{U}$. Thus, a straightforward solution has two stages: the first is to build some known metric index $\mathcal{I}$ [8], and the second is to use $\mathcal{I}$ to solve the n queries. However, this basic scheme can be improved if we take into account these observations:

- We are solving queries for all the elements in $\mathbb{U}$, not for general objects in $\mathbb{X}$. If we solve the n queries jointly we can share costs through the whole process. For instance, we can avoid some calculations by using the symmetry of d.

– We can upper bound some distances by computing shortest paths over the kNNG under construction, maybe avoiding their actual computation. So, we can use the very kNNG in stepwise refinements to improve the second stage.

2.1 The Ingredients of the Recipe

The main data structure. Along all the algorithm, we use the *Neighbor Heap Array* (*NHA*) to store the kNNG under construction. *NHA* can be regarded as the union of *priority queues* NHA_u, of size k, for all $u \in \mathbb{U}$. At any point in the process NHA_u will contain the k elements closest to u known up to then, and their distances to u. Formally, $NHA_u = \{(x_{i_1}, d(u, x_{i_1})), \ldots, (x_{i_k}, d(u, x_{i_k}))\}$ sorted by decreasing $d(u, x_{i_j})$ (i_j is the j-th neighbor identifier).

For each $u \in \mathbb{U}$, we initialize $NHA_u = \{(\perp, \infty), \ldots, (\perp, \infty)\}$, $|NHA_u| = k$. Let $curCR_u = d(u, x_{i_1})$ be the current covering radius of u, that is, the distance from u towards its current farthest neighbor candidate in NHA_u.

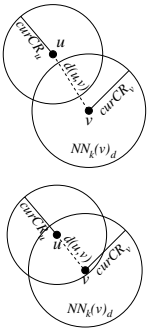
In the first stage, every distance computed to build the index $\mathcal{I}$ populates *NHA*. In the second, we refine *NHA* with the following distance computations. We must ensure that $|NHA_u| = k$ upon successive additions. Hence, if we find some object v such that $d(u, v) < curCR_u$, before adding $(v, d(u, v))$ to NHA_u we extract the farthest candidate from NHA_u. This progressively reduces $curCR_u$ from ∞ to the real covering radius. At the end, *NHA* stores the kNNG of $\mathbb{U}$.

Using *NHA* as a graph. Once we calculate $d_{uv} = d(u, v)$, if $d_{uv} \geq curCR_u$ we discard v as a candidate for NHA_u. Also, due to the triangle inequality we can discard all objects w such that $d(v, w) \leq d_{uv} - curCR_u$. Unfortunately, we do not necessarily have stored $d(v, w)$. However, we can upper bound $d(v, w)$ with the sum of edge weights traversed in the shortest paths over *NHA* from v to all $w \in \mathbb{U}$, $d_{NHA}(v, w)$. So, if $d_{uv} \geq curCR_u$, we also discard all objects w such that $d_{NHA}(v, w) \leq d_{uv} - curCR_u$.

***d* is symmetric.** Every time a distance $d_{uv} = d(u, v)$ is computed, we check both $d_{uv} < curCR_u$ for adding (v, d_{uv}) to NHA_u, and $d_{uv} < curCR_v$ for adding (u, d_{uv}) to NHA_v. This can both reduce $curCR_v$, and cheapen the future query for v, even when we are solving neighbors for another object.

$\mathbb{U}$ is fixed. Assume we are solving query $NN_k(u)$, we have to check some already solved object v, and $curCR_u \leq curCR_v$. Then, if $u \notin NN_k(v) \Rightarrow d(u, v) \geq curCR_v$, so $v \notin NN_k(u)$. Otherwise, if $u \in NN_k(v)$, then we already computed $d(u, v)$. Then, in those cases we avoid to compute $d(u, v)$. Fig. 1(a) illustrates.

Check Order Heap (*COH*). We create the priority queue $COH = \{(u, curCR_u), u \in \mathbb{U}\}$ to complete $NN_k(u)$ queries in increasing $curCR_u$ order, because a small radius query has larger discriminative power and produces candidates that are closer to the query u. This reduces the CPU time and – as d is symmetric – could increase the chance of improving candidate sets in *NHA* for other objects v. This, in turn, could reduce $curCR_v$ and change the position of v in COH.



(a) $\mathbb{U}$ is fixed.

K**NN** (Integer k, ObjectSet $\mathbb{U}$)
Stage 1: Initialize *NHA* and construct the index $\mathcal{I}$
1. **For each** $u \in \mathbb{U}$ **Do** $NHA_u \leftarrow \{(\perp, \infty), \ldots, (\perp, \infty)\}$ // k pairs
2. Create $\mathcal{I}$, all computed distances populate symmetrically *NHA*

Stage 2: Complete the $NN_k(u)$ for all $u \in \mathbb{U}$
3. $COH \leftarrow \{(u, curCR_u), u \in \mathbb{U}\}$
4. **For each** $(u, curCR_u) \in COH$, in increasing $curCR_u$ order **Do**
5. Create the candidate set $\mathcal{C}$ according to $\mathcal{I}$ // exclude NHA_u
6. **While** $\mathcal{C} \neq \emptyset$ **Do**
7. $c \leftarrow$ extract a candidate from $\mathcal{C}$
8. **If** "$\mathbb{U}$ is fixed" does not apply for u and c **Then**
9. $d_{uc} \leftarrow d(u, c)$, try to insert c into NHA_u
10. try to insert u into NHA_c, update c in COH (symmetry)
11. use *NHA* as a graph and $\mathcal{I}$ to discard objects from $\mathcal{C}$
12. **Return** *NHA* as a graph

(b) Sketch of the methodology.

Fig. 1. In 1(a), assume we are solving u, v is already solved, and $curCR_u \leq curCR_v$. On the top, if $u \notin NN_k(v) \Rightarrow d(u, v) \geq curCR_v \geq curCR_u$. On the bottom, if $u \in NN_k(v)$, we already computed $d(u, v)$. Then, in those cases we avoid computing $d(u, v)$. In Fig. 1(b), we sketch the methodology.

The recipe. We split the process into two stages. The first is to build $\mathcal{I}$ to preindex the objects. The second is to use $\mathcal{I}$ and all the ingredients to solve the $NN_k(u)$ queries for all $u \in \mathbb{U}$. Fig. 1(b) depicts the methodology.

For practical reasons, we allow that our algorithms use at most $O(n(\log n + k))$ memory both to index $\mathbb{U}$ and to store the kNNG under construction.

2.2 The Resulting Algorithms

Based on our methodology, we propose two kNNG construction algorithms focused on decreasing the total number of distance computations. They are:

1. Recursive partition based algorithm: In the first stage, we build a preindex by performing a recursive partitioning of the space. In the second stage, we complete the $NN_k(u)$ queries using the order induced by the partitioning.
2. Pivot based algorithm: In the preindexing stage, we build the pivot index. Later, we complete the $NN_k(u)$ queries by performing range-optimal queries.

The experiments confirm that these algorithms are efficient. For instance, in the string space, the pivot-based algorithm requires CPU time of the empirical form $c_t n^{1.85}$, and $c_d n^{1.26}$ in distance computations. In the high-dimensional document space, the recursive partition-based algorithm requires empirically $cn^{1.955}$ both in distance computations and CPU time.

2.3 Recursive Partition Based Algorithm

This algorithm is based on using a preindex slightly different to the *Bisector Tree* (*BST*) [17]. We call our modified *BST* the *Division Control Tree* (*DCT*), which is a binary tree representing the shape of the partitioning. The *DCT* node structure is $\{p, l, r, pr\}$, which represents the parent, left and right children, and partition radius of the node, respectively. The partition radius is the distance from the node towards the farthest node of its partition. (With respect to the *BST* structure, we have added the pointer p to easily navigate trough the tree.)

For simplicity we use the same name for the node and for its representative in the *DCT*. Then, given a node $u \in \mathbb{U}$, u_p, u_l, and u_r, refer to nodes that are the parent, left child, and right child of u in the *DCT*, respectively, and also to their representative nodes in $\mathbb{U}$. Finally, u_{pr} refers to the partition radius of u.

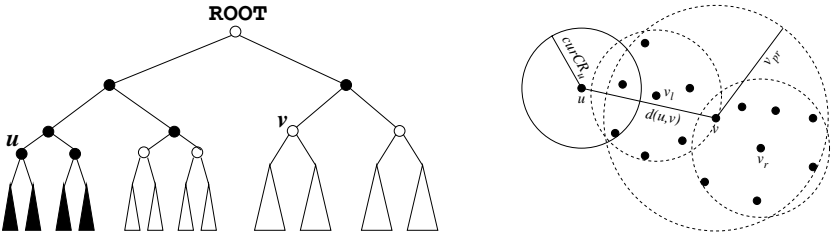
In this algorithm, we use $O(kn)$ space to store the *NHA* and $O(n)$ to store the *DCT*. The remaining memory is used as a cache of computed distances, *CD*, whose size is limited to $O(n \log n)$. Thus, every time we need to compute a distance, we check if it is present in *CD*, in which case we return the stored value. Note that the $CD \subset \mathbb{U}^2 \times \mathbb{R}^+$ can also be seen as graph of all stored distances. The criterion to insert distances into *CD* depends on the stage (see later). Once we complete the $NN_k(u)$, we remove its adjacency list from *CD*.

First stage: construction of *DCT*. We partition the space recursively to construct the *DCT*, and populate symmetrically *NHA* and *CD* with all the computed distances. The *DCT* is built as follows. Given the node *root* and the set S, we choose children objects l and r from S. Then, we generate two subsets: S_l, objects nearest to l, and S_r, objects nearest to r. Finally, we compute both partition radii. The recursion follows with (l, S_l) and (r, S_r), finishing when $|S| < 2$. Once we finish the division, leaves in the *DCT* have partition radii 0. The *DCT* root is fictitious, having no equivalent in $\mathbb{U}$, and partition radius ∞.

Since the *DCT* has n nodes, its expected height is $2 \ln n$ (the *DCT* construction is statistically identical to populating a binary search tree). For each *DCT* level, each node computes two distances towards the splitting nodes, which accounts for $2n$ distances per level. So, we expect to compute $4n \ln n$ distances in the partitioning. As we store 2 edges per distance, we need to store $8n \ln n$ in *CD*. Hence, we fix the maximum space of *CD* as $8n \ln n = O(n \log n)$.

Solving $NN_k(u)$ queries with *DCT*. The construction of *DCT* ensures that every node has already computed distances to all of its ancestors, its ancestor's siblings, and its parent descent. Then, to finish the $NN_k(u)$ query, it is enough to check whether there are relevant objects in all the descendants of u's ancestors' siblings. This corresponds to white nodes and subtrees in Fig. 2(a).

Nevertheless, the *DCT* allows us to avoid some work. Assume we are checking whether v is relevant to u, and the balls $(u, curCR_u)$ and (v, v_{pr}) do not intersect each other, then we discard v and its partition. Otherwise, we recursively check children v_l and v_r. Fig. 2(b) illustrates this. Hence, in the candidate set $\mathcal{C}$, it suffices to manage the set of ancestors' siblings, and if it is not possible to discard the whole sibling's partition we add its children into $\mathcal{C}$. Since it is more likely to



(a) Induced check sequence. (b) DCT can discard partitions.

Fig. 2. Using the DCT to solve $NN_k(q)$ queries. In 2(a), u has been compared with all black nodes and all the descent of its parent. To finish the query, we just process white nodes and subtrees. In 2(b), as $d(u,v) \leq curCR_u + v_{pr}$ the partition of v intersects the ball $(u, curCR_u)$, so we recursively check children v_l and v_r. As v_r's partition does not intersect the ball $(u, curCR_u)$, we discard v_r and its partition. However, we continue the checking on v_l's partition as it intersects the ball $(u, curCR_u)$.

discard small partitions, we process $\mathcal{C}$ in order of increasing radius. This agrees with the intuition that the partition radius of u's parent's sibling is likely the smallest of $\mathcal{C}$, and that some of its descendants could be relevant to u.

Second stage: Completing the queries. As CD can be seen as a graph, we use $NHA \cup CD$ to upper bound distances: when $d(u,v) \geq curCR_u$, we discard objects w such that their shortest path $d_{NHA \cup CD}(v,w) \leq d(u,v) - curCR_u$. We do this by adding them to $\mathcal{C}$ marked as `EXTRACTED`.

In this stage, if we have available space in CD, we cache all the computed distances small enough so as to get into their respective queues in NHA, since these distances can be used in future symmetric queries. Note that adding distances to CD without considering the space limitation could increase its size beyond control, as it is shown by the following average case analysis. With probability $\frac{n-k}{n}$, a random distance is greater than the k-th shortest one (thus, not stored), and with probability $\frac{k}{n}$ it is lower, then it is stored in CD using one cell. The base case uses k cells for the first distances. Then, the recurrence for the average case of edge insertions for each NHA_u is: $T(n,k) = T(n-1,k) + \frac{k}{n}$, $T(k,k) = k$. We obtain $T(n,k) = k(H_n - H_k + 1) = O(k \log \frac{n}{k})$. As we have n priority queues, if we do not consider the limitation, we could use $O(nk \log \frac{n}{k})$ memory cells, which can be an unpractical memory requirement.

Finally, we combine all of these ideas to complete the $NN_k(u)$ queries for all nodes in $\mathbb{U}$. We begin by creating the priority queue COH. Then, for each node u picked from COH we do the following. We add the edges of NHA_u to CD_u, where CD_u refers to the adjacency list of u in CD. (Due to the size limitation it is likely that some of the u's current neighbors do not belong CD_u.) Then, we compute shortest paths from all u's ancestors discarding as many objects as we can. Then, we finish the query $NN_k(u)$, and finally delete CD_u.

To finish the query $NN_k(u)$, we start adding all u's ancestors to $\mathcal{C}$. Later, we take objects c from $\mathcal{C}$ in increasing c_{pr} order, and process c according one of the following rules:

1. If c was already marked as `EXTRACTED`, we add its children $\{c_l, c_r\}$ to $\mathcal{C}$;
2. If "$\mathbb{U}$ is fixed" applies for c and u, and $d(u,c) \notin CD$, we add $\{c_l, c_r\}$ to $\mathcal{C}$; or
3. If we have $d(u,c)$ stored in CD, we retrieve it, else we compute it and use "d is symmetric". Then, if $d(u,c) < curCR_u + c_{pr}$, we have region intersection, so we add $\{c_l, c_r\}$ to $\mathcal{C}$. Next, we use $NHA \cup CD$ as a graph computing shortest paths from c to discard as many object as we can.

2.4 Pivot-Based Algorithm

Pivot-based algorithms have good performance in low dimensional spaces, but worsen quickly as the dimension grows. However, our methodology compensates this failure in medium and high dimensions. In this algorithm we use $O(kn)$ space in NHA and $O(n \log n)$ space to store the pivot index.

First stage: construction of the pivot index. We select at random a set of pivots $\mathcal{P} = \{p_1, \ldots, p_{|\mathcal{P}|}\} \subseteq \mathbb{U}$, and store a table of $|\mathcal{P}|n$ distances $d(p_j, u),\ j \in \{1, \ldots, |\mathcal{P}|\},\ u \in \mathbb{U}$. We give the same space in bytes to the table as that of the cache of distances and the division control tree of the recursive based algorithm. Then, in our implementation we use $|\mathcal{P}| = 12 \ln n + 2.5 = O(\log n)$.

Solving $NN_k(u)$ queries with the pivot table. To perform a range-optimal query for u we use $\mathcal{C}$ as an array to store maximum lower bounds of distances from u to other objects. Because of the triangle inequality, for each $v \in \mathbb{U}$ and $p \in \mathcal{P}$, $|d(v,p) - d(u,p)|$ is a lower bound of $d(u,v)$. Let $\mathcal{C}_v = \max_{p \in \mathcal{P}}\{|d(v,p) - d(u,p)|\}$. So, we can discard non-relevant objects v such that $\mathcal{C}_v \geq curCR_u$.

Then, we store $\mathcal{C}$ values in a priority queue $Sort\mathcal{C} = \{(v, \mathcal{C}_v), v \in \mathbb{U} - (\mathcal{P} \cup NHA_u \cup \{u\})\}$. For each object v picked from $Sort\mathcal{C}$ by ascending $\mathcal{C}_v$, we check if $\mathcal{C}_v < curCR_u$. In such case, when "$\mathbb{U}$ is fixed" applies for u and v we avoid the distance computation and process the next node, else we compute the distance $d_{uv} = d(u,v)$. So, if $d_{uv} < curCR_u$ we add v to NHA_u (this could reduce $curCR_u$). Also, using " d is symmetric" we can refine NHA_v and consequently update v in COH. Finally, we use NHA as a graph computing shortest path from v to extract from $Sort\mathcal{C}$ as many object as we can. Each $NN_k(u)$ query finishes when we reach a node v such that $\mathcal{C}_v \geq curCR_u$, or $Sort\mathcal{C}$ gets empty.

Second stage: Completing the queries. Since pivots $p \in \mathcal{P}$ compute distances towards all objects, once we compute the table, they have already solved their k-nearest neighbors. So, we only have to complete $n - |\mathcal{P}|$ range-optimal queries for objects $u \in \mathbb{U} - \mathcal{P}$. Notice that because of the symmetry of d, these objects already have candidates in their respective queues in NHA.

3 Experimental Results

We have tested our algorithms on synthetic and real-world metric spaces. The first synthetic set is formed by 65,536 points uniformly distributed in the metric space $([0,1]^D, L_2)$ (the unitary real D-dimensional cube with Euclidean distance). This

Table 1. KNNrp and KNNpiv least square fittings for distance evaluations and CPU time for all the metric spaces. CPU time measured in microseconds.

Space	KNNrp Dist. evals.	KNNrp CPU time	KNNpiv Dist. evals.	KNNpiv CPU time
$[0,1]^4$	$10.0n^{1.32}$	$0.311n^{2.24}$	$56.1n^{1.09}$	$0.787n^{2.01}$
$[0,1]^8$	$32.8n^{1.38}$	$0.642n^{2.11}$	$168n^{1.06}$	$15.5n^{1.69}$
$[0,1]^{12}$	$15.1n^{1.59}$	$1.71n^{2.03}$	$116n^{1.27}$	$20.1n^{1.79}$
$[0,1]^{16}$	$5.06n^{1.77}$	$0.732n^{2.14}$	$12.1n^{1.64}$	$6.87n^{1.97}$
$[0,1]^{20}$	$2.32n^{1.88}$	$0.546n^{2.18}$	$2.48n^{1.87}$	$2.77n^{2.10}$
$[0,1]^{24}$	$1.34n^{1.96}$	$0.656n^{2.16}$	$1.23n^{1.96}$	$1.29n^{2.16}$
$[0,1]^D$	$0.455e^{0.19D}n^{1.65}$	$0.571e^{0.01D}n^{2.14}$	$0.685e^{0.23D}n^{1.48}$	$0.858e^{0.11D}n^{2.15}$
Gaussian $\sigma = 0.1$	$74.7n^{1.33}$	$1.13n^{2.07}$	$1260n^{0.91}$	$63.5n^{1.63}$
Gaussian $\sigma = 0.2$	$7.82n^{1.71}$	$1.13n^{2.09}$	$16.3n^{1.60}$	$8.70n^{1.94}$
Gaussian $\sigma = 0.3$	$2.97n^{1.85}$	$0.620n^{2.17}$	$3.86n^{1.81}$	$3.78n^{2.06}$
String	$21.4n^{1.54}$	$1.09n^{2.09}$	$99.9n^{1.26}$	$10.8n^{1.85}$
Document	$0.425n^{1.95}$	$193n^{1.96}$	$0.840n^{1.86}$	$364n^{1.87}$

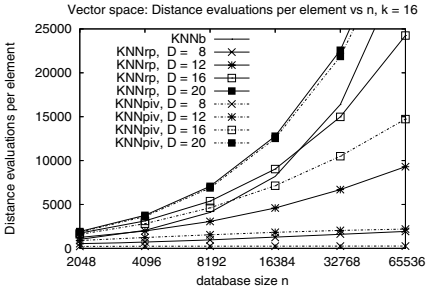
space allows us to measure the effect of the space dimension D on our algorithms. The second set is formed by 65,536 points in a 20-dimensional space with Gaussian distribution forming 256 clusters randomly placed in $([0,1]^{20}, L_2)$. We consider three standard deviations to make more crisp or more fuzzy clusters ($\sigma = 0.1, 0.2, 0.3$). Of course, we have not used the fact that vectors have coordinates, but have treated them as abstract objects.

The first real-world set is the string metric space under the edit distance, a discrete function that measures the minimum number of character insertions, deletions and replacements needed to make the strings equal. We index a random subset of 65,536 words taken from an English dictionary. The second set is the document space under the cosine distance, a function that measures the angle between two documents when they are represented as vectors in a high-dimensional vector model. We index a random subset of 1,215 English documents taken from the TREC-3 collection.

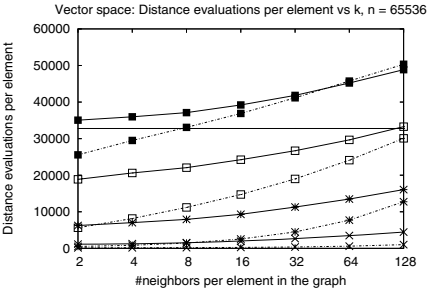
Experiments were run on an Intel Pentium IV of 2 GHz and 512 MB of RAM. We measure distance evaluations and CPU time. For shortness we have called the basic kNNG construction algorithm KNNb, the recursive partition based algorithm KNNrp, and the pivot based algorithm KNNpiv. We are not aware of any published kNNG practical implementation for general metric spaces.

We summarize our experimental results in Fig. 3, where we show distance computations per element, and Table 1 for the least square fittings computed with R [21]. The dependence on k turns out to be so mild that we neglect k in the fittings, thus, costs have the form cn^{α}. Even though in Table 1 we explicit the constant c, from now on, we only refer to the exponent α.

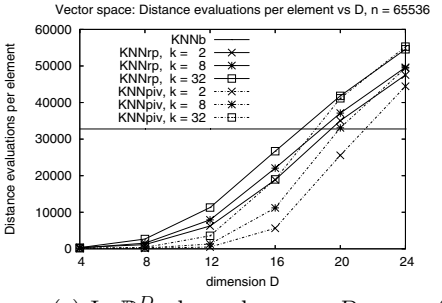
Figs. 3(a), 3(b) and 3(c) show experimental results for $\mathbb{R}^D$. Fig. 3(c) shows that, as D grows, the performance of our algorithms degrade, phenomenon known as the *curse of dimensionality*. For instance, for $D = 4$, KNNpiv uses $cn^{1.10}$ distance evaluations, but for $D = 24$, it is $cn^{1.96}$ distance evaluations. Notice that a metric space with dimensionality $D > 20$ is considered as intractable [8]. Fig. 3(a) shows that for all dimensions our algorithms are subquadratic in distance evaluations, instead of KNNb which is always cn^2. For low and medium



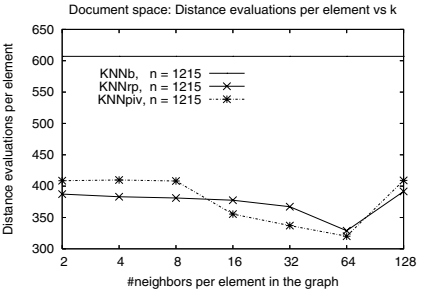
(a) In $\mathbb{R}^D$, dependence on n.



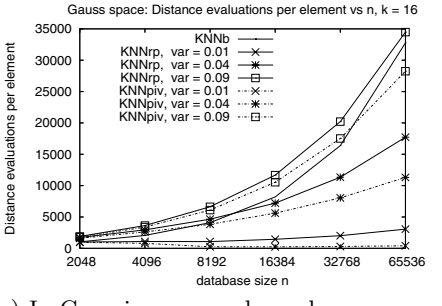
(b) In $\mathbb{R}^D$, dependence on k.



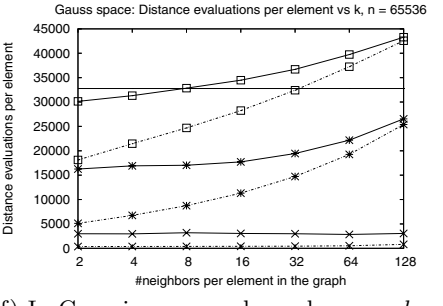
(c) In $\mathbb{R}^D$, dependence on D.



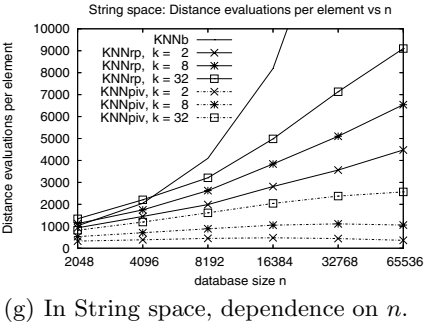
(d) In Document space, dependence on k.



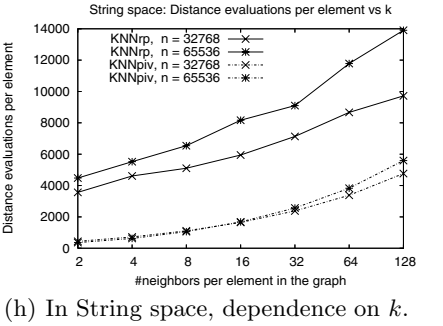
(e) In Gaussian space, dependence on n.



(f) In Gaussian space, dependence on k.



(g) In String space, dependence on n.



(h) In String space, dependence on k.

Fig. 3. Distance evaluations per node during kNNG construction. Fig. 3(b)/3(f) follows the legend of Fig. 3(a)/3(e).

dimensions ($D \leq 16$) ours have better performance than KNNb, being KNNpiv the best of ours. Moreover, for lower dimensions ($D \leq 8$) ours are only slightly superlinear. Fig. 3(b) shows a sublinear dependence on k for all dimensions, however, KNNpiv is more sensitive to k than KNNrp. Also, the dependence on k diminishes as long as D grows, although it is always monotonically increasing on k. Finally, it is shown that for $k \leq 4$, our algorithms behave better than KNNb, even in high dimensional spaces (KNNpiv in $D = 20$).

Figs. 3(e) and 3(f) show results in Gaussian space. For crisp clusters ($\sigma = 0.1$) the performance of our algorithms improves significantly, even for high values of k. It is interesting to note that for $k \leq 8$ our algorithms are more efficient than KNNb for the three variances. Again, KNNpiv has the best performance.

Figs. 3(g) and 3(h) show results for strings. The plots show that both KNNrp and KNNpiv are subquadratic for all $k \in [2, 128]$. For instance, for $n = 65,536$, KNNrp costs 28%, and KNNpiv just 8%, of KNNb to build the 32NNG.

Finally, Fig. 3(d) shows that our methodology save lots of work in the high-dimensional document space. For instance, for $n = 1,215$, KNNrp costs 63%, and KNNpiv costs 67%, of KNNb to build the 8NNG. These two last results show that our methodology is also practical in real-world situations.

All of these conclusions are confirmed in Table 1. We remark that in some practical conditions (vectors in $[0, 1]^D$ with $D \leq 8$ and $k \leq 32$ and Gaussian vectors with $\sigma = 0.01$ and $k \leq 8$), KNNpiv also has better performance than KNNb in CPU time. This is important since the Euclidean distance is very cheap to compute. Note that KNNrp and KNNpiv turn out to be clearly subquadratic on distances evaluations when considering the exponential dependence on D.

Note that in the metric space context, superquadratic CPU time in side computations is not as important as a subquadratic number of computed distances. In fact, in the document space, KNNrp and KNNpiv perform better in CPU time that KNNb, showing that in practice the leading complexity (computing distances) is several orders of magnitude larger than other side computations such as traversing pointers or scanning the pivot table.

4 Conclusions

We have presented a general methodology to construct the k-nearest neighbor graph (kNNG) in general metric spaces. Based on our methodology we give two algorithms. The first is based on a recursive partitioning of the space (KNNrp), and the second on the classic pivot technique (KNNpiv). Our methodology considers two stages: the first indexes the space, and the second completes the kNNG using the index and some metric and graph optimizations.

Experimental results confirm the practical efficiency of our approach in vectorial metric spaces of wide dimensional spectrum ($D \leq 20$), and real-world metric spaces. For instance, in the string space, our algorithms achieve empirical CPU time of the form $c_t n^{1.85}$, and $c_d n^{1.26}$ in distance computations; and in the high-dimensional document space, they reach empirical $cn^{1.87}$ both in distance computations and CPU time. In low dimensional metric spaces, our algorithms behave even better. KNNpiv is in general better than KNNrp for

small and moderate k values, yet KNNrp is less sensitive to larger k values or higher dimensional spaces.

Future work involves developing another kNNG constructing algorithm based on the list of clusters [7] so that we can also obtain good performance in higher dimensional metric spaces. We are also researching how to enhance the data structure to allow dynamic insertions/deletions in reasonable time, so as to maintain an up-to-date set of k-nearest neighbors for each element in the database.

Acknowledgement. We wish to thank Georges Dupret and Marco Patella for their valuable comments.

References

1. S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimension. In *Proc. SODA'94*, pages 573–583, 1994.
2. F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3), 1991.
3. R. Baeza-Yates, C. Hurtado, and M. Mendoza. Query clustering for boosting web page ranking. In *Proc. AWIC'04*, LNCS 3034, pages 164–175, 2004.
4. M. Brito, E. Chávez, A. Quiroz, and J. Yukich. Connectivity of the mutual k-nearest neighbor graph in clustering and outlier detection. *Statistics & Probability Letters*, 35:33–42, 1996.
5. P. Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *Proc. FOCS'93*, pages 332–340, 1993.
6. P. Callahan and R. Kosaraju. A decomposition of multidimensional point sets with applications to k nearest neighbors and n body potential fields. *JACM*, 42(1):67–90, 1995.
7. E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.
8. E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.
9. K. Clarkson. Fast algorithms for the all-nearest-neighbors problem. In *Proc. FOCS'83*, pages 226–232, 1983.
10. K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.
11. M. Dickerson and D. Eppstein. Algorithms for proximity problems in higher dimensions. *Computational Geometry Theory and Applications*, 5:277–291, 1996.
12. R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.
13. H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.
14. D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete & Computational Geometry*, 11:321–350, 1994.
15. K. Figueroa. An efficient algorithm to all k nearest neighbor problem in metric spaces. Master's thesis, Universidad Michoacana, Mexico, 2000. In Spanish.
16. G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report TR 4199, Dept. of Comp. Sci. Univ. of Maryland, Nov 2000.
17. I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Trans. Software Eng.*, 9(5):631–634, 1983.

18. D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proc. STOC'02*, pages 741–750, 2002.
19. R. Krauthgamer and J. Lee. Navigating nets: simple algorithms for proximity search. In *Proc. SODA'04*, pages 798–807, 2004.
20. R. Paredes and E. Chávez. Using the k-nearest neighbor graph for proximity searching in metric spaces. In *Proc. SPIRE'05*, LNCS 3772, pages 127–138, 2005.
21. R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004.
22. P. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbor problem. *Discrete & Computational Geometry*, 4:101–115, 1989.

Fast and Simple Approximation of the Diameter and Radius of a Graph*

Krists Boitmanis, Kārlis Freivalds, Pēteris Lediņš, and Rūdolfs Opmanis

Institute of Mathematics and ComputerScience,
University of Latvia, Rainis blvd. – 29, Riga, Latvia
Krists.Boitmanis@mii.lu.lv

Abstract. The increasing amount of data to be processed by computers has led to the need for highly efficient algorithms for various computational problems. Moreover, the algorithms should be as simple as possible to be practically applicable. In this paper we propose a very simple approximation algorithm for finding the diameter and the radius of an undirected graph. The algorithm runs in $O(m\sqrt{n})$ time and gives an additive error of $O(\sqrt{n})$ for a graph with n vertices and m edges. Practical experiments show that the results of our algorithm are close to the optimum and compare favorably to the 2/3-approximation algorithm for the diameter problem by Aingworth et al [1].

Keywords: algorithm engineering, analysis of algorithms, approximation techniques, graph algorithms, graph diameter.

1 Introduction

Nowadays there are huge amounts of data to be processed by computers, hence highly efficient algorithms for various combinatorial problems are necessary. In many cases the speed of an algorithm is more important than the optimal solution to a problem instance. Also, for an algorithm to be widely accessible, it has to be as simple as possible.

One of such fundamental problems is the computation of the diameter of an undirected graph, which is the subject of this paper. Recall, that the diameter of a graph is the largest distance between any pair of its vertices. A straightforward algorithm leads to the running time $O(nm)$ for a graph with n vertices and m edges, which is prohibitively slow for very large graphs. It actually finds the distance between every pair of vertices and returns the largest distance found. Although this bound on the running time can be improved considerably for dense graphs [18, 6], computing the whole distance matrix still takes $\Omega(n^2)$ time.

Since we are interested in the largest distance only, we could hope that the diameter of a graph can be determined faster than the whole distance matrix. Unfortunately such algorithms have been devised for special classes of graphs only, e.g. for trees [14], maximal outerplanar graphs [13], interval graphs [12, 15],

* This work is partially supported by ESF (European Social Fund).

C. Álvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 98–108, 2006.

ptolemaic graphs [12], strongly chordal graphs, dually chordal graphs [3], and distance-hereditary graphs [10, 11]. So the question of whether computing the diameter of an arbitrary graph is an easier problem than computing the whole distance matrix remains open [5].

Thus, while waiting for a major breakthrough we have to confine ourselves to approximation algorithms. The diameter can be trivially approximated within a factor of 1/2 by simply performing a breadth first search from an arbitrary vertex. It is well known that the depth of such a breadth first search tree is at least half of the diameter. A non-trivial approximation algorithm is due by Aingworth et al [1], which obtains a 2/3-approximation of the diameter. Their algorithm runs in $O(m\sqrt{n \log n} + n^2 \log n)$ time.

In this paper we propose an extremely simple method for approximating the diameter of an undirected graph by utilizing several breadth first searches. The algorithm gives an additive error of $O(n/k)$, if terminated after k breadth first searches, hence an $O(\sqrt{n})$ upper bound on the error can be obtained in $O(m\sqrt{n})$ time. As an extra asset, the radius of the graph can be approximated in very much the same way.

This paper is organized as follows. In the next section we give some background information and formal definitions of the notions involved. In Sect. 3 we explain our algorithm and analyze its performance. Computational experiments described in Sect. 4 make it evident that our algorithm performs very well in practice. It is compared with the algorithm of [1] and some of the linear time algorithms, which have been shown to work almost optimally on various special classes of graphs [8]. Finally, Sect. 5 concludes the paper and discusses possible directions for the future research.

2 Background

In this section we remind to the reader some definitions of graph theory and briefly discuss the breadth first search procedure. We will consider undirected graphs only.

Definition 1. *A* graph *G is a pair of sets $G = (V, E)$, where the elements of E, called* edges*, are unordered pairs of elements from V, called* vertices*.*

Definition 2. *A sequence $(v_0, v_1, \ldots, v_k)$ of distinct vertices of a graph $G = (V, E)$ is called a* path *between v_0 and v_k if $\{v_i, v_{i+1}\} \in E$ whenever $0 \leq i < k$. The* length *of the path is k, which is the number of edges in the path.*

Definition 3. *A graph $G = (V, E)$ is said to be* connected *if there is a path between every pair of vertices $v, u \in V$.*

In the rest of this paper we deal with connected graphs only.

Definition 4. *The* distance *$d(v, u)$ between a pair of vertices v and u is the length of the shortest path between these vertices. More generally, the distance $d(v, S)$ from a vertex v to a subset $S \subseteq V$ of vertices is defined as $d(v, S) = \min_{u \in S} d(v, u)$.*

Definition 5. *The* eccentricity $e(v)$ *of a vertex* v *is the maximum distance from* v *to any other vertex, i.e.* $e(v) = \max_{u \in V} d(v, u)$.

Definition 6. *The maximum and minimum eccentricities among all vertices of a graph* $G = (V, E)$ *are called the* diameter *and the* radius *of the graph, respectively, i.e.*

$$\operatorname{diam} G = \max_{v \in V} e(v) \ ,$$

and

$$\operatorname{rad} G = \min_{v \in V} e(v) \ .$$

2.1 Breadth First Search

We use the breadth first search (BFS) algorithm [7] to determine the eccentricity of a specific vertex u and the distance from u to every other vertex in a graph $G = (V, E)$. Also, the set V of vertices can be easily partitioned into *layers* with the i-th layer defined as $L_i = \{v \in V | d(u, v) = i\}$, $0 \leq i \leq e(u)$.

3 Algorithm

In this section we assume that a connected graph $G = (V, E)$ with n vertices and m edges is given.

3.1 Estimating the Diameter

The idea of our algorithm is fairly simple. We compute the eccentricities of several vertices and take the maximum eccentricity as an estimate for the diameter of the given graph $G = (V, E)$. For this purpose we utilize multiple breadth first searches, hence the name of the Algorithm 1. The next vertex to start the search from is chosen to be the furthest vertex from the already processed vertices. The algorithm terminates after processing k vertices $\{v_1, v_2, \ldots, v_k\}$, where the constant k will be determined later.

Let us consider the sets of processed vertices $U_i = \{v_1, v_2, \ldots, v_i\}$ after each iteration $i = 1, 2, \ldots, k$, and let $L = \max_{v \in V} d(v, U_k)$.

Theorem 1. *The error of the algorithm MultiBFS does not exceed* L*, i.e.* $\operatorname{diam} G - \bar{D} \leq L$.

Proof. Let $x, y \in V$ be a pair of vertices such that $d(x, y) = \operatorname{diam} G$ and let $v \in U_k$ such that $d(x, v) = d(x, U_k)$. We have $\operatorname{diam} G = d(x, y) \leq d(x, v) + d(v, y) \leq L + \bar{D}$, and hence $\operatorname{diam} G - \bar{D} \leq L$. □

Lemma 1. *For each vertex* $v \notin U_k$ *and for every pair of processed vertices* v_i, v_j *with* $1 \leq i < j \leq k$ *we have the inequality* $d(v, U_k) \leq d(v_i, v_j)$.

Algorithm 1. MultiBFS

1: $U \leftarrow \emptyset$ {the set of processed vertices}
2: **for** $i \leftarrow 1$ to k **do**
3: **if** $i = 1$ **then**
4: Let $v_i \in V$ be an arbitrary vertex
5: **else**
6: Let $v_i \in V$ be a vertex with maximal $d(v_i, U)$
7: $U \leftarrow U + v_i$
8: Perform a breadth first search from v_i and find its eccentricity $e(v_i)$
9: return $\bar{D} = \max_{v \in U} e(v)$ as an approximation for the diameter of G

Proof. Since $d(v, U_k) \leq d(v, U_{j-1})$ and the vertex v_j was favored over the vertex v in the j-th iteration of the algorithm, we have $d(v, U_k) \leq d(v_j, U_{j-1})$. The obvious inequality $d(v_j, U_{j-1}) \leq d(v_i, v_j)$ completes the proof. □

Theorem 2

$$L \leq \frac{2(n-1)}{k+1}$$

Proof. Let $v \in V - U_k$ be an arbitrary vertex, which is not processed by the algorithm, and let $l = d(v, U_k)$.

Consider a BFS tree T rooted at the vertex v. Construct a rooted tree T_k from the tree T by repeatedly removing the leaf vertices, which do not belong to U_k. In effect, all leaves of T_k belong to U_k.

Assuming that $k \geq 2$, we construct another rooted tree T_{k-1} from T_k as follows. Let u be a leaf vertex of T_k with the maximum depth, and let p be the deepest proper ancestor of u, such that $p \in U_k$ or p has at least two children in T_k. The tree T_{k-1} is obtained by removing the vertex u and all the inner vertices of the p–u path in the tree T_k. Note, that at least $l/2$ edges have been removed in this way. Indeed, let us assume that $d(u, p) < l/2$ on the contrary. Consider a vertex $w \in U_k - u$, which is a descendant of p in T_k (note, that w may very well be p itself). By the choice of u we have $d(w, p) \leq d(u, p)$. Consequently, $d(u, w) \leq d(u, p) + d(w, p) < l$, contradicting to Lemma 1.

A rooted tree T_{k-2} can be obtained from T_{k-1} in the same fashion. We arrive at the rooted tree T_1 after $k-1$ such operations. Now, T_1 has exactly one vertex u in U_k and l edges. Adding up the number of edges in T_1 and the removed edges, we conclude that the initial tree T has at least $l + (k-1)l/2 = (k+1)l/2$ edges. On the other hand, a tree cannot have more than $n-1$ edges, which leads to the inequality $(k+1)l/2 \leq n-1$. The desired result now follows. □

The bound of Theorem 2 is tight in the sense that there are graphs with $L = \frac{2(n-1)}{k+1}$. For example, take a set S of $k+1$ vertices and connect them with some central vertex by disjoint paths of equal length l. For this star-like graph $n = (k+1)l + 1$, and, if the algorithm chooses the first vertex in S, it processes k vertices of S and we still have $L = 2l$.

Corollary 1. *The error of the algorithm MultiBFS does not exceed* $\frac{2(n-1)}{k+1}$.

Theorem 3. *The diameter of an undirected graph can be approximated in time* $O(km)$ *with an additive error at most* $\frac{2(n-1)}{k+1}$.

Proof. We have just proved the error bound in Corollary 1, so it remains to analyze the performance of Algorithm 1. A single execution of the line 6 can be carried out in $O(n)$ time, provided that the distances $d(v, U)$ are maintained in an array $d[v]$. The breadth first search in line 8 takes $O(m)$ time. Note, that besides the eccentricity $e(v_i)$ the breadth first search from v_i computes all distances $d(v_i, v)$, so it is easy to update the distance values $d[v]$ at the same time, namely, $d[v] \leftarrow \min(d[v], d(v_i, v))$. The running time of the algorithm is obviously dominated by those two lines, and since $n \leq m + 1$ for connected graphs, we have the total running time $O(km)$. □

By choosing $k = \sqrt{n}$, we obtain a good approximation in a reasonable amount of time.

Corollary 2. *The diameter of an undirected graph can be approximated in time* $O(m\sqrt{n})$ *with an additive error* $O(\sqrt{n})$.

The next section shows that most of the time the algorithm performs considerably better than the error bound we just derived. So, if the desired error bound ϵ is given in advance, we can apply an alternative strategy, where the number of iterations is determined dynamically. It follows from Theorem 1 that we can terminate the algorithm as soon as $L \leq \epsilon$.

3.2 Estimating the Radius

It is easy to modify the algorithm to return the radius of the graph. Just change the line 9 to return $\bar{R} = \min_{v \in U} e(v)$ as an approximation for the radius of G.

Theorem 4. *The radius of an undirected graph can be approximated in time* $O(km)$ *with an additive error at most* $\frac{2(n-1)}{k+1}$.

Proof. Again, let $L = \max_{v \in V} d(v, U)$. We just have to prove that $\bar{R} - \operatorname{rad} G \leq L$ and the rest of the theorem will follow from Theorems 2 and 3.

Suppose that $v \in V$ is a central vertex and $u \in U$ such that $d(v, u) = d(v, U)$. It is easy to see that $e(u) - e(v) \leq d(v, u)$, and since $e(u) \geq \bar{R}$, $e(v) = \operatorname{rad} G$, and $d(v, u) \leq L$, we have $\bar{R} - \operatorname{rad} G \leq L$. □

4 Experimental Results

To evaluate the practical behavior of our algorithm, we tested it on three families of random graphs and a collection of graphs used in testing various graph drawing algorithms [2].

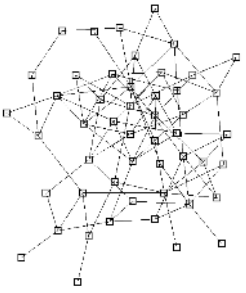


Fig. 1. Random graph

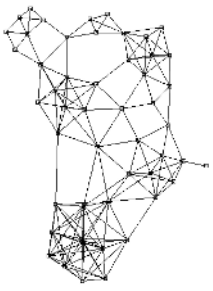


Fig. 2. Geometric graph

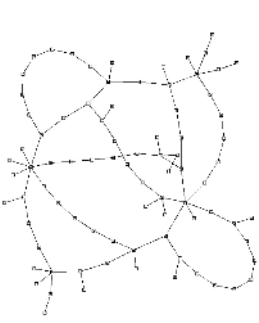


Fig. 3. Bridge graph

We compared the results with the algorithm by Aingworth et al [1] (denoted by ACIM in honor of the authors), and several linear time algorithms from [8]: Breadth First Search (BFS), Last Layer Minimum Degree (LL+) and Lexicographic Breadth First Search (LBFS). There are algorithms for the approximation of distances between all pairs of vertices (and consequently the diameter as well) [9], but due to their high running times $\Omega(n^2)$ they are of little interest here.

The algorithm ACIM has a little similarity with our approach in the sense that it performs several breadth first searches from a carefully chosen subset of vertices.

Each of the linear time algorithms searches for some vertex, whose eccentricity is returned as an approximate value of the diameter, namely,

- BFS algorithm performs a breadth first search and uses the last vertex visited,
- LL+ (Last layer, Minimum degree) performs a breadth first search and uses a vertex from the last layer, which has the minimum number of neighbors in the previous layer, and
- LBFS algorithm performs a lexicographic breadth first search [16] and uses the last vertex visited.

It must be noted that we chose the starting vertex in these algorithms randomly.

On the random graph families we chose the vertex count n from 10 to about 4000 and created 100 graphs for each size. We run all algorithms on these 100 graphs and calculated the maximum error from the true diameter, the average error and the number of times the algorithms failed to be exact. The results are given in Tables 1-3. Additionally, Figures 4-6 show the average error of the algorithms depending on the graph size. Our experiments revealed that all three linear-time algorithms perform very similar on the tested graph families, hence the figures show the data of LBFS algorithm only.

The first graph family we tested is random graphs generated with the algorithm "random edges" from [17]. Given a set V of vertices, it creates m random pairs $\{u, v\}$ of vertices in V as edges of G. We used $m = 2n$ in our experiments and took the largest connected component, if the generated graph was not

connected. See Fig. 1 for an example of a random graph. The maximum error of our algorithm never exceeded 1 on these graphs, while ACIM and the linear time algorithms had an error up to 2 and 3, respectively. The average error depending on the graph size is shown in Fig. 4. The proposed algorithm is clearly a winner. We also tested the algorithms on random graphs with larger edge density, but the results were similar, with the difference that diameters became smaller with increasing density and hence the errors of all algorithms also decreased significantly without changing their relative magnitudes considerably.

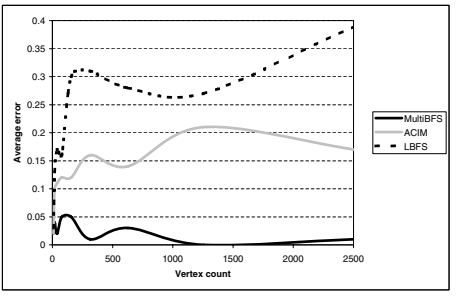


Fig. 4. Average error on random graphs

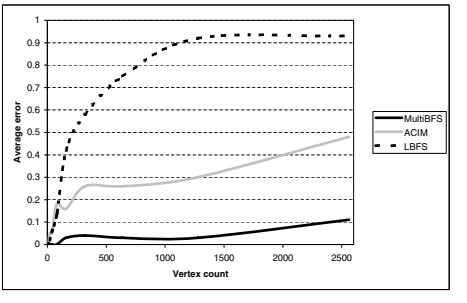


Fig. 5. Average error on geometric graphs

The second family is random geometric graphs [17]. In order to generate a geometric graph we place the vertices of the graph randomly in the unit circle. Then we include an edge between each two vertices that are within distance δ, where δ is the minimum value such that the resulting graph is connected. See Fig. 2 for an example of a geometric graph. Algorithm performance on geometric graphs is given in Fig. 5. Only the MultiBFS algorithm managed to keep the maximum error not exceeding 1, while the second best is ACIM with the maximum error of 4. Also, MultiBFS algorithm has significantly lower average error than the others on these graphs.

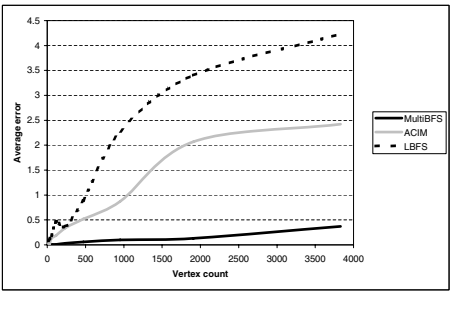


Fig. 6. Average error on bridge graphs

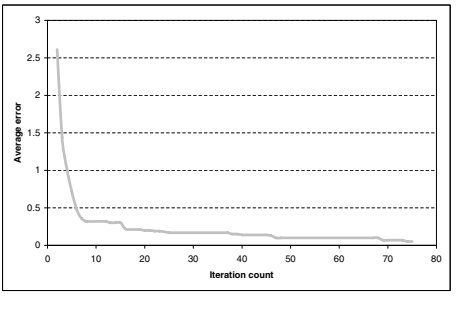


Fig. 7. Average error dependency of iteration count

The third family is what we call bridge graphs, a specially designed family of random graphs that have large diameter and hence is potentially hard for the diameter algorithms. To generate a bridge graph $B_{n,p,l}$, we take a random tree with n vertices as a starting point and add p paths of random length in range 1 to l between randomly chosen pairs of tree vertices. Here we choose $p = l = \sqrt{n}$. The trees are uniformly generated by using Prüfer sequences [4]. See Fig. 3 for an example of a bridge graph. Figure 6 shows the algorithm average error on these graphs. Only MultiBFS manages to keep it as low as 0.37. For the other algorithms the average error exceeds 2. Similarly, MultiBFS is a winner also in maximum error and error count measures.

Table 1. Algorithm average error, maximum error, and number of errors on random graphs

Vertex	Edge	MultiBFS			ACIM			BFS			LL+			LBFS		
count	count	Avg	Max	No	Avg	Max	No	Avg	Max	No	Avg	Max	No	Avg	Max	No
9	19	0.02	1	2	0.02	1	2	0.06	1	6	0.05	1	5	0.03	1	3
19	39	0.04	1	4	0.1	1	10	0.19	2	16	0.19	2	18	0.13	2	12
39	79	0.02	1	2	0.11	1	11	0.21	2	20	0.18	2	17	0.17	2	16
78	159	0.05	1	5	0.12	1	12	0.24	2	23	0.31	2	29	0.16	2	15
156	319	0.05	1	5	0.12	1	12	0.35	2	32	0.46	2	41	0.3	2	29
313	639	0.01	1	1	0.16	1	16	0.27	2	22	0.47	2	39	0.31	2	26
627	1279	0.03	1	3	0.14	1	14	0.36	2	34	0.52	2	46	0.28	2	26
1254	2558	0	0	0	0.21	1	21	0.3	2	28	0.47	2	39	0.27	3	23
2509	5117	0.01	1	1	0.17	2	16	0.3	2	26	0.65	3	50	0.39	2	34

Table 2. Algorithm average error, maximum error, and number of errors on geometric graphs

Vertex	Edge	MultiBFS			ACIM			BFS			LL+			LBFS		
count	count	Avg	Max	No	Avg	Max	No	Avg	Max	No	Avg	Max	No	Avg	Max	No
10	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	46	0	0	0	0.05	1	5	0.04	1	4	0.04	1	4	0.02	1	2
40	120	0	0	0	0.05	1	5	0.08	2	7	0.1	2	9	0.06	2	5
80	275	0	0	0	0.18	4	13	0.25	3	18	0.17	2	12	0.13	3	9
160	605	0.03	1	3	0.16	2	14	0.38	4	23	0.35	4	22	0.42	4	27
320	1361	0.04	1	4	0.26	3	18	0.61	4	40	0.61	4	40	0.58	6	38
640	2897	0.03	1	3	0.26	3	18	0.98	6	47	0.88	6	45	0.75	4	48
1280	6255	0.03	1	3	0.3	4	21	1.08	10	58	1.07	7	58	0.92	6	56
2560	13801	0.11	1	11	0.48	3	35	0.92	5	58	1.05	5	63	0.93	6	60

We also tested the algorithms on tree graphs but do not include the results here since all algorithms except ACIM managed to find the exact diameter. Indeed, it can be easily proved that BFS, LL+ and LBFS algorithms find the exact diameter of tree graphs, and that our algorithm finds the diameter after two iterations.

The results of the graph collection are similar to results of random graphs. The collection contains 11579 connected graphs. The results are given in Table 4. Our algorithm is clearly a leader in all given error measures.

Regarding the quality of results of MultiBFS algorithm depending on the iteration count k, our algorithm was run on the random, geometric and bridge graphs for different values of k and error measures were calculated. Average error dependency on the iteration count on bridge graphs of average node count 1515 is shown in Figure 7. For this graph size our choice of iteration count is $k = \sqrt{n} = 39$, and it can be seen from the figure that such choice provides a reasonable compromise of the quality and running time. The results on random and geometric graphs are similar.

Table 3. Algorithm average error, maximum error, and number of errors on bridge graphs

Vertex	Edge	MultiBFS			ACIM			BFS			LL+			LBFS		
count	count	Avg	Max	No	Avg	Max	No	Avg	Max	No	Avg	Max	No	Avg	Max	No
14	16	0.02	1	2	0.03	1	3	0.14	2	12	0.09	1	9	0.08	1	8
30	33	0	0	0	0.01	1	1	0.17	4	9	0.07	2	6	0.12	3	7
57	62	0.01	1	1	0.13	2	12	0.16	3	13	0.14	3	11	0.18	3	15
116	124	0.01	1	1	0.19	4	14	0.36	4	20	0.49	4	29	0.47	4	27
237	249	0.03	1	3	0.34	6	17	0.85	6	39	0.78	5	39	0.37	4	21
472	489	0.06	2	5	0.52	7	23	1.54	10	52	1.42	9	48	0.89	6	34
952	976	0.1	2	8	0.87	10	32	1.73	10	51	1.8	10	49	2.28	13	51
1908	1943	0.13	4	8	2.07	13	44	3.35	14	60	3.24	14	62	3.4	15	65
3831	3881	0.37	5	19	2.42	15	48	4.01	22	63	3.98	22	65	4.23	19	71

Table 4. Algorithm average error, maximum error, and number of errors on the collection

	Avg	Max	Number of errors
ACIM	0.073	4	779
LBFS	0.156	6	1494
MultiBFS	0.010	2	111

5 Conclusions and Further Research

In this paper we have proposed an efficient algorithm for approximately determining the diameter and the radius of a graph. The approximation accuracy can be chosen arbitrary with appropriate trade-off in running time. In order to obtain a good approximation in reasonable time, we have chosen the maximum additive error of $O(\sqrt{n})$, leading to an algorithm with $O(m\sqrt{n})$ running time. From theoretical point of view our algorithm cannot be directly compared with 2/3-approximation algorithm for the diameter problem by Aingworth et al [1] since their error bound is multiplicative rather than additive. For larger diameters our algorithm gives lower error bounds, with the opposite being true for

small diameters. The algorithm by Aingworth et al has also worse running time for sparse graphs.

We have shown that the proposed algorithm performs very well on several families of graphs, giving much lower error rate than other known approximation algorithms.

As a further work we would like to extend our algorithm to directed graphs and weighted graphs. Also it would be worth investigating whether our approach leads to new approximation algorithms for the all-pairs-shortest-paths problem. And, of course, the problem of computing the exact diameter of a graph in sub quadratic time remains as challenging as before.

Acknowledgments

We would like to thank Austris Krastiņš and Viesturs Zariņš for the considerable amount of work on implementing our testing environment, and to Paulis Ķikusts for his valuable comments on this paper.

References

1. D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast Estimation of Diameter and Shortest Paths (without Matrix Multiplication). *SIAM J. on Computing*, 28 (1999), pages 1167-1181.
2. G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari and F. Vargiu. An Experimental Comparison of Three Graph Drawing Algorithms. *Proc. 11th Annu. ACM Sympos. Comput. Geom*, 1995, pages 306 - 315.
3. A. Brandstädt, V.D. Chepoi, and F.F. Dragan. The Algorithmic Use of Hyper-Tree Structure and Maximum Neighborhood Orderings. In E.W. Mayr, G. Schmidt, and G. Tinhofer (Eds.), *Proceedings of WG'94 Graph-Theoretic Concepts in Computer Science*, LNCS 903, pages 65-80, 1995.
4. G. Chartrand and L. Lesniak. Graphs & Digraphs. *Chapman & Hall*, 1996.
5. Fan R.K. Chung. Diameters of Graphs: Old Problems and New Results. *Congressus Numerantium*, 60 (1987), pages 295-317.
6. D. Coppersmith and S. Winograd. Matrix Multiplication via Arithmetic Progression. *Proc 19th ACM Symp on Theory of Computing*, 1987, pages 1-6.
7. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. Introduction to Algorithms. *The MIT Press*, 1990.
8. D.G. Corneil, F.F. Dragan, and E. Köhler. On the Power of BFS to Determine a Graph's Diameter. *Networks*, 42(4), pages 209-222, 2003.
9. D. Dor, S. Halperin, and U. Zwick. All Pairs Almost Shortest Paths. *Electronic Colloquium on Computational Complexity*, vol. 4, 1997.
10. F.F. Dragan. Dominating Cliques in Distance-Hereditary Graphs. In E.M. Schmidt and S. Skyum (Eds.), *Proceedings of SWAT'94 Fourth Scandinavian Workshop on Algorithm Theory*, LNCS 824, pages 370-381, 1994.
11. F.F. Dragan and F. Nicolai. LexBFS-orderings of Distance-Hereditary Graphs with Application to the Diametral Pair Problem. *Discrete Appl. Math.*, 98 (2000), pages 191-207.
12. F.F. Dragan, F. Nicolai, and A. Brandstädt. LexBFS-orderings and Powers of Graphs. *Proceedings of the WG'96*, LNCS 1197, pages 166-180, 1997.

13. A.M. Farley and A. Proskurowsky. Computation of the Center and Diameter of Outerplanar Graphs. *Discrete Appl. Math.*, 2 (1980), pages 185-191.
14. G. Handler. Minimax Location of a Facility in an Undirected Tree Graph. *Transp. Sci.*, 7 (1973), pages 287-293.
15. S. Olariu. A Simple Linear-Time Algorithm for Computing the Center of an Interval Graph. *Int. J. Comput. Math.*, 34 (1990), pages 121-128.
16. D. Rose, R.E. Tarjan, and G. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs. *SIAM J. Comput.*, 5 (1976), pages 266283.
17. R. Sedgewick. Algorithms in C, Part 5: Graph Algorithms, 3rd Edition. Addison-Wesley, 2002.
18. R. Seidel. On the All-Pair-Shortest-Paths Problem. *Proc 24th ACM Symp on Theory of Computing*, 1992, pages 745-749.

Lists on Lists: A Framework for Self-organizing Lists in Environments with Locality of Reference

Abdelrahman Amer and B. John Oommen*

School of Computer Science, Carleton University, Ottawa, Canada
{aamer, oommen}@scs.carleton.ca

Abstract. We examine the problem of self-organizing linear search lists in environments with the locality of reference phenomenon, when the queries exhibit a probabilistic dependence between themselves. We introduce a novel list organization framework that we call *Lists on Lists (LOL)*, which regards the list as a set of sublists that are manageable in the same way that individual records are. A LOL organization involves a reorganization operation on the accessed record level, as well as another on the sublist which it belongs to (the record's *context*). We show that it is beneficial to consider the reorganization of the context together with the accessed record, since other records within the context are likely to be accessed in the near future. With the aid of an automaton-based *partitioning algorithm*, we demonstrate that we can accurately classify the different contexts of the sublist. Using this framework, we were able to empirically achieve asymptotic search costs that are significantly superior to the move-to-front heuristic, widely acknowledged as the best algorithm for such environments.

1 Introduction

For data retrieval from a list of n elements, a *linear* or *sequential search* is performed, with a time complexity of $O(n)$. It can be shown that the best ordering of the records would be in terms of the descending order of their respective access probabilities. As such, since the access probabilities are rarely known in advance, one has to resort to a heuristic to approximate an optimal behaviour. One way to significantly reduce the search cost (without estimating the access probabilities) is by making the list *self-organizing*, such that it dynamically adapts to the query distribution to minimize the search cost, by running a reorganization algorithm. A *self-organizing linear search list* is a list that runs a reorganization heuristic with (possibly) each access query with the hope of optimizing the linear search cost.

Research on self-organizing linear search lists began with McCabe's work in 1965 [1]. He introduced two such heuristics: the move-to-front rule (MTF), which moves the queried record to the front of the list, and the transposition rule (TR), which swaps the queried record with its predecessor. The MTF is characterized

* *Professor* and *Fellow of the IEEE*.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 109–120, 2006.

by quick convergence rates and the ability to quickly adapt to changes in the environment. The TR, on the other hand, is more likely to find a better asymptotic arrangement for the list than the MTF, through its incremental conservative changes. However, its convergence rate and ability to respond to changes in the environment are poor.

In essence, a lot of the research that followed McCabe's work has attempted to combine the two algorithms, to benefit from the speed of the MTF and the accuracy of the TR. Some algorithms are hybrids of the two (e.g. [2], [3], [4]). Other "batched" methods invoke the reorganization heuristic every kth access [1] or after k accesses in a row [5]. More recent work in this field include the work of Schulz [6] and Bachrach et al. [7]. Randomized algorithms (e.g. [8], [9], [10]) are another means to reorganize lists that, generally speaking, try to move the accessed record in a more conservative way than the MTF.

An interesting behaviour of the algorithms manifests when the environment has the property of *locality of reference*, because of the probabilistic dependence among the access queries. In other words, during a given time interval, only a subset of the query set is predominantly accessed on the list, and then this subset changes with time. This property is also sometimes referred to as dependent accesses. Examples of the locality of reference phenomenon are indeed abundant in computer science, such as in program execution, paging and caching, memory management, file systems, and database systems.

Lam et al. [11] studied the the expected search cost of the MTF in a dependent environment, by considering a Markovian model for dependent accesses. Chassaing [12] has shown that the expected search time for MTF in environments with locality of reference is not greater than that of any other sequential strategy. Later, in 2002, Bachrach et al. [7] backed this result with a comprehensive experiment that tested almost all the algorithms reported in the literature up to the time of the study. They have shown that the MTF outperforms all other algorithms that they tested in such environments, as the dependence factor increases. This result seems to be the general consensus within the literature. To our knowledge, no other algorithm has been reported to outperform the MTF in environments with higher dependence degrees. Comparison against the MTF is therefore a very good metric for the performance of an algorithm in environments with locality of reference.

One common approach for algorithms dealing with environments possessing such a property is to minimize the access cost for the required resource as well as its surrounding "context," since the odds that a nearby element will soon be requested are high. Applied to self-organizing lists, this approach would reorganize some of the elements around the accessed record as well as the record itself, i.e. by essentially reorganizing a *sublist* that the accessed record belongs to, within the list. This is especially effective in lists since the reorganization cost for a sublist is the same as that of a single record (by updating few pointers), without incurring additional cost, provided that the sublist boundaries are known. We call such an approach a *List on List (LOL)* organization.

A LOL algorithm is denoted by the form X-Y, where X would be the reorganization of the accessed record within the sublist, and Y would be the reorganization of sublists among themselves. For example, consider the LOL scheme MTF-MTF. In one variation of this scheme, we divide the list into a number of sublists k, of length m each. When a requested record is found, it is moved to the front of the sublist it belongs to, and then the entire sublist is moved to the front of the list. If the sublist contains records within the current environment's context, we may just be lucky enough to have minimized the access costs for a host of forthcoming requests until the environment's context changes. At the same time, by doing this, we are bringing records that are more frequently accessed *within this context* to the head of the list, thereby reducing the cost even more.

What remains is to ensure that the sublists actually reflect the environment's various contexts. It would be of no benefit for us to move a number of records to the beginning of the list if they are not likely to be accessed in the near future. To achieve this goal, we used a *dependence capturing* or *partitioning* automaton, alongside with the LOL reorganization algorithm. This is done by running the current queried key (at time t) with the last queried key (at time $t-1$) against the partitioning algorithm. If the two elements belong to the same partition within the partitioning automaton's internal representation, this automaton is "rewarded" and the LOL reorganization takes place. Otherwise, the automaton is "penalized," and instead of performing a LOL reorganization, the automaton learns a better partitioning and the sublists are modified accordingly to match that partitioning. In this paper we use two such algorithms to attain the partitioning, the Object Migration Automaton (OMA) [13], and the Modified Linear Reward-Penalty reinforcement rule ($\mathrm{ML_{RP}}$) [14]. A LOL scheme that uses a partitioning automaton is denoted by the form X-Y-Z, where Z is the partitioning algorithm. The above example would be called MTF-MTF-OMA.

2 Models of Dependence

To initiate the study, we assume that the set of n distinct query elements can be divided into k equal disjoint subsets with m elements, where $n = k \cdot m$. These subsets are sometimes referred to as *local sets* or *local contexts* [7]. Elements of the same subset are "dependent" on each other, i.e. if an element from set i is requested at time t, the likelihood that the element requested at time $t+1$ is from the same local context is significant.

The environment can be regarded as having a set of finite states $\{Q_i | 1 \leq i \leq k\}$. The way the environment changes states defines the type of dependence model.

In a *Markovian switching environment*, the states of the environments are also the states of an (unkown) Markov chain. After generating the request, the environment stays in the same state with probability α, and makes a transition to another state with probability $(1-\alpha)/(k-1)$. After being at any environment state (local set), the query generation follows a fixed probability distribution to facilitate analysis. This model is the one most often used to study dependent

environments, such as in [7], [11], [15]. Bachrach et al. [7] observed that the expected number of subsequent requests in a local set is given by $1/(1-\alpha)$.

In a *periodic switching environment*, the environment changes states in a round-robin fashion. After every T queries, the environment changes state from Q_i to $Q_{i+1 \bmod k}$. Within these T queries, all the requests belong to exactly one local set. There are two variations of this environment. In the first, the algorithm which manipulates the data structures is aware of the time after which the environment changes its state. In the second, the algorithm does not know about this time period. Algorithms with this extra information are expected to yield better performance than those without it.

3 Sublist Manipulation

We propose the idea of dividing the self-organizing list of size n into a set of k sublists. We examine both sublists of the same size (Sect. 4.1) and different sizes (Sect. 4.3). We would then apply self-organization algorithms on the elements within the sublist and then between the sublists *themselves*. For example, we may choose to transpose the record within the sublist and then move the sublist to the front of the original list. Following the naming scheme introduced above, this would be called TR-MTF. Another example would be to move the record to the front of the sublist, and then move the sublist to the front of the list, which would be an MTF-MTF scheme (see Fig. 1).

The rationale behind such a strategy can be explained as follows. When an element is accessed in a dependent environment, the likelihood that another element within its local set is going to be chosen in the immediate future is high. It is therefore beneficial to make use of this information by promoting the entire set to an advanced position within the list, thereby reducing the search time if such an element is requested. This is somewhat related to the benefits one would gain from using the MTF rule in a non-stationary environment, since the

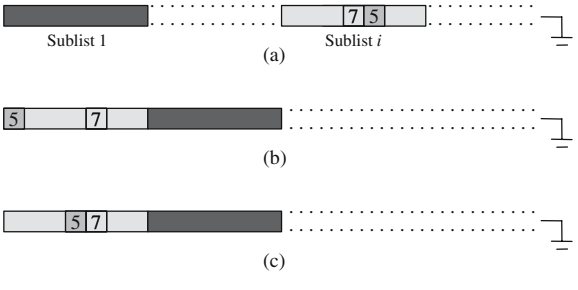


Fig. 1. Illustration of the MTF-MTF and the TR-MTF. (a) A record with key 5 is accessed in sublist i. (b) In MTF-MTF, the record is moved to the front of the sublist, and then the sublist is moved to the front of the list. (c) In TR-MTF, the record is transposed with the preceding record, and then the sublist is moved to the front of the list.

frequently accessed records are promoted to locations near the front of the list. The LOL approach has several benefits, however. Unlike the MTF, records are promoted en masse, and therefore the prospects of the access time being reduced in forthcoming accesses increase. Second, if a record outside of the local context is accessed, it is displaced much further from the front in subsequent calls than in the MTF case, because the group of all records that obey the context may be promoted to precede it. In the case of the MTF, however, such an access may cause the record to linger at the front of the list for a longer duration, until the entire context is moved to the front of the list one element at a time.

To capture the dependence between the access queries, we use a *partitioning scheme*, such that records of the same local context reside in the same sublist division. We used the Object Migration Automaton (OMA) [13], which partitions objects into equal size groups. We also used the Modified Linear Reward-Penalty scheme ($\mathrm{ML_{RP}}$) [14], which can produce different-sized partitions albeit at greater time and space costs. Both schemes are based on the field of learning automata [15].

Algorithm 1. Generic LOL algorithm X-Y-Z

Notation:
q: Input query
R_q: Record corresponding to q
$L(R_q)$: Yields the sublist to which R_q belongs
X: Reorganization scheme on the record level, within a sublist
Y: Reorganization scheme on the sublist level
Z: Partitioning algorithm

```
loop
  Input query q
  q1 ⇐ q2
  q2 ⇐ q
  Search for R_q2
  if not the first time then
    Run Z(q1, q2)
    if Reward then
      Run X(R_q2, L(R_q2))
      Run Y(L(R_q2))
    else
      Rearrange R_q1 or R_q2 as dictated by the partitioning Z
    end if
  end if
end loop
```

Algorithm 1 presents a high level algorithmic version for the generic LOL algorithm X-Y-Z. Clearly, the time complexity of this algorithm is dependent on those of X, Y and Z. However, we note that the cost of X is often constant, since it is likely a simple MTF or TR operation within the sublist. The cost of Z is often constant (as in the case of the OMA for example). This is especially true

after convergence, because the automaton's state does not change. The cost of Y is either constant or $O(m)$ (in the case where pointers to sublist boundaries are maintained). Therefore, it is often the case that the only cost incurred is that of the linear search. However, this is dependent on the choice of algorithms used and their details.

4 Empirical Results

Throughout our experiments, we used a data source with $n = 128$ distinct elements, labeled 1 through 128. To produce dependent access sequences, we divided the query space into k local contexts $(Q_1 \dots Q_k)$ (unknown to the list organization scheme), each of size m. We then applied the Markovian and periodic models of dependence discussed in Section 2. For example, if $k = 8$ and $m = 16$, Q_1 would include the elements $1 \dots 16$, Q_2 would include $17 \dots 32$, etc. To model the probability distribution within the local context, we used Zipf's law, which has been used to model the probability distribution within the local context of dependent environments in [7], as well as four additional distributions. The access probability s_i for record i, $1 \leq i \leq m$, for the different distributions is given by:

1. Zipf's distribution:

$$s_i = \frac{1}{iH_m} \text{ where } H_m = \sum_{k=1}^{m} (1/k).$$

2. 80-20 distribution:

$$s_i = \frac{1}{i^{(1-\theta)} H_m^{(1-\theta)}} \text{ where } H_m^{(1-\theta)} = \sum_{k=1}^{m} (1/k^{(1-\theta)}), \text{ and } \theta = \frac{\log 0.80}{\log 0.20} \approx 0.1386.$$

3. Lotka's distribution:

$$s_i = \frac{1}{i^2 H_m^2} \text{ where } H_m^2 = \sum_{k=1}^{m} (1/k^2).$$

4. Exponential distribution:

$$s_i = \frac{1}{2^i K} \text{ where } K = \sum_{k=1}^{m} (1/2^k).$$

5. Linear distribution:

$$s_i = K(m - i + 1) \text{ where } K = \sum_{k=1}^{m} k.$$

4.1 OMA-Based Algorithms

Number of sublists. We tested the MTF-MTF-OMA algorithm in a Markovian environment ($\alpha = 0.9$) for varying number of sublists (k= 2, 4, 8, 16, 32, 64). Setting α to 0.9 compares our algorithms against the strongest performance of the MTF, which excels with increasing dependence. Figure 2 plots the ratio of the asymptotic cost of the MTF-MTF-OMA algorithm to that of the MTF, for different values of k. A value greater than unity indicates a performance worse than MTF, while a value less than unity indicates a superior performance.

Observe that for all distributions with the exception of the Lotka and exponential distributions, the MTF-MTF-OMA produced results that were consistently better than the MTF for most numbers of sublists, achieving results that are twice as good in some cases. Also observe that the curves are U-shaped, because the algorithms are not optimal for very long and very short sublists. When the sublists are long (lower values of k), the partitioning of the OMA is non-optimal. As a result, the algorithm may spend a lot of time searching for a record that was wrongly partitioned out of the current working set. For very short sublists (higher values of k), any error in the partitioning algorithm would prove to be too costly in comparison to a plain MTF operation. Therefore, the optimum

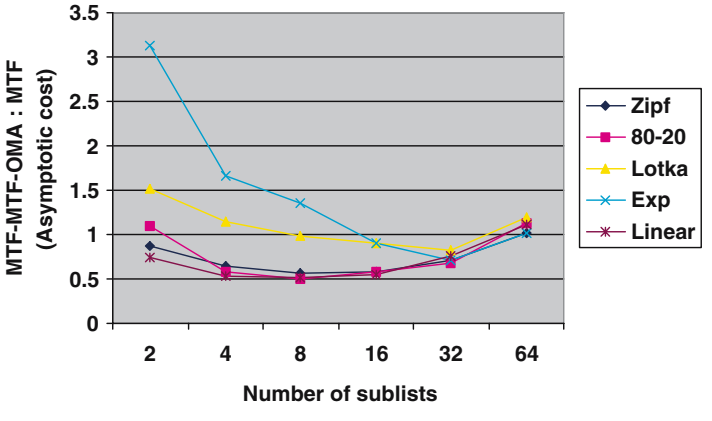


Fig. 2. The ratio of the asymptotic cost of the MTF-MTF-OMA to that of the MTF, for different numbers of sublists k and different probability distributions, in a Markovian environment with $\alpha = 0.9$

Table 1. Asymptotic cost for LOL algorithms with $k = 16$ and $\alpha = 0.9$

Distrib.	Zipf	80-20	Lotka	Exp	Linear
MTF	28.5	29.7	16.4	16.1	30.2
TR	38.5	42.0	18.4	22.6	46.1
MTF-MTF-OMA	16.4	17.1	14.7	14.5	16.7
MTF-MTF-MLRP	15.8	16.2	13.8	14.4	16.7

number of sublists (and records within a sublist) is in the center, and hence the U-shape of the curves in Fig. 2. In fact, for $k = 16$ and $k = 34$, the results of LOL algorithms were better in *all* distributions tested (see Table 2).

Degree of dependence. We have also studied the performance of the MTF-MTF-OMA with respect to varying degrees of locality of reference (α). Figure 3 shows curves that reproduce the behaviour seen in the experiments of Bachrach et al. [7], in which the MTF outperformed the TR (and all other algorithms) as α increased. All other algorithms that they tested had curves that lay between these two.

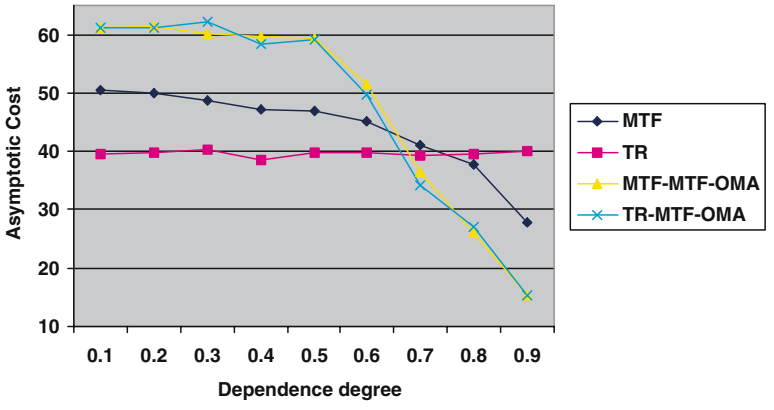


Fig. 3. The asymptotic cost of various algorithms, in a Markovian environment with different degrees of dependence α, for $k = 16$ sublists under the Zipf probability distribution

Table 2. Asymptotic cost for LOL algorithms with $k = 16$ and varying α under the Zipf probability distribution

α	0.5	0.6	0.7	0.8	0.9
MTF	46.9	43.4	41.0	37.7	27.9
TR	39.9	39.4	39.3	39.6	40.1
MTF-MTF-OMA	59.3	56.8	36.6	26.0	15.4
MTF-MTF-MLRP	48.5	43.9	35.8	26.0	15.5

However, at higher dependence degrees, the curve for the MTF-MTF-OMA dives well below that of the MTF (see Table 2). This result is significant since it empirically shows that the LOL algorithms yield better asymptotic search cost than the MTF, which was shown by Bachrach et al. [7] to be better than all other algorithms they tested in a strongly dependent environment, using the same dependence model.

4.2 Periodic Variations

We tested the MTF-MTF-OMA in a periodic environment with a period of 10 accesses. This produces a comparable environment to the Markovian one (with $\alpha = 0.9$), since the average number of accesses from the same working set in the Markovian environment is $1/(1-\alpha) = 10$. In a periodic environment, the MTF-MTF-OMA delivers performance that is just as good as in a Markovian one with strong locality of reference. This is because periodic environments can be thought of as being Markovian ones with very strong locality of reference ($\alpha = 1$) for a specified number of queries. When compared to MTF, the curves produced are of a similar nature to the ones produced in a Markovian environment, and all the arguments can be carried over.

If we have advance knowledge of the period T, we can simply move the first sublist to the *end* of the list after T queries. If the OMA has correctly converged, no records from this sublist are expected to be accessed except after $(k-1)T$ queries. We call this algorithm the MTF-MTF-OMA-P (for "periodic"). If we do not know the period in advance, however, we choose to move the first sublist to the rear if two successive queries are not dependent. That is, if the OMA penalizes two successive queries, we interpret that as the end of the period and move the sublist to the rear accordingly. We call this algorithm the MTF-MTF-OMA-UP (for "unknown period").

The MTF-MTF-OMA-P is *ideal* for the periodic environment. For smaller numbers of sublists ($k = 2, 4$), the MTF-MTF-OMA-P significantly outperforms the MTF-MTF-OMA, already shown to be better than the MTF. For larger numbers of sublists (e.g. $k = 32$, Zipf distribution), the difference is dramatic: 39.5 for the asymptotic cost of the MTF, 27.0 for the MTF-MTF-OMA, and 2.6 for the MTF-MTF-OMA-P (Fig. 4).

As expected, in a Markovian environment, the MTF-MTF-OMA-P does not perform as well as MTF-MTF-OMA. Needless penalties are incurred since the

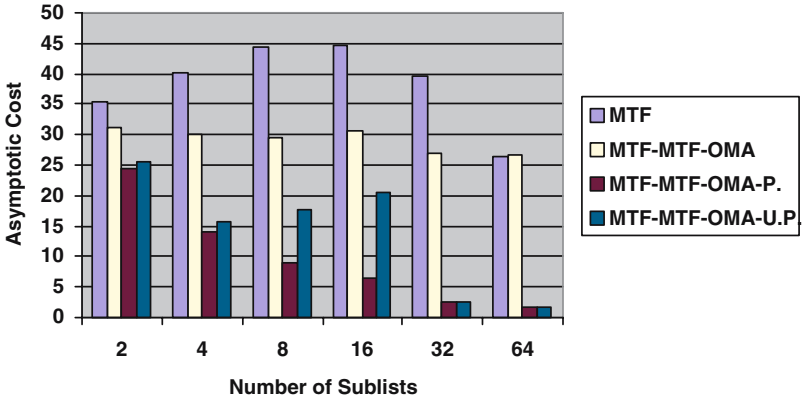


Fig. 4. The asymptotic cost of different algorithms plotted against the number of sublists for the Zipf distribution, in a periodic environment with period = 10

algorithms only consider the period T, rather than the query dependence, for moving the first sublist to the rear. However, the performance of the MTF-MTF-OMA-UP was often on a par to that of the MTF-MTF-OMA. This is because when the local set changes in a Markovian environment, all the other local sets are equally likely to become the current local set, and therefore the position where the first sublist is moved to is irrelevant.

4.3 $\mathrm{ML_{RP}}$-Based Algorithms

We have seen that the MTF-MTF-OMA produced results that are better than the MTF in dependent environments. However, the OMA requires the sizes of the environment's local contexts to be equal. But the LOL organization strategy and the dependence capturing mechanism need not be tightly coupled. As long as the interface is well-defined, any dependence capturing mechanism can be used with the algorithms described above, instead of the OMA.

As a proof of concept, we implemented the MTF-MTF-$\mathrm{ML_{RP}}$, where the $\mathrm{ML_{RP}}$ was used for the partitioning of the the environment inputs and determining the sublists. The $\mathrm{ML_{RP}}$ can partition objects into different sized groups, thereby offering more flexibility than the OMA, with comparable accuracy. However, we have seen that the time and space requirements are higher than that of the OMA.

The MTF-MTF-$\mathrm{ML_{RP}}$ algorithm is very similar to the MTF-MTF-OMA, except that in initialization, every record in the list sits in a sublist of its own. The sublists are then gradually merged according to the access sequences to form bigger sublists, that eventually match the dependence model of the environment. Notice that, when an element is moved from one sublist to another, no other element is moved back to the original sublist.

In Tables 2 and 2 we show that the costs for the MTF-MTF-$\mathrm{ML_{RP}}$ and the MTF-MTF-OMA are very close to each other. We have also noticed that the variance for $\mathrm{ML_{RP}}$-based algorithms is much smaller than that for OMA-based algorithms, particularly for the Lotka and exponential distributions. The $\mathrm{ML_{RP}}$ seems to be more likely to choose a more correct partitioning than the OMA for these distributions.

The most interesting result about the MTF-MTF-$\mathrm{ML_{RP}}$ is the rate of change in its amortized cost, illustrated in Fig. 5. Narendra and Thathatchar [15] define the optimality of a system in a non-stationary environment by its ability to minimize the amortized cost. To get a feeling for the rate of convergence of the algorithms, we plotted the amortized cost as the number of queries increased, for the first 100,000 queries, in a Markovian environment with $\alpha = 0.9$, $k = 16$, for the Zipf distribution (Fig. 5). Starting at roughly the same cost as OMA-based algorithms, the MTF-MTF-$\mathrm{ML_{RP}}$ curve proceeds with an impressive convergence speed to quickly reach the asymptotic value well before the MTF-MTF-OMA. Although the literature ([13], [14]) did not provide comparisons between the $\mathrm{ML_{RP}}$ and the OMA, we believe that the former yields much faster convergence speed, which in turn could be well worth the additional time and space penalties incurred.

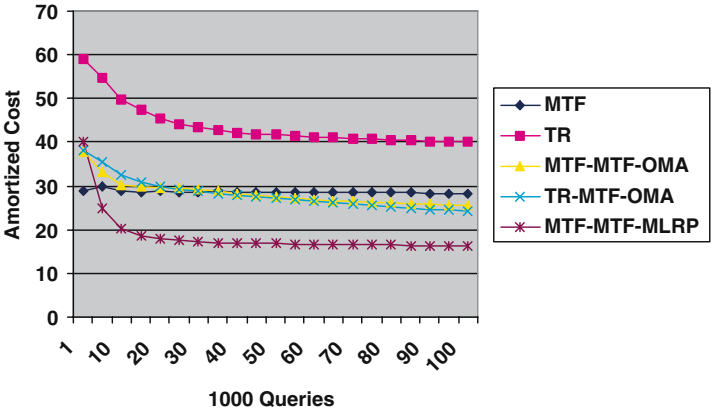


Fig. 5. The amortized cost of various algorithms, for 100,000 queries, in a Markovian environment with $\alpha = 0.9$, for $k = 16$ sublists under the Zipf probability distribution

5 Conclusion

We have introduced a strategy for reorganizing linear search lists under environments with locality of reference that we call *List on Lists (LOL)*. Under such schemes, the list can be viewed as a set of sublists that could be manipulated and reorganized, just as the individual records are. With the help of a partitioning algorithms, we can arrange the different sublists to try to have the sublists match the different environment contexts.

Experimental results with simulated data for LOL based algorithms have shown superior results to those of the MTF, widely acknowledged as the best heuristic for use in dependent environments. This was consistent for most of the probability distributions and different sublist sizes tested. Additionally, by using the $\mathrm{ML}_{\mathrm{RP}}$, we were able to achieve very fast convergence rates. As well, our periodic variations of LOL algorithms were shown to be very well suited to both periodic and Markovian environments. These results need to be backed up with experiments on real data.

The most significant contribution of the paper is the novel way of viewing the list as a set of sublists, that are manageable using the same strategy used to manage individual records. To our knowledge, such an approach has not been explored before. The formal analysis of the schemes, though open, will be far from trivial.

References

1. McCabe, J.: On serial files with relocatable records. Operations Research **12** (1965) 609–618
2. Bitner, J.R.: Heuristics that dynamically organize data structures. SIAM Journal on Computing **8**(1) (1979) 82–110

3. Rivest, R.: On self-organizing sequential search heuristics. Communications of the ACM **19**(2) (1976) 63–67
4. Tenenbaum, A.M., Nemes, R.M.: Two spectra of self-organizing sequential search algorithms. SIAM Journal on Computing **11** (1982) 557–566
5. Kan, Y.C., Ross, S.M.: Optimal list order under partial memory constraints. Journal of Applied Probability **17** (1980) 1004–1015
6. Schulz, F.: Two new families of list update algorithms. In: Proceedings of the 9th International Symposium on Algorithms and Computation (ISAAC), LNCS 1533 (1998) 99–108
7. Bachrach, R., El-Yaniv, R., Reinstadtler, M.: On the competitive theory and practice of online list accessing algorithms. Algorithmica **32**(2) (2002) 201–245
8. Irani, S.: Two results on the list update problem. Information Processing Letters **38**(6) (1991) 301–306
9. Reingold, N., Westbrook, J., Sleator, D.D.: Randomized competitive algorithms for the list update problem. Algorithmica **11**(1) (1994) 15–32
10. Albers, S.: Improved randomized on-line algorithms for the list update problem. SIAM Journal on Computing **27** (1998) 670–681
11. Lam, K., Leung, M.Y., Siu., M.K.: Self-organizing files with dependent accesses. Journal of Applied Probability **21** (1984) 343–359
12. Chassaing, P.: Optimality of move-to-front for self-organizing data structures. Annals of Applied Probability **3**(4) (1993) 1219–1240
13. Oommen, B.J., Ma, D.C.Y.: Deterministic learning automata solutions to the equipartitioning problem. IEEE Transactions on Computers **37**(1) (1988)
14. Oommen, B.J., Ma, D.C.Y.: Stochastic automata solutions to the object partioning problem. The Computer Journal **35** (1992) A105–A120
15. Narendra, K.S., Thathachar, M.A.L.: Learning Automata: An Introduction. Prentice Hall (1989)

Lists Revisited: Cache Conscious STL Lists

Leonor Frias*, Jordi Petit**, and Salvador Roura***

Departament de Llenguatges i Sistemes Informàtics,
Universitat Politècnica de Catalunya,
Campus Nord, edifici Omega,
08034 Barcelona, Spain
{lfrias, jpetit, roura}@lsi.upc.edu

Abstract. We present three cache conscious implementations of STL standard compliant lists. Up to now, one could either find simple double linked list implementations that easily cope with standard strict requirements, or theoretical approaches that do not take into account any of these requirements in their design. In contrast, we have merged both approaches, paying special attention to iterators constraints. In this paper, we show the competitiveness of our implementations with an extensive experimental analysis. This shows, for instance, 5-10 times faster traversals and 3-5 times faster internal sort.

1 Introduction

The Standard Template Library (STL) is the algorithmic core of the C++ standard library. The STL is made up of containers, iterators and algorithms. Containers consist on basic data structures such as lists, vectors, maps or sets. Iterators are a kind of high-level pointers used to access and traverse the elements in a container. Algorithms are basic operations such as sort, reverse or find. The C++ standard library [1] specifies the functionality of these objects and algorithms, and also their temporal and spatial efficiency, using asymptotical notation.

From a theoretical point of view, the knowledge required to implement the STL is well laid down on basic textbooks on algorithms and data structures (e.g. [2]). In fact, the design of current widely used STL implementations (including SGI, GCC, VC++, ...) is based on these.

Nevertheless, the performance of some data structures can be improved taking advantage of the underlying memory hierarchy of modern computers. Not in vain, in the last years the algorithmic community has realized that the old unitary cost memory model is turning more inaccurate with the changes in computer architecture. This has raised an interest on cache conscious algorithms

* This author has been supported by grant number 2005FI 00856 of the *Agència de Gestió d'Ajuts Universitaris i de Recerca* with funds of the European Social Fund and ALINEX project under grant TIN2005-05446.

** This author has been supported by GRAMARS project under grant TIN2004-07925-C03-01 and ALINEX project under grant TIN2005-05446.

*** This author has been supported by AEDRI-II project under grant MCYT TIC2002-00190 and ALINEX project under grant TIN2005-05446.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 121–133, 2006.

and data structures that take into account the existence of a memory hierarchy, mainly studied under the so-called cache aware (see e.g. [3, 4]) and cache oblivious models (see e.g. [5, 6]).

However, if these data structures are to be part of a standard software library, they must conform to its requirements. As far as we know, no piece of work has taken this into account. Our aim in this paper is to propose standard compliant alternatives that perform better than traditional implementations in most common settings. Specifically, we have analyzed one of the most simple but essential objects in the STL: lists. We have implemented and experimentally evaluated three different variants of cache conscious lists supporting fully standard iterator functionality. The diverse set of experiments shows that great speedups can be obtained compared to traditional double linked lists found for instance in the GCC STL implementation and in the LEDA library [7].

The remainder of the paper is organized as follows: In Sect. 2, we describe STL lists and the cache behavior of a traditional double linked implementation. The observations drawn there motivate the design of cache conscious STL lists that we present in Sect. 3. Our implementations are presented and experimentally analyzed in Sect. 4. Conclusions are given in Sect. 5.

2 Motivation for Cache Conscious STL Lists

A list in the STL library is a generic sequential container that supports forward and backward traversal using iterators, as well as single insertion and deletion at any iterator position in $O(1)$ time. Additionally, it offers internal sorting, several splice operations, and others (see further documentation in [8]). Finally, it must also be able to deal with an arbitrary number of iterators on it and ensure that operations cannot invalidate them. That is, iterators must point to the same element after any operation has been applied (except if the element is deleted).

In order to fulfill all these requirements, a classical double linked list together with pointers for iterators suffices. Indeed, this is what all known STL implementations do.

The key property of any pointer-based data structure as this is that even though the physical position of each element is permanent, its logical position can be changed just modifying the pointers in the data structure. Consequently, iterators are not affected by these movements.

Further, pointer-based data structures use memory allocators to get and free nodes. These allocators typically answer consecutive memory requests with consecutive addresses of memory (whenever possible). In the list case, if we add elements at one end (and no other allocations are performed at the same time), there is a good chance that logically consecutive elements are also physically consecutive, which leads to a good cache performance. However, if elements are inserted at random points or if the list is shuffled, logically consecutive elements will be rarely at physically nearby locations. Therefore, a traversal may incur in a cache miss per access, thus increasing dramatically the execution time.

In order to give evidence of the above statement, we have performed the following experiment with the GCC list implementation: Given an empty list, n random integers are pushed back one by one. Then, we measure the time to fully traverse it. Afterwards, we modify the list, and again we measure the time to fully traverse it. The modification consists either on sorting (thus randomly shuffling the links between nodes), or on k iterations of the so-called *k-insertion-erase* test. In the i-th iteration of this test ($1 \leq i \leq k$): first, the list is traversed and an element is inserted at each position with probability $1/(3+i)$, then the list is traversed again and each element is erased with probability $1/(4+i)$. Traversal times before modifying the list and after each kind of modification are shown in Fig. 1. Except for very small lists, it can be seen that the traversal of the shuffled list is about ten times slower than the traversal of the original list; and the only difference can be in the memory layout (and so, in the number of cache misses). Besides, note that four iterations of the insertion-erase test are enough to register half the worst case time.

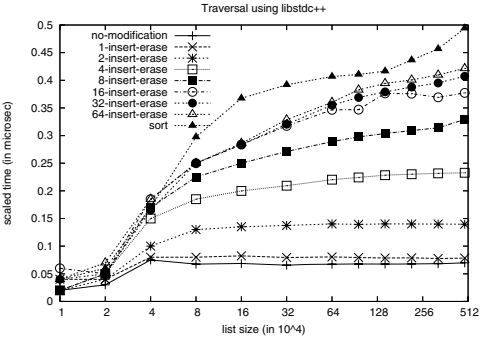


Fig. 1. Time measurements for list traversal before modifying it and after being modified in several ways. The vertical axis is scaled to the list size (that is, time has been divided by the list size before being plotted).

Taking into account that lists are used when elements are often reorganized (e.g. sorted) or inserted and deleted at arbitrary positions (if we only wished to perform insertions at the ends, we would better have used a vector, stack, queue or dequeue rather than a list), it is worth to try to improve the performance of lists using a cache conscious approach.

3 Design of Cache Conscious STL Lists

In this section we first consider previous work on cache conscious lists. Then, we present the main issues on combining them with STL list requirements. Finally, we present our proposed solutions.

3.1 Previous Work

Cache conscious lists have already been analyzed before; see a good summary in [5]. The operations taken into account are traversal (as a whole), insertion and deletion and their cost measured as the number of memory transfers.

Let be n the list size and B be the cache line size. The *cache aware* solution consists on a partition of $\Theta(n/B)$ pieces, each between $B/2$ an B consecutive elements, achieving $O(n/B)$ amortized traversal cost and constant update cost. The *cache oblivious* solutions are based on the packed memory structure [9], basically an array of $\Theta(n)$ size with uniformly distributed gaps. To guarantee this uniformity updates require $O((\log^2 n)/B)$, which can be slightly lowered by partitioning the array in smaller arrays. Finally, *self-organizing structures* [9] achieve the same bounds as the cache aware but amortized. There, updates breaking the uniformity are allowed until the list is reorganized when traversed.

Therefore, theory shows that cache conscious lists fasten scan based operations and hopefully, do not rise significantly update costs compared to traditional double linked lists. However, none of the previous designs take into account common requirements of software libraries. In particular, combining iterator requirements and cache consciousness rule out some of the more attractive choices.

3.2 Preliminaries

Before proceeding to the actual design, the main problems to be addressed must be identified. In our case, these concern to iterators. Secondly, it may be useful to determine common scenarios in which lists appear to guide the design.

Iterators concerns. In cache conscious structures, the physical and logical locations of an element are heavily related. In the case of STL list, this makes difficult to implement iterators trivially with pointers while enforces being able to reach iterators to keep them coherent whenever a modification in the list occurs.

The main issue is that an unbounded number of iterators can point to the same element. Therefore, $\Theta(1)$ modifying operations can be guaranteed only if the number of iterators is arbitrarily restricted, or if iterators pointing to the same element share some data that is updated when a modification occurs.

Hypotheses on common list usages. From our experience as STL programmers, it can be stated that a lot of common list usages are in keeping with the following:

- A particular list instance has typically only a few iterators on it.
- Given that lists are based on sequential access, many traversals are expected.
- The list is often modified and at any position: insertions, deletions, splices.
- The stored elements are not very big (e.g. integers, doubles, ...).

Note that the last hypothesis, which also appears implicitly or explicitly in general cache conscious data structures literature, can be checked in compile time. In case it did not hold, a traditional implementation could be used instead and this can be neatly achieved with template specialization.

3.3 Our Design

Our design combines cache efficient data access with full iterator functionality and (constant) worst case costs compliant with the Standard. Besides, our approach is specially convenient when the hypotheses on common list usages hold.

The data structure core is inspired by the cache aware solution previously mentioned (note that self-organizing strategies are not convenient here because STL lists are not traversed as a whole but step by step via iterators). Specifically, it is a double linked list of *buckets*. A bucket contains a small array of *bucket capacity* elements, pointers to the next and previous buckets, and extra fields to manage the data in the array. This data structure ensures locality inside the bucket, but logically consecutive buckets are let to be physically far.

Finally, it must be decided a) how to arrange the elements inside a bucket, b) how to reorganize the buckets when inserting or deleting elements, and c) how to manage iterators. Besides, the appropriate bucket capacity must be fixed (this has been studied experimentally, see end of Sect. 4.1).

a) *Arrangement of elements.* We devise three possible ways to arrange the elements inside a bucket:
   - *Contiguous*: The elements are stored contiguously from left to right and so, insertions and deletions must shift all the elements on its left or right.
   - *With gaps*: Elements are still stored from left to right but gaps between elements are allowed. In this way, we expect to reduce the average number of shifts. However, an extra field per element is needed to distinguish real elements from gaps. Additionally, more computation may be needed.
   - *Linked*: The order of the elements inside the bucket is set by internal links instead of the implicit left to right order. This requires extra space for the links, but avoids shifts inside the bucket. Thus, this solution is scalable for large bucket (and cache line) sizes.

b) *Reorganization of buckets.* The algorithms involved in the reorganization of buckets preserve the data structure invariant after an insertion or deletion. This includes: keeping a minimum bucket occupancy to guarantee the locality of accesses, preserving the arrangement coherency (e.g. if contiguous arrangement is used, gaps between the elements cannot be created),...

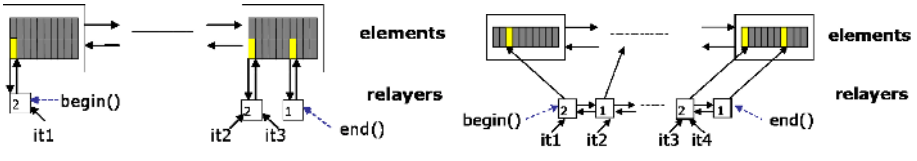
   The main issue is keeping a good balance between high occupancy, few bucket accesses per operation, and few elements movements. Besides, it should be guaranteed that no sequence of alternated insertions and deletions can toggle infinitely between creating and destroying a bucket. This is a must for performance unless we fully manage bucket allocation/deallocation.

c) *Iterator management.* Finally, it must be decided how iterators are implemented. Recall from Sect. 3.2 that this cannot be done trivially with pointers.

   Specifically, we have decided to identify all the iterators referred to an element with a dynamic node (relayer) that points to it. The relayer must be found in constant time and keep count of how many iterators are referring the element (so that it can be destroyed when there are none). Besides, we only need to update the relayer when the physical location of the element changes. We propose two possible solutions (see Fig. 2):

– *Bucket of pairs*: In this solution, for each element, a pointer to its relayer is kept. This is easy to do and still uses less space than a traditional double linked list because it needs two pointers per element.
– *2-level*: In this solution, we maintain a double linked list of active relayers. Note that $O(1)$ time access to the relayers can be guaranteed because STL lists are always accessed through iterators. This solution uses less space compared to the previous one (if there are not much iterators).

Unfortunately, the locality of iterator accesses decreases with the number of elements with iterators because relayers addresses can be unrelated. Anyway, dealing with just a few iterators is not a big matter because in particular, there is a good chance to find them in cache memory. In any case, our two approaches conform the Standard whatever the number of iterators.



(a) Bucket of pairs (b) 2-level list

Fig. 2. Standard compliant iterators policies

4 Performance Evaluation

We developed three implementations. Two of them use contiguous bucket arrangement, one of which uses *bucket of pairs* iterator solution and another *bucket of pairs*. The last implementation uses a linked bucket arrangement and the *2-level* iterator solution. All these can be found under http://www.lsi.upc.edu/∼lfrias/lists/lists.zip. Notice that in contrast to a flat double linked list, our operations deal with several cases and each of them with more instructions. This makes our code 3 or 4 times longer (in code lines).

In this section, we experimentally analyze the performance of our implementations and show their competitiveness in a lot of common settings.

The results are shown for a Sun workstation with Linux and an AMD Opteron CPU at 2.4 GHz, 1 GB main memory, 64 KB + 64 KB 2-associative L1 cache, 1024 KB 16-associative L2 cache and 64 bytes per cache line. The programs were compiled using the GCC 4.0.1 compiler with optimization flag `-O3`. Comparisons were made against the current STL GCC implementation and LEDA 4.0 (in the latter case the compiler was GCC 2.95 for compatibility reasons).

All the experiments were carried with lists of integers considering several list sizes that fit in main memory. Besides, all the plotted measurements are scaled to list size for a better visualization.

With regard to performance measures, we collected wall-clock times, that were repeated enough times to obtain significative averages (variance was always

observed to be very low). Furthermore, to get some insight on the behavior of the cache, we used Pin [10], a tool for the dynamic instrumentation of programs. Specifically, we have used a Pin tool that simulates and gives statistics of the cache hierarchy (using typical values of the AMD Opteron).

In the following we present the most significant results. Firstly, we analyze the behavior of lists with no iterators involving basic operations and common access patterns. Then, we consider lists with iterators. Finally, we compare our implementations against LEDA, and consider other hardware environments.

4.1 Basic Operations with No Iterator Load

Insertion at the back and at the front. Given an initially empty list, this experiment compares the time to get a list of n elements by successively applying n calls to either the `push_back` or `push_front` methods.

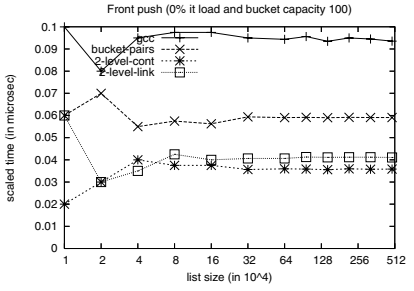
The results for `push_front` are shown in Fig. 3(a); a similar behavior was observed for `push_back`. In these operations, we observe that our three implementations perform significantly better than GCC. This must be due to manage memory more efficiently: firstly, the allocator is called only once for all elements in a bucket and not for every element. Secondly, our implementations ensure that buckets get full or almost full in these operations, and so, less total memory space is allocated.

Traversal. Consider the following experiment: First, build a list; then, create an iterator at its begin and advance it up to its end four times. At each step, add the current element to a counter. We measure the time taken by all traversals.

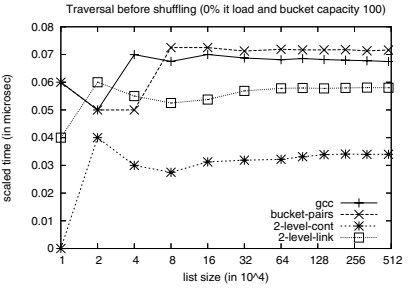
Here, the way to construct the list plays an important role. If we just build the list as in the previous experiment, the traversal times are those summarized in Fig. 3(b). These show that performance does not depend on list size and that our 2-level contiguous list implementation is specially efficient even compared to the other 2-level implementation. Our linked bucket implementation is slower than the contiguous implementation because, firstly, its buckets are bigger for the same capacity and so, there are more memory accesses (and misses). Secondly, the increment operation of the linked implementation requires more instructions.

Rather, if we sort this list before doing the traversals, and then measure the time, we obtain the results shown in Fig. 3(c). Now, the difference between GCC's implementation and ours becomes very significant and increases with list size (our implementation turns to be more than 5 times faster). Notice also that there is a big slope just beginning at lists with 20000 elements.

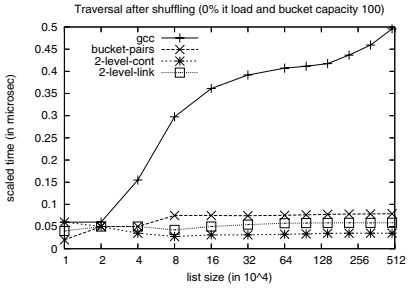
The difference in performance is due to the different physical arrangement of elements in memory (in relation to their logical positions). To prove this claim, we repeated the same experiment using the Pin tool, counting the number of instructions and L1 and L2 cache accesses and misses. Some of these results are given in Fig. 4(a). Firstly, these show that indeed our implementations incur in less caches misses (both in L1 and L2). Secondly, the scaled ratio of L1 misses is almost constant because even small lists do not fit in L1. Besides, the big slope in time performance for the GCC implementation coincides with a sudden rise



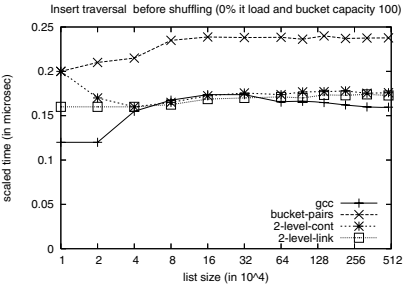
(a) `push_front`



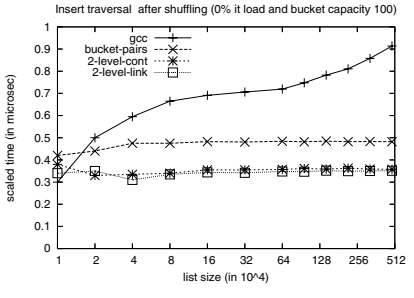
(b) Traversal before shuffling



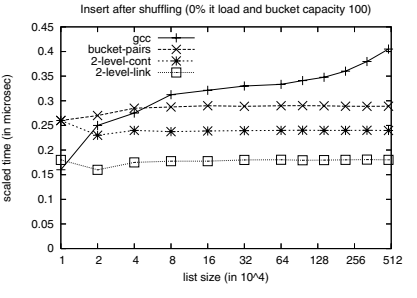
(c) Traversal after shuffling



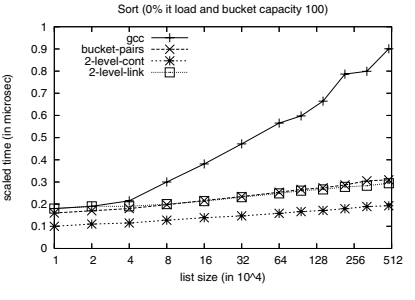
(d) Insertion before shuffling



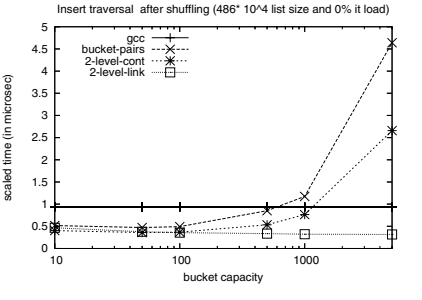
(e) Insertion after shuffling



(f) Intensive insertion

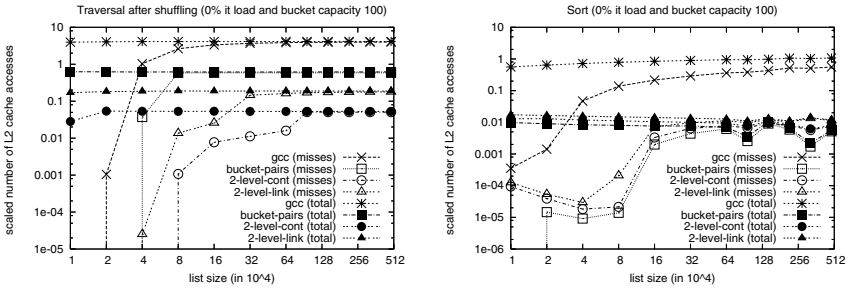


(g) Internal sort



(h) Effect of bucket capacity: Insertion after shuffling (list size 486000)

Fig. 3. Experimental results for basic operations with no iterator load



(a) Traversal (b) Internal Sort

Fig. 4. Simulation results on the cache performance (the vertical axis is logarithmic)

in L2 cache miss ratio, which leads to a state in which almost every access to L2 is a miss. This transition also occurs in our implementations, but much more smoothly. Nevertheless, the L2 access ratio (that is, L1 miss ratio) is much lower because logically close elements are in most cases in the same bucket and so, already in the L1 cache (because bucket capacity is not too big).

Insertion. In this experiment, we deal with insertions at arbitrary points. Firstly, a list is built (using the two abovementioned ways). Then, it is forwardly traversed four times. At each step, with probability $\frac{1}{2}$, an element is inserted before the current. We measure the time of doing the traversal plus the insertions.

Results are shown in Figs. 3(d) and 3(e), whose horizontal axis corresponds to the initial list size. Similar results were obtained with the erase operation.

Analogously to plain traversal, performance depends on the way the list is built. However, as in this case the computation cost is greater, the differences are smoother. In fact, when the list has not been shuffled, the bucket of pairs list performs worse than GCC's. Our two other implementations perform similarly to GCC's though. On the other hand, when the list has been shuffled, GCC's time highly increases, while ours is almost not affected.

It is interesting to observe that the linked arrangement implementation does not outperform the contiguous ones even though it does not require shifting elements inside the bucket. This must be due to the fact that more memory accesses (and misses) are performed and this is still dominant. This was confirmed performing the analogous Pin experiment. Instead, if an intensive insertion test is performed, in which a lot of insertions per element are done and almost no traversal is performed, then this gain is not negligible. This is shown in Fig. 3(f).

Internal sort. The STL requires an $O(n \log n)$ `sort` method that preserves the iterators on its elements. Our implementations use a customized implementation of merge sort.

Results of executing the sort method are given in Fig. 3(g). These show that our implementations are between 3 and 4 times faster than GCC. Taking into account that GCC also implements a merge sort, we claim that the significant speedup is due to the locality of data accesses inside the buckets. To confirm

this, Fig. 4(b) shows the Pin results. Indeed, GCC does about 30 times more cache accesses and misses than our implementations.

Effect of bucket capacity. The previous results were obtained for buckets with capacity of 100 elements. Anyway, this choice did not appear to be critical. Specifically, we repeated the previous tests with lists with other capacities, and observed that once the bucket capacity was not very small (less than 8-12 elements), a wide range of values behaved neatly. Note that a bucket of integers with capacity of 8 elements is yet 40-80 bytes long (depending on the implementation and address length) and a typical cache line is 64 or 128 bytes long.

To illustrate the previous claims, we show in Fig. 3(h) insertion results on a shuffled list with initially about 5 million elements. These show that for contiguous arrangement implementations, time decreases until a certain point and then starts to increase. In these cases, increasing the bucket size increases the intrabucket movements which finally results more costly than the achieved locality of accesses. In contrast, the linked arrangement implementation seems to be not affected because no such operations are performed, accesses of a bucket do not interfere between them, and our insert reorganization algorithm takes into account at most three buckets at a time.

If we perform the last test with several instances at the same time, a smooth rise in time for all implementations can be seen, in particular for big bucket capacities. In fact, it is common dealing with several data structures at the same time. In this case, some interferences within the different objects accesses can occur, which are more probable as the number of instances grows. Therefore, it is advisable to keep a relatively low bucket size.

4.2 Basic Operations with Iterator Load

Now, we repeat the previous experiments on lists that do have iterators on their elements. We use the term *iterator load* to refer to the percentage of elements of a list that have one or more iterator on them.

Results are shown for tests in which elements have already been shuffled, iterator loads range from 0% to 100% and a big list size is fixed (about 5 million elements) because then is crucial to manage data in the cache efficiently.

Traversal. When there are no iterators on the list, our implementations traversal is very fast because the increment operation is simple and elements are accessed with high locality. However, when there are some iterators, it may turn slower because the increment operation depends whether there are other iterators pointing to the element or its successor. In contrast, the increment operation on traditional double linked lists is independent of it, and so, performance must be not affected. When the list has not been shuffled, this is exactly the case.

In contrast, when the elements are shuffled, which changes iterators logical order, iterators accesses may score low locality. Results for this case are shown in Fig. 5(a), which show indeed that the memory overhead become the most important factor in performance. Nevertheless, the good locality of accesses to the

elements themselves makes our implementations more competitive than GCC's up to 80% iterator load even for relatively small lists (about 100000 elements).

Insertion. When an element is inserted in a bucket with several iterators, some extra operations must be done but are much less than in the traversal case in relative terms. Therefore, performance should be less affected.

Results are shown after the elements have been shuffled in Fig. 5(b).

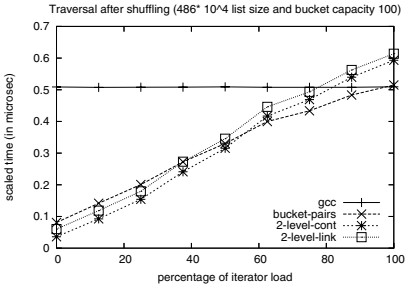
The results are analogous to the traversal test but with smoother slopes, as happened with no iterator load. Specifically, when the list has been shuffled, our implementations are more convenient up to 80% iterator load.

Internal sort. Guaranteeing iterators consistency in our customized merge sort is not straightforward, specially in the case of 2-level approaches that need some (though small) auxiliary arrays. Performance results are shown in Fig. 5(c).

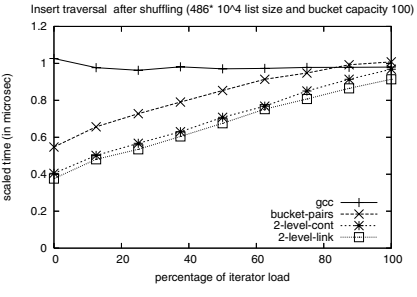
The results indeed show that the 2-level implementations are more sensitive to the iterator load. Anyway, any of our implementation are faster than GCC for iterators loads lower than 90%.

4.3 Comparison with LEDA

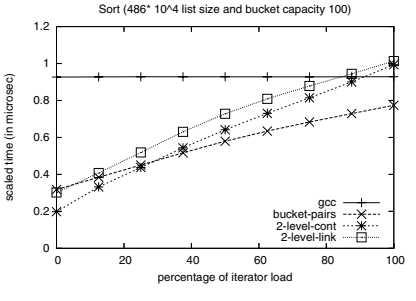
Here, we compare our lists with the LEDA well-known implementation, which uses a customized memory allocator. Although LEDA does not follow the STL, its interface is very similar and as GCC, it uses classical double linked lists.



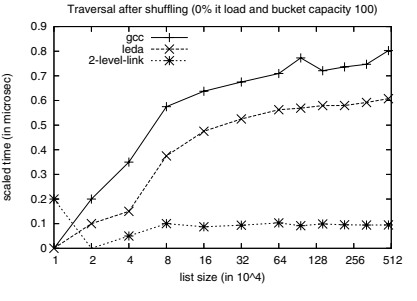
(a) Traversal after shuffling (I)



(b) Insertion after shuffling (I)



(c) Internal sort (I)



(d) Traversal after shuffling (L)

Fig. 5. Experimental results depending on the iterator load for a list of size $4.86{*}10^6$ (I) and LEDA results (L)

In Fig. 5(d), we show the results for traversal operation after shuffling. These make evident the limitations in performance of using a double linked list compared to our cache conscious approach. LEDA's times are just slightly better than GCC's, but remain worse than our implementations.

We omit the rest of plots with LEDA, because its results are just slightly better than GCC. The only exception is its internal sort (a quicksort) which is very competitive. Nevertheless, it requires linear extra space, does not keep iterators (items in LEDA jargon) and is not faster than ours.

4.4 Other Environments

The previous experiments have been run in a AMD Opteron machine. We have verified that the results we claim also hold on other environments. These include an older AMD K6 3D Processor at 450 MHz with a 32 KB + 32 KB L1 cache, 512 KB L2 off-chip (66 MHz) and a Pentium 4 CPU at 3.06 GHz, with a 8KB + 8KB L1 cache and 512 KB L2 cache. On both machines, similar results are obtained in relative terms, and better as newer the machine and compiler.

5 Conclusions

In this paper we have presented three cache conscious lists implementations that are compliant with the C++ standard library. Cache conscious lists were studied before but did not cope with library requirements. Indeed, these goals enter in conflict, particularly preserving both constant costs and iterators requirements.

This paper shows that it is possible to combine efficiently and effectively cache consciousness with STL requirements. Furthermore, our implementations are useful in many situations, as is shown by our wide range of experiments. The experiments compare our implementations against double linked list implementations such as GCC and LEDA. These show for instance that our lists can offer 5-10 times faster traversals, 3-5 times faster internal sort and even with an (unusual) big load of iterators be still competitive. Besides, in contrast to double linked lists, our data structure does not degenerate when the list is shuffled.

Further, the experiments show that the 2-level implementations are specially efficient. In particular, we would recommend using the linked bucket implementation, although its benefits only evince when the modifying operations are really frequent, because it can make more profit of eventually bigger cache lines.

Given that the use of caches is growing in computer architecture (in size and in number) we believe that cache conscious design will be even more important in the future. Therefore, we think that it is time that standard libraries take into account this knowledge. In this sense, this article sets a precedence but there is still a lot of work to do. To begin with, similar techniques could be applied to more complicated data structures. Moreover, current trends indicate that in the near future it will be common to have multi-threaded and multi-core computers. So, we should start thinking how to enhance these features in modern libraries.

References

1. International Standard ISO/IEC 14882: Programming languages — C++. 1st edn. American National Standard Institute (1998)
2. Cormen, T.H., Leiserson, C., Rivest, R., Stein, C.: Introduction to algorithms. 2 edn. The MIT Press, Cambridge (2001)
3. Lamarca, A.: Caches and algorithms. PhD thesis, University of Washington (1996)
4. Sen, S., Chatterjee, S.: Towards a theory of cache-efficient algorithms. In: SODA '00, SIAM (2000) 829–838
5. Demaine, E.: Cache-oblivious algorithms and data structures. In: EEF Summer School on Massive Data Sets. LNCS. (2002)
6. Frigo, M., Leiserson, C., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: FOCS '99, IEEE Computer Society (1999) 285
7. Mehlhorn, K., Naher, S.: LEDA — A platform for combinatorial and geometric computing. Cambridge University Press (1999)
8. Josuttis, N.: The C++ Standard Library : A Tutorial and Reference. Addison-Wesley (1999)
9. Bender, M., Cole, R., Demaine, E., Farach-Colton, M.: Scanning and traversing: Maintaining data for traversals in memory hierarchy. In: ESA '02. (2002) 152–164
10. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: PLDI '05, Chicago, IL (2005)

Engineering the LOUDS Succinct Tree Representation*

O'Neil Delpratt, Naila Rahman, and Rajeev Raman

Department of Computer Science, University of Leicester,
Leicester LE1 7RH, UK
{ond1, naila, r.raman}@mcs.le.ac.uk

Abstract. Ordinal trees are arbitrary rooted trees where the children of each node are ordered. We consider *succinct*, or highly space-efficient, representations of (static) ordinal trees with n nodes that use $2n + o(n)$ bits of space to represent ordinal trees. There are a number of such representations: each supports a different set of tree operations in $O(1)$ time on the RAM model.

In this paper we focus on the practical performance the fundamental Level-Order Unary Degree Sequence (LOUDS) representation [Jacobson, *Proc. 30th FOCS*, 549–554, 1989]. Due to its conceptual simplicity, LOUDS would appear to be a representation with good practical performance. A tree can also be represented succinctly as a balanced parenthesis sequence [Munro and Raman, *SIAM J. Comput.* **31** (2001), 762–776; Jacobson, *op. cit.*; Geary et al. *Proc. 15th CPM Symp.*, LNCS 3109, pp. 159–172, 2004]. In essence, the two representations are complementary, and have only the basic navigational operations in common (parent, first-child, last-child, prev-sibling, next-sibling).

Unfortunately, a naive implementation of LOUDS is not competitive with the parenthesis implementation of Geary et al. on the common set of operations. We propose variants of LOUDS, of which one, called *LOUDS++*, is competitive with the parenthesis representation. A motivation is the succinct representation of large static XML documents, and our tests involve traversing XML documents in various canonical orders.

1 Introduction

Ordinal trees are arbitrary rooted trees where the children of each node are ordered. We consider *succinct*, or highly space-efficient, representations of (static) ordinal trees with n nodes. An information-theoretically optimal representation of such trees would require $2n - O(\log n)$ bits. There are a number of representations that use $2n + o(n)$ bits of space, and support various navigational and other operations in $O(1)$ time on the RAM model of computation [6, 1, 5, 9].

This paper compares the practical performance of the fundamental *level-order unary degree sequence* succinct representation (hereafter LOUDS) [6] with non-succinct ordinal tree representations, as well as the *parenthesis* succinct representation (hereafter PAREN) [9], which supports a complementary set of operations to LOUDS. In practice, one must consider the lower-order terms in the space

* Delpratt is supported by PPARC e-Science Studentship PPA/S/E/2003/03749.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 134–145, 2006.

bound, which come from augmenting a bit-string of $2n + O(1)$ bits representing the tree with a number of *directories*, or auxiliary data structures, that are used to support operations in $O(1)$ time. The space used by each directory is, of course, asymptotically $o(n)$ bits, but is usually a function like $\Theta(n \log \log n / \log n)$ (and sometimes worse). For this and other reasons, the directories can use much more space than the representation of the tree, for practical values of n.

With this in mind, we consider the operations supported by the two succinct representations above, assuming a 'minimal' set of directories. Both support the basic navigational operations of parent, first-child, last-child, prev-sibling and next-sibling. However, LOUDS supports additional $O(1)$-time operations such as degree(x) (reporting the number of children of x), childrank(x) (the position of x among the children of its parent), child(x, i) (reporting the i-th child of x — recall that ordinal trees can have unbounded degree) and can enumerate all nodes at the same depth as a given node x, in time proportional to the number of such nodes. PAREN, on the other hand, readily supports operations such as desc(x) (report the number of children descended from x). LOUDS essentially numbers the nodes of the tree with integers in a level-order (breadth-first) numbering, while PAREN uses a depth-first (pre- or post- order) numbering.

The functionality of these 'minimal' representations can be expanded at negligible *asymptotic* cost. For example, PAREN can support degree(x) in $O(1)$ time by augmenting it with additional $o(n)$-bit directories [2], or level-ancestor queries in $O(1)$ time using yet another directory [10]. The *depth-first unary degree sequence* representation [1] comes close to being a 'union' of LOUDS and PAREN. However, its directories are also an (almost disjoint) union of the directories of both LOUDS and PAREN. Quite apart from the fact that none of these augmented data structures subsumes each other, adding additional directories may lead to poor practical performance. As noted above, directories consume significant space in practice. Different directories may have different memory access patterns, and adding additional ones can make it difficult to organise data to minimise cache misses. This motivates the study of alternative 'minimal' tree representations, such as LOUDS and PAREN, so that the one that best suits an application may be chosen, rather than a single 'universal' representation.

Although we defer a complete description of LOUDS to Section 3, we give a brief overview, in order to summarise the main issues and contributions. We first explain the task which we use to evaluate the data structures (the motivation is in the sub-section on XML below). We store, along with the tree, an array of size n, which stores a *satellite* symbol associated with each node. We traverse the nodes of the tree in pre-order, reverse pre-order and breadth-first order. As the traversal visits a node, we find the associated symbol in the array and gather some simple statistics (e.g. the number of nodes with a particular symbol). This set of tasks tests the first-child, last-child, prev-sibling and next-sibling operations[1], all of which are supported in $O(1)$ time by LOUDS and PAREN.

LOUDS stores an n-node ordinal tree as a bit-string of $2n + 1$ bits. Navigation on the tree is performed by rank and select operations on the bit-string:

[1] We currently use a recursive pre-order traversal, so parent is not tested.

$\mathsf{rank}_1(x)$ Returns the number of **1** bits to the left of, and including, position x in the bit-string.

$\mathsf{select}_1(i)$ Given an index i, returns the position of the i-th **1** bit in the bit-string, and -1 if $i \leq 0$ or i is greater than the number of **1**s in the bit-string.

The operations rank_0 and select_0 are defined analogously for the **0** bits in the bit-string; the operations are collectively referred to as rank and select. We refer to a data structure that supports (a nonempty subset of) rank and select operations on a bit-string as a *bit-vector*.

A bit-vector is a fundamental data structure and is used in many succinct and compressed data structures. A bit-vector that supports rank and select in $O(1)$ time can be obtained by augmenting a bit-string of length k with directories occupying $o(k)$ space [6, 3]. Unfortunately, rank and select, though $O(1)$-time asymptotically, are certainly not free in practice. Using the approach of [3] in practice is very slow [7]. In fact, Kim et al. [7] argue that their $3k + o(k)$-bit data structure is more practical than approaches based on [3]. Even in this well-engineered data structure, a select is over three times as slow as a rank, and a rank is somewhat slower than a memory access. Given this, it was perhaps not surprising that a direct implementation of LOUDS (using either of the bit-vectors of [4, 7]) was sometimes over twice as slow as the implementation of PAREN by [4], when parameters were chosen to make the space usages of the data structures somewhat similar. This rather negative result prompted our attempt at engineering LOUDS.

For LOUDS, Jacobson [6] suggested a numbering of nodes from 1 to $2n$. Using his numbering, parent, first-child and last-child all require just one call each to rank and select, and next-sibling and prev-sibling only require the inspection of a bit in the representation of the tree. As nodes are numbered from 1 to $2n$, rather than 1 to n, to access an array of size n that contains information associated with a given node, one has to perform a rank operation on its node number. We first observe that, due to the way rank and select calls are made in LOUDS, one may eliminate calls to rank altogether. The idea, called *double-numbering*, not only speeds up the navigational operations, it also numbers the nodes from 1 to n in level-order, making it easy to access information associated with a node. The resulting data structure, LOUDS1, is indeed much faster than LOUDS, but remains slower and more space-expensive than PAREN.

We then note that, in practice, ordinal trees have a high proportion of leaves: e.g., our XML trees have often about 67% leaves, and a random tree has about 50% leaves. Thus, a rapid test that determines whether a node is a leaf could speed up the first-child operation of LOUDS considerably in practice. We propose a numbering of nodes from 1 to $2n$, which is different from that of [6]. Using this numbering, testing whether a node is a leaf is quick. Applying double-numbering, we again require no rank operations to support navigation in this representation. In this scheme, parent and first-child (for non-leaf nodes), next-sibling and prev-sibling all require up to two select operations, compared with at most one in LOUDS1, and last-child requires one select. This data structure is called LOUDS0. Unfortunately, the performance advantages, such as speeding up first-child for

non-leaf nodes, does not gain enough to make up for the slow next-sibling and prev-sibling operations. The overall speed is poor for pre-order and BFS traversals, but is better (but still poor) for traversals in reverse pre-order.

Finally, we present a new variant of LOUDS, called LOUDS++, which partitions the bit-string representing the tree into two separate bit-strings, and stores them as bit-vectors. The advantages of LOUDS++ are:

- Testing if a node is a leaf, as well as next-sibling and prev-sibling, require only the inspection of bits in one of the bit-strings.
- The other navigational operations (first-child, last-child and parent) are slightly slower than LOUDS1, requiring a rank and a select_1 operation.
- The other operations, degree, childrank and $\mathsf{child}(x, i)$ are as easy as LOUDS1.
- Each of the bit-vectors only needs to support select_1 and rank, while the LOUDS1 bit-vector needs to support both select_1 and select_0 (but not rank, because of double-numbering). This gives significant space savings in practice, since the select directories are usually much larger; in addition, bit-vectors such as that of [7] need the rank directory to implement select.

The rest of this paper is as follows. Immediately following is a short background on XML files. Section 2 discusses bit-vectors, Section 3 introduces LOUDS, including all our variants. Section 4 contains our experimental results.

Representing XML Documents. Our motivation is in the use of this data structure for the representation of (large, static) XML documents. The correspondence between XML documents and ordinal trees is well-known (see e.g. Fig. 1). In this paper we focus on storing the tree structure. The XML Document Object Model (DOM) is a standard interface (see `www.w3.org`) through which applications can access XML documents. DOM implementations store an entire XML document in memory, with its tree structure preserved, but this can take many times more memory than the raw XML file. This 'XML bloat' seriously impedes the scalability and performance of XML query processors [12].

DOM allows tree navigation through the `Node` interface, which represents a single node in the tree. The node interface contains attributes to store information about the node, as well as navigational methods `parentNode`, `firstChild`,

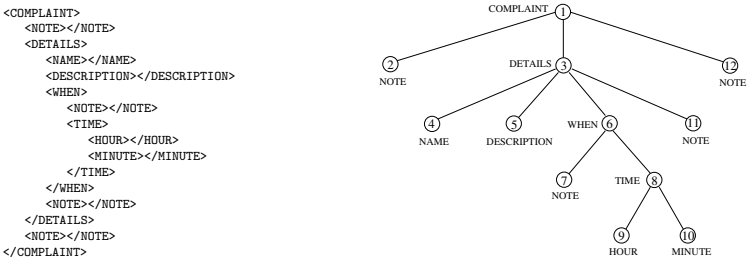


Fig. 1. Left: Small XML fragment (only tags shown). Right: Corresponding tree representation, nodes numbered in depth-first order.

`lastChild`, `previousSibling` and `nextSibling`. The usual way of storing the tree in DOM implementations is to store with each node a pointer to a (subset of) the parent, the first/last child, and the previous/next sibling. We store the tree succinctly, and simulate access to node information by accessing an array. Traversals are important primitives in DOM. There are two main orders of traversals: *document order*, which corresponds to pre-order, and *reverse document order*, which corresponds to reverse pre-order. There is no recognised equivalent of BFS traversal in DOM.

2 Bit-Vector Implementations

We now discuss the space usage of the two bit-vector implementations that we use. The formulae in Table 1 are implicit in [4, 7]. We break the space usage down into constituent parts, as this is important to understand the space-efficiency of LOUDS++. In what follows, k is the size of the bit-string to be represented. The implementations assume a machine with word-size 32 bits (and hence $k \leq 2^{32}$). The space usage figures given below do not include the space for pre-computed tables which are used to perform computations on short bit-strings of 8 or 16 bits (the 'four Russians' trick).[2]

Table 1 gives the space usage of the implementation of Clark-Jacobson bit-vector in [4]. The implementation has three parameters, L, B and s; we show the space usage when $B = 64$, $L = 256$ and $s = 32$. In Table 1, k_0 and k_1 are the numbers of **0**s and **1**s in the bit-string, and l_0 and l_1 are values that depend upon the precise bit-string (the number of so-called 'long gaps'). It is easy to show that $\max\{l_0, l_1\} \leq k/L$, but in practice, the number of long gaps is rather small in most cases (see Section 4). Note that since rank_0 trivially reduces to rank_1 and vice-versa, a single directory suffices to support both rank queries.

We now state the space usage of the 'byte-based' bit-vector Kim et al. [7], again broken down into its various components. Referring to their paper, the space usage of the the directory for select_1 comprises the space usage of its constituent components, including the *delimiter* bit-string and its rank directory (we use a block size of 64 rather than 32). The final terms (c_0 and c_1) are the space usages of the *clump delimiter* and the *clump array*. Their values depend upon the precise distribution of **0**s and **1**s in the bit-string. In the worst case, $c_1 \leq k_0 + 0.043k$, but (as the authors suggest and we confirm) it is much smaller than this upper bound. In order to support select_0 in addition to select_1, we augment the bit-string with a symmetric directory for indexing **0**s, whose space is shown in the last line. The similarity between the space bounds in Table 1 is highlighted in the following remark.

Remark 1. If $\mathsf{select}i$ is to be supported, for only one $i \in \{0, 1\}$, the space bound for the bit-vector implementations is of the form $k + fr(k) + fs(k) + gs(k_i) + hs_i(A)$,

[2] This may seem like cheating, but it is standard practice, since the size of tables is determined by factors such as the size of the cache, independently of k.

Table 1. The space usage of the two bit-vector implementations used

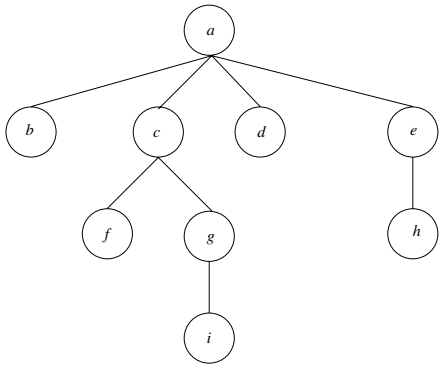
	Clark-Jacobson	Kim et al.
Input bit-string	k	k
$\mathsf{rank}_0/\mathsf{rank}_1$ directory	$0.5k$	$0.25k$
select_0 directory	$0.023k + k_0 + 1024l_0$	$1.25k_1 + c_1$
select_1 directory	$0.023k + k_1 + 1024l_1$	$1.25k_0 + c_0$

where fr, fs and gs are linear functions, indicating the space required for rank and two kinds of select directories respectively, and A is the input bit-string.

For example, if only rank and select_0 are to be supported, the Clark-Jacobson implementation of [4], as described in Table 1 has $fr(k) = 0.5k$, $fs(k) = 0.023k$, $gs(k_0) = k_0$, and $hs_0(A) = 1024l_0$.

3 The LOUDS Representation

The LOUDS bit-string (LBS) is defined as follows. We begin with an empty string. We visit every node in level-order, starting from the root. As we visit a node v with $d \geq 0$ children, we append $\mathbf{1}^d\mathbf{0}$ to the bit-string. Finally, we prefix the bit-string with a **10**, which is the degree of an imaginary 'super-root,' which is the parent of the root of the tree (see Figure 2).



(a)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**

(b)

Vertex	a	b	c	d	e	f	g	h	i
BFS	1	2	3	4	5	6	7	8	9
1-based	1	3	4	5	6	9	10	13	16
0-based	7	8	11	12	14	15	17	18	19

(c)

Vertex	a	b	c	d	e	f	g	h	i
$R0$	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**
$R1$		**1**	**0**	**0**	**0**	**1**	**0**	**1**	**1**	**1**

(d)

Fig. 2. An example ordinal tree (a) and its representations ((b)–(d)).(b) the LOUDS bit-string (LBS); the vertical bars in the LBS have been inserted for readability. The numbers above are the positions of the bits. The initial **10** is for the 'super-root'. (c) Zeros- and ones-based numberings. (d) Partitioned bitvector.

Proposition 1. *The LBS of a tree T with n nodes has n **1**s and $n + 1$ **0**s. The i-th node of T in level-order is represented twice: as the i-th **1**, which lies within the encoding of the degree of its parent, and is associated with the edge that attaches it to its parent, and also as the $i + 1$-st **0**, which marks the end of its own degree sequence.*

Ones-based numbering. Jacobson [6] suggests numbering the i-th node in level-order by the position of the i-th **1** bit. This gives a node a number from $\{1, \ldots, 2n+1\}$. To access data associated with a node numbered x, calculating $\mathsf{rank}_1(x)$ numbers the nodes from $\{1, \ldots, n\}$ in level-order.

Zeros-based numbering. Proposition 1 suggests that a node may also be represented by a **0** bit, namely the bit that ends the unary sequence of that node's degree. Again, this is a number from $\{1, \ldots, 2n+1\}$, and a rank_0 operation may be needed to map the nodes to numbers from $\{1, \ldots, n\}$. Figure 3 indicates how the navigational operations might work on the zero-based numbering. Although the operations seem more complex, with the notable exception of isleaf, there is hope that the practical performance may not be too poor, as many of the operations apply select_0 to consecutive zeros in the LBS. Also, next-sibling (prev-sibling) requires only one select for the last (first) child; there are as many last (first) children as non-leaf nodes.

Ones-based numbering

```
isleaf(x)
  See first-child

parent(x)
  select1(rank0(x))

first-child(x)
  y := select0(rank1(x))+1
  if A[y] = 0 then -1 else y

last-child(x)
  y := select0(rank1(x)+1)-1
  if A[y] = 0 then -1 else y

next-sibling(x)
  if A[x+1] = 0 then -1
```

Zero-based numbering

```
isleaf(x)
  (A[x-1] = 0) and (A[x] = 0)

parent(x)
  select0(rank0(select1(rank0(x)-1)+1))

first-child(x)
  if (isleaf(x)) then -1
  else select0(rank1(select0(rank0(x)-1))+2)

last-child(x)
  if (isleaf(x)) then -1
  else select0(rank1(x)+1)

next-sibling(x)
  y := select1(rank0(x)-1)+1
  if A[y] = 0 then -1 else select0(rank0(x))
```

Fig. 3. Navigation operations for zeros-based and ones-based numberings (A is the LBS). prev-sibling is analogous to next-sibling.

3.1 Double-Numbering

Both the ones-based and the zeros-based numberings benefit from the following proposition:

Proposition 2. *Computing* $y = \mathsf{select}_i(x)$, *for* $i = 0$ *or* 1, *also computes* $\mathsf{rank}_0(y)$ *and* $\mathsf{rank}_1(y)$.

Proof. If $y = \mathsf{select}_0(x)$ then $\mathsf{rank}_0(y) = x$ and $\mathsf{rank}_1(y) = y - x$. select_1 is similar.

This allows us to maintain the following invariant. We represent a node as a pair $\langle x, y \rangle$, where y is the position of the node in level-order, and x is the position of the representation of the node in the bit-string. Clearly, depending on whether the numbering is one-based or zero-based, $x = \mathsf{rank}_1(y)$ or $x = \mathsf{rank}_0(y) - 1$. It follows that all computations of the form $\mathsf{rank}(\mathsf{select}(\cdot))$ are really just select operations. Also, since the final step in any (nontrivial) navigation operation is always a select, it follows that the invariant can be maintained at the end of each navigational operation. For example, the call to rank_0 in the parent function in the ones-based numbering in Figure 3 can be implemented as follows:

```
parent(<x,y>)
  rzerox := y - x
  newy   := select1(rzerox)
  newx   := newy - rzerox
  return(<newx, newy>)
```

We refer to the ones-based and zeros-based representations with double-numbering as LOUDS1 and LOUDS0 respectively.

3.2 Partitioned Representation

We now describe a new representation that has the simplicity of LOUDS1 and also allows the isleaf test in $O(1)$ time. The idea is to encode the runs of zeros and ones in the LBS in two separate bit-strings, which we will call R0 and R1. Specifically, if there are runs of **0**s of length $l_1, l_2, \ldots, l_z$ in the LBS, then the bit-string R0 is simply $\mathbf{0}^{l_1-1}\mathbf{1}\mathbf{0}^{l_2-1}\mathbf{1}\ldots\mathbf{0}^{l_z-1}\mathbf{1}$. R1 is defined analogously. Noting that the LBS begins with a **1** and ends with a **0**, it is clearly possible to reconstruct it from R0 and R1. It is now trivial to access the i-th **1** or the $i+1$-st **0** that represents the node numbered i in level-order. This means, in particular, that operations such as isleaf are trivial: the node numbered x in level order is a leaf iff the x-th and $x+1$-st **0** belong to the same run of **0**s, which is easily tested by probing the appropriate bits of R0. Likewise next-sibling and prev-sibling are trivial to implement by looking at R1. LOUDS++ is simply R0 and R1, each augmented with directories to support select_1 and rank^- operations, where:

$\mathsf{rank}^-(x)$ returns the number of **1** bits strictly to the left of position x in the bit-vector. ($\mathsf{rank}^-(x) = \mathsf{rank}_1(x-1)$ except when $x = 1$.)

We now observe:

Proposition 3. select *operations on the LOUDS bit-vector can be simulated by a* select_1 *and a* rank^- *on R0 and R1.*

Proof. We claim that $\mathsf{select}_1(\mathrm{LBS}, i) = \mathsf{select}_1(\mathrm{R0}, \mathsf{rank}^-(\mathrm{R1}, i)) + i$. Note that $\mathsf{rank}^-(\mathrm{R1}, i)$ equals the number of completed runs of **1**s before the run that i is in. There must be an equal number of completed runs of **0**s before i. The select on R0 then gives the total length of these runs, which is then added to i to give the position of the i-th **1**. $\mathsf{select}_0(\mathrm{LBS}, i)$ is similar.

Corollary 1. LOUDS++ *supports the operations* parent, first-child *and* last-child.

Proof. We look at the implementation of these operations in LOUDS1. Due to double-numbering, these operations only have a single select call, which can be simulated as in Proposition 3.

Proposition 4. *The number of* **1***s in R0 and R1 is equal to the number of non-leaf nodes in the input tree plus one.*

Proof. A run of **1**s in the LBS is a node of degree > 0, i.e. a non-leaf node (with the exception of the super-root). The number of **1**s in R1 is the number of runs in the LOUDS bit-string. The number of runs of **0**s in the LBS equals the number of runs of **1**s.

This proposition is key to the good space usage of LOUDS++: not only do we need to support just select_1, but also, the number of non-leaf nodes is usually just a small fraction of the number of nodes. In particular, the (usually considerable) space usage represented by functions $gs()$ in Remark 1 is much reduced. The above representation also gives a nice bound on the number of non-leaf nodes in a random n-node ordinal tree:

Proposition 5. *For any constant $c > 0$, with probability greater than $1 - 1/n^c$, the number of non-leaf nodes in a random n-node ordinal tree is $n/2 \pm o(n)$.*

Proof. (outline) The bit-strings R0 and R1 can be represented using $\lg \binom{n}{t}$ bits, where t is the number of non-leaf nodes. If, for a random random ordinal tree, t deviates significantly from $n/2$, the bit-strings R0 and R1 can be represented using significantly less than n bits, thus giving a representation of the random tree's LBS that uses significantly less than $2n$ bits. However, a simple counting argument shows that no representation of an ordinal tree can represent a random ordinal tree using less than $2n - O(\log n)$ bits, with probability greater than $1 - 1/n^c$, for any constant $c > 0$.

4 Experimental Evaluation

To test our data structures we obtained ordinal trees from the following 6 real-world XML files: xcdna.xml and xpath.xml, which contain genomic data, and mondial-3.0.xml, orders.xml, nasa.xml and treebank_e.xml [14]. We also tested the data structures on randomly generated XML files. These were obtained by using the algorithm described in [11] to generate random parentheses strings. A random parentheses string was converted to an XML file by replacing the opening and closing parentheses of non-leaf nodes by opening and closing tags. The parentheses for leaf nodes were replaced with short text nodes. Our real-world and random files were selected to get some understanding of the behaviour of the data structures as the file size varied with respect to the size of the hardware cache and as the structure of the trees varied. In all cases, the type of each node (element, text node etc.) was stored as a 4-bit value in an accompanying array.

			LOUDS++				LOUDS0/LOUDS1				
			KNKP-BV		CJ-BV		KNKP-BV		CJ-BV		
File	nodes	%leaf	total	clump DS	total	long gap	total	clump DS	total	long gap	Paren
mondial-3.0	57372	78	3.12	0.07	3.82	0.34	5.11	0.11	5.65	0.55	3.73
orders	300003	50	3.78	0.03	5.64	1.60	5.07	0.07	5.10	NEG	3.73
nasa	1425535	67	3.37	0.05	4.27	0.57	5.09	0.09	5.42	0.33	3.73
xpath	2522571	67	3.37	0.04	3.99	0.27	5.08	0.08	5.63	0.53	3.73
treebank_e	7312612	67	3.37	0.04	3.77	0.06	5.08	0.08	5.10	0.01	3.73
xcdna	25221153	67	3.35	0.02	3.80	0.08	5.11	0.11	5.48	0.38	3.73
R62K	62501	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	3.73
R250K	250001	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	3.73
R1M	1000001	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	3.73
R4M	4000001	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	3.73
R16M	16000001	49	3.81	0.04	4.07	NEG	5.08	0.08	5.10	NEG	3.73

Fig. 4. Space Usage. Test file, number of nodes, %leaf node. For LOUDS++ and for LOUDS0/LOUDS1 together: total space per node and space per node for the clump data structure using KNKP-BV; total space per node and space per node to support long gaps using the CJ-BV. For PAREN: space per node, where the numbers are obtained using a generic formula, that does not take into account tree-specific parameters. In [4] this formula was shown to be quite accurate for a wide variety of XML files.

We used Centerpoint XML's DOM [13] implementation to parse the XML files. Our experiments were to traverse the trees and to count the total number of nodes of a particular XML type by accessing the nodetype array. We tested with three different types of traversal, breath-first order, *BFO*, recursive depth-first order, *DFO*, and recursive reverse depth-first order, *RDO*, where we first visit the last child at each nodes and then each of its previous siblings in turn. We compared the three LOUDS data structures with CenterPoint XML's DOM [13] and the PAREN implementation of [4].

We implemented the data structures in C++ and tested them on a dual processor Pentium 4 machine and a Sun UltraSparc-III machine. The Pentium 4 has 512MB RAM, 2.8GHz CPUs and a 512KB L2 cache, running Debian Linux. The compiler was g++ 3.3.5 with optimisation level 2. The UltraSparc-III has 8GB RAM, a 1.2GHz CPU and a 8MB cache, running SunOS 5.9. The compiler was g++ 3.3.2 with optimisation level 2.

For rank and select we used an optimised version of the Clark-Jacobson bit-vector [4], with $B = 64$ and $s = 32$. We refer to this as CJ-BV. We also implemented the bit-vector described in [7], which we refer to as the KNKP-BV. In this data structure we use 256-bit superblocks and 64-bit blocks.

Figure 4 summarises the space usage per node. We see that LOUDS++ generally uses less space than the other LOUDS data structures and with the KNKP-BV its space usage is competitive with the PAREN. Note that LOUDS++ using CJ-BV uses more space than LOUDS1 for the file orders.xml. The structure of the file is such that the number of long gaps in the partitioned bit-strings is relatively large, but there are no long gaps in the LBS.

The performance measure we report is the slowdown relative to DOM of the succinct data structures. We first determine which bit-vector to use. The table below gives the slowdown relative to DOM of LOUDS++ using the KNKP-BV and using the CJ-BV for a DFO traversal on a Pentium 4. The CJ-BV based LOUDS++ outperforms the KNKP-BV based data structure. We saw the same relative performance for LOUDS1 and LOUDS0 and for RDO and BFS traversals. This is not too surprising since the KNKP-BV was designed for sparse bit-vectors, the bit-vectors here are dense. In the remaining experimental results the LOUDS data structures use CJ-BV.

	mond	order	nasa	xpath	treeb	R62K	R250K	R1M	R4M	R16M
KNKP-BV	1.82	3.24	2.82	3.13	3.26	3.63	3.73	3.77	4.09	2.14
CJ-BV	1.46	2.15	2.18	2.23	2.53	2.78	2.84	2.93	3.12	1.73

We now consider RDO traversals. At each node DOM stores a pointer to the parent, first child and next sibling in the tree. So the operation `getLastChild()` requires a traversal across all the children and `getPrevSibling()` at the i-th child requires a traversal across $i-1$ children. At a node with d children DOM performs $O(d^2)$ operations. In the real-world files orders.xml, xpath.xml and treebank_e.xml there is at-least one node with over 2^{14} children and for these files the slowdown relative to DOM of the LOUDS data structure is 0 (to two decimal points), for the other real-world XML files it is between 0.14 and 0.45.

Figure 5 summarises the performance of the data structures for DFO and BFO traversals. We see that LOUDS++ is faster than LOUDS0 or LOUDS1. LOUDS++ is also almost always faster than the PAREN when comparing performance of the basic tree navigation operations.

	Pentium 4								Sun UltraSparc-III							
	DFO				BFO				DFO				BFO			
File	L1	L0	L++	Par	L1	L0	L++	Par	L1	L0	L++	Par	L1	L0	L++	Par
mond	1.99	2.96	**1.46**	1.67	1.08	1.08	**0.80**	0.94	2.47	3.80	**2.15**	2.27	1.91	2.77	1.73	**1.67**
order	2.48	4.04	**2.15**	2.20	1.83	1.83	**1.67**	1.69	1.34	2.35	1.51	**1.33**	0.80	1.25	0.85	**0.74**
nasa	2.80	4.30	**2.18**	2.24	1.38	1.38	**1.11**	1.29	1.20	1.94	**1.16**	1.17	0.66	1.00	0.67	**0.59**
xpath	2.83	4.37	**2.23**	2.29	2.15	2.15	**1.39**	1.60	1.20	1.98	**1.18**	**1.18**	0.71	1.04	0.69	**0.61**
treeb	3.02	4.92	**2.53**	2.62	1.28	1.28	**1.01**	1.44	1.22	1.92	**1.18**	1.29	0.65	0.97	**0.72**	**0.72**
xcdna									1.21	1.95	1.15	**1.13**	0.75	1.03	0.65	**0.61**
R62K	3.17	5.04	**2.78**	3.16	2.06	3.24	**1.75**	3.07	2.30	3.58	**2.40**	2.82	2.14	3.45	**2.32**	3.22
R250K	3.22	5.10	**2.84**	3.22	2.02	3.21	**1.71**	3.01	1.53	2.40	**1.60**	1.85	1.42	2.25	**1.54**	2.12
R1M	3.29	5.25	**2.93**	3.19	1.92	3.08	**1.69**	2.96	1.23	1.90	**1.31**	1.75	1.12	1.78	**1.21**	1.71
R4M	3.46	5.61	**3.12**	3.21	1.13	1.86	**0.97**	3.01	1.24	1.93	**1.34**	1.77	0.99	1.59	**1.08**	1.57
R16M	1.76	2.97	**1.73**	1.84	0.50	0.80	0.44	**0.30**	1.22	1.93	**1.32**	1.81	0.61	0.96	**0.65**	1.17

Fig. 5. Performance evaluation on Pentium 4 and Sun UltraSparc-III.: Test file, slowdown relative to DOM for depth-first order (DFO) and breath-first order (BFO) traversals for LOUDS1 (L1), LOUDS0 (L0), LOUDS++ (L++) all using CJ-BV and for PAREN. The fastest data structure for each result is set in bold font. DOM could not fit XCNDA.xml into the internal memory of the Pentium 4.

5 Conclusions and Future Work

We have presented a partitioned version of Jacobson's [6] LOUDS representation, called LOUDS++, that appears to outperform other succinct tree representations in practice. Although we have demonstrated experimentally that LOUDS++ uses less space than LOUDS, this could be understood on a firmer theoretical basis. It would be interesting to see whether the partitioning idea generalises to other applications.

Acknowledgement. We thank Richard Geary for useful discussions.

References

1. D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman and S. S. Rao, Representing trees of higher degree. *Algorithmica* **43** (2005), pp. 275-292.
2. Chiang, Y.-T., Lin, C.-C. and Lu, H.-I. Orderly spanning trees with applications. *SIAM Journal on Computing*, **34** (2005), pp. 924-945.
3. D. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. 7th ACM-SIAM SODA*, pp. 383–391, 1996.
4. R. F. Geary, N. Rahman, R. Raman and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. 15th Symposium on Combinatorial Pattern Matching*, Springer LNCS 3109, pp. 159–172, 2004.
5. R. F. Geary, R. Raman and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. 15th ACM-SIAM SODA*, pp. 1–10, 2004.
6. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, 549-554, 1989.
7. D. K. Kim, J. C. Na, J. E. Kim and K. Park. Efficient implementation of Rank and Select functions for succinct representation. In *Proc. 4th Intl. Wkshp. Efficient and Experimental Algorithms*, LNCS 3505, pp. 315–327, 2005.
8. J. I. Munro. Tables. In *Proc. 16th FST&TCS conference*, LNCS 1180, 37-42, 1996.
9. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Computing*, **31** (2001), pp. 762–776.
10. J. I. Munro, and S. S. Rao, Succinct representations of functions. In *Proc. 31st ICALP*, LNCS 3142, pp. 1006-1015, 2004.
11. H. W. Martin and B. J. Orr. A random binary tree generator. *Proceedings of the 17th ACM Annual Computer Science Conference*, pp. 33–38, 1989.
12. http://xml.apache.org/xindice/FAQ.
13. Centerpoint XML, http://www.cpointc.com/XML.
14. UW XML Repository. `http://www.cs.washington.edu/research/xmldatasets/`.
15. http://www.w3.org/TR/2000/CR-DOM-Level-2-20000510/traversal.html

Faster Adaptive Set Intersections for Text Searching

Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu

David R. Cheriton School of Computer Science,
University of Waterloo, Waterloo, ON N2L 3G1, Canada
{jbarbay, alopez-o, ttlu}@uwaterloo.ca

Abstract. The intersection of large ordered sets is a common problem in the context of the evaluation of boolean queries to a search engine. In this paper we engineer a better algorithm for this task, which improves over those proposed by Demaine, Munro and López-Ortiz [SODA 2000/ALENEX 2001], by using a variant of interpolation search. More specifically, our contributions are threefold. First, we corroborate and complete the practical study from Demaine *et al.* on comparison based intersection algorithms. Second, we show that in practice replacing binary search and galloping (one-sided binary) search [4] by interpolation search improves the performance of each main intersection algorithms. Third, we introduce and test variants of interpolation search: this results in an even better intersection algorithm.

Topics: Evaluation of Algorithms for Realistic Environments, Implementation, Testing, Evaluation and Fine-tuning of Algorithms, Information Retrieval.

1 Introduction

The intersection of large ordered sets is a common problem in the context of the evaluation of relational queries to databases as well as boolean queries to a search engine. The worst case complexity of this problem has long been well understood, dating back to the algorithm by Hwang and Lin from over three decades ago [13]. In 2000, Demaine *et al.* improved over this by proposing a faster method for computing the intersection of k sorted sets [7] using an adaptive algorithm. Their algorithm has optimal worst-case behaviour on a much finer analysis than simply worst-case input size. We refer the reader to [7] for the precise details on the adaptive measure used.

In a followup study they showed that the adaptive theoretical optimal algorithm is not always best in practice in the context of search engines [8]. In that study, they compared a straightforward implementation of an intersection algorithm, termed `SvS`, with their adaptive algorithm, termed `Adaptive`, and showed that on the given data `Adaptive` is superior only for queries involving two or three terms, while thereafter `SvS` outperforms it by a constant factor. Their study uses what at the time was a sizable collection of plain text from

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 146–157, 2006.

web pages. Using this data set, Demaine *et al.* engineered an algorithm, termed `Small Adaptive`, that combines the best aspects of both `Adaptive` and `SvS`. They showed experimentally that on the given data set this algorithm outperforms both the `Adaptive` and `SvS` algorithm.

In this paper we revisit that study. Our contributions are threefold. First, we corroborate the practical study from [8] by considering a much larger web crawl and extend their study to include a more recent algorithm, introduced in [3]. The results are similar to those of the original study: the algorithm termed `Small Adaptive` is the one which performs the best. Second, we study the impact of replacing binary searches and galloping (one-sided binary) searches [4] by interpolation searches, for each of the main intersection algorithms. Our results show that this improves the performance of each intersection algorithm. The optimal algorithm, `Interpolation Small Adaptive`, is based on `Small Adaptive`, and our results show that the relative performance of the intersection algorithms are the same when using interpolation search than when using binary search and galloping. Third, we introduce several parameterized variants of extrapolation search, which combine the concepts of interpolation search and galloping, taking advantage of both. We evaluate the performance of each of those variants using `Small Adaptive` as a base, and we identify the best variant, termed `Extrapolate Ahead Small Adaptive`, which at each step computes the position of the next comparison using the values of elements at distance l of each other, and which performs the best when l is logarithmic on the size of the set. This results in an intersection algorithm which performs even better in practice than simply introducing interpolation.

The paper is structured as follows: in the next section we describe the data set on which we evaluated the various algorithms discussed. In Section 3 we describe in detail the intersection algorithms studied, and the basis of the interpolation algorithms. In Section 4 we present our experimental results. We conclude in Section 5 with a summary of the results.

2 Dataset

The intersection of sets in the context of search engines is a driving application for this work. Thus we test our algorithms using a web crawl together with a representative query log from a search engine. Each set corresponds to a keyword occurring in a query, and the elements of each set refer to integer document identifiers of those web pages containing the keyword. We use a sample web corpus from Google of 6.85 gigabytes of text as well as a 5000 entry query log, also from Google. The query log is the same as in [8], while the web crawl is a substantially larger and more recent data set. In the past we empirically verified that the relative performance of the algorithms did not change when run on corpora varying in size by orders of magnitude. Our results using this new larger set are consistent with this observation.

The web corpus was indexed into an inverted word index, which lists a set of document identifiers in increasing order for each word appearing in the corpus. The total number of web pages indexed is approximately 600,000. The size of the

resulting inverted word index is 1.06 gigabytes with HTML markup removed, and the number of words in the index is 2,604,335. Note that words consists of only alphanumerical characters.

In the sample query log from Google, we do not consider queries that contain words not found in our index nor queries that consists of a single keyword since no set intersection need be performed in this case. We refer the reader to [8] for a more thorough discussion on the query log.

3 Algorithms

3.1 Intersection Algorithms

Various algorithms for the intersection of k sets have been introduced in the literature [3, 7, 8]. In this study we focus on four particular ones, described below. We do not consider, however, the most naïve sequential (linear merging) algorithm as both theoretical and experimental analysis show that its performance in the comparison model is significantly worse than the ones studied here.

Algorithm 1. Pseudo-code for `Adaptive`

```
1: Choose eliminator e = set[0][0], in the set elimset ← 0.
2: Consider the first set, i ← 1
3: while the eliminator e ≠ ∞ do
4:    perform one step of the galloping search in set[i].
5:    if the gallop overshot then
6:       binary search in set[i] for e.
7:       if e was found then
8:          increase the occurrence counter, and let i ← i + 1 mod k, i ≠ elimset.
9:          if the value of occurrence counter is k then
10:            output e and let e ← set[i][succ(e)],   elimset ← i
               i ← i + 1 mod k, i ≠ elimset.
11:         else
12:            set e to the first element in set[i] which is larger than e.
13:            update the set elimset ← i and consider the next set i ← i+1 mod k, i ≠
               elimset.
14:         end if
15:      end if
16:   end if
17: end while
```

The theoretical study in [7] introduced an information theoretical optimum algorithm, which was implemented in [8] under the name `Adaptive`. This algorithm performs a search in *all* other sets for an element from one set, using a one-sided binary search or "galloping" search. The element being searched for is updated using a greedy technique. For the details we refer the reader to [7].

The experimental study in [8] introduced more algorithms, simulating fourteen different algorithms to study their practical performance on a query set provided by Google and a data set obtained through their own web crawl. Of those, we focus on two particular ones: `SvS` and `Small Adaptive`. `SvS` is a straightforward algorithm widely used in practice, which intersects the sets two at a time in increasing order by size, starting with the two smallest. It uses a binary search procedure to determine if an element in the first set appears in the second set.

`Small Adaptive` is a hybrid algorithm, which combines the best properties of `SvS` and `Adaptive`. For each element in the smallest set, it performs a galloping

Algorithm 2. Pseudo-code for `SvS`

Sort the sets by size ($|set[0]| \leq |set[1]| \leq \ldots \leq |set[k]|$).
Let the smallest set $s[0]$ be the candidate answer set.
for each set $s[i]$, $i = 1 \ldots k$ **do** initialize $\ell[k] = 0$.
for each set $s[i]$, $i = 1 \ldots k$ **do**
  for each element e in the candidate answer set **do**
    binary search for e in $s[i]$ in the range $\ell[i]$ to $|s[i]|$,
    and update $\ell[i]$ to the last position probed in the previous step.
    if e was not found **then**
      remove e from candidate answer set, and advance e to the next element in the answer set.
    end if
  end for
end for

Algorithm 3. Pseudo-code for `Small Adaptive`

Sort the sets by size ($|set[0]| \leq |set[1]| \leq \ldots \leq |set[k]|$).
Choose an eliminator $e = set[0][0]$ in the set $elimset \leftarrow 0$.
Consider the first set, $i \leftarrow 1$.
while the eliminator $e \neq \infty$ **do**
  gallop once in $set[i]$.
  if the gallop overshot **then**
    binary search in $set[i]$ for e.
    if e was found **then**
      increase the occurrence counter and let $i \leftarrow i + 1 \bmod k, i \neq elimset$.
      if the value of occurrence counter is k **then**
        add e to answer.
        resort the sets, and let $e \leftarrow set[0][succ(e)]$, $elimset \leftarrow 0$, $i \leftarrow 1$
      end if
    else
      resort the sets.
      if $i = 0$ or $i = 1$ **then** consider the set $i \leftarrow 1 - i$,
      else consider the first set: $elimset \leftarrow 0$, $i \leftarrow 1$. **end if**
    end if
  end if
end while

one-sided search on the second smallest set. If a common element is found, a new search is performed in the remaining $k-2$ sets to determine if the element is indeed in the intersection of all sets, otherwise a new search is performed. Observe that the algorithm computes the intersection from left to right, producing the answer in increasing order. After each step, each set has an already examined range and an unexamined range. `Small Adaptive` selects the two sets with the smallest unexamined range and repeats the process described above until there is a set that has been fully examined.

Algorithm 4. Pseudo-code for `Sequential`

Choose an eliminator $e = set[0][0]$, in the set $elimset \leftarrow 0$.
Consider the first set, $i \leftarrow 1$.
while the eliminator $e \neq \infty$ **do**
  Gallop for e in $set[i]$ till overshot
  binary search in $set[i]$ for e
  if the binary search found e **then**
    increase the occurrence counter.
    if the value of occurrence counter is k **then** output e **end if**
  end if
  if the value of the occurrence counter is k, or e was not found **then**
    update the eliminator to $e \leftarrow set[i][succ(e)]$.
  end if
  Consider the next set in cyclic order $i \leftarrow i + 1 \bmod k$.
end while

The theoretical study in [3] introduces a fourth algorithm, called `Sequential`, which is optimal for a different measure of difficulty, based on the non-deterministic complexity of the instance. It cycles through the sets performing one entire gallop search at a time in each (as opposed to a single galloping *step* in `Adaptive`), so that it performs at most k searches for each comparison performed by an optimal non-deterministic algorithm.

The pseudo-code for the algorithms described above is given in Algorithms 1 to 4. Each of those algorithms has linear time worst case behaviour, and each performs better than the others on at least one instance. `Adaptive` performs well on instances with an intersection certificate that can be encoded in a small amount of space, while `Sequential` performs well on instances whose intersection certificate contains a small number of comparisons. `SvS` reduces the number of sets by intersecting the two smallest sets, searching for the elements of the smallest set in the larger set; `Small Adaptive` performs similarly so long as no element is found to be in the intersection of the two sets, at which point it checks for it in the other sets, and after which it updates which sets are the smallest. Note that `Small Adaptive` and `SvS` are the only algorithms taking active advantage of the difference of sizes of the sets, and that `Small Adaptive` is the only one which takes advantage of how this size varies as the algorithm eliminates elements: `Adaptive` and `Sequential` ignore this information.

All of these algorithms are based on galloping and binary search, and use only comparisons between the elements: we study the impact on the performance of replacing those searches with interpolation search, or a suitably engineered variant of interpolation search, as described in the next section.

3.2 Search Algorithms

Interpolation search has long been known to perform significantly better than binary search on data randomly drawn from a uniform distribution, hence it is only natural to test if this holds using web crawled data. Moreover, recent developments suggest that interpolation search is also a reasonable technique for non-uniform data [6]. Our experiments, which we describe in the next Section, confirm this conjecture.

Recall that interpolation search for an element of value e in an array $set[i]$ on the range a to b probes a position as given by the formula:

$$I(a,b) = \left\lfloor \frac{e - set[i][a]}{set[i][b] - set[i][a]} \right\rfloor + a$$

In each of `Adaptive`, `Small Adaptive` and `Sequential` we replace each galloping step by an interpolation probe, and we replace binary search with interpolation search. In essence, the two changes are equivalent to performing an interpolation search in $set[i]$ for the eliminator. The index probed is $I(\ell[i], n_i)$, where $\ell[i]$ is the current position in $set[i]$ and n_i is the size $|set[i]|$ of $set[i]$.

4 Experimental Results

We compare the performance of each of the four algorithms described in the previous section by focusing on the number of comparisons performed by the algorithms. For large data sets such as in search engines, the run time is dominated by external memory accesses. It has long been known that the number of comparisons by an algorithm generally shows high correlation with the number of I/O operations, so we follow this convention. Our model has certain other simplifications; for example posting sets are likely to be stored in a compressed form, albeit one suitable for random access. We posit that most such refinements and other system specific improvements are likely orthogonal to the *relative* performance of the search algorithms presented here (see for example [5] for a discussion of these issues).

4.1 Comparing Intersection Algorithms

Here we present the part of our study which corroborates the study of [8], as we measure the performance of the algorithms on a larger data set, and completes it as we compare one more intersection algorithm (`Sequential`).

Figures 1 and 2 show that, when using binary search, `Small Adaptive` outperforms `Sequential`.

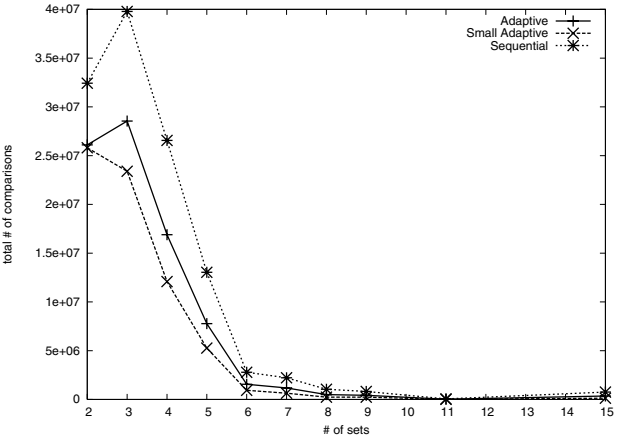


Fig. 1. Performance of various Intersection algorithms when using binary search

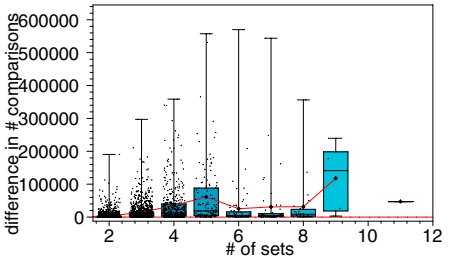


Fig. 2. `Small Adaptive` (high wins) vs. `Sequential` (low wins). The algorithm `Small Adaptive` is always better.

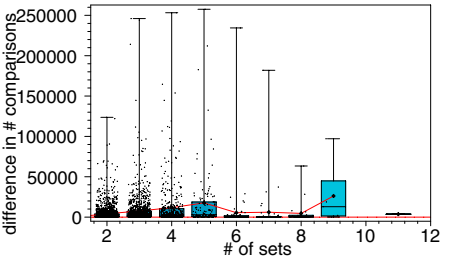


Fig. 3. `Interpolation Small Adaptive` (high wins) vs. `Small Adaptive` (low wins). Interpolation improves on all instances, and is consistent over k.

4.2 Comparing Interpolation and Binary Search

Here we present a first approach of the impact of replacing the binary searches and galloping by interpolation searches in the intersection algorithms. It is well known that interpolation search outperforms binary search, on average on arrays whose elements are well behaved (uniformly distributed). Thus it is expected that replacing binary search by interpolation search would improve the performance of the intersection algorithms. As gallop search [4] is a local search algorithm, it is not necessarily outperformed by interpolation search: we show here that in practice it is.

Figure 3, 4 and 5 show the clear advantage of using interpolation search over binary search, as each of the three intersection algorithm using interpolation search has a clear advantage over its variant using binary search, outperforming it on almost all instances.

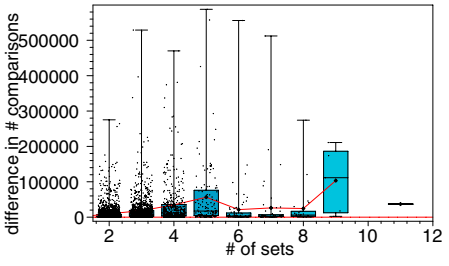


Fig. 4. `Interpolation Sequential` (high wins) vs. `Sequential` (low wins). Interpolation search provides roughly a two-fold improvement.

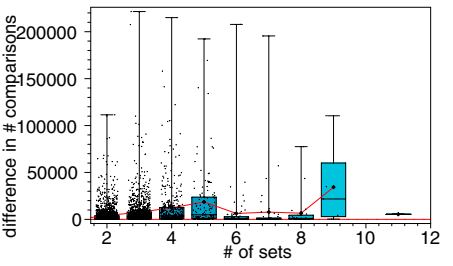


Fig. 5. `Interpolation Adaptive` (high wins) vs. `Adaptive` (low wins). The improvement is more noticeable if k is smaller.

As a side note, the study of the *ratio* of the performances (not shown here because of space limitations) shows that the ratio between the performance of `Interpolation Adaptive` and `Adaptive`, while always larger than one, decreases when k increases. This is likely due to the fact that the algorithm continually cycles through the sets trying to find a set which does not contain the eliminator [8]. Thus, the overhead caused by the cycling, which performs one interpolation going through each set (as opposed to galloping), is dominating the number of comparisons when k is relative large. Note that, in contrast, since `Small Adaptive` does not cycle through the sets, the average ratio between the performance of `Small Adaptive` and `Interpolation Small Adaptive` stays fairly constant with respect to k.

The experiments suggest that web crawled data is amenable to interpolation search, and hence using this technique gives a noticeable reduction in the number of comparisons required.

4.3 Introducing and Comparing Extrapolation Variants

In this section, we introduce an adaptation of interpolation search, which we named *extrapolation*, and some variants of it. We test those variants on our data set. Interestingly, our experimental results show that the difference in performance between search algorithms is independent of the intersection algorithm chosen. Since `Small Adaptive` is the fastest algorithm among those tested in [8] (when using binary search) and in our measures (when using binary search as well as when using interpolation search), we use it as a reference to show the performance of different interpolation techniques (See Figure 6).

The first variant, which we call `Extrapolation Small Adaptive`, involves extrapolating on the current and previous positions in $set[i]$. Specifically, the extrapolation step probes the index $I(p'_i, p_i)$, where p'_i is the previous extrapolation probe. This has the advantage of using "explored data" as the basis for calculating the expected index: this strategy is similar to galloping, which uses the previous jump value as the basis for the next jump (i.e. the value of the next jump is the double of the value of the current jump). Figure 7 shows that

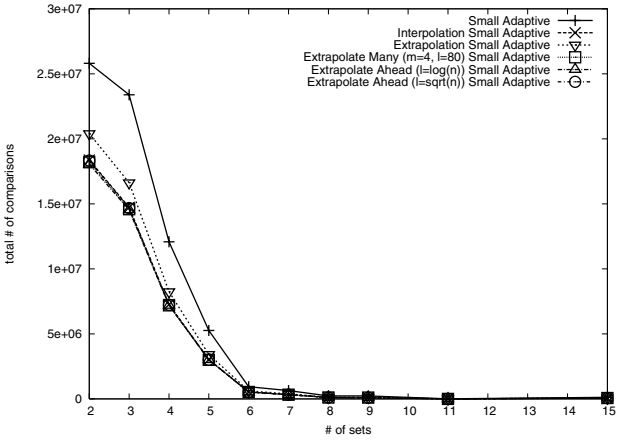


Fig. 6. Relative performance of search algorithms in `Small Adaptive`: binary search is outperformed by both interpolation search and Extrapolation-based algorithms

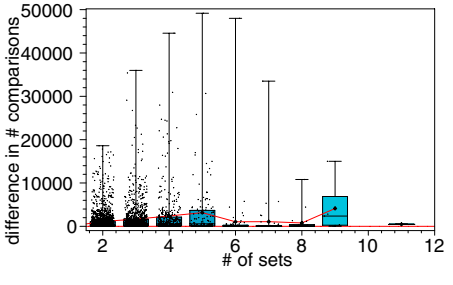


Fig. 7. `Extrapolation` (low wins) vs `Interpolation` (high wins). On `Small Adaptive`, using extrapolation on previous explored data is less accurate.

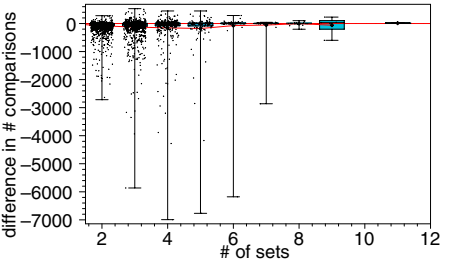


Fig. 8. `Extrapolate Ahead` ($l = 50$) (low wins) vs. `Interpolation` (high wins). On `Small Adaptive`, the look-ahead range improves the performance.

extrapolation alone does worse than interpolation. Those results suggest that using the previous "explored data" for extrapolation is not as accurate as using a standard interpolation probe, given by $I(p_i, n_i)$, on the remaining elements in $set[i]$.

The second variant, `Extrapolate Ahead Small Adaptive`, is similar to `Extrapolation Small Adaptive`, but rather than basing the extrapolation on the current and previous positions, we base it on the current position and a position that is further ahead. Thus, our probe index is calculated by $I(p_i, p_i + l)$ where l is a positive integer that essentially measures the degree to which the extrapolation uses local information. The algorithm uses the local distribution as a representative sample of the distribution between $set[i][p_i]$ and the eliminator: a large value of l corresponds to an algorithm using more global information, while a small value of l correspond to an algorithm using only local information. If the index of the successor $succ(e)$ of e in $set[i]$ is not far from p_i, then

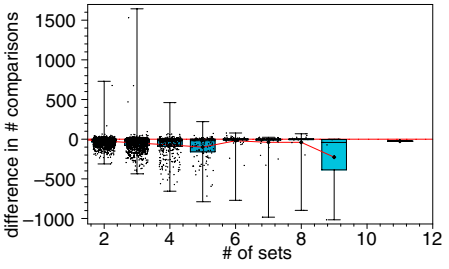


Fig. 9. `Extrapolate Ahead` ($l{=}\sqrt{n_i}$) (low wins) vs. `Interpolation Small Adaptive` (high wins). Deterioration of performance for a polynomial look-ahead range.

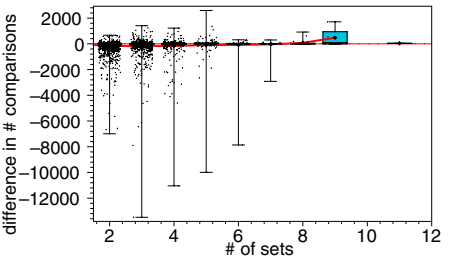


Fig. 10. `Extrapolate Ahead` ($l{=}\lg n_i$) (low wins) vs. `Interpolation Small Adaptive` (high wins). More complex results with a logarithmic look-ahead range.

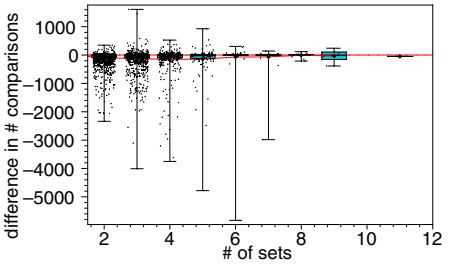


Fig. 11. `Extrapolate Many` ($m{=}4$, $l{=}80$) (low wins) vs `Interpolation Small Adaptive` (high wins)

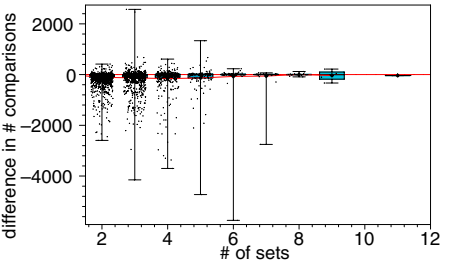


Fig. 12. `Extrapolate Many` ($m{=}8$, $l{=}80$) (low wins) vs `Interpolation Small Adaptive` (high wins)

the distribution between $set[i][p_i]$ and $set[i][p_i + l]$ is expected to be similar to the distribution between $set[i][p_i]$ and $set[i][succ(e)]$, and the estimate will be fairly accurate. Figure 8 shows that for $l = 50$, `Extrapolate Ahead Small Adaptive` performs as well as `Interpolation Small Adaptive`, and that their performance stays close when it is worse. Figure 9 shows a similar result for $l = \sqrt{n_i}$.

Figure 10 shows that choosing a smaller value for the look-ahead range l, such as $l = \lg n_i$, deteriorates slightly the performance: the algorithm has a much less precise approximation of the distribution of the values in the array.

The third variant involves extrapolating many times, which we call `Extrapolate Many Small Adaptive`. We calculate the index by taking the average of several extrapolations, which is based on the current position and several positions ahead. That is, our probe index can be calculated by $\frac{1}{m}\sum_{j=1}^{m} I(p_i, p_i + j\frac{l}{m})$, where m is the number of times we extrapolate and l is the farthest reach of the extrapolations. This has the advantage of a more accurate extrapolation and could result in less comparisons. Figures 11 and 12 show that it is not the case, as `Interpolation Small Adaptive` is still better, if only by a small mar-

gin. This is perhaps due to the fact that the extrapolations with larger values of j in $I(p_i, p_i + j\frac{l}{m})$ is more accurate than those with smaller values of j, thus when taking the average of all extrapolations, the ones with small values of j contribute more to the inaccuracy of the estimate.

5 Conclusions and Open Questions

We showed that using binary search, the intersection algorithm `Small Adaptive` outperforms all the other intersection algorithms including `Sequential` the most recent intersection algorithm proposed in the theory community, which heretofore had not been compared in practice. Our results also confirm the superiority of `Small Adaptive` over all other algorithms as reported in [8], even on a data set substantially larger than the one used in that study. Considering variants of those intersection algorithms using interpolation search instead of binary search and galloping, we showed that for any fixed intersection technique, such as `Small Adaptive`, using interpolation search always improves the performance. Finally, we combine the two concepts of interpolation search and galloping to define the extrapolation search and several variants of it. Comparing the practical performance of these on the intersection algorithm `Small Adaptive`, we found one that is particularly effective. This results in an even better intersection algorithm, termed `Extrapolate Ahead Small Adaptive`, which at each step computes the position of the next comparison using the values of elements at distance l of each other, and which performs the best when $l = \lg n_i$.

For completeness we summarize the results across all algorithms on the whole data set in Table 1. We would like to highlight some further experiments and open questions. First, it would be interesting to run the experiments over other data, such as the TREC corpus, particularly on the web slice of the collection. Second, to measure actual running times as opposed to the on number of comparisons alone. We expect that I/O and caching effects would

Table 1. Total number of comparisons performed by each algorithm over the data set. `Extrapolate Ahead Small Adaptive` with look-ahead range $l = \lg n$ is best.

Algorithm	# of comparisons
`Sequential`	119479075
`Adaptive`	83326341
`Small Adaptive`	68706234
`Interpolation Sequential`	55275738
`Interpolation Adaptive`	58558408
`Interpolation Small Adaptive`	44525318
`Extrapolation Small Adaptive`	50018852
`Extrapolate Many Small Adaptive` ($m = 4$, $l = 80$)	44119573
`Extrapolate Many Small Adaptive` ($m = 8$, $l = 80$)	44087712
`Extrapolate Ahead Small Adaptive` ($l = 50$)	44133783
Extrapolate Ahead Small Adaptive ($l = \lg n$)	43930174
`Extrapolate Ahead Small Adaptive` ($l = \sqrt{n}$)	44379689

have a significant impact on the reported times of each algorithm. Third, to study a broader range of intersection algorithms, as some combining the techniques proposed in [1, 2] with orthogonal techniques from other intersection algorithms.

References

1. Ricardo A. Baeza-Yates. A Fast Set Intersection Algorithm for Sorted Sequences. In *Proceedings of 15th Annual Symposium on Combinatorial Pattern Matching* (CPM), 400–408, 2004.
2. Ricardo A. Baeza-Yates, Alejandro Salinger. Experimental Analysis of a Fast Intersection Algorithm for Sorted Sequences. In *Proceedings of 12th International Conference on String Processing and Information Retrieval* (SPIRE), 13–24, 2005.
3. Jérémy Barbay and Claire Kenyon. Adaptive Intersection and t-Threshold Problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), 390–399, 2002.
4. Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.
5. Daniel K. Blandford and Guy E. Blelloch. Compact Representations of Ordered Sets. *ACM/SIAM Symposium on Discrete Algorithms* (SODA), 11–19, 2004.
6. Erik D. Demaine, Thouis R. Jones, Mihai Patrascu. Interpolation search for non-independent data. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), 529–530, 2004.
7. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), 743–752, 2000.
8. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Experiments on Adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments* (ALENEX), 91–104, 2001.
9. V. Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4) 441–476, 1992.
10. W. Frakes and R. Baeza-Yates. *Information Retrieval*. Prentice Hall, 1992.
11. G. Gonnet, L. Rogers, and G. George. An algorithmic and complexity analysis of interpolation search. *Acta Informatica*, 13(1) 39–52, 1980.
12. Frank K. Hwang, Shen Lin. Optimal Merging of 2 Elements with n Elements. *Acta Informatica*, v.1, 145–158, 1971.
13. Frank K. Hwang, Shen Lin. A Simple Algorithm for Merging Two Disjoint Linearly-Ordered Sets. *SIAM Journal of Computing*, v.1, 31–39, 1972.
14. Frank K. Hwang. Optimal Merging of 3 Elements with n Elements. *SIAM Journal of Computing*, v.9, 298–320, 1980.
15. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Symposium on Discrete Algorithms* (SODA), 319–327, 1990.
16. Y. Perl, A. Itai, and H. Avni. Interpolation search–A $\log \log n$ search. *CACM*, 21(7) 550–554, 1978.

Compressed Dictionaries: Space Measures, Data Sets, and Experiments*

Ankur Gupta**, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter**

Department of Computer Sciences, Purdue University,
West Lafayette, IN 47907-2066, USA
{agupta, wkhon, rahul, jsv}@cs.purdue.edu

Abstract. In this paper, we present an experimental study of the space-time tradeoffs for the *dictionary problem*, where we design a data structure to represent *set data*, which consist of a subset S of n items out of a universe $U = \{0, 1, \ldots, u-1\}$ supporting various queries on S. Our primary goal is to reduce the space required for such a dictionary data structure. Many compression schemes have been developed for dictionaries, which fall generally in the categories of combinatorial encodings and data-aware methods and still support queries efficiently. We show that for many (real-world) datasets, data-aware methods lead to a worthwhile compression over combinatorial methods. Additionally, we design a new data-aware building block structure called BSGAP that presents improvements over other data-aware methods.

1 Introduction

The recent proliferation of data has challenged our ability to organize, store, and access data from various real-world sources. Massive data sets from biological experiments, Internet routing information, sensor data, and audio/video devices require new methods for managing data. In many of these cases, the information content is relatively small compared to the size of the original data. We want to exploit the huge potential to save space in these cases. However, in many applications, the data also needs to be indexed for fast query processing. The new trend in data structures design considers space and time efficiency together. The ultimate goal is to design structures that require a minimum of space, while still performing queries efficiently.

Ideally, the space required to store any particular data should be defined with respect to its Kolmogorov complexity (the size of the smallest program which can generate that data). Unfortunately, the Kolmogorov complexity is undecidable for arbitrary input, making it an inconvenient measure for practical use. Thus, other measures of compressibility are used as a framework for data compression, like *entropy* for textual data.

* Support was provided in part by the National Science Foundation through research grant IIS–0415097.

** Support was provided in part by the Army Research Office through grant DAAD20–03–1–0321.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 158–169, 2006.

One fundamental type of data is *set data*, which consist of a subset S of n items from the universe $U = \{0, 1, \ldots, u-1\}$. Some specific examples include IP addresses, UPC barcodes, and ISBN numbers; set data also appear in inverted indexes for libraries and web pages as well as results from scientific experiments. In many cases, S is not a random subset of U and can be compressed. (For instance, consider a set S with a few tightly clustered items spread throughout U.) The *gap* measure [3] (described formally in Section 2) has been used extensively as a reasonable data-aware measure in the context of inverted indexes [15], and we will use it as a measure in this paper.

We use these notions of compressibility to design *compressed data structures* that index the data in a succinct way and also allow fast access. In particular, we address the fundamental *dictionary problem*, where we design a data structure to represent a subset S that supports various queries on S. Two fundamental queries, *rank* and *select*, are of particular interest [9]. Earlier work on the dictionary problem for these two queries, such as Jacobson [9], Munro [11], Brodnik et al. [5], and Raman et al. [12], focuses on combinatorial compression methods of set data. In particular, they develop dictionaries that take about $\lceil \log \binom{u}{n} \rceil$ bits of space. Note that $\lceil \log \binom{u}{n} \rceil \approx n \log(u/n)$ is known as the information-theoretic (combinatorial) lower bound because it is the minimum number of bits required to differentiate between the $\binom{u}{n}$ possible subsets of n items out of a universe of size u. These dictionaries use the same number of bits for each subset of size n, and thus, do not compress the data in a data-aware manner. Another focus of these papers is to achieve constant-time bounds for *rank* and *select* queries. In order to do this, they require an additional term of $\Omega(u \log \log u / \log u)$ bits. When $n \ll u$, these structures are not space-efficient since the additional term will be much (perhaps exponentially) larger than the information-theoretic minimum term $\lceil \log \binom{u}{n} \rceil$, dwarfing any savings achieved by combinatorial compression.

Another line of work focuses on achieving space that is polynomial in n and $\log u$. However, the lower bounds on the predecessor problem imply that we can no longer achieve constant query times [2]. Recent results [4, 8] exploit some properties of the underlying data and scale their space accordingly, thus potentially saving a lot of space in practice. In the case of sparse set data where $n \ll u$, [8] provides the first dictionary to take just $O(\log \binom{u}{n})$ bits of space in the worst-case. In this paper, we adopt the same view and focus on achieving near-optimal space in practice, while minimizing query time. Briefly, we mention the following new contributions.

In this paper, we motivate the importance of data-aware compression methods in practice through careful experimentation with real-world data sets. In particular, we show that a gap-style encoding method saves about 10–40% space over combinatorial encoding methods. We then develop a **b**inary-**s**earchable **gap** encoding method called **BSGAP** and show that its space is competitive with simple sequential encoding schemes [10], both in theory and practice. We also show a space-time tradeoff for BSGAP with respect to block encoding methods [4].

Table 1. Time and space bounds of dictionaries for *rank* and *select* queries

	Theoretical		Practical[a]
Paper	Time	Space (bits)	Space (bits)
this paper	$AT(u,n)$	$gap + o(\log \binom{u}{n})$ when $n \ll u$	$\leq 1,830,959$
[8]	$AT(u,n)$	$gap + o(\log \binom{u}{n})$ when $n \ll u$	$\leq 2,001,367$
[4]	$AT(u,n)$	$2gap + \Theta(u^\epsilon)$	$\leq 1,855,116$
[13][b]	$O(\log\log u)$	$\Theta(n \log u)$	$> 3,200,000$
[1]	$AT(u,n)$	$\Theta(n \log u)$	$> 3,200,000$
[2]	$BF(u,n)$	$\Theta(n^2 \log u)$	$> 320,000,000,000$
[9]	$O(1)$	$u + \Theta(u \log\log u/\log u)$	$> 4,429,185,024$
[12]	$O(1)$	$\log \binom{u}{n} + \Theta(u \log\log u/\log u)$	$> 136,217,728$
[7]	$O(1)$	$gap + \Theta(u \log\log u/\log u)$	$> 136,017,728$

[a] The practical space bounds are for indexing our upc_32 file, with $n = 100{,}000$ and $u = 2^{32}$. The values for [13, 2, 9, 12, 7] are estimated by their reported space bounds. For these methods, we relaxed their query times to $O(\log\log u)$ to provide a fairer comparison in space usage.

[b] The theoretical space bound is from Willard's y-fast trie implementation [14].

Table 1 lists the theoretical results with practical estimates for the space required to represent the various compressed dictionaries we mentioned. Here, we define $AT(u,n) = O(\min\{\sqrt{(\log n)/(\log\log n)}, (\log\log u)(\log\log n)/(\log\log\log u), \log\log n + (\log n)/(\log\log u)\})$, and $BF(u,n) = O(\min\{(\log\log u)/(\log\log\log u), \sqrt{(\log n)/(\log\log n)}\})$. Note that $BF(u,n) \leq AT(u,n)$ for any u and n. Please see [8] for a more comprehensive look at the methods in Table 1.

2 Dictionaries on Set Data

Let $S = \langle s_1, \ldots, s_n \rangle$ be an ordered subset of n items, with items chosen from a universe $U = \{0, 1, \ldots, u-1\}$ of size u; that is, $i < j$ implies $s_i < s_j$. A dictionary on S is a data structure that supports queries on S. In particular, we are interested in the following queries:

- $member(S, a)$, which returns 1 if $a \in S$, and 0 otherwise;
- $rank(S, a)$, which returns the number of items $x \in S$ that are at most a; and
- $select(S, i)$, which returns the ith smallest item of S.

The normal concern of a dictionary is how fast one can answer a query, but space usage is also an important consideration. We would like the dictionary to use the minimum space for representing S, as if it were not being indexed. There are some common measures to describe this minimum space. The first measure is $n \log u$, which is the number of bits needed to store the items s_i explicitly in an array. The second measure is the information-theoretic minimum $\lceil \log \binom{u}{n} \rceil \approx n \log(u/n)$, which is the worst-case number of bits required to differentiate between any two distinct n-item subsets of universe U.

Another well-known measure is the *gap measure* defined as

$$gap(S) = \sum_{i=1}^{n} \lceil \log(g_i + 1) \rceil,$$

where $g_1 = s_1$, and $g_i = s_i - s_{i-1}$ for $i > 1$. The gap measure is related to the space needed to represent S in *gap encoding* [3], which stores the stream of gaps $G = g_1, \ldots, g_n$ along with the value n instead of S. Note that we cannot merely store each g_i in $\lceil \log(g_i+1) \rceil$ bits and decode the stream uniquely; we also need to know the separation boundaries between successive items. One popular technique to "mark" these separators is by using a prefix code such as the δ code [6]. In δ coding, we represent each g_i in $\lceil \log(g_i + 1) \rceil + 2 \lceil \log \lceil \log(g_i + 1) \rceil \rceil$ bits, where the first $\lceil \log \lceil \log(g_i + 1) \rceil \rceil$ bits are the unary encoding of the number $\lceil \log \lceil \log(g_i + 1) \rceil \rceil$, the next $\lceil \log \lceil \log(g_i + 1) \rceil \rceil$ bits are the binary representation of the number $\lceil \log(g_i + 1) \rceil$, and the final $\lceil \log(g_i + 1) \rceil$ bits are the binary representation of g_i. Given any prefix code, we can uniquely decode the stream $G = g_1, g_2, ..., g_n$ by simply concatenating the prefix encoding of each g_i. For our theoretical results in this paper, we make use of the δ code. Another example of a prefix code is the nibble code proposed in [4]. In this paper, we will primarily use a variation of the nibble code called *nibble4* in our experiments. For this scheme, we write a "nibble" part of $\lceil \lceil \log(g_i + 1) \rceil / 4 \rceil$ in unary, which is then followed by $4 \cdot \lfloor \lceil \log(g_i + 1) + 3 \rceil / 4 \rfloor$ bits to write the binary representation of g_i, padded out to multiples of four bits. (Later, we describe *nibble4fixed*, which we use for 64-bit data. It encodes the first part in binary in four bits, since for a universe size of 2^{64}, we would need to write $64/4 = 16$ different lengths.)

The gap measure is also related to the space needed to represent S in the *prefix omission method* (POM) described in [10], which is often used to represent bitstrings of arbitrary length. Consider the bitstrings sorted lexicographically. In POM, each bitstring t_i is represented with respect to the previous bitstring t_{i-1} by omitting the common prefix of the two bitstrings. We denote the total length of this stream of (compressed) bits as $trie(S)$. It is shown in [8] that $trie(S) \geq gap(S)$, and in the worst case, $trie(S)$ is close to $2gap(S)$; however, if we pick a random number $k \in U$ and add it (modulo u) to all the numbers in S (which we call shifting by k), $trie(S)$ is expected to be very close to $gap(S)$. To encode in this amount of space, we need to find a good k.

We summarize the relationship between these measures in the following fact.

Fact 1. *Both* $\log \binom{u}{n}$ *and* $gap(S)$ *are smaller than* $n \log u$. *Also,* $gap(S) \leq trie(S)$. *When* $n = o(u)$, $gap(S) \leq \log \binom{u}{n}$.

We provide some experimental results on real data sets in Figure 1, which bears out the theoretical statements of Fact 1. Here, the files tested are described in Section 4.1, and the space is reported (in bits) along the y-axis. The figure on the left shows data files with a universe of size $u \leq 2^{32}$, and the figure on the right shows data files with $u \leq 2^{64}$.

Notice that $gap(S)$ is significantly smaller than $\log \binom{u}{n}$ for real data. In fact, nibble4 is a decodeable gap encoding that also outperforms the information-theoretic minimum. For the IP data files, $gap(S)$ performs relatively poorly,

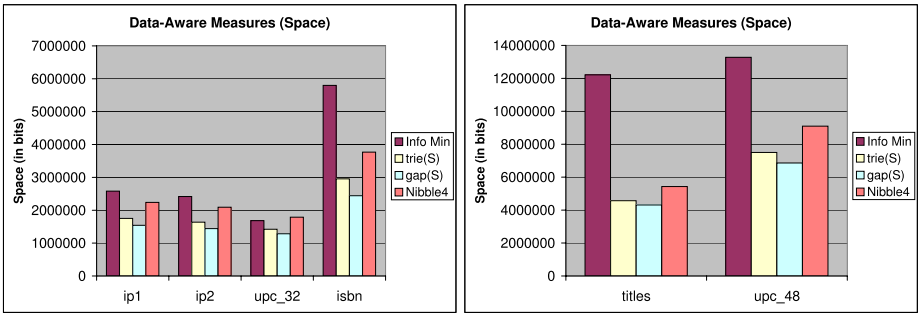


Fig. 1. Comparison of $\log \binom{u}{n}$, $trie(S)$, $gap(S)$, and a gap stream encoded according to the nibble4 code for the data files in Section 4.1

because although IPs tend to be clustered together by domains, within a domain, addresses tend to be more uniformly distributed. On the other hand, the titles file tends to have very tightly-clustered entries, since many books start with the same (or similar) words. As such, this represents a good case for gap encoding.

Since $gap(S)$ is less than $trie(S)$ for all of the files, we are free to use gap encoding throughout the remainder of the paper. Also, the impact of $gap(S)$ (and therefore its prefix encodings) is more dramatic for larger universe sizes: the figure on the right showcases this observation for the listed files. In Section 4.2, we show tradeoffs between various prefix codes for both the space required and their encoding/decoding time. It turns out that nibble4 is the method of choice (which is why we included it here).

3 The Binary-Searchable Gap Encoding Scheme (BSGAP)

In this section, we describe our BSGAP data structure that compresses a balanced binary search tree T on S and still supports queries in $O(\log n)$ time. The main point of this section is in showing that a binary-searchable representation requires about the same number of bits as linear encoding schemes [10]. (In Section 4.2, we show that this observation also holds in practice.)

The basic idea to achieve compression for T is to store each of the n nodes in the binary search tree for S in less than $\log u$ bits. In particular, an item s corresponding to node v in the binary search tree will be more succinctly stored if we can store the *difference* between s and some other item along the path from v to the root. The best such item s' would minimize $|s - s'|$. By the properties of binary search trees, s' must either be v's left parent or v's right parent.[1] Generally, let s'_i represent this best ancestor along the path from the root to the node v_i corresponding to item s_i.

Formally, let the subsets $S_L = \langle s_1, s_2, ...s_{\lceil n/2 \rceil - 1} \rangle$ and $S_R = \langle s_{\lceil n/2 \rceil + 1}, ..., s_n \rangle$ represent the left and right subtrees of the root of the balanced binary search tree

[1] The left (right) parent of v is the first node on the path P from v to root that is to the left (right) of v. In a binary search tree, it contains the largest (smallest) item that is smaller (larger) than s among the nodes on P.

T, which stores the item $s_{\lceil n/2 \rceil}$. For the BSGAP of S, denoted as $\texttt{BSGAP}(S)$, let $|\texttt{BSGAP}(S)|$ be the number of bits used to encode $\texttt{BSGAP}(S)$, where we δ-encode each value required. Then, the encoding of BSGAP of S is defined recursively as a concatenation of encoding of four components:

$$\texttt{BSGAP}(S) = \langle s_{\lceil n/2 \rceil} - s'_{\lceil n/2 \rceil};\ \ |\texttt{BSGAP}(S_L)|;\ \ \texttt{BSGAP}(S_L);\ \ \texttt{BSGAP}(S_R) \rangle,$$

where we define $s'_{\lceil n/2 \rceil} = 0$ to handle encoding the root. The value $s_{\lceil n/2 \rceil} - s'_{\lceil n/2 \rceil}$ is called the key-value of $\texttt{BSGAP}(S)$. Note the sign of this key-value determines whether the best ancestor is the left parent or the right parent. This value is encoded in variable-length encoding, by the use of a prefix code (such as δ or nibble4) to allow compression. The term $|\texttt{BSGAP}(S_L)|$ is needed as a *pointer* to jump to the the right half of the set while searching, and thus constitutes additional overhead. We shall refer to this space overhead as the *pointer cost*. In fact, we actually store the encoding of $\min\{|\texttt{BSGAP}(S_L)|, |\texttt{BSGAP}(S_R)|\}$, along with an additional bit to indicate which value we have stored. (This improvement saves space both in theory and practice, since the only time one spends any extra bits over the original encoding is when $|\texttt{BSGAP}(S_L)| = |\texttt{BSGAP}(S_R)|$.)

The search in $\texttt{BSGAP}(S)$ follows exactly the same steps as a search in the original (uncompressed) binary search tree, with the exception that we must decode item values in each node on the fly. In order to maintain an $O(\log n)$ query time, we have to consider the issue of decoding a δ-coded number (or similar prefix code) in the RAM model in constant time. We assume that in the RAM model, the word size of the machine is at least $\log u$ bits, and that we are allowed to perform addition, subtraction, multiplication, and bitshift operations on words in $O(1)$ time. We also assume that we can calculate the position of the leftmost 1 of a subword x of $\log \log u$ bits in $O(1)$ time. (This is equivalent to calculating the value $\lceil \log(x+1) \rceil$ when the word x is seen as an integer.) This latter assumption allows us to decode a δ-coded number (or similar prefix code) in $O(1)$ time in our data structure. If this is not the case, we can simulate the decoding by storing the decoding result of every possible $\log \log u$-bit number in a table with $\log u$ entries. Note that this table takes $O(\log u \log \log \log u)$ bits, which is negligible when compared to the other space terms in our data structure.

In summary, we have the following theorem, where the proof is analogous to Lemma 2 in [8], except that we avoid having to find the best shift k that minimizes their space usage.

Theorem 1. *The* $\texttt{BSGAP}(S)$ *representation requires* $gap(S) + O(n \log \log(u/n))$ *bits and supports membership, rank and select queries in* $O(\log n)$ *time.*

Proof. (sketch) Recall that a key-value in the BSGAP structure is storing the difference between an item s_i and its best ancestor s'_i in the binary search order. Though encoding a particular key-value $s_i - s'_i$ can take more space than encoding the corresponding gap value g_i, we can show that the total space for encoding all the key-values in the BSGAP structure is at most $gap(S) + O(n \log \log(u/n))$ bits using a counting technique similar to Lemma 2 of [8]. The remaining space, including the pointer cost and overhead from using δ encoding, can be bounded by $O(n \log \log(u/n))$ bits using Jensen's inequality. □

Now, we describe our main structure, which uses BSGAP as a black box for succinct representation and fast decoding of small blocks. We describe our structure in a bottom-up way. At the bottom level, we group every b items into a block and encode the items in each block by a BSGAP structure. The ith BSGAP structure corresponds to those s_j's in S with j in $[ib+1, ib+b]$. We keep an array P of n/b entries, where $P[i]$ points to the ith BSGAP structure. The array P requires $O((n/b)\log u)$ bits of space. At the top level, we collect the first item in each block, and build an instance of Andersson and Thorup's predecessor structure [1] on these first items, which takes $(n/b)\log u$ bits of space. Setting $b = \log^2 n$, we achieve the result in the theorem below, which is a slight improvement over Theorem 2 in [8], since we avoid finding a random shift k. As a companion result, we also achieve a worst-case analysis in Corollary 1, since $gap(S)$ and $O(n \log\log(u/n))$ are bounded by $O(n \log(u/n))$.

Theorem 2. *Given a set S of n items from $[1, u]$, we implement a dictionary in $gap(S) + O(n \log(u/n)/\log n) + O(n \log\log(u/n))$ bits so that rank takes $AT(u, n)$ time and select takes $O(\log\log n)$ time.*

Proof. (sketch) To answer *select*, we only need to traverse one BSGAP structure, thus requiring $O(\log\log n)$ time. To answer *rank*, the time is dominated by the predecessor query at the top level, which takes $AT(u, n)$ time. For our space bounds, the $n/\log^2 n$ BSGAP data structures require a total of $gap(S) + O(n \log\log(u/n))$ bits. The predecessor structure and P take $O((n/b)\log u) = O(n \log(u/n)/\log n)$ bits of space, thus achieving the stated space bound. □

Corollary 1. *We implement a dictionary in at most $O(n \log(u/n))$ bits of space so that rank takes $AT(u, n)$ time and select takes $O(\log\log n)$ time.*

In practice, we replace [1] with a simple binary search tree, and introduce a new parameter $h = O(\log\log n)$ that does not affect the theoretical time for BSGAP but provides a noticeable improvement in practice. Inside each BSGAP-encoded block, we further tune our structure to resort to a simple sequential encoding scheme when there are at most h items left to search, where $h < b$. Theoretically, the time required to search in the BSGAP structure is still $O(\log\log n)$. We employ this technique when sequential decoding is fast enough, to save space on the BSGAP pointers used to jump to the right half of the set. In our experiments, we actually let h range up to b, to see the point at which a sequential decoding of h items becomes impractical. It turns out that these few adjustments to our theoretical work result in a fast and succinct practical dictionary.

4 Experimental Results

In this section, we present our experimental results. Section 4.1 describes the experimental setup. In Section 4.2, we discuss various issues with the space requirements of our BSGAP structure and give some intuition about how to encode the various parts of the BSGAP structure efficiently. In Section 4.3, we describe a further tweakable parameter for our BSGAP structure and use it as a black box to succinctly encode blocks of data.

Apart from the δ code, the nibble code [4], and the nibble4 code we have mentioned in Section 2, in this section, we also refer to a number of variations of prefix codes as follows:

- The *delta squared code* encodes the value $\lceil \log(g_i + 1) \rceil$ using δ codes, which is followed by the binary representation of g_i. For instance, the delta squared code for 170 is **001 00 1000 10101010**.
- The *nibble4Gamma* encodes the "nibble" part of the nibble4 code using γ code instead of unary.[2] For instance, the nibble4Gamma code for 170 is **01 0 10101010**.
- In case the universe size of the data set is at most 2^{32}, we will also have the *fixed5 code* which encodes the value $\lceil \log(g_i + 1) \rceil$ in binary using five bits. For instance, 170 here is encoded as **01000 10101010**.
- For larger universe sizes (such as our 2^{64}-sized ones), we use the *nibble4fixed code*, a mix of the nibble4 code and the fixed5 code. Here, we encode the "nibble" part of the nibble4 code using four bits.

For each of these codes, we create a small table of values so that we can decode them quickly when appropriate. As described in Section 3, these tables add negligible space, and we have accounted for this (and other) table space in the experimental results that we describe throughout the paper.

4.1 Experimental Setup

Our source code is written in `C++` in an object-oriented style. The experiments were run on a Dell PowerEdge 650 with 3 GB of RAM. The machine was running Centos 4.1, with a `gnu g++ 3.4.4` compiler. The data sets used were as follows:

- **IP1:** List of IP addresses obtained from Duke University's Computer Science Department. The list refers to 159,690 IP addresses that hit the Duke CS pages in the month of January 2005.
- **IP2:** Similar to IP1, but this list consists of 148,700 IP addresses that hit the Duke CS pages in February 2005.
- **UPC_32:** List of 100,000 upc codes obtained from items sold by the Wal-Mart supermarket that fit in a universe of size 2^{32}.
- **ISBN:** List of 390,000 ISBNs of books at the Purdue Libraries in a 32-bit format.
- **UPC_48:** List of 432,223 upc codes in the original 48-bit format obtained from items sold by the Wal-Mart supermarket.
- **Title:** List of 256,391 book titles from Purdue Libraries, converted into a numeric value out of a universe of size 2^{64}.

4.2 Code Comparisons for Encodings and Pointers

We performed experiments to compare the space/time tradeoffs of using different encodings in place of nibble4. We summarize those experiments in Figure 2.

[2] The "nibble" part will be an integer from $[1, 16]$. The γ code for an integer x is a unary encoding of $\lceil \log x \rceil$ followed by the binary encoding of x in $\lceil \log(x + 1) \rceil$ bits.

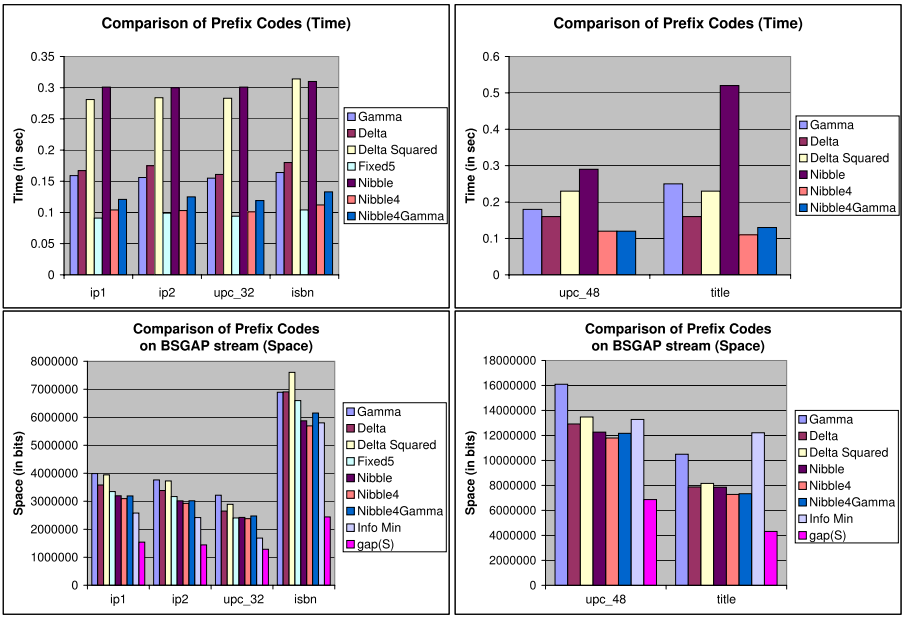


Fig. 2. Comparison of codes and measures for the data files in Section 4.1

The figures in the top row show the time required to process 10,000 randomly generated *rank* queries with a BSGAP structure using the codes listed, averaged over 10 trials. The figures in the bottom row show the space (in bits) required to encode the BSGAP data structure using the listed prefix codes. Each of the bottom two rows also has the information-theoretic minimum and $gap(S)$ listed for reference.

It is clear that both fixed5 and nibble4 are very good codes in the BSGAP structure for the 32-bit case; fixed5 is slightly faster than nibble4, and nibble4 is slightly more space-efficient. (For the ISBN file, nibble4 is significantly more space-efficient.) For 64-bit files, nibble4 is the clear choice. Since our focus is on space efficiency, the rest of the paper will build BSGAP structures with nibble4. (For our 64-bit data sets, we will actually use nibble4fixed.)

Next, we investigate the cost of these BSGAP pointers and see if a different choice of code *for just the pointers* can improve its cost. We summarize the space/time tradeoffs in Figure 3. The figure shows the pointer costs (in bits) of each BSGAP structure. As we can see, nibble4 and nibble are both space-efficient for the pointer distribution. However, nibble4 is again the logical choice, since it is both the most space-efficient and very fast to decode. If we remove these pointer costs from the total space cost for the BSGAP structure, we see that this space is *about the same as encoding the gap stream sequentially*; as such, we can think of the pointer overhead for BSGAP as a cost to support fast searching.

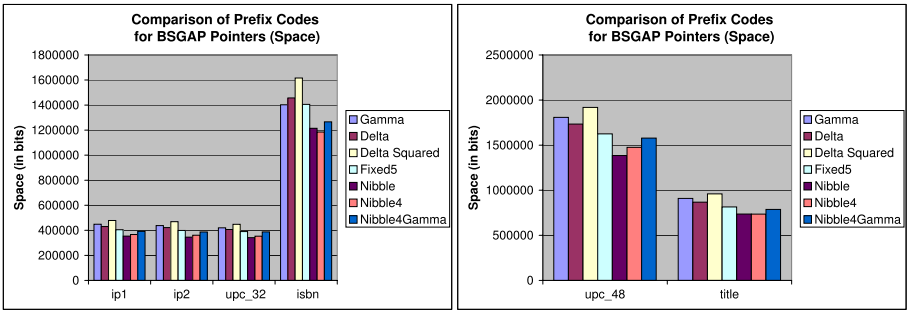


Fig. 3. Comparison of prefix codes for BSGAP pointers for the data files in Section 4.1

4.3 BSGAP: The Succinct Binary-Searchable Black Box

In this section, we focus on the practical implementation of our dictionary described in Section 3, which is based on a two-level scheme with a binary search tree at the top and BSGAP structures at the bottom. Recall that there is a parameter b that governs the number of items contained in each BSGAP structure and a parameter h that controls the degree of sequential encoding within a BSGAP data structure. We denote a particular configuration of our dictionary structure by $D(b, h)$. Let BB refer to the data structure in [4]. In this framework, BB is a special case of our dictionary $D(b, h)$ when $h = b$.

In Figure 4, we show a space/time tradeoff for BB and our dictionary. Each graph plots space vs. time, where the time is that required to process 10,000 randomly generated *rank* queries, averaged over five trials. Here, we tune BB to operate on the same number of items in each block to avoid extra costs for padding and give them the same benefits as BSGAP receives. For each graph in Figure 4, we let the blocksize b range from $[2, 256]$ and the hybrid value range from $[2, b]$. We collect time and space statistics for each $D(b, h)$ data structure. The BB curve is generated from the 256 points corresponding to $D(b, b)$. For the BSGAP curve, we partition the x-axis into 300 partitions and choose the most time-efficient implementation of $D(b, h)$ taking that much space. Notice that our BSGAP structure converges to BB as we allow more space for the data structures, but we have some improvement for extremely small space.

Since BB is a subcase of our BSGAP structure, one might think that our space-time curve should never be higher than BB's. However, the curve is generated with actual data structures $D(b, h)$ taking a particular space and time. So, the existence of a point above the BB curve on our BSGAP curve simply means that there exists one configuration of our data structure $D(b, h)$ which has those particular results.

The parameter h is crucial to achieving a good space/time tradeoff. Notice that as h increases, the space of $D(b, h)$ decreases because we store fewer pointers in each BSGAP data structure. One may think of transferring this saved space into entries in the top level binary search tree to speed up the query time. On the other hand, the time required to search at the bottom of each BSGAP

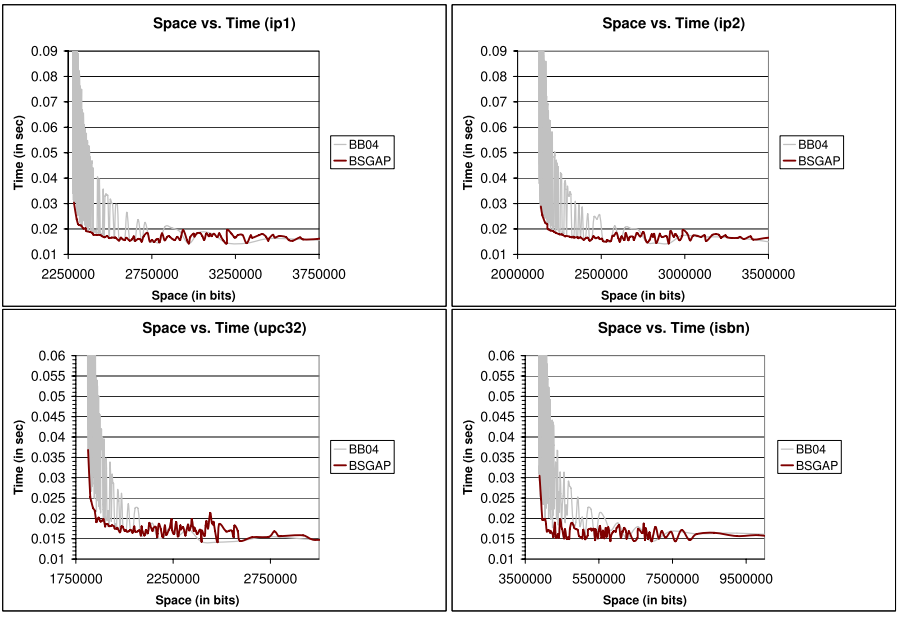


Fig. 4. Comparison of BB and BSGAP on 32-bit data files in Section 4.1

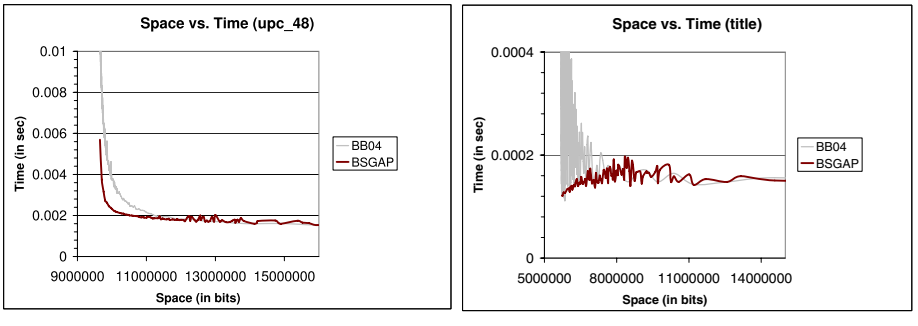


Fig. 5. Comparison of BB and BSGAP on 48-bit and 64-bit data files in Section 4.1

structure increases linearly with h. So, there must be some moderate value of h that balances these costs and arrives at the best space/time tradeoff. Hence, we collect all (b, h) pairs and evaluate the best candidates among them.

In Figure 5, we compare BB and our dictionary for 64-bit data. We plot space vs. time, where the time is that required to process 1,000 randomly generated *rank* queries, averaged over five trials. We collect data for $D(b, h)$ as before, where the range for b and h for upc_48 is $[2, 512]$ and title is $[2, 2048]$. Notice that our data structure provides a clear advantage over BB as the universe size increases.

5 Conclusion

In this paper, we have shown evidence that data-aware measures (such as *gap*) tend to be smaller than combinatorial measures on real-life data. Employing techniques that exploit the redundancy of the data can lead to more succinct data structures and a better understanding of the underlying information. As such, we encourage researchers to develop theoretical results with a data-aware analysis. In particular, our BSGAP data structure, along with BB (proposed in [4]) are extremely succinct for sparse data sets. In addition, we provide some evidence that BSGAP is less sensitive than [4] to an increase in the size of the universe. Finally, we provide some useful information on the relative performance of prefix codes with respect to compression space and decompression time.

References

1. A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the ACM Symposium on Theory of Computing*, 2000.
2. P. Beame and F. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the ACM Symposium on Theory of Computing*, 1999.
3. T. C. Bell, A. Moffat, C. G. Nevill-Manning, I. H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44(9):508–531, 1993.
4. D. Blandford and G. Blelloch. Compact representations of ordered sets. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2004.
5. A. Brodnik and I. Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.
6. P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21:194–203, 1975.
7. R. Grossi and K. Sadakane. Squeezing succinct data structures into entropy bounds. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2006.
8. A. Gupta, W. Hon, R. Shah, and J. Vitter. Compressed data structures: Dictionaries and data-aware measures. In *Proceedings of the IEEE Data Compression Conference*, 2006.
9. G. Jacobson. Succinct static data structures. Technical Report CMU-CS-89-112, Carnegie-Mellon University, 1989.
10. S. T. Klein and D. Shapira. Searching in compressed dictionaries. In *Proceedings of the IEEE Data Compression Conference*, 2002.
11. J. I. Munro. Tables. *Foundations of Software Technology and Theoretical Computer Science*, 16:37–42, 1996.
12. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.
13. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.
14. D. E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–394, 1984.
15. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, 1999.

Efficient Bit-Parallel Algorithms for (δ, α)-Matching

Kimmo Fredriksson[1,⋆] and Szymon Grabowski[2]

[1]Department of Computer Science, University of Joensuu,
PO Box 111, FIN–80101 Joensuu, Finland
kfredrik@cs.joensuu.fi
[2]Technical University of Łódź, Computer Engineering Department,
Al. Politechniki 11, 90–924 Łódź, Poland
sgrabow@kis.p.lodz.pl

Abstract. We consider the following string matching problem. Pattern $p_0p_1p_2 \ldots p_{m-1}$ (δ, α)-matches the text substring $t_{i_0}t_{i_1}t_{i_2} \ldots t_{i_{m-1}}$, if $|p_j - t_{i_j}| \leq \delta$ for $j \in \{0, \ldots, m-1\}$, where $0 < i_{j+1} - i_j \leq \alpha + 1$. The task is then to find all text positions i_{m-1} that (δ, α)-match the pattern. For a text of length n, the best previously known algorithms for this string matching problem run in time $O(nm)$ and in time $O(n\lceil m\alpha/w \rceil)$, where the former is based on dynamic programming, and the latter on bit-parallelism with w bits in computer word (32 or 64 typically). We improve these to take $O(n\delta + \lceil n/w \rceil m)$ and $O(n\lceil m \log(\alpha)/w \rceil)$, respectively, worst case time using bit-parallelism. On average the algorithms run in $O(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$ and $O(n)$ time. Our experimental results show that the algorithms work extremely well in practice. Our algorithms handle general gaps as well, having important applications in computational biology.

Keywords: approximate string matching, music information retrieval, protein matching, bit-parallelism, nondeterministic finite automata.

1 Introduction

Background and problem setting. Many notions of approximateness have been proposed in string matching literature, usually motivated by some real problems. One of seemingly underexplored problems with applications in music information retrieval and molecular biology is (δ, α)-matching [4] and its variations. In this problem, the pattern $p_0p_1 \ldots p_{m-1}$ is allowed to match a substring of the text $t_0t_1 \ldots t_{n-1}$ with α-limited gaps, and the respective pairs of matching characters may be different, only if their numerical values do not differ by more than δ. Translating this model into a music (melody seeking) application, we can allow for small distortions of the original melody because the (presumably unskilled) human user may sing or whistle the melody imprecisely. The gaps, on

⋆ Supported by the Academy of Finland, grant 202281.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 170–181, 2006.

the other hand, allow to skip over ornamenting notes (e.g., arpeggios), which appear especially in classical music. Other assumptions here, that is, monophonic melody and using pitch values only (without note durations), are reasonable in most practical cases. In biology, somewhat relaxed version of the α-matching problem is important for protein matching, especially together with allowing for matching classes of characters. Fortunately, in all the new algorithms we are going to present in this paper, δ-matching can be straightforwardly changed into matching classes of characters, without any penalty in the complexities if the size of the character class is of the order of δ.

Previous work. The first algorithm for the problem [4] is based on dynamic programming, and runs in $O(nm)$ time. This algorithm was later reformulated [2] to allow to find all pattern occurrences, instead of only the positions where the occurrence ends. This needs more time, however. The algorithm in [3] improves the average case of the one in [2] to $O(n)$, assuming constant α. More general forms of gaps were considered in [10], retaining the $O(nm)$ time bounds. For the α-matching with classes of characters there exists an efficient bit-parallel nondeterministic automaton solution [9]. In this algorithm the gap limits for each pattern character may be of different length, in particular, it is assumed that for many characters it is zero. This algorithm can be trivially generalized to handle (δ, α)-matching [3], but the time complexity becomes $O(n\lceil \alpha m/w \rceil)$ in the worst case, where w is the length of the machine word. For small α the algorithm can be made to run in $O(n)$ time on average. Sparse dynamic programming can be used to solve the problem in $O(n + |\mathcal{M}| \min\{\log(\delta + 2), \log\log m\})$ time, where $\mathcal{M} = \{(i, j) \mid |p_i - t_j| \leq \delta\}$, and $|\mathcal{M}| \leq nm$ [6]. This can be extended for the harder problem variant where transposition invariance and character insertions, substitutions or mismatches are allowed together with (δ, α)-matching [7].

Our results. We improve the dynamic programming based algorithm to run in $O(\lceil n/w \rceil m + n\delta)$ worst case time, where w is the number of bits in a machine word. This can be improved to take $O(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$ time time on average, where σ is the size of the alphabet. We improve the nondeterministic finite automaton based algorithm to take only $O(n\lceil m \log(\alpha)/w \rceil)$ worst case time, i.e. $O(n)$ worst case time for $m = O(w/\log(\alpha))$. For small α the algorithm can be made to run in $O(n)$ time on average regardless of m. The algorithms can be generalized to handle general gaps and character classes as well, see Sec. 3.3. This has important applications in computational biology.

2 Preliminaries

Let the pattern $P = p_0 p_1 p_2 \ldots p_{m-1}$ and the text $T = t_0 t_1 t_2 \ldots t_{n-1}$ be numerical strings, where $p_i, t_j \in \Sigma$ for $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. The number of distinct symbols in the pattern and in the text are denoted by σ_{P} and σ_{T}, respectively. Moreover, we use $\sigma_{\text{P}\cap\text{T}}$ to denote the number characters that occur both in P and T simultaneously. Note that $\sigma_{\text{P}\cap\text{T}} \leq \sigma_{\text{P}}, \sigma_{\text{T}}, m$.

In δ-approximate string matching the symbols $a, b \in \Sigma$ match, denoted by $a =_\delta b$, iff $|a - b| \leq \delta$. Pattern P (δ, α)-matches the text substring $t_{i_0} t_{i_1} t_{i_2} \dots t_{i_{m-1}}$, if $p_j =_\delta t_{i_j}$ for $j \in \{0, \dots, m-1\}$, where $0 < i_{j+1} - i_j \leq \alpha + 1$. If string A (δ, α)-matches string B, we sometimes write $A =_\delta^\alpha B$.

In all our analysis we assume uniformly random distribution of characters in T and P, and constant δ and σ.

3 Bit-Parallel Dynamic Programming

In this section we show how bit-parallelism can be used to bring the worst case complexity of dynamic programming down to $O(n\delta + \lceil n/w \rceil m)$, where w is the number of bits in computer word (typically 32 or 64).

We number the bits from the least significant bit (0) to the most significant bit $(w-1)$. C–like notation is used for the bit-wise operations of words; & is bit-wise `and`, | is `or`, $\sim$ negates all bits, << is shift to left, and >> shift to right, both with zero padding.

Let us first define a matrix D. Let $D_{i,j} = 1$ if $p_0 p_1 \dots p_i =_\delta^\alpha t_h t_{h+1} \dots t_j$. Otherwise, $D_{i,j} = 0$. This can be expressed as:

$$D_{i,j} = \begin{cases} 1, p_i =_\delta t_j \text{ AND } \exists j' : 0 < j - j' \leq \alpha + 1 \text{ AND } D_{i-1,j'} = 1 \\ 0, \text{otherwise.} \end{cases} \quad (1)$$

At a first glance it seems that this recurrence would lead to $O(\alpha nm)$ time. However, we show how to compute $O(w)$ columns in each row of the matrix in $O(1)$ time, independent of α, leading to $O(\lceil n/w \rceil m)$ total time.

To this end, assume that in the preprocessing phase we have computed a helper bit-matrix (whose efficient computation we will consider later) V:

$$V_{i,j} = \begin{cases} 1, p_i =_\delta t_j \\ 0, \text{otherwise.} \end{cases} \quad (2)$$

The computation of D will proceed column-wise, w columns at once. Each matrix element takes only one bit of storage, so we can store w columns in a single machine word. Assume that we have computed all rows of the columns $(j-1)w \dots jw - 1$, and columns $jw \dots (j+1)w - 1$ up to row $i-1$, and we want to compute the columns $jw \dots (j+1)w - 1$ at row i. Assume also that $\alpha < w$. We adopt the notation $D^w_{i,j} = D_{i,jw \dots (j+1)w-1}$, and analogously for V. The goal is then to produce $D^w_{i,j}$ from $V^w_{i,j}$, $D^w_{i-1,j}$ and $D^w_{i-1,j-1}$. $D^w_{i,j}$ does not depend on any other D^w element, according to the definition of D, and given our assumption that $\alpha < w$.

Now, according to Eq. (1), the kth bit in $D^w_{i,j}$ should be set iff (i) the kth bit in $V^w_{i,j}$ is set (i.e. $p_i =_\delta t_{jw+k}$), and (ii) any of the bits $k - \alpha - 1 \dots k - 1$ in $D^w_{i-1,j}$ or any of the bits $k + w - \alpha - 1 \dots w - 1$ in $D^w_{i-1,j-1}$ is set (i.e. the gap length to the previous match is at most α). To compute item (ii) efficiently we assume that we have available function $M(x)$:

$$M(x) = M(x, \alpha) = (x << 1) \mid (x << 2) \mid \dots \mid (x << (\alpha + 1)). \quad (3)$$

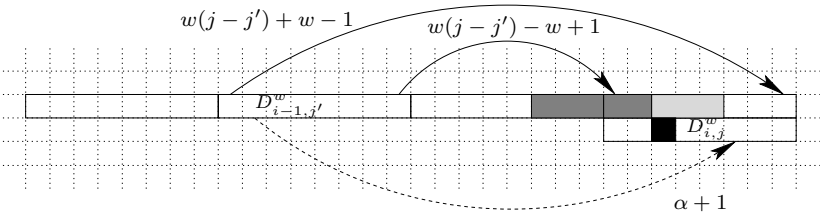


Fig. 1. Tiling the dynamic programming matrix with $w \times 1$ vectors ($w = 8$). The black cell of the current tile depends on the dark gray cells of the two tiles in the previous row ($\alpha = 4$). The light gray cells depict a negative gap, together with the dark gray cells the gap is $-3 \ldots 4$. The arrows illustrate some bit distances for the case $\alpha \geq w$.

In other words, $M(x)$ copy-propagates all bits in x to left $1 \ldots \alpha + 1$ positions. This means that if the 1 bits in x correspond to the matching positions of a pattern prefix, then $M(x)$ will have those 1 bits aligned in all positions where the matching prefix could be extended. Note that the representation of $M(x)$ needs $w + \alpha + 1$ bits, i.e. at most $2w$ bits (2 computer words) for $\alpha < w$. We can now write the recurrence for D^w:

$$D^w_{i,j} = V^w_{i,j} \;\&\; (M(D^w_{i-1,j}) \mid (M(D^w_{i-1,j-1}) >> w)). \tag{4}$$

Fig. 1 illustrates the bits affecting the current row.

We are not able to compute $M(x)$ in constant time, hence we use a precomputed look-up table instead. Since w can be too large to make this approach feasible, we can precompute the answers e.g. to only $w/2$ or $w/4$ bit numbers, and correspondingly compute $M(x)$ in 2 or 4 pieces without affecting the time complexity (in our tests we used $w/2 = 16$ bit numbers for computing $M(x)$).

We also need to compute V efficiently. This is easy with table look-ups as we have an integer alphabet. We first compute a table L, such that for all $c \in \Sigma$ the list $L[c]$ contains all the distinct characters p_i that satisfy $p_i =_\delta c$. Using this table we build a table V', which we will use as a terse representation of V, namely we have that $V'[p_i] = V_i$. This can be done by scanning through the text, and setting the jth bit of the bitvector $V'[c]$ to 1 for each $c \in L[t_j]$. This process takes $O(\lceil n/w \rceil \sigma_{\text{P}} + m + \sigma + \delta\sigma_{\text{P}} + \delta n) = O(\lceil n/w \rceil \sigma_{\text{P}} + \delta n)$ worst case time. The probability that two characters δ-match is at most $(2\delta + 1)/\sigma$, and hence the expected number of matching pattern characters for each text character is $O(\delta\sigma_{\text{P}\cap\text{T}}/\sigma_{\text{T}})$. Therefore, the average case complexity of the preprocessing is $O(\lceil n/w \rceil \sigma_{\text{P}} + n(\delta\sigma_{\text{P}\cap\text{T}}/\sigma_{\text{T}} + 1))$. Searching clearly takes only $O(\lceil n/w \rceil m)$ time.

3.1 Fast Algorithm on Average

We make the following observation: if $D_{i \ldots m-1, j-\alpha \ldots j} = 0$, for some i, j, then $D_{i+1 \ldots m-1, j+1} = 0$. This is because there is no way the recurrence can introduce any other value for those matrix cells. In other words, if $p_0 \ldots p_i$ does not (δ, α)-match $t_h \ldots t_{j-k}$ for any $k = 0 \ldots \alpha$, then the match at the position $j + 1$ cannot

Alg. 1. DA-bpdp$(T, n, P, m, \delta, \alpha)$

```
1   V ← DA-bpdp-preprocess(T, n, P, m, δ, α)
2   w' ← w/2; msk ← (1 << w') − 1
3   for i ← 0 to (1 << w') − 1 do
4       M[i] ← 0
5       for j ← 0 to α do M[i] ← M[i] | (i << (j + 1))
6   top ← m − 1
7   D_0 ← V[p_0][0]
8   for i ← 1 to top do
9       D_i ← V[p_i][0] & (M[D_{i−1} & msk] | (M[D_{i−1} >> w'] << w'))
10  if D_{m−1} ≠ 0 then report matches
11  for j ← 1 to ⌈n/w⌉ do
12      D'_0 ← V[p_0][j]
13      i ← 1
14      while i ≤ top do
15          x ← M[D'_{i−1} & msk] | (M[D'_{i−1} >> w'] << w')
16          y ← M[D_{i−1} >> w'] >> w'
17          D'_i ← V[p_i][j] & (x | y)
18          if i = top AND top < m − 1 AND D'_i & (∼0 >> 1) ≠ 0 then
19              D_i ← 0
20              top ← top + 1
21          i ← i + 1
22      if top = m − 1 AND D'_{m−1} ≠ 0 then report matches
23      while top > 0 AND D'_top & (∼0 << (w − α − 1)) = 0 do top ← top − 1
24      if top < m − 1 then top ← top + 1
25      Dt ← D; D ← D'; D' ← Dt;
```

be extended to $p_0 \ldots p_{i+1}$. This can be utilized by keeping track of the highest row number top of the current column j such that $D_{top,j} \neq 0$, and computing the next column only up to row $top + 1$. More formally, we define (for D^w) the maximum row top^w_j for the column j as:

$$top^w_j = \text{argmax}_i\{D^w_{i-1,j-1} \;\&\; a \neq 0 \text{ OR } D^w_{i-1,j} \;\&\; (\sim 0 >> 1) \neq 0\}, \qquad (5)$$

where the bitmask $a = \sim 0 << (w-\alpha-1)$. Consider first the part $D^w_{i-1,j-1}$ & $a \neq 0$. The rationale is as follows. When we are computing $D^w_{i,j}$, only the $\alpha+1$ highest non-zero bits of $D^w_{i-1,j-1}$ can affect the bits in $D^w_{i,j}$. These are selected by the & a operation. However, since we are computing w columns in parallel, the $w-1$ least significant set bits in $D^w_{i-1,j}$ (the second part), i.e. in the previous row of the *current* set of columns, can affect the bits in $D^w_{i,j}$ as well. Obviously, this second part cannot be computed at column $j-1$. We solve this simply by computing the first part of top^w_j after the column $j-1$ have been computed, and when processing the column j, we increase top^w_j if needed according to the second part ($D^w_{i-1,j}$ & $(\sim 0 >> 1) \neq 0$).

Alg. 1 gives the pseudo code. It uses $w' = w/2$ bits for the precomputed table for the $M(\cdot)$ function. For simplicity, the code also assumes that $\alpha < w'$ (but w columns are still processed in parallel). The average case running time of this algorithm depends on what is the average value of top^w. For $w = 1$ it can be shown that $\text{avg}(top^1) = O(\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}})$ [3]. This is $O(\alpha\delta/\sigma)$ for $\delta/\sigma < 1-\alpha^{-1/(\alpha+1)}$, so the average time is $O(n\lceil\alpha\delta/\sigma\rceil)$. We are not able to analyze $\text{avg}(top^w)$ exactly, but we have trivially that $\text{avg}(top^1) \leq \text{avg}(top^w) \leq \text{avg}(top^1) + w - 1$, and hence the amortized average search time of Alg. 1 is at most $O(\lceil n/w\rceil\lceil\alpha\delta/\sigma\rceil + n)$.

The $O(\lceil n/w \rceil \sigma_{\text{P}} + \delta n)$ (worst case) preprocessing time can be the dominating factor in some cases. We now present an alternative preprocessing variant. The idea is to partition the alphabet into $\lceil \sigma/\delta \rceil$ disjoint intervals of width δ. Let us first redefine V as V^δ:

$$V^\delta_{i,j} = \begin{cases} 1, |\lfloor p_i/\delta \rfloor - \lfloor t_j/\delta \rfloor| \leq 1 \\ 0, \text{otherwise.} \end{cases} \tag{6}$$

Obviously, if $V_{i,j} = 1$, then also $V^\delta_{i,j} = 1$. On the other hand, the converse is not true, i.e. it is possible that $V^\delta_{i,j} = 1$ but $V_{i,j} = 0$. This means that we can use V^δ in place of V, but the search algorithm becomes a filter, and the candidate occurrences must be verified using some other algorithm, e.g. plain dynamic programming, which makes the total complexity $O(nm)$ in the worst case. However, the benefit is that V^δ is very simple to compute, taking only $O(n)$ time. The initialization time drops to $O(\lceil n/w \rceil \min(\sigma_{\text{P}}, \sigma/\delta))$, since it takes $O(\lceil n/w \rceil)$ for each distinct $\lfloor p_i/\delta \rfloor$.

Note that one can use the definition $V^\delta_j[p_i/\delta] = 1$ iff $\lfloor p_i/\delta \rfloor = \lfloor t_j/\delta \rfloor$ instead of $V^\delta_{i,j}$, and then use the fact that $V^\delta_{i,j} = V^\delta_j[p_i/\delta - 1] \mid V^\delta_j[p_i/\delta] \mid V^\delta_j[p_i/\delta + 1]$ in the search phase. This speeds up the preprocessing by a constant factor, but slows down the search correspondingly. We use this approach in our experiments.

This can be still improved by interweaving the preprocessing and search phases, so that we initialize and preprocess V^δ only for top^w_j length prefixes of the pattern for each j. At the time of processing the column j, we only know top^w_{j-1}, so we use an estimate $\varepsilon \times top^w_{j-1}$ for top^w_j, where $\varepsilon > 1$ is a small constant. If this turns out to be too small, we just increase the estimate and re-preprocess for the current column. The total preprocessing cost on average then becomes only $O(\lceil n/w \rceil \sigma_{top^w} + n)$, where σ_{top^w} is the alphabet size of top^w length prefix of the pattern. Hence the initialization time is at most $O(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$. We require that $\delta < \sigma/3$, as otherwise the probability of a match becomes 1. The average number of verifications decreases exponentially for $m > \text{avg}(top^w)$, making their cost negligible, so the total preprocessing, filtering and verification time is $O(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$. For larger δ or smaller m the filter becomes useless.

3.2 Handling Large α in $O(1)$ Time

Alg. 1 assumes that $\alpha < w$. For larger α the time increases by $O(\alpha/w)$ factor, as the gap may span over several machine words. We now show how to remove this limit while maintaining the $O(1)$ cost for processing w columns.

Let us define *Last Prefix Occurrence*:

$$LPO_{i,j} = \begin{cases} j', & \max j' \leq j \text{ such that } D^w_{i,j'} \neq 0 \\ -\alpha - 1, & \text{otherwise.} \end{cases} \tag{7}$$

I.e. for $LPO_{i,j} = j'$, $D^w_{i,j'}$ is the vector that corresponds to the last (δ, α)-match(es) of the prefix $p_0 \ldots p_i$ in the text area $t_0 \ldots t_{wj-1}$. If such vector does not exist (e.g. when $j = 0$) we set $LPO_{i,j} = -\alpha - 1$.

Assume that $\alpha \geq w$ and consider the computation of $D^w_{i,j}$. The recurrence becomes

$$D^w_{i,j} = V^w_{i,j} \ \& \ (M(D^w_{i-1,j}, w) \mid ov). \tag{8}$$

The vector ov is computed according to the last prefix occurrence information. Let $j' = LPO_{i-1,j-1}$. We have the following four cases (see also Fig. 1):

1. $j' < 0$: no matching prefixes have been found, hence $ov = 0$.
2. $w(j - j') - w + 1 > \alpha + 1$: no bit of $D^w_{i-1,j'}$ can affect any bit in $D^w_{i,j}$, hence we set $ov = 0$.
3. $w(j - j') + w - 1 \leq \alpha + 1$: any set bit in $D^w_{i-1,j'}$ is close enough to affect any bit in $D^w_{i,j}$, hence we set $ov = \sim 0$.
4. Otherwise some bits of $D^w_{i-1,j'}$ can be close enough to affect some bits of $D^w_{i,j}$, and we set $ov = (M(D^w_{i-1,j'}, \alpha \bmod w) >> w)$.

Note that since $\alpha \geq w$, the function $M(\cdot, \cdot)$ is now much easier to compute. $M(D^w_{i-1,j}, w) = 2^w - 2 \times LSB(D^w_{i-1,j})$, where $LSB(x)$ extracts the least significant set bit of x. The first subtraction operation then propagates the LSB to every higher position as well, while the second subtraction then clears the least significant bit of the result. The solution for $LSB(x)$ is part of the computing folklore, and can be computed as $LSB(x) = (x \ \& \ (x - 1)) \ \hat{} \ x$ in $O(1)$ time. Likewise, it is easy to see that $M(D^w_{i-1,j'}, \alpha \bmod w) >> w = 2^s - 1$ for $s = \alpha \bmod w - (w - \lfloor \log_2(D^w_{i-1,j'}) \rfloor - 1) + 1$, where $\lfloor \log_2(x) \rfloor$ effectively gets the *index* of the most significant set bit of x. In other words, s tells the number of bit positions the most significant bit of $D^w_{i-1,j'}$ propagates to to fill the least significant bits of ov. If $s < 0$, we just set $ov = 0$.

Finally, $LPO_{i,j}$ can be easily maintained in constant time for each i, j. $LPO(i, -1)$ is initialized to $-\alpha - 1$ for all i, which takes $O(m)$ time. Then, the computation of D^w proceeds column-wise. After $D^w_{i,j}$ is computed, we simply set $LPO_{i,j} = j$ iff $D^w_{i,j} \neq 0$, otherwise we set $LPO_{i,j} = LPO_{i,j-1}$. In practice we can store only the latest value of LPO for each row, so only $O(m)$ space is needed. Hence we can conclude that the value of α does not affect the running time of the algorithm.

3.3 Relaxing δ and α

Alg. 1 can be generalized to the case where the gap limit can be of different length for each pattern character [10], and where the δ-matching is replaced with character classes, i.e. each pattern character is replaced with a set of characters. More precisely, pattern $p_0 p_1 p_2 \ldots p_{m-1}$, where $p_j \subset \Sigma$, matches $t_{i_0} t_{i_1} t_{i_2} \ldots t_{i_{m-1}}$, if $t_{i_j} \in p_j$ for $j \in \{0, \ldots, m-1\}$, where $a_j \leq i_{j+1} - i_j \leq b_j + 1$, where a_j and b_j are the minimum and maximum gap lengths permitted for a pattern position j. This problem variant has important applications e.g. in protein searching, see [9]. Yet a stronger model [8] allows gaps of *negative* lengths, i.e. a_j (and b_j) can be negative. In other words, parts of the pattern occurrence can be overlapping in the text, see Fig. 1. First note that handling character classes is trivial, since it only requires a small change in the computation of V. As for the gaps, consider

first the situation where (i) $a_i \geq 0$; or (ii) $b_i \leq 0$. In either case we have $a_i \leq b_i$. Handling the case (i) is just what our algorithm already does. The case (ii) is just the dual of the case (i), and conceptually it can be handled by just scanning the current row from right to left, and using the limits $-b_i - 2, -a_i - 2$ instead of a_i, b_i, and handling the gap -1 as a special case.

The core of Alg. 1 is the use of $M(x)$ (Eq. (3)) to select the positions from the previous row where a matching pattern prefix ends. To handle gaps of the form $a_i \geq 0$ we use

$$M_i^{\mathrm{L}}(x) = (x << (a_i + 1)) \mid (x << (a_i + 2)) \mid \ \ldots \ \mid (x << (b_i + 1)). \tag{9}$$

For the negative gaps $b_i < 0$ we just align the bits from right, and hence define:

$$M_i^{\mathrm{R}}(x) = (x >> -b_i - 1) \mid (x >> -b_i) \mid \ \ldots \ \mid (x >> -a_i - 1). \tag{10}$$

The general case $a_i < 0 \leq b_i$ is handled as a combination of these:

$$M_i(x) = (x >> -a_i - 1) \mid (x >> -a_i) \mid \ \ldots \ \mid (x << (b_i + 1)). \tag{11}$$

The final simple modification that we need is to take $D^w_{i-1,j+1}$ into account while computing $D^w_{i,j}$, since the negative gaps may span into it. Hence we modify Eq. (4) to:

$$D^w_{i,j} = V^w_{i,j} \ \ \& \ \ ((M_i^{\mathrm{L}}(D^w_{i-1,j-1}) >> w) \mid M_i(D^w_{i-1,j}) \mid \tag{12}$$

$$(M_i^{\mathrm{R}}(D^w_{i-1,j+1} << w) >> w)). \tag{13}$$

4 Non-deterministic Finite Automata

In this section we present an algorithm based on non-deterministic finite automaton. The problem of the algorithm in [9] is that it needs $m + (m-1)\alpha$ bits to represent the search search state. Our goal is to reduce this to $O(m \log \alpha)$, and hence the worst case time to $O(n\lceil (m \log \alpha)/w \rceil)$.

At a very high level, the algorithm can be seen as a novel combination of Shift-And and Shift-Add algorithms [1]. The 'automaton' has two kinds of states: Shift-And states and Shift-Add states. The Shift-And states keep track of the pattern characters, while the Shift-Add states keep track of the gap length between the characters. The result is a systolic array rather than automaton; a high level description of a building block for character p_i is shown in Fig. 2. The final array is obtained by concatenating one building block for each pattern character. We call the building blocks *counters*.

To efficiently implement the systolic array in sequential computer, we need to represent each counter with as few bits as possible while still being able to update all the counters bit-parallelly.

We reserve $\ell = \lceil \log_2(\alpha + 1) \rceil + 1$ bits for each counter, and hence we can store $\lfloor w/\ell \rfloor$ counters into a single machine word. We use the value $2^{\ell-1} - (\alpha + 1)$ to initialize the counters, i.e. to represent the value 0. (This representation has

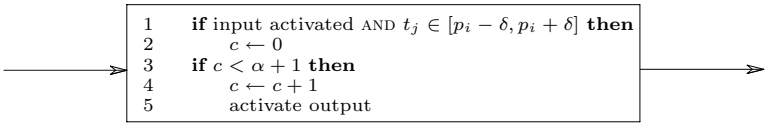


Fig. 2. A building block for a systolic array detecting δ-matches with α-bounded gaps

been used before, e.g. in [5].) This means that the highest bit (ℓth bit) of the counter becomes 1 when the counter has reached a value $\alpha + 1$, i.e. the gap cannot be extended anymore. Hence the lines 3—4 of the algorithm in Fig. 2 can be computed bit-parallelly as

$$C \leftarrow C + ((\sim C >> (\ell - 1)) \ \& \ msk),$$

where msk selects the lowest bit of each counter. That is, we negate and select the highest bit of each counter (shifted to the low bit positions), and add the result the the original counters. If a counter value is less than $\alpha + 1$, then the highest bit position is not activated, and hence the counter gets incremented by one. If the bit was activated, we effectively add 0 to the counter.

To detect the δ-matching characters we need to preprocess a table B, so that $B[c]$ has $i\ell$th bit set to 1, iff $|p_i - c| \leq \delta$. We can then use the plain Shift-And step:

$$D' \leftarrow ((D << \ell) \mid 1) \ \& \ B[t_i],$$

where we have reserved ℓ bits per character in D as well. Only the lowest bit of each field has any significance, the rest are only for aligning D and C appropriately. The reason is that a state in D may be activated also if the corresponding gap counter has not exceeded $\alpha+1$. In other words, if the highest bit of a counter in C is not activated (the gap condition is not violated), then the corresponding bit in D should be activated:

$$D \leftarrow D' \mid ((\sim C >> (\ell - 1)) \ \& \ msk).$$

The only remaining difficulty to solve is how to reinitialize (bit-parallelly) some subset of the counters to zero, i.e. how to implement the lines 1–2 of the algorithm in Fig. 2. The bit vector D' has value 1 in every field position that survived the Shift-And step, i.e. in every field position that needs to be initialized in C. Then

$$C \leftarrow C \ \& \sim(D' \times ((1 << \ell) - 1))$$

$$C \leftarrow C \mid (D' \times ((1 << (\ell - 1)) - (\alpha + 1)))$$

first clears the corresponding counter fields, and then copies the initial value $2^{\ell-1} - (\alpha + 1)$ to all the cleared fields.

This completes the algorithm. Alg. 2 gives the pseudo code. Alg. 2 runs in $O(n)$ worst case time, if $m(\lceil \log_2(\alpha + 1) \rceil + 1) \leq w$. Otherwise, several machine words are needed to represent the search state, and the time grows accordingly. However, by using the well-known folklore idea, it is possible to obtain $O(n)$

Alg. 2. DA-mloga-bits$(T, n, P, m, \delta, \alpha)$

```
1   ℓ ← ⌈log2(α + 1)⌉ + 1
2   for i ← 0 to σ − 1 do B[i] ← 0; B'[i] ← 0
3   for i ← 0 to m − 1 do B'[p_i] ← B'[p_i] | (1 << (i × ℓ))
4   for i ← 0 to σ − 1 do if B'[i] ≠ 0 then
5       for j ← max(0, i − δ) to min(i + δ, σ − 1) do B[j] ← B[j] | B'[i]
6   msk ← 0
7   for i ← 0 to m − 1 do msk ← msk | (1 << (i × ℓ))
8   am ← (1 << (ℓ − 1)) − (α + 1)
9   D ← 0; C ← (am + α + 1) × msk
10  msk ← msk >> ℓ
11  mm ← 1 << ((m − 1) × ℓ)
12  for i ← 0 to n − 1 do
13      C ← C + ((∼C >> (ℓ − 1)) & msk)
14      D' ← ((D << ℓ) | 1) & B[t_i]
15      D ← D' | ((∼C >> (ℓ − 1)) & msk)
16      C ← C & ∼((D' << ℓ) − D')
17      C ← C | (D' × am)
18      if (D & mm) = mm then report match
```

average time for long patterns not fitting into a single word by updating only the "active" (i.e. non-zero) computer words. This works in $O(n)$ time on average as long as $\delta/(\sigma(1-\delta/\sigma)^{\alpha+1}) = O(w/\log\alpha)$. The preprocessing takes $O(m + (\sigma + \delta\sigma_{\text{P}})\lceil m\log(\alpha)/w\rceil)$ time, which is $O(m + (\sigma + \delta\min\{m, \sigma\})\lceil m\log(\alpha)/w\rceil)$ in the worst case.

5 Experimental Results

We have run experiments to evaluate the performance of our algorithms. The experiments were run on Pentium4 2GHz with 512Mb of RAM, running GNU/Linux 2.4.18 operating system. We have implemented all the algorithms in C, and compiled with `icc 7.0`.

For the text we used a concatenation of 7543 music pieces, obtained by extracting the pitch values from MIDI files. The total length is 1,828,089 bytes. The pitch values are in the range $[0 \ldots 127]$. This data is far from random; the six most frequent pitch values occur 915,082 times, i.e. they cover about 50% of the whole text, and the total number of different pitch values is just 55. A set of 100 patterns were randomly extracted from the text. Each pattern was then searched for separately, and we report the average user times.

We compared the following algorithms: **DP**: The plain Dynamic Programming algorithm [4], $O(nm)$ time and $O(m)$ space (a column-wise variation); **SS**: Sequential Sampling algorithm [2], $O(nm)$ time and $O(\alpha m)$ space; **DP Cut-off**: "Cut-off" version of DP (similar as the SS cut-off [3]); **BP Cut-off**: Bit-parallel "cut-off" (Alg. 1); **BP Cut-off 3-L**: Fast preprocessing time variant of BP Cut-off (Sec. 3.1); **NFA alpha**: The nondeterministic finite automaton ([9]), slightly optimized version; **NFA log alpha**: The nondeterministic finite automaton (Alg. 2).

Fig. 3 shows the timings for different pattern lengths and δ, α values. **DP** and **SS** were slow even for small m, hence we omit the plots from those. For small α **NFA alpha** is hard to beat due to its simplicity. Interestingly, the

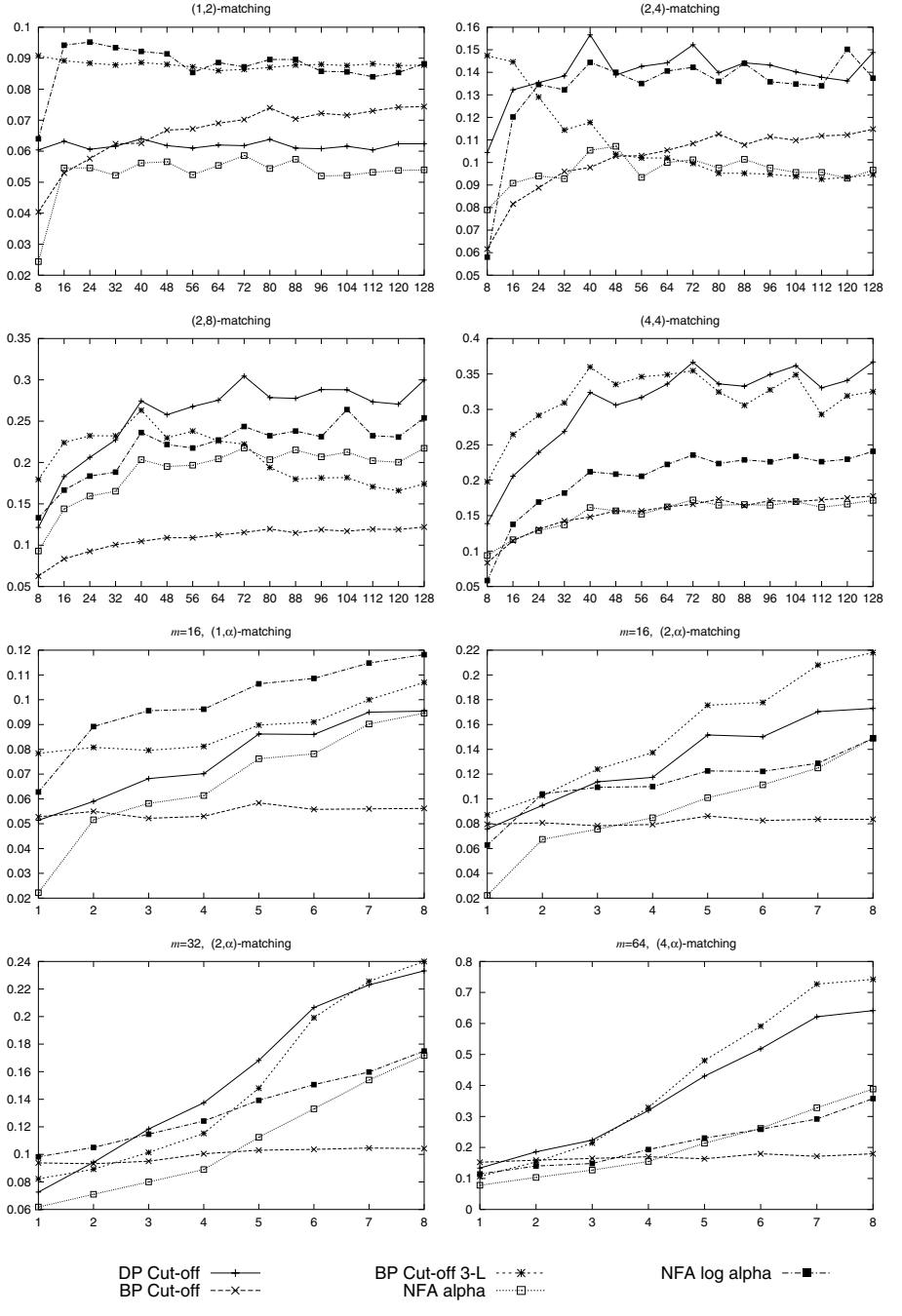


Fig. 3. Top: Running times in seconds for different pattern lengths. Bottom: Running times in seconds for $\alpha = 1 \ldots 8$ and different pattern lengths.

more complex **NFA log alpha** beats it only in few cases. However, as δ and α increase, **BP Cut-off** eventually becomes the winner. As the effective alphabet size is small, the variant with faster preprocessing is almost always slower due to the verifications. However, for random texts over large alphabets with flat distribution this algorithm is clearly faster especially for large m. We omit the plots due to lack of space. For large (δ, α) the differences between the algorithms become smaller. The reason is that a large fraction of the text begins to match the pattern. However, this means that these large parameter values are not interesting anymore.

6 Conclusions

We have presented new efficient algorithms for string matching with bounded gaps and character classes. Our algorithms are based on pre-emptying the computation early where the match cannot be extended, bit-parallelism and nondeterministic finite automata. Besides having theoretically good worst and average case complexities, the algorithms are shown to work well in in practice. We have concentrated on music retrieval but our algorithms have important applications in computational biology as well.

References

1. R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.
2. D. Cantone, S. Cristofaro, and S. Faro. An efficient algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In *Proceesings of WEA'05*, volume 3503 of *LNCS*, pages 428–439. Springer, 2005.
3. D. Cantone, S. Cristofaro, and S. Faro. On tuning the (δ, α)-sequential-sampling algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In *Proceedings of ISMIR'05*, 2005.
4. M. Crochemore, C. Iliopoulos, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas. Approximate string matching with gaps. *Nordic Journal of Computing*, 9(1):54–65, 2002.
5. M. Crochemore, C. Iliopoulos, G. Navarro, Y. Pinzon, and A. Salinger. Bit-parallel (δ, γ)-matching suffix automata. *Journal of Discrete Algorithms (JDA)*, 3(2–4):198–214, 2005.
6. V. Mäkinen. *Parameterized approximate string matching and local-similarity-based point-pattern matching*. PhD thesis, Department of Computer Science, University of Helsinki, August 2003.
7. V. Mäkinen, G. Navarro, and E. Ukkonen. Transposition invariant string matching. *Journal of Algorithms*, 56(2):124–153, 2005.
8. E. W. Myers. Approximate matching of network expression with spacers. *Journal of Computational Biology*, 3(1):33–51, 1996.
9. G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *Journal of Computational Biology*, 10(6):903–923, 2003.
10. Y. J. Pinzón and S. Wang. Simple algorithm for pattern-matching with bounded gaps in genomic sequences. In *Proceedings of ICNAAM'05*, pages 827–831, 2005.

Tiny Experiments for Algorithms and Life

Jon Bentley

Avaya Labs Research
jbentley@avaya.com

Abstract. Algorithmic experiments come in all sizes. A jumbo testbed for the Traveling Salesman Problem, for instance, can take years to build, and additional years can be spent designing and running insightful experiments. This talk concentrates on tiny algorithmic experiments that can be conducted in a few minutes. Such experiments include parameter estimation, hypothesis testing, determining functional forms, and conducting *horse races*. This talk also describes how tiny *Math, Science and Engineering* (MSE) can be done in one's head or on the back of the proverbial envelope, and shows how to apply it to professional problems and problems in everyday life.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, p. 182, 2006.

Evaluation of Online Strategies for Reordering Buffers*

Matthias Englert, Heiko Röglin, and Matthias Westermann

Department of Computer Science,
RWTH Aachen, D-52056 Aachen, Germany
{englert, roeglin, marsu}@cs.rwth-aachen.de

Abstract. A sequence of objects which are characterized by their color has to be processed. Their processing order influences how efficiently they can be processed: Each color change between two consecutive objects produces costs. A reordering buffer which is a random access buffer with storage capacity for k objects can be used to rearrange this sequence online in such a way that the total costs are reduced. This concept is useful for many applications in computer science and economics.

The strategy with the best known competitive ratio is MAP. An upper bound of $O(\log k)$ on the competitive ratio of MAP is known and a non-constant lower bound on the competitive ratio is not known [2]. Based on theoretical considerations and experimental evaluations, we give strong evidence that the previously used proof techniques are not suitable to show an $o(\sqrt{\log k})$ upper bound on the competitive ratio of MAP. However, we also give some evidence that in fact MAP achieves a competitive ratio of $O(1)$.

Further, we evaluate the performance of several strategies on random input sequences experimentally. MAP and its variants RC and RR clearly outperform the other strategies FIFO, LRU, and MCF. In particular, MAP, RC, and RR are the only known strategies whose competitive ratios do not depend on the buffer size. Furthermore, MAP achieves the smallest constant competitive ratio.

1 Introduction

Frequently, a number of tasks has to be processed and their processing order influences how efficiently they can be processed. Hence, a reordering buffer can be expedient to influence the processing order. This concept is useful for many applications in computer science and economics. In the following, we give an example (for further examples see [1, 2, 3, 4, 5, 7]).

In computer graphics, a rendering system displays 3D scenes which are composed of primitives. In current rendering systems, the state changes performed by the graphics hardware are a significant factor for the performance. A state change occurs when two consecutively rendered primitives differ in their attribute

* The first and the last author are supported by the DFG grant WE 2842/1. The second author is supported by the DFG grant VO 889/2.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 183–194, 2006.

values, e. g., in their texture or shader program. These state changes slow down a rendering system. To reduce the costs of the state changes, a reordering buffer can be included between application and graphics hardware. Such a reordering buffer which is a random access buffer with limited memory capacity can be used to rearrange the incoming sequence of primitives online in such a way that the costs of the state changes are reduced [6].

1.1 The Model

An *input sequence* $\sigma = \sigma_1\sigma_2\cdots$ of objects which are only characterized by a specific attribute has to be processed. To simplify matters, we suppose that the objects are characterized by their color, and, for each object σ_i, let $c(\sigma_i)$ denote the color of σ_i. A *reordering buffer* which is a random access buffer with storage capacity for k objects can be used to rearrange the input sequence in the following way.

The first object of σ that is not handled yet can be stored in the reordering buffer, or objects currently stored in the reordering buffer can be removed. These removed objects result in an *output sequence* $\sigma_{\pi^{-1}} = \sigma_{\pi^{-1}(1)}\sigma_{\pi^{-1}(2)}\cdots$ which is a permutation of σ. We suppose that the reordering buffer is initially empty and, after processing the whole input sequence, the buffer is empty again.

For an input sequence σ, let $C^A(\sigma)$ denote the *costs of a strategy* A, i. e., the number of color changes in the output sequence. The goal is to minimize the costs $C^A(\sigma)$.

The notion of an online strategy is intended to formalize the realistic scenario, where the strategy does not have knowledge about the whole input sequence in advance. The online strategy has to serve the input sequence σ one after the other, i. e., a new object is not issued before there is a free location in the reordering buffer. Online strategies are typically evaluated in a competitive analysis. In this kind of analysis the costs of the online strategy are compared with the costs of an optimal offline strategy. For an input sequence σ, let $C^{\mathrm{OPT}}(\sigma)$ denote the costs produced by an optimal offline strategy. An online strategy is denoted as α-*competitive* if it produces costs at most $\alpha \cdot C^{\mathrm{OPT}}(\sigma) + \kappa$, for each sequence σ, where κ is a term that does not depend on σ. The value α is also called the *competitive ratio* of the online strategy.

1.2 The Strategies

We only consider *lazy* strategies, i. e., strategies that fulfill the following two properties.

- An *active color* is selected, and, as long as objects with the active color are stored in the buffer, a lazy strategy does not make a color change.
- If an additional object can be stored in the buffer, a lazy strategy does not remove an object from the buffer.

Hence, a lazy strategy has only to specify how to select a new active color. Note that every (in particular every optimal offline) strategy can be transformed into a lazy strategy without increasing the costs.

First-In-First-Out (FIFO). This strategy assigns time stamps to each color stored in the buffer. Initially, the time stamps of all colors are undefined. When an object is stored in the buffer and the color of this object has an undefined time stamp, the time stamp is set to the current time. Otherwise, it remains unchanged. FIFO selects as new active color the color with the oldest time stamp and resets this time stamp to undefined. This is a very simple strategy that does not analyze the input stream. The buffer acts like a sliding window over the input stream in which objects with the same color are combined.

Least-Recently-Used (LRU). Similar to FIFO, this strategy assigns time stamps to each color stored in the buffer. Initially, the time stamps of all colors are undefined. When an object is stored in the buffer, the time stamp of its color is set to the current time. LRU selects as new active color the color with the oldest time stamp and resets this time stamp to undefined. LRU and also FIFO tend to remove objects from the buffer too early [7].

Most-Common-First (MCF). This fairly natural strategy tries to clear as many locations as possible in the buffer, i. e., it selects as new active color a color that is most common among the objects currently stored in the buffer. MCF also fails to achieve good performance guarantees since it keeps objects with a rare color in the buffer for a too long period of time [7]. This behavior wastes valuable storage capacity that could be used for efficient buffering otherwise.

Maximum-Adjusted-Penalty (MAP). This strategy, which is introduced and analyzed in a non-uniform variant of our model [2], provides a trade-off between the storage capacity used by objects with the same color and the chance to benefit from future objects with the same color. We present an adapted version of MAP for our uniform model which is similar to the Bounded-Waste strategy [7]. A penalty counter is assigned to each color stored in the buffer. Informally, the penalty counter for color c is a measure for the storage capacity that has been used by all objects of color c currently stored in the buffer. Initially, the penalty counters for all colors are set to 0. MAP selects as new active color a color with maximal penalty counter and the penalty counters are updated as follows: The penalty counter for each color c is increased by the number of objects of color c currently stored in the buffer, and the penalty counter of the new active color is reset to 0.

Random-Choice (RC). Since the computational overhead of MAP is relatively large, we present more practical variants of MAP. RC which is a randomized version of MAP selects as new active color the color of an uniformly at random chosen object from all objects currently stored in the buffer. Note that RC can also be seen as a randomized version of MCF. Even if RC is much simpler than MAP, random numbers have to be generated.

Round-Robin (RR). This strategy is a very efficient variant of RC. It uses a selection pointer which points initially to the first location in the buffer. RR selects as new active color the color of the object the selection pointer points to

and the selection pointer is shifted in a round robin fashion to the next location in the buffer. We suppose that RR has the same properties on typical input sequences as RC.

1.3 Previous Work

Räcke, Sohler, and Westermann [7] show that several standard strategies are unsuitable for a reordering buffer, i. e., the competitive ratio of FIFO and LRU is $\Omega(\sqrt{k})$ and the competitive ratio of MCF is $\Omega(k)$, where k denotes the buffer size. Further, they present the deterministic Bounded-Waste strategy (BW) and prove that BW achieves a competitive ratio of $O(\log^2 k)$.

Englert and Westermann [2] study a non-uniform variant of our model: Each color change to color c produces non-uniform costs b_c. As main result, they present the deterministic MAP strategy and prove that MAP achieves a competitive ratio of $O(\log k)$.

The offline variant of our model is studied in [1, 5]. However, the goal is to maximize the number of saved color changes. Note that an approximation of the minimum number of color changes is preferable, if it is possible to save a large number of color changes. Kohrt and Pruhs [5] present a polynomial-time offline algorithm that achieves an approximation ratio of 20. Further, they mention that optimal algorithms with running times $O(n^{k+1})$ and $O(n^{m+1})$ can be obtained by using dynamic programming, where k denotes the buffer size and m denotes the number of different colors. Bar-Yehuda and Laserson [1] study a more general non-uniform maximization variant of our model. They present a polynomial-time offline algorithm that achieves an approximation ratio of 9.

Khandekar and Pandit [4] consider reordering buffers on a line metric. This metric is motivated by an application to disc scheduling: Requests are categorized according to their destination track on the disk, and the costs are defined as the distance between start and destination track. For a disc with n uniformly-spaced tracks, they present a randomized strategy and show that this strategy achieves a competitive ratio of $O(\log^2 n)$ in expectation against an oblivious adversary.

Krokowski et al. [6] examine the previously mentioned rendering application. They use a small reordering buffer (storing less than hundred references) to rearrange the incoming sequence of primitives online in such a way that the number of state changes is reduced. Due to its simple structure and its low memory requirements, this method can easily be implemented in software or even hardware. In their experimental evaluation, this method typically reduces the number of state changes by an order of magnitude and the rendering time by roughly 30%. Furthermore, this method typically achieves almost the same rendering time as an optimal, i. e., presorted, sequence without a reordering buffer.

1.4 Our Contributions

In Section 2, we study the worst case performance of MAP. Recall that an upper bound of $O(\log k)$ on the competitive ratio of MAP is known and a non-constant lower bound on the competitive ratio is not known [2]. Hence, a natural

question is whether it is possible to improve the upper bound on the competitive ratio of MAP. The proof of the upper bound consists of two parts. First, it is shown that the competitive ratio of MAP_{4k} against OPT_k is 4, where A_n denotes the strategy A with buffer size n and OPT denotes an optimal offline strategy. Finally, it is proven that the competitive ratio of OPT_k against OPT_{4k} is $O(\log k)$. As we see, the logarithmic factor is lost solely in the second part of the proof.

Based on theoretical considerations and experimental results, we give strong evidence that the competitive ratio of OPT_k against OPT_{4k} is $\Omega(\sqrt{\log k})$. This implies that the previously used proof techniques are not suitable to prove an $o(\sqrt{\log k})$ upper bound on the competitive ratio of MAP. However, we also give some evidence that in fact MAP achieves a competitive ratio of $O(1)$.

In Section 3, we evaluate the performance of several strategies on random input sequences experimentally. MAP and its variants RC and RR clearly outperform the other strategies FIFO, LRU, and MCF. In particular, MAP, RC, and RR are the only known strategies whose competitive ratios do not depend on the buffer size.

2 Worst Case Performance of MAP

In Section 2.1, we give an alternative proof that the competitive ratio of OPT_k against OPT_{4k} is $O(\log k)$ in our uniform model. This proof is based on a potential function. In Section 2.2, we exploit properties of this potential function to generate deterministic input sequences that give strong evidence that this result cannot be improved much. In more detail, based on our experimental evaluation in Section 2.3, we conjecture that the competitive ratio of OPT_k against OPT_{4k} is $\Omega(\sqrt{\log k})$. As a consequence, the proof technique in [2], which is also implicitly contained in the proof of [7], is not suitable to show an $o(\sqrt{\log k})$ upper bound on the competitive ratio of MAP.

2.1 Theoretical Foundations

In this section, we give an alternative proof for the following theorem.

Theorem 1. *The competitive ratio of* OPT_k *against* OPT_{4k} *is* $O(\log k)$.

Proof. Fix an input sequence σ and an optimal offline strategy OPT_{4k}. Let $\sigma_{\pi^{-1}}$ denote the output sequence of OPT_{4k}. Suppose that $\sigma_{\pi^{-1}}$ consists of m color blocks $B_1, \ldots B_m$, i. e., $\sigma_{\pi^{-1}} = B_1 \cdots B_m$ and all objects in each color block have the same color and the objects in each color block B_i have a different color than the objects in color block B_{i+1}. Let $c(B_i)$ denote the color of the objects in color block B_i. W. l. o. g. assume that $c(B_1) = 1$, $c(B_2) = 2$, $\ldots$ $c(B_m) = m$, i. e., the color of each color block is different from the colors of the other color blocks. This does not change the costs of OPT_{4k} and can obviously only increase the costs of OPT_k.

Consider the execution of a strategy A. Fix a time step. We denote a color c as *finished* if all objects of color c have occurred in the output sequence of

A. Otherwise, color c is denoted as *unfinished*. Let $f = \min\{c|c \text{ is unfinished}\}$ denote the *first unfinished color*, and let $d(c) = c - f$ denote the *distance of color c*. Then, the potential of color c is defined as $\Phi(c) = n(c) \cdot d(c)$, where $n(c)$ denotes the number of objects of color c currently stored in the buffer of A. For each color c, we define a counter $p(c)$, initially set to 0. Intuitively, the counter $p(c)$ indicates how many objects with a color strictly larger than c have occurred in the output sequence of A. Whenever A moves an object of color c to the output sequence, for each $f \leq i < c$, $p(i)$ is increased by one.

Now, we describe the simple algorithm GREEDY$_k$ (f, $d(c)$, $n(c)$, $\Phi(c)$, and $p(c)$ are defined w. r. t. GREEDY$_k$). Note that the accumulated potential Φ which is initially set to 0 is introduced but not used in the algorithm.

1. Calculate the first unfinished color f. As long as $n(f) \neq 0$, move objects of color f to the output sequence.
2. Calculate a color $q = \arg\max_c \Phi(c)$ with maximum potential. Move $n(q)$ objects of color q to the output sequence. Increase the accumulated potential Φ by $\Phi(q)$. Proceed with step 1.

Observe that GREEDY$_k$ is an offline algorithm since it has to know the output sequence of OPT$_{4k}$. In the following, it is shown that the competitive ratio of GREEDY$_k$ against OPT$_{4k}$ is $O(\log k)$.

The following lemma provides an upper bound on the counters. It implies for the accumulated potential $\Phi \leq 8k \cdot m$ since the accumulated potential Φ can also be expressed as $\Phi = \sum_c p(c)$.

Lemma 2. *For each color c,* $p(c) \leq 8k$.

Proof. Observe that $p(f) \geq p(f+1) \geq \cdots \geq p(m)$ and that counters for colors less than f do not change their values anymore. Hence, it suffices to show that $p(f) \leq 8k$. This is done by induction over the iterations of GREEDY$_k$. Fix an iteration of GREEDY$_k$. We distinguish the following two cases.

- Suppose that $p(f) \leq 7k$ at the beginning of this iteration.
  Then, $p(f) \leq 8k$ at the end of this iteration since $p(f)$ is increased by at most k in step 2. Note that the counters are only increased in step 2.
- Suppose that $p(f) > 7k$ at the beginning of this iteration.
  Then, GREEDY$_k$ has moved more than $7k$ objects with a color larger than f to its output sequence. Due to its buffer size, OPT$_{4k}$ has moved more than $3k$ of these objects to its output sequence. However, this implies that OPT$_{4k}$ has moved the last object of color f to its output sequence more than $3k$ time steps ago. Hence, the last object of color f has already entered the buffer of GREEDY$_k$. As a consequence, the unfinished color f becomes finished in step 1 of this iteration. □

Due to the following lemma, each iteration of GREEDY$_k$ increases the accumulated potential Φ by at least $\frac{k}{1+\ln k}$.

Lemma 3. *If* $n(f) = 0$ *and the buffer contains* k *objects,* $\max_c \Phi(c) \geq \frac{k}{1+\ln k}$.

Proof. First of all, observe that $\sum_{c>f} n(c) = k$, since for each color $c \leq f$, $n(c) = 0$ and the buffer contains k objects. Define $q = \max_c \Phi(c)$. Obviously, for each $i \geq 1$, $n(f+i) \leq \lfloor q/i \rfloor$. In particular, for each $i > q$, $n(f+i) = 0$. Hence,

$$k = \sum_{i=1}^{q} n(f+i) \leq \sum_{i=1}^{q} \frac{q}{i} = q \cdot H_q \ ,$$

where $H_q = \sum_{i=1}^{q} \frac{1}{i}$ denotes the q-th harmonic number.

Suppose that $q < \frac{k}{1+\ln k}$. Then

$$k \leq q \cdot H_q < q \cdot H_k \leq q \cdot (1 + \ln k) < k$$

which is a contradiction. □

Combining the results of the two lemmata above yields that there are at most $8m \cdot (1 + \ln k)$ iterations of GREEDY_k while its buffer contains k objects. Since each iteration generates two color changes, GREEDY_k generates at most $16m \cdot (1 + \ln k) + k$ color changes. Recall that OPT_{4k} generates $m - 1$ color changes. This concludes the proof of the theorem. □

2.2 Generating Input Sequences

In this section, we describe our approach to generate deterministic input sequences for which $\text{MAP}_{\lfloor k/4 \rfloor+1}$ loses a logarithmic factor compared to OPT_{2k+1}. To some extend, the buffers sizes are chosen arbitrarily. Our construction can be generalized canonically to MAP_a and OPT_b, for each $a < b$.

The main idea is to use the accumulated potential Φ defined in the proof of Theorem 1. The generated input sequences consist of objects with m different colors, and at most $2k$ objects of each of the m colors. The sequences are intended to have the property that $\text{MAP}_{\lfloor k/4 \rfloor+1}$ can increase the accumulated potential Φ by only $O(k/\log k)$ with each color change, and the accumulated potential Φ is $\Omega(m \cdot k)$ after the sequences are processed. As a consequence, $\text{MAP}_{\lfloor k/4 \rfloor+1}$ makes $\Omega(m \cdot \log k)$ color changes for these input sequences. However, OPT_{2k+1} is able to rearrange these input sequences in such a way that the objects of each color form a consecutive block, i. e., the number of color changes made by OPT_{2k+1} is $m - 1$. Hence, $\text{MAP}_{\lfloor k/4 \rfloor+1}$ loses a $\Omega(\log k)$ factor compared to OPT_{2k+1}.

The following algorithm for generating deterministic input sequences is based on the proof of Lemma 3. The first $2k$ objects are, for each $1 \leq i \leq \Theta(k/\log k)$, $\lceil q/i \rceil$ objects of color i, with $q = 2k/\log k$. Then, the algorithm proceeds in phases corresponding to the last unfinished color f. At the beginning of phase f, let $n(c)$ denote the number of objects of color c currently stored in the buffer of $\text{MAP}_{\lfloor k/4 \rfloor+1}$, and let $s(c)$ denote the number of objects of color c included in the input sequence so far. In phase f, $s(f)$ objects of colors larger than f followed by the last object of color f are appended to the input sequence.

At the beginning of phase f, the algorithm tries to restore a situation in which the accumulated potential Φ can only be increased by $O(k/\log k)$ and OPT_{2k+1}

is still able to rearrange the input sequence properly. At the beginning of phase f, the length of the input sequence created so far is $2k+s(1)+s(2)+\cdots+s(f-1)$. Observe that $s(1)+s(2)+\cdots+s(f-1)$ of these objects have colors smaller than f and $2k$ of these objects have colors larger or equal to f. Hence, the number of objects having a color larger than f so far is $2k-s(f)$. Due to the restriction that OPT_{2k+1} is able to rearrange the input sequence into an output sequence with only $m-1$ color changes, at most $2k$ objects with a color larger than f can precede the last object of color f. Hence, at most $s(f)$ objects of colors larger than f can be appended before the last object of color f is appended to the input sequence.

According to Lemma 3, the algorithm should achieve $n(f+i) \approx q/i$. Hence, $\max\{0, \lceil q \rceil - n(f+1)\}$ objects of color $f+1$ are appended to the input sequence, $\max\{0, \lceil q/2 \rceil - n(f+2)\}$ objects of color $f+2$ are appended to the input sequence, ... until altogether $s(f)$ objects have been appended in this phase. Then, the phase is finished by appending the last object of color f to the input sequence.

We expect that the accumulated potential Φ is $\Omega(m \cdot k)$ after the input sequence has been processed by $\mathrm{MAP}_{\lfloor k/4 \rfloor+1}$. To see this, define the potential of a color c slightly differently by $\Phi(c) = n'(c) \cdot d(c)$. This potential is not based on the number $n(c)$ of objects of color c currently stored in the buffer, but on the number $n'(c)$ of objects of color c which are moved to the output sequence when changing to color c. Observe that $n(c)$ and $n'(c)$ differ only if during moving the objects of color c to the output sequence additional objects of this color arrive. Recall that, for each color f, the last object of color f is preceded by $2k$ objects of colors larger than f. Hence, GREEDY_k has to move at least k of these objects to the output sequence before moving the last object of color f to the output sequence, and, as a consequence, $p(f) \geq k$. Then, after the sequence has been processed, the accumulated potential Φ is $\Omega(m \cdot k)$. We expect that for the generated input sequences $n(c)$ and $n'(c)$ usually do not differ much.

2.3 Experimental Evaluation

Figure 1 depicts the competitive ratios of $\mathrm{MAP}_{\lfloor k/4 \rfloor+1}$ against OPT_{2k+1} on the generated input sequences for buffer sizes $k_1, \ldots k_{92}$ with $k_1 = 540$ and $k_i = \lfloor k_{i-1} \cdot 11/10 \rfloor + 1$. A regression analysis with functions of the type $a \cdot \ln k + b$ results in $0.92127 \cdot \ln k + 1.30714$ where the sum of the squared residuals is 0.0705539. Using functions of the type $a \cdot \ln k + b \cdot \ln \ln k + c$ yields $0.837668 \cdot \ln k + 0.857676 \cdot \ln \ln k + 0.19425$ where the sum of the squared residuals is only 0.0185538.

Further, Figure 1 depicts the competitive ratios of MAP_k against OPT_{2k+1} on the generated input sequences for buffer sizes $k_1, \ldots k_{79}$. Unfortunately, there are periodic fluctuations in these competitive ratios which makes a small sum of squared residuals impossible. However, a regression analysis with functions of the type $a \cdot \ln k + b \cdot \ln \ln k + c$ results in $0.418333 \cdot \ln k + 1.40659 \cdot \ln \ln k - 0.337541$

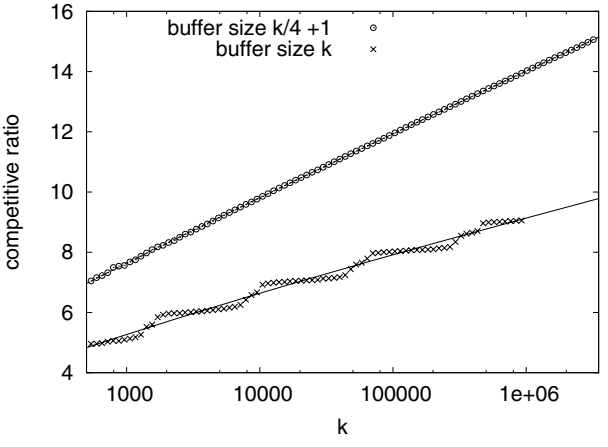


Fig. 1. Competitive ratios of $\mathrm{MAP}_{\lfloor k/4 \rfloor + 1}$ and MAP_k against OPT_{2k+1} on the generated input sequences and resulting functions for regression analysis with $a \cdot \ln k + b \cdot \ln \ln k + c$.

where the sum of the squared residuals is 1.63387 and no residual is greater than 0.266742476.

Based on the experimental evaluation, we conjecture the following.

Conjecture 4. The competitive ratio of MAP_{4k} against OPT_{32k} is $\Omega(\log k)$.

Now, we can conclude the following theorem. If we take the experimental evaluation for smaller factors between the buffer sizes into account, we can make the stronger conjecture that the competitive ratio of MAP_{4k} against OPT_{8k} is $\Omega(\log k)$, and then the $o(\sqrt[3]{\log k})$ term in the theorem improves to $o(\sqrt{\log k})$.

Theorem 5. OPT_k *cannot achieve a competitive ratio of* $o(\sqrt[3]{\log k})$ *against* OPT_{4k} *if Conjecture 4 holds.*

Proof. Suppose for contradiction that the competitive ratio of OPT_k against OPT_{4k} is $o(\sqrt[3]{\log k})$. Then, the competitive ratio of OPT_k against OPT_{64k} is $o(\log k)$. In the first part of the proof of Theorem 4 in [2] it is shown that the competitive ratio of MAP_{4k} against OPT_k is 4. As a consequence, the competitive ratio of MAP_{4k} against OPT_{64k} is $o(\log k)$ which is a contradiction to Conjecture 4. □

Our actual interest is the competitive ratio of MAP. Is it possible to show a non-constant lower bound on the competitive ratio of MAP or to improve the upper bound? Based on our experimental evaluation, the proof technique in [2, 7] is not suitable to show an $o(\sqrt{\log k})$ upper bound on the competitive ratio of MAP since this would require a competitive ratio of OPT_k against OPT_{4k} of $o(\sqrt{\log k})$.

However, we have evidence that MAP achieves in fact a competitive ratio of $O(1)$ in our uniform model. MAP is always optimal, i. e., it achieves a competitive ratio of 1, for the generated input sequences. In addition to the following

observations, this indicates a small competitive ratio of MAP. Each $\Omega(\sqrt{\log k})$ lower bound on the competitive ratio of MAP implies an $\Omega(\sqrt{\log k})$ lower bound on the competitive ratio of OPT_k against OPT_{4k}. Hence, the input sequences used in such a lower bound have to assure that the potential gained in step 2 of GREEDY_k is not too large. However, our sequences are constructed to have exactly this property. As a consequence, any major modification to our generated input sequences will probably fail to show an $\Omega(\sqrt{\log k})$ lower bound on the competitive ratio of MAP.

3 Random Input Sequences

In this section, we evaluate the performance of several strategies on random input sequences experimentally. Since an efficient optimal offline algorithm is not known, we cannot simply generate random input sequences and evaluate the performance of the strategies by comparing their number of color changes with the optimal number of color changes. Hence, we first introduce a technique to generate random input sequences with known optimal number of color changes. Finally, the experimental evaluation is presented in detail.

3.1 Input Sequences with Known Optimum

Fix an input sequence σ and an optimal offline strategy OPT_k. Let $\sigma_{\pi^{-1}}$ denote the output sequence of OPT_k. Suppose that $\sigma_{\pi^{-1}}$ consists of m color blocks $B_1, \ldots B_m$, i. e., $\sigma_{\pi^{-1}} = B_1 \cdots B_m$ and all objects in each color block have the same color and the objects in each color block B_i have a different color than the objects in color block B_{i+1}. W. l. o. g. assume that the color of each color block is different from the colors of the other color blocks. This does not change the costs of OPT_k and can obviously only increase the costs of any other strategy.

The following result is given in [2]: For each input sequence σ, the permutation $\sigma_{\pi^{-1}}$ of σ is an output sequence of a strategy with buffer size k if and only if $\pi^{-1}(i) < i + k$, for each i. Hence, a random input sequences with known optimal number of color changes can be generated as follows. First, we determine an output sequence σ^{opt} of OPT_k. This output sequence is completely characterized by the number of color blocks m and the color block lengths $l_1, \ldots l_m$, i. e., l_i denotes the number of objects in the i-th color block. Then, a permutation π with $\pi^{-1}(i) < i + k$, for each i, is chosen uniformly at random among all permutations with this property. This way, we get a random input sequence $\sigma_\pi^{\mathrm{opt}}$ for which OPT_k makes $m - 1$ color changes. Observe that usually different permutations lead to the same input sequence.

3.2 Experimental Evaluation

We evaluate the performance of MCF, LRU, FIFO RC, RR, and MAP on different kinds of random input sequences experimentally.

Constant color block lengths. We evaluate the competitive ratios of the strategies for buffer sizes $k_1, \ldots, k_{139}$ with $k_1 = 10$ and $k_i = \lfloor k_{i-1} \cdot 21/20 \rfloor + 1$ on generated input sequences with $m = 2k_i$ and color block lengths $l_1 = \cdots = l_m = 2k_i$. For each buffer size, we average over 50 runs. The variances are very small and decreasing with increasing buffer sizes. For buffer sizes larger than 1000, the variances are below 0.002.

The competitive ratios of LRU and FIFO increase with the buffer size on these non-malicious inputs. RC and RR achieve small constant competitive ratios. A regression analysis with functions of the type $a - b \cdot \exp(-k^c)$ results in $1.14098 - 0.43676 \cdot \exp(-k^{0.486582})$ for RC where the sum of the squared residuals is 0.00435474 and in $1.14157 - 0.350151 \cdot \exp(-k^{0.465377})$ for RR where the sum of squared residuals is 0.00401949. Hence, RC and RR achieve a competitive ratio of 1.14. MCF and MAP achieve the best competitive ratios. MCF is optimal for all buffer sizes, and, for buffer sizes greater than 250, MAP is also optimal.

Uniformly chosen color block lengths. Figure 2 depicts the competitive ratios of the strategies for buffer sizes $k_1, \ldots, k_{131}$ on the following generated input sequences. Let $u_1, u_2, \ldots$ denote a sequence of independent random variables distributed uniformly between 1 and $2k$. Then, $m = \max_i\{u_1 + \cdots + u_i < 4k^2\} + 1$ and, for $1 \leq i < m$, $l_i = u_i$ and $l_m = 4k^2 - (u_1 + \cdots + u_{m-1})$. For each buffer size, we average over 50 runs. The variances, except for MCF, are very small and decreasing with increasing buffer sizes. For buffer sizes larger than 1000, the variances, except for MCF, are below 0.004.

The competitive ratios of LRU, FIFO, and, in contrast to the first set of input sequences, MCF increase with the buffer size on these non-malicious input sequences. RC, RR, and MAP achieve small constant competitive ratios. A regression analysis with functions of the type $a - b \cdot \exp(-k^c)$ results in

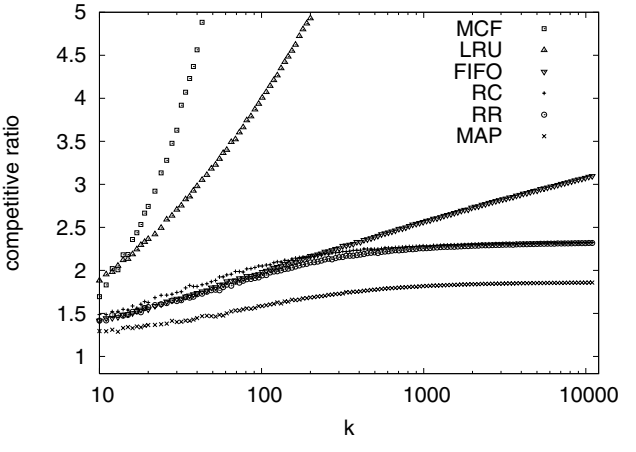


Fig. 2. Competitive ratios on random input sequences with uniformly chosen color block lengths

$2.33508 - 4.78793 \cdot \exp(-k^{0.200913})$ for RC where the sum of squared residuals is 0.0461938, in $2.32287 - 4.90995 \cdot \exp(-k^{0.229328})$ for RC where the sum of squared residuals is 0.022163, and in $1.88434 - 3.02955 \cdot \exp(-k^{0.186283})$ for MAP where the sum of the squared residuals is 0.0401868. Hence, RC, RR, and MAP achieve competitive ratios of 2.33, 2.32, and 1.88, respectively.

Different buffer sizes. We evaluate the competitive ratio of $\text{MAP}_{\lfloor k_i/4 \rfloor}$ against OPT_{2k_i+1} for $k_1, \ldots, k_{132}$ on generated input sequences with $m = 2k_i$ and color block lengths $l_1 = \cdots = l_m = 2k_i$. For each buffer size, we average over 25 runs. The variances are very small and decreasing with increasing buffer sizes. For buffer sizes larger than 1000, the variances are below 0.014.

These experiments justify the sophisticated generation of deterministic input sequences we used to obtain Conjecture 4, as they show that random input sequences do not suffice for that purpose. A regression analysis with functions of the type $a - b \cdot \exp(-k^c)$ results in $13.8829 - 42.4326 \cdot \exp(-k^{0.254565})$ for MAP where the sum of the squared residuals is 3.08368. Hence, $\text{MAP}_{\lfloor k/4 \rfloor + 1}$ achieves a constant competitive ratio against OPT_{2k+1}.

References

1. R. Bar-Yehuda and J. Laserson. 9-approximation algorithm for sorting buffers. In *Proceedings of the 3rd Workshop on Approximation and Online Algorithms*, 2005.
2. M. Englert and M. Westermann. Reordering buffer management for non-uniform cost models. In *Proceedings of the 32st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 627–638, 2005.
3. K. Gutenschwager, S. Spieckermann, and S. Voss. A sequential ordering problem in automotive paint shops. *International Journal of Production Research*, 42(9):1865–1878, 2004.
4. R. Khandekar and V. Pandit. Online sorting buffers on line. In *Proceedings of the 23th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 584–595, 2006.
5. J. Kohrt and K. Pruhs. A constant approximation algorithm for sorting buffers. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 193–202, 2004.
6. J. Krokowski, H. Räcke, C. Sohler, and M. Westermann. Reducing state changes with a pipeline buffer. In *Proceedings of the 9th International Fall Workshop Vision, Modeling, and Visualization (VMV)*, pages 217–224, 2004.
7. H. Räcke, C. Sohler, and M. Westermann. Online scheduling for sorting buffers. In *Proceedings of the 10th European Symposium on Algorithms (ESA)*, pages 820–832, 2002.

Scheduling Unrelated Parallel Machines Computational Results*

Burkhard Monien and Andreas Woclaw

Faculty of Computer Science, Electrical Engineering and Mathematics,
University of Paderborn, Fürstenallee 11, 33102 Paderborn, Germany
{bm, wocland}@uni-paderborn.de

Abstract. Scheduling n independent jobs on m unrelated parallel machines without preemption belongs to the most difficult scheduling problems. Here, processing job i on machine j takes time p_{ij}, and the total time used by a machine is the sum of the processing times for the jobs assigned to it. The objective is to minimize makespan. In this paper we present an experimental study on the Unsplittable-Truemper algorithm. This purely *combinatorial* approach computes 2-approximate solutions in the best worst-case running time known so far. The goal of our simulations was to prove its efficiency in practice. We compare our technique with algorithms and heuristics used in practice, especially with those based on the *two-step* approach. The experiments show that for large and difficult instances the Unsplittable-Truemper algorithm has a clear advantage over methods based on *linear programming*. Moreover, it requires much less operational memory, and thus is more effective and easier to handle.

1 Introduction

We consider the scheduling problem where n independent jobs have to be assigned to a set of m unrelated parallel machines without preemption. Processing job i on machine j takes time p_{ij}. A *schedule* (assignment) of jobs to machines is defined by a function $\alpha : [1..n] \to [1..m]$. The *load* $\delta_j(\alpha)$ induced by assignment α on machine j is the sum of processing times p_{ij} for the jobs that are assigned to machine j. The makespan of a schedule is the maximum load computed over all machines. The objective is to find a schedule that minimizes makespan. The problem can be formulated as following mixed integer program:

$$\begin{array}{lll} \mathsf{MIP}: & \min T & \\ & \text{s.t. } \sum_{i\in[n]} p_{ij}x_{ij} \le T & \forall\, j \in [m] \\ & \sum_{i\in[m]} x_{ij} = 1 & \forall\, i \in [n] \\ & x_{ij} \in \{0,1\} & \forall\, i \in [n], j \in [m] \end{array} \tag{1}$$

* This work has been partially supported by the DFG-Sonderforschungsbereich 376 Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen, by the European Union within the 6th Framework Program under contract 001907 (DELIS) and by the DFG Research Training Group GK-693 of the Paderborn Institute for Scientific Computation (PaSCo).

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 195–206, 2006.

Here, x_{ij} is a 0/1 assignment variable which is equal to 1 (respectively to 0) if job i is assigned (respectively not assigned) to machine j. The objective is to minimize the non-negative variable T which corresponds to the makespan. The first type of constraints associated with machine j assures that the load on machine j is at most T. The second type of constraints associated with job i means that job i has to be completed.

Following the standard three-field notation introduced by Graham et al. [8], the problem is equivalent to $R||C_{\max}$. It is known to be $\mathcal{NP}$-*complete* since PARTITION polynomially reduces to the special case $P2||C_{\max}$ where $p_{ij} = p_i$ and $m = 2$. It can be solved with an approximation factor of $2 - \frac{1}{m}$ using the algorithm from [14] and it is non-approximate in polynomial time within a factor $\frac{3}{2} - \epsilon$ of the optimum [10]. The vast majority of the approximation algorithms for this problem is based on the classical two-step approach. They compute first an optimal fractional solution by using, e.g., linear programming (LP), and then apply rounding techniques to get an integral solution.

In the paper we present an experimental study on Unsplittable-Truemper, which we have recently developed for $R||C_{\max}$ [5]. This purely combinatorial approach computes 2-approximate solutions within the best worst-case running time known so far. The primary goal of our research was to evaluate its efficiency in practice. We present two implementations of the algorithm (with and without heuristical improvements) and compare them with algorithms developed for this problem over the past decades. We are especially interested in those based on the two-step approach motivated by their usage in practice. As measures of interest, the total computation time, the value of makespan and the usage of operational memory were chosen.

2 Algorithms and Heuristics

In order to evaluate the Unsplittable-Truemper algorithm, several algorithms and heuristics [3, 7, 11, 12, 19] for $R||C_{\max}$ have been implemented and tested. Here however, because of space limit, we present only four implementations in details. In the following, we give a short description for each of them. We begin with two algorithms using the two-step approach and end with two implementations of Unsplittable-Truemper.

2.1 LP Relaxation&Rounding

The algorithm. A common strategy to solve $R||C_{\max}$ is to apply the two-step approach directly. In the first step, an LP relaxation of problem (1) is solved. Here, the integrality constrains of assignment variables are replaced by weaker non-negativity conditions as given in (2), so that the resulting LP can be solved in polynomial time [17]. The fractional optimal solution to (2) provides also a lower bound for the original problem. In the second step, the fractional solution is rounded up, so that an integral solution corresponding to the assignment is obtained. This method gives an approximation factor of 2 [10]. It is easy to implement and delivers solutions of good quality.

Algorithm 1. CPLEX(**P**)

Input: matrix of processing times **P**
Output: assignment α
1: solve LP relaxation;
2: round the fractional solution;
3: **return** integral solution;

$$\begin{array}{lll} \mathsf{LP}: & \min T & \\ & \text{s.t. } \sum_{i\in[n]} p_{ij}x_{ij} \le T & \forall\, j \in [m] \\ & \sum_{j\in[m]} x_{ij} = 1 & \forall\, i \in [n] \\ & x_{ij} \ge 0 & \forall\, i \in [n], j \in [m] \end{array} \tag{2}$$

Implementation Details. The LP relaxation is solved with the *sifting* algorithm from the ILOG CPLEX 8.0 package. In the second step, we round the fractional solution by applying the matching algorithm from [10].

2.2 Scheduling by Column Generation

The algorithm. A big drawback of Algorithm 1 is the large number of assignment variables in large-scale problems. These applications need much more operational memory to represent $R||C_{\max}$ and thus are harder to handle throughout all computations. In the worst case, it cannot even be possible to state all variables of the problem. Furthermore, in each iteration of the simplex algorithm, we look explicitly for a non-basic variable to price out and enter the basis. This operation becomes too costly when the number of variables is large. Consequently, the overall performance decreases dramatically.

To overcome this unfavorable behavior, we apply a *column generation* approach to solve the LP relaxation. Algorithm 2, which we use in our experiments, is based on the ideas from [19] where a *branch&bound* approach combined with column generation is used to solve the $P||\sum_{j\in[m]} C_j$ problem. In our case, the *master problem* (MP) of the column generation schema is defined as an LP relaxation of $R||C_{\max}$ and is given in (3) in standard form. It is easy to prove that both formulations, (2) and (3), are equivalent.

$$\begin{array}{lll} \mathsf{MIP}: & \min c^T x & \\ & \text{s.t. } Ax = b & \\ & x \ge 0 & \end{array} \tag{3}$$

where

$$x = \left[\, x_{11} \cdots x_{1m}\, x_{21} \cdots x_{2m} \cdots x_{n1} \cdots x_{nm}\, T\, s_1 \cdots s_m \,\right]^{\mathrm{T}} \in \mathbb{R}^{nm+1+m}_{\ge 0}$$
$$c = \left[\, 0 \cdots 0\ 1\ 0 \cdots 0 \,\right] \in \{0,1\}^{nm+1+m}$$
$$b = \left[\, 0 \cdots 0\ 1 \cdots 1 \,\right]^{\mathrm{T}} \in \{0,1\}^{m+n}$$

$$A = \begin{bmatrix}
p_{11} & & 0 & p_{21} & & 0 & & p_{n1} & & 0 & -1 & 1 & & 0 \\
& \ddots & & & \ddots & & & & \ddots & & \vdots & & \ddots & \\
0 & & p_{1m} & 0 & & p_{2m} & & 0 & & p_{nm} & -1 & 0 & & 1 \\
1 & \cdots & 1 & 0 & \cdots & 0 & & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
0 & \cdots & 0 & 1 & \cdots & 1 & & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
& & & & & & \ddots & & & & & & & \\
0 & \cdots & 0 & 0 & \cdots & 0 & & 1 & \cdots & 1 & 0 & 0 & \cdots & 0
\end{bmatrix} \in \mathbb{Z}_{\geq -1}^{(m+n)(nm+1+m)}$$

The appealing idea of column generation is to work only with a reasonably small subset of variables, forming the *restricted master problem* (RMP). More variables are added only when needed. We define RMP by a subset $\mathbf{a}$ of columns from A, i.e., $\mathbf{a} = \{a_k \in A | k \in I' \subseteq I\}$, where a_k is the k-th column from A corresponding to variable $x_k \in x$, and I is a set of indices of all columns from A, $|I| = nm + 1 + m$. Note that the number of columns in RMP, $|I'|$, induces the same number of variables in RMP.

$$\begin{aligned} \text{PSP: find } & k \\ \text{s.t. } & \bar{c}_k = c_k - a_k^T \mu(\mathbf{a}) < 0 \\ & a_k \in A \qquad\qquad \forall\, k \in I - I' \end{aligned} \tag{4}$$

In column generation technique, an iteration consists, first, of optimizing the RMP in order to determine dual multipliers of the current optimal solution, and second, of finding, if there still exists, a variable with negative *reduced cost*. More specifically, let $x(\mathbf{a})$ and $\mu(\mathbf{a})$ be the primal and the dual optimal solution of the current RMP defined by $\mathbf{a}$, respectively. To determine if $x(\mathbf{a})$ is also the optimal solution for the MP, we solve the *pricing subproblem* (PSP) given in (4), where $\bar{c}_k$ denotes the reduced cost of variable x_k. If the PSP returns no feasible k, i.e., the reduced costs of all variables from $I - I'$ are non-negative, then the solution $x(\mathbf{a})$ to the RMP optimally solves the MP as well (see [2] for details), and we are done. Otherwise, variable x_k, identified by the PSP, is added to the RMP, and optimization of the RMP is repeated.

In what regards convergence, note that $a_k \in A$ is generated at most once since no variable in an optimal RMP has negative reduced cost. When dealing with a finite set A of columns, the column generation algorithm converges to the optimal solution.

Implementation Details. In the first step, in order to find an initial set of columns for the RMP, a hybrid method is used. It needs $O(nm \log n)$ running time. Most of the initial columns are generated with the algorithm by Jaffe and Davis [3]. Additional columns are computed with two simple heuristics. The RMP is solved with the primal simplex algorithm from ILOG CPLEX 8.0. Because of the sparse structure of matrix A, the reduced costs $\bar{c}_k$ can be computed very fast in time $O(nm)$ by enumeration. We add up to 20 new columns to $\mathbf{a}$ in each step provided at least one column with negative reduced cost exists. In the second step, to solve the MIP, we use the *branch&cut* algorithm from CPLEX 8.0.

If the optimal integral solution cannot be found within time limit t, the rounding procedure from [10] is called for the fractional solution and a 2-approximate solution is computed.

2.3 Unsplittable-Truemper Algorithm

The algorithm. In contrast to the LP methods, the unrelated scheduling problem can also be solved with a purely combinatorial approach. Here, the fractional scheduling problem (equivalent to the LP relaxation) can be formulated as a *generalized flow problem*, where the network is defined by the scheduling problem and the capacity of some edges corresponds to the makespan. The algorithm by Radzik [13] is so far the fastest combinatorial algorithm for generalized (fractional) flow problem with running time of $O(nE(E + n\log(n)))$, where E is the number of pairs (i, j) with $p_{ij} \neq \infty$. The minimization of the makespan can then be done by binary search. It needs at most $O(\log(nU))$ calls to the algorithm for generalized flow problem. U denotes the maximal p_{ij}. Finding an integral solution for $R||C_{\max}$ can also be formulated as an *unsplittable* generalized flow problem. Several authors, e.g., [4, 9] have studied the unsplittable flow problem for usual flow networks.

Algorithm 3 is an implementation of the Unsplittable-Truemper algorithm from [5]. It is based on generalized and unsplittable network flows, and computes schedules with approximation factor of 2 in $O(m^2 E \log(m) \log(nU))$ running time. The algorithm solves an unsplittable flow problem in a generalized bipartite network defined by $R||C_{\max}$. The graph of this bipartite network consists of nodes corresponding to machines and jobs. There is a directed edge from job node i to machine node j if job i can be processed on machine j, i.e., $p_{ij} \neq \infty$. An edge which starts on machine node j and ends on job node i means, that job i is assigned to machine j. To guarantee the correctness of the assignment, there can be only one edge between given job and machine nodes, regardless of its direction. The generalized flow problem can then be transformed to a minimum cost flow problem [16] and solved with the primal-dual approach [1]. To

Algorithm 2. COLUMN(A, b, c, t)

Input: matrix A and vectors b and c
time limit t
Output: assignment α

1: find initial set of columns $\mathbf{a}$;
2: (x, μ) := solve RMP$(\mathbf{a}, b, c)$;
3: compute reduced costs $\bar{c}_k$ from dual multipliers μ and columns a_k, for $k \in I - I'$;
4: **if** $\min_{k \in I - I'} \bar{c}_k < 0$ **then**
5: generate new column(s) from A add them to $\mathbf{a}$;
6: update I' and goto 2;
7: **end if**
8: solve MIP defined by $\mathbf{a}$ with time limit t;
9: **if** time t elapsed **then** round the fractional solution x **end if**;
10: **return** integral solution;

Algorithm 3. UNSPLITTABLE-TRUEMPER($\mathbf{P}$)

Input: matrix of processing times $\mathbf{P}$
Output: assignment α

1: $l := \max_{i\in[n]} \min_{j\in[m]} p_{ij}$, $u := \sum_{i\in[n]} \min_{j\in[m]} p_{ij}$;
2: **while** $l + 1 \neq u$ **do**
3: $T := \lceil \frac{l+u}{2} \rceil$;
4: compute initial assignment α;
5: let $G_\alpha(T)$ be a bipartite graph induced by matrix $\mathbf{P}$, assignment α and parameter T;
6: let M^+ and M^- be the set of overloaded and underloaded machines in $G_\alpha(T)$, resp.;
7: **while** $\exists$ path from M^+ to M^- in $G_\alpha(T)$ **do**
8: compute shortest paths from all nodes to the set of sinks M^- in $G_\alpha(T)$;
9: compute admissible graph $G^0_\alpha(T)$;
10: $\alpha :=$ UNSPLITTABLE-BLOCKING-FLOW($G^0_\alpha(T), \alpha$);
11: update $G_\alpha(T)$;
12: **end while**
13: **if** $M^+ = \emptyset$ **then** $u := T$ **else** $l := T$ **end if**;
14: **end while**
15: **return** assignment α;

compute a blocking flow among the edges with zero reduced costs (they constitute an admissible graph), it uses an adapted version of the Unsplittable-Blocking-Flow algorithm from [4].

Given some candidate value for the makespan, T, and an initial assignment α, the inner `while`-loop of Algorithm 3 (lines 7-12) finds an approximate solution (if there exists) for the generalized flow problem in the bipartite network denoted by $G_\alpha(T)$. The bipartite graph $G_\alpha(T)$ corresponds to the scheduling problem where all $p_{ij} \leq T$ and all jobs are scheduled according to assignment α. Throughout the execution, the algorithm *always* maintains an integral assignment α, i.e, each job is always assigned to exactly one machine. Each assignment α defines a partition of the machines into underloaded, M^+, and overloaded machines, M^-. The load of overloaded machines is at least twice as large as T, whereas the load of underloaded machines is not greater than T. Overloaded and underloaded machines are treated as sources and sinks, respectively. To obtain an assignment with smaller makespan, the algorithm pushes flow (unsplittable jobs) from a machine in M^+ to a machine in M^- through the bipartite network (line 10). To push unsplittable jobs, Unsplittable-Blocking-Flow is called. It receives as input an admissible graph $G^0_\alpha(T) \subseteq G_\alpha(T)$ and an assignment α. $G^0_\alpha(T)$ consists of edges on shortest paths from M^+ to M^- in $G_\alpha(T)$. When Unsplittable-Blocking-Flow terminates, it returns a new assignment α, having the property that in updated $G^0_\alpha(T)$ there is no path from M^+ to M^-, i.e., a blocking flow in $G^0_\alpha(T)$ has been found.

Unsplittable-Truemper leaves the inner `while`-loop when it can either derive a lower bound on the optimal makespan, or it found an assignment with $M^+ \neq \emptyset$ for given candidate T. By doing binary search (the outer `while`-loop

of Algorithm 3) on $T \in [\max_{i\in[n]} \min_{j\in[m]} p_{ij}, \sum_{i\in[n]} \min_{j\in[m]} p_{ij}]$, we identify the smallest T such that in the returned assignment α, $M^+ = \emptyset$. Since for $T-1$ the algorithm terminates with $M^+ \neq \emptyset$, we can prove that assignment α is a 2-approximate solution for the scheduling problem. For more details on the Unsplittable-Truemper algorithm we refer to [5].

Implementation Details. We have developed two versions of Unsplittable-Truemper. The first one, UTA-GENERIC, is a generic implementation of the algorithm. Here, two different data structures are used to represent the bipartite graph. We have a job-machine oriented structure which we use in binary search and in the Unsplittable-Blocking-Flow algorithm [4]. This data structure consists of machine and job nodes. The information about edges adjacent to a given node is saved in this node as a double-linked list. The other, machine oriented structure, is used to speed up the computations of admissible graphs. It consists only of machine nodes. Information about job nodes and edges is saved in machine nodes. Here, for each machine node, we use a double-linked list to save the job nodes connected with this machine. Each job on the list contains pointer to the machine node on which it is assigned. The second structure is thus a contracted version of the first one and can be computed in $O(E)$. The usage of two different data structures was motivated by huge differences in size between the graphs used globally in the binary search and the admissible graphs used by Unsplittable-Blocking-Flow.

In the second version of the algorithm, UTA-IMPROVED, we use a better interphase (between two consecutive binary search steps) initialization of the algorithm. By doing this, not all results computed in the previous phase get lost with the beginning of the next phase (i.e., computed assignment and node potentials). Since our algorithm uses the primal-dual approach, the initial assignment must maintain the reduced cost optimality condition, i.e., for each edge (i, j) in the bipartite graph, the reduced cost $c^{\pi}_{ij} \geq 0$ [1]. To initialize a new binary search step with the previous solution, we need to check if this condition is fulfilled. There are two cases to consider when a binary search step terminates. In the first case, Unsplittable-Blocking-Flow terminates with $M^+ \neq \emptyset$, and the next binary search step is initialized with increased T. When T increases, new edges can be added to the bipartite graph. For each such edge we check if its reduced cost is nonnegative. Here we use node potentials from the previous solution. If only one edge does not fulfill the reduced cost optimality condition, we initialize the next step like in UTA-GENERIC, i.e., each job is assigned to a machine where its processing time is minimum. In the second case, Unsplittable-Blocking-Flow terminates with $M^+ = \emptyset$, and the next binary search step is initialized with T of smaller value. When T decreases, some edges must be deleted from the bipartite graph because their processing times are bigger than T. In particular, some jobs may become unassigned. For each such job node i, we choose machine node j on which it has the minimal processing time. Afterward, using node potential from the previous solution, we check for each edge (j, i) if the reduced cost optimality condition is fulfilled. If this is the case, we use the previous solution as initialization. Otherwise, we initialize the next step as in UTA-GENERIC. As we will

see in Section 4, the refined initialization significantly improves the convergence of the computations and thus the overall efficiency of the algorithm, especially for difficult instances.

3 Test Models and Experimental Setup

To test the algorithms, we used artificially generated test instances. Each instance belongs to one of the three different test models:

Model A: no correlation between the machines and the jobs. p_{ij} are generated from uniform distribution [1..100].
Model B: the jobs are correlated. $p_{ij} = \beta_i + \delta_{ij}$, where β_i and δ_{ij} are two randomly generated integers from uniform distributions on [1..100] and [1..20], respectively. Intuitively, β_i can be seen as a mean value of processing time and δ_{ij} as its deviation.
Model C: the machines are correlated. $p_{ij} = \alpha_j + \delta_{ij}$, where α_j and δ_{ij} are two randomly generated integers from uniform distributions on [1..100] and [1..20].

Similar models were used previously in the literature, e.g, in [6, 12, 15]. Both versions of Unsplittable-Truemper were implemented in C using standard libraries and compiled with GNU compiler. The *simplex* and *branch&cut* procedures, both included in the ILOG CPLEX 8.0 Callable Library, have been used together with the ILOG Concert Technology to implement CPLEX and COLUMN algorithms. All techniques were tested on a Sun Fire 3800 machine equipped with eight 900MHz ultraSPARC III processors and 8GB RAM working under Solaris 9 operating system.

4 Computational Results

In the following, we present computational results obtained from the experiments on the algorithms from in Section 2. All tested algorithms deliver solutions with approximation factor of 2. Here, we evaluate the quality of makespan computed by these algorithms for practical test instances. We use two types of diagrams to present the results. In Section 4.1 we present performance diagrams, both for the computation time and the makespan. They show, for each tested method, the percentage of all tested instances, for which this method returned the best result. In Section 4.2 we present diagrams showing the total computation time in dependence of the number of machines. In the last subsection, we show the computation time in function of the number of machines, while the number of jobs stays fixed. The *size* of instance is given as a tuple (n, m). The growth of the size is then defined as an increase in n, or m, or in both.

4.1 Performance Diagrams for Computation Time and Makespan

For each model the following instance sizes were used: $m \in \{10, 50, 100, 200, 500\}$ and $n = 10 \cdot m$. For each size, 10 different instances were generated and solved. To

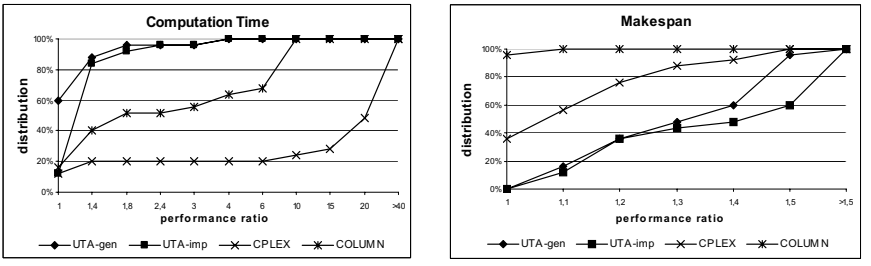


Fig. 1. Performance diagrams for model A

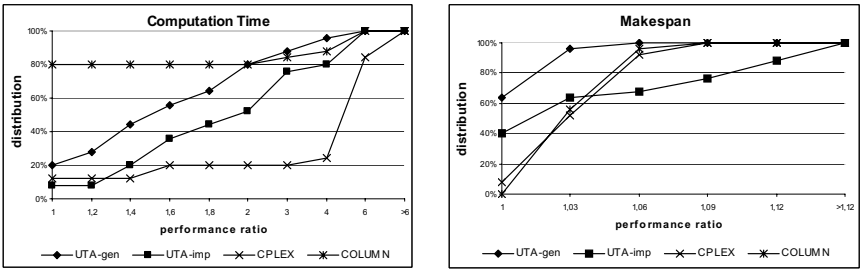


Fig. 2. Performance diagrams for model B

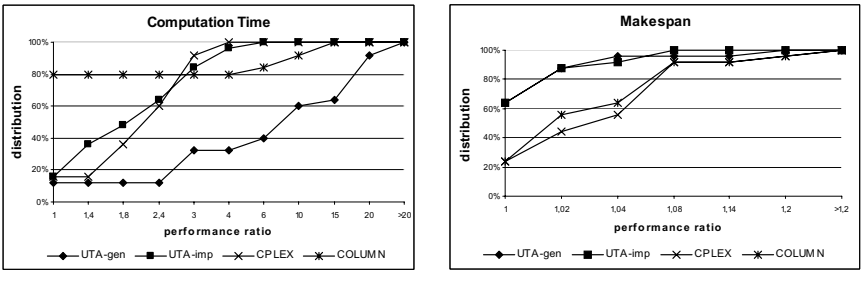


Fig. 3. Performance diagrams for model C

our best knowledge, it is the first time when instances of such sizes were considered in the experimental research on $R||C_{\max}$. The largest instances considered so far contained up to 200 jobs and 20 machines [11, 12, 18]. Obtained results were used to construct performance diagrams, both for the total computation time and the makespan. The value of performance ratio is given always on the x-axis. It shows, for each result of a given instance, the ratio between this result and the best result among all methods for this instance. The y-axis shows, for each method, the percentage of instances which performance ratio is not greater than a given value of performance ratio. Note that the more upper left is a given graph located, the better is the result.

The graphs in Fig. 1, 2 and 3 show the performance diagrams for model A, B and C, resp. In Model A, UTA-GENERIC performed much better (in about 60% of all instances) than the other methods. Clear to see is also its advantage over CPLEX, which achieved here, and in Model B too, the worst processing times. UTA-IMPROVED was, similarly to UTA-GENERIC, in about 90% of all instances at most 1.4 times worse than the best solution. The schedules with the best makespans in Model A, however, were computed with COLUMN. This method achieved also the best computation times in models B and C. Only for small instances ($n = 500, m = 50$), COLUMN was outperformed by other methods. In model B, UTA-GENERIC performed much better than CPLEX. In model C, however, UTA-IMPROVED was better than CPLEX, and much better than UTA-GENERIC. This fact can be explained by the usage of better interphase initialization (a detailed explanation is given in Section 4.3). In model B and C, Unsplittable-Truemper based algorithms returned schedules with the best makespans.

4.2 Computation Time (in Dependence of the Number of Machines)

Figure 4 shows for each model the average computation time in function of the number of machines m. The number of jobs $n = 10m$. The diagrams show on x-axis the number of edges, E, in the bipartite graph of the scheduling problem. Note that the number of edges is equal to nm. Therefore, an increase of E can be interpreted as a growth of the scheduling problem. In the case when the instances grew, both versions of the Unsplittable-Truemper performed in model A and B far better than CPLEX. Moreover, in model A, the sizes of instances for CPLEX were drastically limited. Because of the heavy usage of operational memory, CPLEX was not able to solve instances with more than 5000 edges. Also in cases where it was still possible to deliver solutions, the computation times were very long in comparison with other methods. It can again be explained by the usage of too many memory operations. In model C, unfortunately, UTA-GENERIC was the worst technique. In model B and C, COLUMN was the best method. In model C, the performance of UTA-IMPROVED was due to improved interphase initialization comparable with that of CPLEX. Moreover, for difficult instances, like those from model B and C, both Unsplittable-Truemper implementations needed far less memory, and thus performed more stable and were easier to handle.

4.3 Computation Time (Instances with Constant Number of Jobs)

Figure 5 shows for each model the average computation time in function of the number of machines while the number of jobs stays constant. This describes a situation when the scheduling system evolves and the jobs have more possibilities to be processed. In model A, both versions of Unsplittable-Truemper performed similar, and were much better than COLUMN when the number of machines grew (please notice the logarithmic scale of y-axis). In model B all methods performed quite similar. The results for model C indicate a huge influence of the interphase initialization on the computation time. Here, UTA-GENERIC was much worse than UTA-IMPROVED. The bad performance of UTA-GENERIC can be

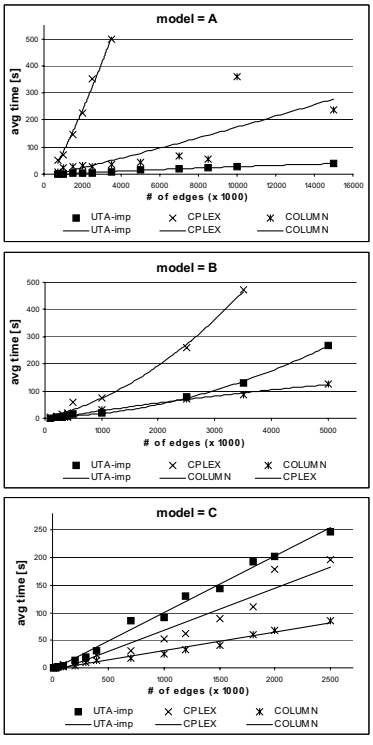


Fig. 4. Computation time with variable number of edges

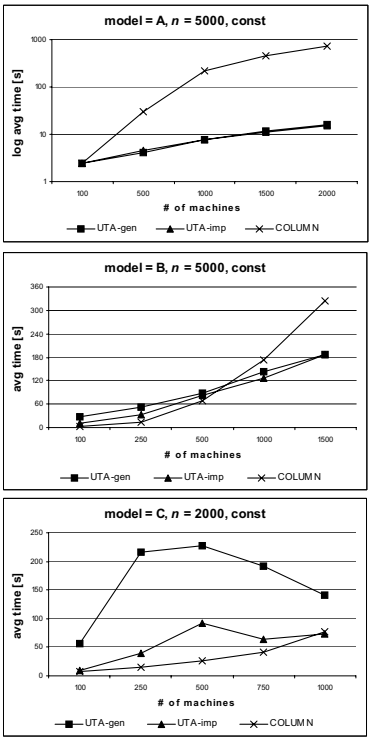


Fig. 5. Computation time with constant number of jobs

explained by the enormous number of inner `while`-loops caused by unbalanced initial assignments. This can be explained by the special property which the initial assignment has to fulfill (the reduced costs must be maintained [5]) and the character of model C (machines with small α_j allow very low processing times), the jobs can be assigned initially only to a few machines, leaving other machines empty. This in turn results in a highly unbalanced initial assignment with a large makespan. Furthermore, the same initialization is used throughout all computations, at the beginning of each binary search step. Consequently, much larger workload in comparison with other models is given to UTA-GENERIC.

5 Conclusions

The experiments show that the Unsplittable-Truemper has an advantage over other methods, especially over CPLEX. In particular, for large test instances, it delivers solutions of the same or better makespans than other approaches, and in most cases is much faster than the LP-based technique. For large instances with correlated machines, however, the column generation approach is the fastest solution method. Moreover, our experiments show that the implementations

of Unsplittable-Truemper require much less operational memory than CPLEX or *branch&bound* methods, what makes it more efficient and easier to handle, especially when the size of the problem grows.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
2. G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.
3. E. Davis and J. M. Jaffe. Algorithms for scheduling tasks on unrelated parallel processors. *Journal of ACM*, 28:721–736, 1981.
4. M. Gairing, T. Lücking, M. Mavronicolas, and B. Monien. Computing nash equilibria for scheduling on restricted parallel links. In *Proceedings of the 36th Annual ACM Symposium on the Thoery of Computing (STOC'04)*, pages 613–622, 2004.
5. M. Gairing, B. Monien, and A. Woclaw. A faster combinatorial approximation algorithm for scheduling unrelated parallel machines. In *Proceedings of 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*, pages 828–839, 2005.
6. P. Glass, C. Potts, and P. Shade. Unrelated parallel machine scheduling using local search. *Mathematical Computing Modeling*, 20(2):41–52, 1994.
7. R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.
8. R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Ken. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.
9. J. Kleinberg. Single-source unsplittable flow. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS'96)*, pages 68–77, 1996.
10. J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.
11. E. Mokotoff and P. Chrétienne. A cutting plane algorithm for the unrelated parallel machine scheduling problem. *European Journal of Operational Research*, 141:515–525, 2002.
12. E. Mokotoff and J. L. Jimeno. Heuristics based on partial enumeration for the unrelated parallel processor scheduling problem. *Annals of Operations Research*, 117:133–150, 2002.
13. T. Radzik. Improving time bounds on maximum generalised flow computations by contracting the network. *Theoretical Computer Science*, 312(1):75–97, 2004.
14. E. V. Shchepin and N. Vakhania. An optimal rounding gives a better approximation for scheduling unrelated machines. *Operations Research Letters*, 33:127–133, 2005.
15. F. Sourd. Scheduling tasks on unrelated machines: Large neighborhood improvement procedures. *Journal of Heuristics*, 7:519–531, 2001.
16. K. Truemper. On max flows with gains and pure min-cost flows. *SIAM Journal on Applied Mathematics*, 32(2):450–456, 1977.
17. P. M. Vaidya. Speeding up linear programming using fast matrix multiplication. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS'89)*, pages 332–337, 1989.
18. S. L. van de Velde. Duality-based algorithms for scheduling unrelated parallel machines. *ORSA Journal on Computing*, 5(2):182–205, 1993.
19. J. M. van den Akker, J.A. Hoogeveen, and S. L. van de Velde. Parallel machine scheduling by column generation. *Operations Research*, 47(6):862–872, 1999.

Implementation of Approximation Algorithms for the Max-Min Resource Sharing Problem*

Mihhail Aizatulin, Florian Diedrich, and Klaus Jansen

Institut für Informatik und Praktische Mathematik,
Universität zu Kiel, Olshausenstr. 40, 24098 Kiel, Germany

Abstract. We implement the algorithm for the max-min resource sharing problem described in [7], using a new line search technique for determining a suitable step length. Our line search technique uses a modified potential function that is less costly to evaluate, thus heuristically simplifying the computation. Observations concerning the quality of the dual solution and oscillating behavior of the algorithm are made. First numerical observations are briefly discussed. In particular we study a certain class of linear programs, namely the computational bottleneck of an algorithm from [8] for solving *strip packing* with an approach from [10, 13]. For these, we obtain practical running times. Our implementation is able to solve instances for small accuracy parameters ϵ for which the methods proposed in theory are out of practical interest. More precisely, the technique from [8] improves the known runtime bound of $O(M^6 \ln^2(Mn/(at)) + M^5 n/t + \ln(Mn/(at)))$ to the more favourable bound $O(M(\epsilon^{-3}(\epsilon^{-2} + \ln M) + M(\epsilon^{-2} + \ln M)))$, where n denotes the number of items, M the number of distinct item widths, a the width of the narrowest item and t is a desired additive tolerance. *Keywords:* Algorithm Engineering, Implementation, Testing, Evaluation and Fine-tuning, Mathematical Programming.

1 Introduction

We consider the max-min resource sharing problem

$$\lambda^* = \max\{\lambda \mid f(x) \geq \lambda e, x \in B\} \tag{R}$$

where $f : B \to \mathbb{R}^M$ is a vector of $M \geq 2$ nonnegative continuous concave functions f_m defined on $B \subseteq \mathbb{R}^M$ (called *block*) which is a nonempty convex compact set; without loss of generality we assume $\lambda^* > 0$. Let $\lambda(x) = \min_{1 \leq m \leq M} f_m(x)$ for any $x \in B$ and let $P = \{p \in \mathbb{R}^M_+ \mid e^T p = 1\}$ be the set of *price vectors*, where e denotes the vector of all ones. For each instance of (R) and $p \in P$ the associated *block problem* is $\Lambda(p) = \max\{p^T f(x) \mid x \in B\}$. We suppose that there is an approximate block solver $ABS(p, t)$ that for any $p \in P$ and $t \in (0, 1)$ computes

* Research of the authors was supported in part by EU Project CRESCCO, Critical Resource Sharing for Cooperation in Complex Systems, IST-2001-33135 and by DFG Project, Entwicklung und Analyse von Approximativen Algorithmen für Gemischte und Verallgemeinerte Packungs- und Überdeckungsprobleme, JA 612/10-1.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 207–218, 2006.

$\hat{x} = \hat{x}(p) \in B$ such that $p^T f(\hat{x}) \geq (1-t)\Lambda(p)$. We study an implementation of the algorithm described in [7] that, provided the existence of $ABS(p,t)$, finds for any $\epsilon \in (0,1)$ a solution to the following problem.

$$\text{compute } x \in B \text{ such that } f(x) \geq (1-\epsilon)\lambda^* e \qquad (R_\epsilon)$$

The number of calls to the block solver is called the *coordination complexity*.

Previous Results and Related Problems. Plotkin et al. in [15], Könemann in [12] and Young in [17] studied the (linear feasibility variant of the) fractional covering problem, respectively, yielding data dependent runtime bounds. Grigoriadis et al. in [7] proposed an algorithm with a runtime bound that is data independent; see [1] for a more detailed comparison. Closely related problems are the fractional packing problem [3, 4, 15, 17] and min-max resource sharing problem [5, 6, 9, 16]; these problems have been studied with approaches similar to [7]. Experimental results have been obtained in [2] where the choice of the step length τ is regarded as an essential decision. Similar results can be found in [14, 9]. We refer the reader to [1] for a more detailed survey of similar studies.

Applications. Many combinatorial optimization problems can be modelled as a max-min resource sharing problem with an exponential number N of variables and a polynomial number M of constraints; we refer the reader to [1] for a more detailed survey. In these applications the block problem is hard to solve or to approximate but can be approximated by a (general) approximate block solver. The running time of these algorithms is dominated by the number of iterations and the running time of the approximate block solver.

New Results. We implement the algorithm for the max-min resource sharing problem described in [7]. Our implementation uses a simplified line search technique for determining a suitable step length. For solving special cases of linear programs (namely relaxations of *strip packing*) the runtime bound of $O(M^6 \ln^2(Mn/(at)) + M^5 n/t + \ln(Mn/(at)))$ from the approach in [10, 13] is improved to $O(M(\epsilon^{-3}(\epsilon^{-2} + \ln M) + M(\epsilon^{-2} + \ln M)))$ with an approach from [8], which has been fine-tuned to yield practical running times. Observations concerning the quality of the dual solution and oscillating behavior of the algorithm are made. Numerical results for random instances are briefly discussed.

In Sect. 2 we describe the algorithm from [7] that we implemented and in Sect. 3 we comment on our implementation. In Sect. 5 we present results from [11] their modification from [8] as well as some computational results. Finally we present some observations concerning performance in Sect. 4.

2 Algorithm Description

The algorithm solves (R) approximately by iteratively computing a sequence of vectors $x_0, \ldots, x_n \in B$. In each step a price vector $p = p(x_i) \in P$ for the current vector $x_i \in B$ is computed and the block solver is called to generate an approximate solution $\hat{x} \in B$ of the block problem. The next vector is set as

$x_{i+1} = (1-\tau)x_i + \tau\hat{x}$ with an appropriate step length $\tau \in (0,1)$. For computing the price vector $p(x)$ the standard logarithmic potential function

$$\Phi_t(\theta, x) = \ln\theta + \frac{t}{M}\sum_{m=1}^{M} \ln(f_m(x) - \theta) \tag{1}$$

is used, where $x \in B$, $\theta \in (0, \lambda(x))$ are variables and t is a tolerance parameter, the same as used for $ABS(p,t)$. The potential function has an unique maximizer $\theta(x)$ for each $x \in B$. The *reduced potential function* $\phi_t(x) = \Phi_t(\theta(x), x)$ measures the quality of the solution. The price vector $p = p(x)$ is defined by

$$p_m(x) = \frac{t}{M}\frac{\theta(x)}{f_m(x) - \theta(x)}, \quad m \in \{1, \ldots, M\}, \tag{2}$$

The parameter $\nu(x, \hat{x}) = (p^T f(\hat{x}) - p^T f(x))/(p^T f(\hat{x}) + p^T f(x))$ is used for deciding the stopping rule; the algorithm can be outlined as follows.

(1) compute initial solution $x^{(0)}$, $s := 0$, $\epsilon_0 := 1/4$;
(2) **repeat** {scaling phase}
  (2.1) $s := s + 1$; $\epsilon_s := \epsilon_{s-1}/2$; $t = \epsilon_s/6$; $x := x^{(s-1)}$;
  (2.2) **while** *true* **do begin** {coordination phase}
    (2.2.1) compute $\theta(x)$ and $p(x)$;
    (2.2.2) $\hat{x} := ABS(p(x), t)$;
    (2.2.2) compute $\nu(x, \hat{x})$;
    (2.2.3) **if** $\nu(x, \hat{x}) \le t$ **then begin** $x^{(s)} := x$; **break**; **end;**
    (2.2.4) compute step length τ and set $x := (1-\tau)x + \tau\hat{x}$;
  end;
  (2.3) **until** $\epsilon_s \le \epsilon$;
(3) return$(x^{(s)})$.

We use $x^{(0)} = 1/M \sum_{m=1}^{M} ABS(e_m, 1/2)$, where e_m denotes the m-th unit vector. The step length from [7] is

$$\tau = \frac{t\theta\nu}{2M(p^T f(\hat{x}) + p^T f(x))} \tag{3}$$

and is chosen in order to guarantee a provable increase of the reduced potential. In [7] computing τ by a line search to maximize $\phi_t(x + \tau(\hat{x} - x))$ is recommended without suggestion of any particular method, a problem that we address in Subsect. 3.1. The following theorem from [7] bounds the coordination complexity.

Theorem 1. *For any given relative accuracy $\epsilon \in (0,1)$ the algorithm above computes a solution x of (R_ϵ) in $N = O(M(\ln M + \epsilon^{-2}))$ coordination steps.*

3 Implementation

We have implemented the algorithm from Sect. 2 in C++. Our implementation uses abstract classes which need to be implemented for each specific application; the following types of problems have been tested.

1. Linear problems with $n = 1$ where the block vector is a number from an interval and the block solver returns a value near to one of the interval bounds as a block solution $\hat{x}$.
2. A multidimensional linear case implemented with CPLEX. Vectors and functions are implemented using CPLEX data structures and the block problem is solved by the CPLEX optimizer. Input is read in a standard MPS-Format supported by CPLEX. This permits easy verification of results by running the input through a CPLEX optimizer that solves the max-min problem.
3. The fractional strip packing problem as described in [11, 8]. The block solver uses an FPTAS for the *unbounded knapsack* problem; the results are presented in Sect. 5.

Type 1 is a special case of type 2, but the simpler block solver allowed tests with more iterations quickly, while running a CPLEX block solver permits about 20 calls per second for small problems.

3.1 Choice of the Step Length

We use a line search to find a step length τ that maximizes the reduced potential ϕ_t instead of using (3). We simplify ϕ_t in order to speed up its evaluation as follows. Let $x, \hat{x}, t$ be as in step **(2.2.4)** and $\theta = \theta(x)$; for each $\tau \in (0, 1)$ and $x' = (1 - \tau)x + \tau\hat{x}$ we define $\tilde{\phi}_t$ as follows.

$$\tilde{\phi}_t(\tau) = \begin{cases} t/M \sum_{m=1}^{M} \ln(f_m(x') - \theta) & \text{if } \theta < f(x') \\ -\infty & \text{otherwise} \end{cases}$$

In the case that $\theta < f(x')$ it is thus simply $\tilde{\phi}_t(\tau) = \Phi_t(\theta, x') - \ln(\theta)$. As $\tilde{\phi}_t(\tau)$ is convex for $\theta < f(x')$, we can use binary search to approximate its maximum. For evaluation we assume that f is linear and compute $f(x') = (1 - \tau)f(x) + \tau f(\hat{x})$ which spares us the actual evaluation of $f(x')$ which could be expensive, e.g. if x' had many non-zero components. In particular we study in the case where f is linear since this is true in Sect. 5. This search technique yields a dramatic improvement of the runtime, as shown in Sect. 4 and Sect. 5. The increase of ϕ_t in each step is usually several orders greater than the one theoretically predicted. See Subsubsect. 4.3 and Subsect. 5.2 for a numerical comparison of both methods. There are several advantages of using $\tilde{\phi}_t$ over calculating with $\phi_t(\tau) = \Phi_t(\theta(x'), x')$. Usage of the same θ for all evaluations of $\tilde{\phi}_t$ allows us to spare the expensive calculation of $\theta(x')$ in every step of the search. It also allows us to leave out the constant summand $\ln(\theta)$ of the potential function which improves the numerical quality; $\ln(\theta)$ often dominates the value of the potential function with the rest being relatively small. Computing with a smaller function

makes the algorithm more sensitive to changes in the function. Concerning the quality of the solution, we observed that there is little difference between τ computed using ϕ_t and $\tilde{\phi}_t$. We neglect the fact that performing search adds to numerical overhead, as the coordination complexity is of main interest.

3.2 An Additional Stopping Rule

The stopping rule using $\nu(x, \hat{x})$ is based on a comparison of primal and dual points. We give an additional criterion that is based on the the quality of the final iterate of the previous scaling phase (or initial solution in the first scaling phase), similar to [9]. We introduce the parameter

$$\omega_s = \begin{cases} 2M(1-\epsilon_1) & \text{for the first scaling phase} \\ (1-\epsilon_s)/(1-2\epsilon_s) & \text{for all other phases} \end{cases}$$

and terminate the current scaling phase as soon as $\lambda(x) \geq \omega_s \lambda(y)$, where x is the current iterate and y is the final iterate of the previous scaling phase (or initial solution in the first scaling phase). An analysis similar to [9] shows that the inequality implies that the last iterate of each phase has the requested quality.

4 Performance

Our tests with CPLEX for small M were carried out with $B = [-100, 100]^n$, $\epsilon = 1/1000$ and exact block solvers returning vertices of B. We used linear functions f_m with uniformly random coefficients. The domains of the coefficient distributions were selected so that the values of the functions lie in $[0, 200]$. The tests were made with $\epsilon_0 = 10$ instead of $\epsilon_0 = 1/4$ as in [7]. This was done so that $t_1 = 5/6$ and the initial scaling phase would not begin with a too small t, causing slow progess. Observations presented in this subsection refer to both the algorithm from [7] and our modification where τ is determined by line search. Our primary goal was to test the dependence of the number of iterations on n and M under such conditions. We present some observations made during testing.

4.1 Improvement of the Dual Solution

We noticed that the primal solution computed by the algorithm from Sect. 2 often has a much higher precision than requested; the algorithm does not terminate as soon as the solution x reaches the desired precision but continues to iterate. Consider the instance

$$n = 1,\ M = 2,\ B = [-100, 100],\ f(x) = (x + 100, -x + 100) \tag{4}$$

where the optimum is $x^* = 0$ yielding $\lambda^* = \lambda(x^*) = 100$. Suppose that in some iteration $x = -0.1$ and $\hat{x} = 100$. The stopping rule gets satisfied if $\nu - t \leq 0$. Plotting $\nu - t$ against t shows that this is not the case for $t \leq 0.02$. However the

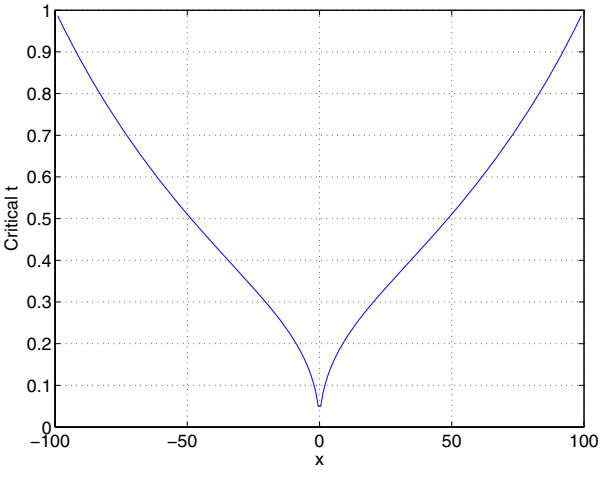


Fig. 1. Dependence of critical t on x

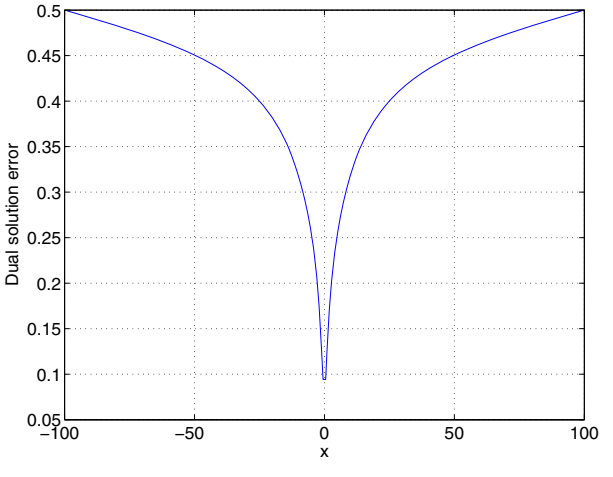


Fig. 2. Error of the dual solution

value $\lambda(x)$ approximates $\lambda^* = 100$ with relative precision of 0.001, which is 20 times smaller. Moreover the ratio of the minimal t for which the stopping rule is satisfied (in the following we shall call such t *critical*) to the actual precision of the current solution differs for different values x of the solution. This is illustrated by the plot of critical t in Fig. 1.

This behavior can be explained by the fact that ν depends on the quality of the dual solution in addition to the quality of the primal solution. Furthermore the dual solution is often improving slower than the primal solution. As a way to measure the quality of the dual solution let us consider the error $\hat{l}$ with which $p^T f(\hat{x})$ approximates λ^*, that is $\hat{l} = 1 - \lambda^*/p^T f(\hat{x})$. This expression evaluated over B (with $\lambda^* = 100$ and $\hat{x} = -100 \cdot \mathrm{sign}(x)$) is shown in Fig. 2, whereas critical

t was used in calculation of p. Using values of t smaller than the critical t results in even worse quality of the dual solution. The behavior described above leads to the fact that primal solutions get calculated to a precision which is several orders higher than the precision actually requested. This effect has been observed in all instances we tested with our algorithm. This is a drawback if one is interested only in the primal solution, which is often the case.

4.2 Oscillations

A typical behavior of the algorithm from Sect. 2 is that the current solution x oscillates around some value that very slowly converges to the optimum. One such example with $n = 2$ and $M = 10$ is shown in Fig. 3. Points connected by segments correspond to the sequence of iterations generated by the algorithm. Points marked with a circle are those in which the tolerance t gets decreased. A contour plot of $\lambda(x)$ is added to the diagram. Notice how the amplitude of the oscillations decreases together with t. Such oscillating behavior occurs when block solutions produced by the block solver lead the algorithm in a direction different from the one in which the potential ϕ_t mainly grows. Small steps are made so that ϕ_t would not get reduced and thus the algorithm makes a lot of iterations while trying to follow the growth of ϕ_t. The interesting fact is that following the growth of ϕ_t often has very little to do with approaching the actual solution. Consider the following example.

$$n = 2,\ M = 3,\ B = [-100, 100] \times [0, 200]$$
$$f_1(x, y) = x + 100, \quad f_2(x, y) = -x + 100, \quad f_3(x, y) = y$$

The optimal set is $N = \{x^* \in B \mid \lambda(x^*) = \lambda^*\} = \{0\} \times [100, 200]$. A graphical presentation of the algorithmic behaviour is omitted due to lack of space, we

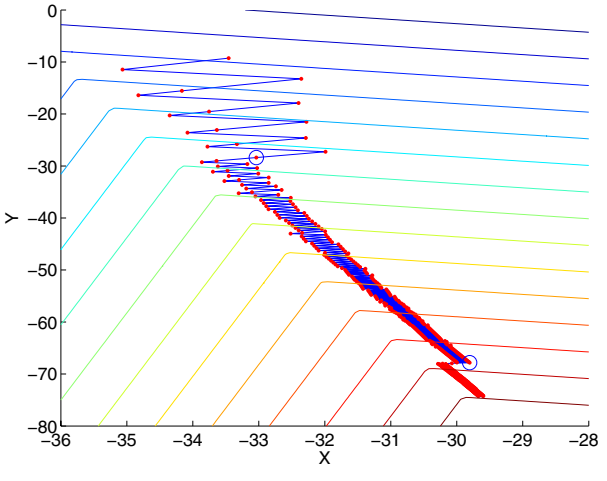


Fig. 3. Oscillations, $n = 2$ and $M = 10$

refer the reader to [1]; the algorithm crosses N in each iteration, but instead of stopping there the search for maximal ϕ_t takes it further resulting in oscillating behavior. This is caused by the function f_3, which plays a significant role in calculating ϕ_t even for $y > 100$. As $f_3(x, y)$ grows with y, so does ϕ_t thus effectively misleading the algorithm, which then tries to increase y more than necessary in each iteration. It is worth noting that oscillations also occur when θ is computed exactly in step length search (see Sect. 3.1) or when fixed step length as in (3) is used. We call functions like f_3 above for which $f(x) > \lambda(x)$ but which still play a significant role in determining ϕ_t *shadow functions*. Oscillations described here actually occur in all nontrivial instances with $n = 2$ that we tested so far. By nontrivial we mean that the solution does not lie in the vertex of B (otherwise it is usually reached within about 10 steps). The described behavior was also observed in cases with $n \gg M$, there however the oscillating pattern is not so distinct. Still it seems that cases with $n \gg M$ are less subject to the problem described in this section. In our tests 10 random instances with $n = 2000$ and $M = 10$ got solved in under 200 iterations, whereas 9 of 10 instances with $n = 4$ and $M = 10$ caused a timeout with more than 900 iterations. The role that oscillations play in large instances needs further investigation.

4.3 Numerical Results

In this subsection we discuss the different choices of τ and the effect of M on the running time.

Comparison of Strategies for Step Length Choice. We compared the runtime of the algorithm with line search for determining τ to the runtime of the algorithm that uses fixed step length from (3). Using our quick block solver for the case $n = 1$ we tested random instances with 4 different values of M, setting $\epsilon = 1/100$. For each M we took the mean of the complexity over 20 instances. The results for line search (rounded to nearest integer) are shown in Table 1. For fixed step size most of the instances had a timeout with tens of thousands of iterations. In case of $n > 1$ the version with fixed step length timeouts with more than 1000 iterations on almost all instances, whereas the version with line search often can solve them in less than 10 iterations.

Dependence of Runtime on M. We performed a series of tests with fixed $n = 500$ and 18 values of M ranging from 2 to 170. For each value of M we took

Table 1. Coordination complexity for line search, $n = 1, \epsilon = 1/100$

M	Mean	Standard deviation
2	9	7
10	16	6
100	34	20
1000	116	127

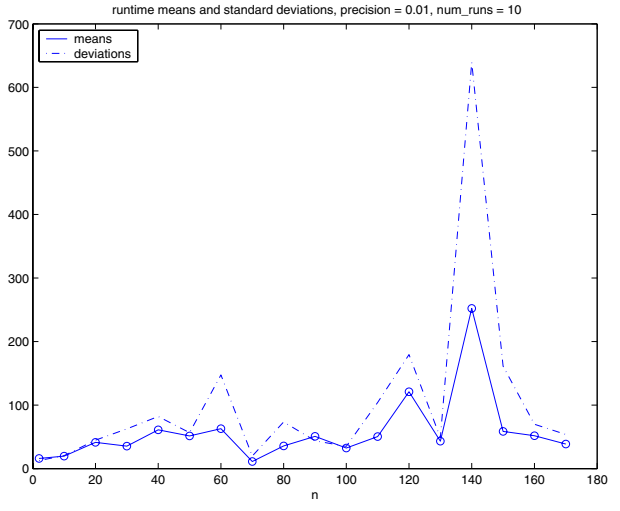


Fig. 4. Plot of mean coordination complexities and standard deviations

average coordination complexity over 10 tests. The results did not reveal any certain dependence of coordination complexity on M – the standard deviation is too high. This is due to the fact that some tests are solved almost instantly (within under 10 iterations) which usually happens when the solution lies in the vertex of the block. However, some single tests need thousands of iterations. This probably happens when oscillations with small step length occur. For plots of mean coordination complexities and standard deviations see Fig. 4. We are currently performing bigger tests with more instances to reveal the dependence on M.

5 Application for Strip Packing

We used our implementation to solve the computational bottleneck of an approximation algorithm for *strip packing* described below using an approach from [8] and [11];we proceed with experimental results.

5.1 Solving Strip Packing Via Fractional Covering

In *strip packing* we are given a list L of n rectangles with widths w_i and heights $h_i \in (0, 1]$ and want to generate an axis-aligned arrangement of L into the strip $[0, 1] \times [0, \infty)$ such that the interiors of the rectangles are disjoint and the packing height is minimized. This problem is NP-hard, but permits an AFPTAS [11]. See [11] and [8] for details; we focus on a corresponding LP model. Suppose M distinct widhts $w'_1, \ldots, w'_M$ occur. A *configuration* is a multiset of widths which sum up to less than 1, i.e. corresponding items can occur at the same level. Let q be the number of *all* configurations $C_1, \ldots, C_q$ and let α_{ij} denote the number of occurences of w'_i in configuration C_j. For each $i \in \{1, \ldots, M\}$ let

β_i denote the sum of all heights of items in L of width w'_i. The *fractional strip packing* problem is defined as

$$\text{minimize } e^T x \text{ subject to } x \geq 0 \text{ and } Ax \geq b \qquad (C)$$

where $x, e \in \mathbb{R}^q$, $A \in \mathbb{R}^{M \times q}$ with $A_{ij} = \alpha_{ij}$ for each i and j and $b \in \mathbb{R}^M$ with $b_i = \beta_i$ for each i. Note that q may grow exponentially in M, causing implementational difficulties which are circumvented by solving

$$\text{compute } x \in B \text{ that satisfies } Ax \geq b \text{ and } e^T x \leq (1+\epsilon)h^* \qquad (C_\epsilon)$$

where h^* denotes the minimal packing height and $B = \mathbb{R}^q_+$. In [8], the algorithm from [7] is used to solve

$$\text{compute } x \in P \text{ that satisfies } Ax \geq (1-\epsilon)\lambda^* b$$

where $P \subseteq \mathbb{R}^q_+$ is a standard simplex and apply some scaling afterwards to obtain a solution of (C_ϵ). Using the algorithm from [7] the block solver is implemented with an FPTAS for the *unbounded knapsack problem* which has a runtime bound of $O(n+\epsilon^{-3})$. The algorithm from [7] is implemented by column generation. We need $O(M\epsilon^{-2} + M \ln M)$ coordination steps, thus the runtime complexity is $O(M(\epsilon^{-3}(\epsilon^{-2} + \ln M) + M(\epsilon^{-2} + \ln M)))$. This is more efficient than $O(M^6 \ln^2(Mn/(at)) + M^5 n/t + \ln(Mn/(at)))$ as obtained in [10, 13].

5.2 Computational Experiments

We used random instances (where *random* means using a pseudo-random number generator) of 100, 1000 and 10000 items, discretized them as in [11] according to $\epsilon \in (0, 1)$ and solved the resulting instances of (C_ϵ) with the algorithm from Section 2 where τ was both statically chosen and determined by line search. We observed that the latter results in a significant reduction of the coordination complexity, see for instance Fig. 5, where we present the averaged number of iterations. In [1] we present more similar results. The termination criterion described in Subsection 3.2 was *never* satisfied – the bound $\omega_s \lambda(y)$ is lower if the quality of y from the previous scaling phase is worse. To put it the other way round, the bound gets worse if the quality of y is better than required. We have to deal with opposing effects – on one hand, we want to leave each scaling phase with a good solution; on the other hand, the better the solution, the less suitable it is for obtaining a good bound for the next scaling phase. We conclude that the criterion from Subsection 3.2 here is of limited heuristic value. In Subsection 4.2 the oscillative behavior of the algorithm is discussed. However, for larger values of M, it is difficult to present oscillation graphically. In the application for *fractional strip packing* we have therefore considered the proportion of block solver calls in which a block solution $\hat{x}$ was returned that had been used before in a previous iteration; see [1] for a more detailed discussion. We observed that the selected configurations are by no means evenly distributed; we omit a graphical presentation due to space limitations. This suggests further investigation. Comparing the results for τ statically chosen to τ determined by line search, the latter does not seem to have any significant effect on this behaviour.

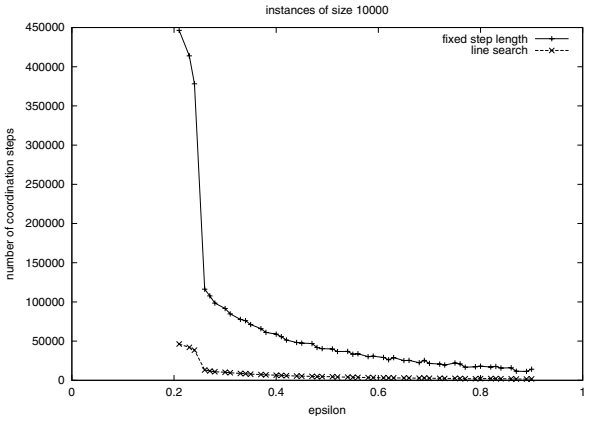


Fig. 5. Averaged number of iterations for instances of size 10000

6 Conclusion

We have implemented an approximation algorithm from [7]; we tested our modified line search for approximating an optimal step length, which turned out to be far superior to using (3). We analyse the improvement rate of the dual solution and find that its precision is growing slower than that of the primal solution. We observed that the runtime for a special class of instances is considerably improved. Interestingly, oscillations are shown, which is undesirable and has not been addressed before; our observation might inspire future research. The results on random instances suggest that the runtime is greatly dependent on the instance, sometimes the problem being solved in several steps and sometimes requiring more than thousand of iterations. Finally we kindly thank the anonymous referees for many helpful comments.

References

1. M. Aizatulin, F. Diedrich, and K. Jansen, Experimental Results in Approximation of Max-Min Resouce Sharing, `http://www.informatik.uni-kiel.de/~fdi/`
2. D. Bienstock, Potential Function Methods for Approximately Solving Linear Programming Problems: Theory and Practice, *Kluwer* (2002).
3. M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. Plotkin, Approximating a finite metric by a small number of tree metrics, *Proceedings 39th IEEE Symposium on Foundations of Computer Science*, FOCS 1998, 379–388.
4. N. Garg and J. Könemann, Fast and simpler algorithms for multicommodity flow and other fractional packing problems, *Proceedings 39th IEEE Symposium on Foundations of Computer Science*, FOCS 1998, 300–309.
5. M. D. Grigoriadis and L. G. Khachiyan, Fast approximation schemes for convex programs with many blocks and coupling constraints, *SIAM Journal on Optimization*, 4 (1994), 86–107.
6. M. D. Grigoriadis and L. G. Khachiyan, Coordination complexity of parallel price-directive decomposition, *Mathematics of Operations Research*, 2 (1996), 321–340.

7. M. D. Grigoriadis, L. G. Khachiyan, L. Porkolab, and J. Villavicencio, Approximate max-min resource sharing for structured concave optimization, *SIAM Journal on Optimization*, 41 (2001), 1081–1091.
8. K. Jansen, Approximation algorithms for min-max and max-min resource sharing problems and applications, *E. Bampis, K. Jansen, and Claire Kenyon (Eds), Efficient Approximation and Online Algorithms*, LNCS 3484 (2006), Springer, 156–202.
9. K. Jansen and H. Zhang, Approximation algorithms for general packing problems with modified logarithmic potential function, *Proceedings 2nd IFIP International Conference on Theoretical Computer Science*, TCS 2002, Kluwer, 255–266.
10. N. Karmarkar and R. M. Karp, An efficient approximation scheme for the one-dimensional bin-packing problem, *Proceedings 23rd IEEE Symposium on Foundations of Computer Science*, FOCS 1982, 312–320.
11. C. Kenyon and E. Rémila, Approximate strip packing, *Mathematics of Operations Research* 25 (2000), 645–656.
12. J. Könemann, Fast combinatorial algorithms for packing and covering problems, *Diploma Thesis, Max-Planck-Institute for Computer Science Saarbrücken*, 2000.
13. B. Korte and J. Vygen, Combinatorial Optimization: Theory and Algorithms, *Algorithms and Combinatorics* 2 (2000), Springer.
14. Q. Lu and H. Zhang, Implementation of Approximation Algorithms for the Multicast Congestion Problem, *Proceedings of the 4th International Workshop on Efficient and Experimental Algorithns*, WEA 2005, LNCS 3503, Springer, 152–164.
15. S. A. Plotkin, D. B. Shmoys, and E. Tardos, Fast approximation algorithms for fractional packing and covering problems, *Mathematics of Operations Research*, 20 (1995), 257–301.
16. J. Villavicencio and M. D. Grigoriadis, Approximate Lagrangian decomposition with a modified Karmarkar logarithmic potential, *Network Optimization, P. Pardalos, D. W. Hearn, and W. W. Hager (Eds), Lecture Notes in Economics and Mathematical Systems* 450 (1997), Springer, 471–485.
17. N. E. Young, Randomized rounding without solving the linear program, *Proceedings 6th ACM-SIAM Symposium on Discrete Algorithms*, SODA 1995, 170–178.

Column Generation Based Heuristic for a Helicopter Routing Problem

Lorenza Moreno[1], Marcus Poggi de Aragão[1], and Eduardo Uchoa[2]

[1] Departamento de Informática, PUC-Rio, Rua Marques de São Vicente, 225, Rio de Janeiro, RJ, 22451-900, Brazil
{lorenza, poggi}@inf.puc-rio.br
[2] Departamento de Engenharia de Produção, Universidade Federal Fluminense, Rua Passo da Pátria 156, Niterói, RJ, 24210-240, Brazil
uchoa@producao.uff.br

Abstract. This work presents a column generation based heuristic algorithm for the problem of planning the flights of helicopters to attend transport requests among airports in the continent and offshore platforms on the Campos basin for the Brazilian State Oil Company (Petrobras). We start from a previous MIP based heuristic for this Helicopter Routing Problem and add column generation procedures that improve the solution quality. This is done by extending the earlier formulation and providing an algorithm to find optimal passenger allocation to fixed helicopter routes. A post optimization procedure completes the resulting algorithm, which is more stable and allows consistently finding solutions that improve the safety and the cost of the one done by the oil company experts.

1 Introduction

Helicopter routing problems often comprise pickups and deliveries of passengers. This characteristic brings a packing aspect difficult to capture to a routing problem. The particular Helicopter Routing problem here addressed generalizes most similar routing problems in the sense that it considers the activities of a fleet of aircrafts during a day comprising several subsequent routing problems. Moreno et al. [7] proposed to find quality solutions to this problem with a heuristic algorithm that uses a mixed integer program with exponentially many columns. This heuristic consists of constructing, a priori, two large sets of columns obtaining a good integer solution to the resulting MIP and applying a local search to find its best solution. No column generation was used, i.e. no information from the linear programming relaxation was taken into account when constructing the potential helicopter routes (columns). The purpose of the present work is to fill this gap, investigating whether or not column generation provides a better and more stable algorithm.

The resulting algorithm was developed for the Brazilian State Oil Company (Petrobras) and is now operating at the flight control center in the city of Macaé. This company concentrates most of its oil exploration and production activities

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 219–230, 2006.

in an offshore area - the Campos Basin. The personnel transportation to and from drilling platforms in this area (42,000 passengers per month) is done by a mixed fleet of 35 helicopters with an average of 70 flights per day. Planning these flights is a difficult task since transport requests must be attended on time, there are usually few helicopters available per day and many safety policies must be observed.

The Helicopter Routing Problem (HRP) tackled in this work is: given a set of locations composed by bases (or airports) and offshore platforms, a set of helicopters, and a set of transport requests which are distributed over departure times associated to a list of platforms that can be served, construct a flight schedule satisfying the following constraints: (i) each flight starts and finishes in a base; (ii) the helicopters capacity can not be exceeded during each flight; (iii) a helicopter must have a preparation time between flights. The goal is to minimize the total cost.

In other words, the HRP plans all the flights for each day, which corresponds to the activities of each helicopter (the sequence of stops, the time they occurred, and the passengers that boarded and unboarded).

In the case studied, the basin has 2 airports and 65 offshore locations. Platform crews can demand transportation for one of a few (nine) flight departure times and their requests are either (partially) attended on time or ignored, since delays are not allowed. These passengers can go from base to platform, from platform to base or from one platform to another. There are a few passengers that change from one platform to another. They are usually grouped into a longer flight with special rules such as more landings and offshore refueling and, for this reason, they are treated apart. There is a high cost for leaving passengers unattended, because oil exploration activities can be compromised.

The helicopters are paid per hour in flight and have distinct sizes and costs, i.e. the fleet is not homogeneous. The helicopter capacity (number of passengers that can be transported) depends on the length of the flight because the allowed take-off weight must include not only passengers weight but also the fuel weight. A helicopter can fly at most five times per day but it must be checked before each flight and it must stop for an hour in the middle of the day to give the pilot a lunch break.

The helicopters do not belong to the oil company. They are operated by other companies which maintain different contracts regarding flight hour costs for each helicopter. The two airports do not share helicopters, i.e. each one has its own fleet. This allows solving one separated problem for each airport. This is possible as long as intersecting departure times from different airports do not serve a same platform. Should this happen, we would have to consider a coupling constraint limiting the number of flights serving this shared platform.

Furthermore, the following rules must be respected: the number of landings for each passenger and for each flight is limited; at each platform, the aggregate number of landings for flights with the same departure time is also limited. The built flight schedule must indicate helicopter, route, passengers and duration of each flight.

This Helicopter Routing Problem is NP-hard. It is so since it can be easily seen as a generalization of the Split Delivery Vehicle Routing Problem (SDVRP) (Dror and Trudeau [4], Dror, Laporte and Trudeau [3]) which was proved to be NP-Hard when the vehicle capacities are 3 or more by Archetti, Mansini and Speranza [1]. In the SDVRP the fleet is homogeneous, there is just one departure time and, most of all, there are only deliveries.

In the 1980's, Galvão and Guimarães [5] worked on this problem in Petrobras. They proposed an algorithm for building routes of the same departure time which used different strategies to create the routes, and at the end selected the set of routes with minimum cost. In their algorithm, the fleet used in each departure time had to be chosen by the user, which is not the case in the present work. Their paper addresses also the issue of the relationship among users, project technical staff and the management group inside the oil company. They depict a situation where users feared loosing their jobs and management feared the quality of the automated solutions would not match the ones obtained by hand. Fifteen years later, there has been a clear evolution in the understanding of optimization tools and their potentiality. Despite that, management considers the testing to be critical to make sure the solutions obtained by an automated tool can be implemented and are at least as good as the ones assembled by hand.

Another similar experience can be found in a Dutch gas exploration company, Tjissen [8] used SDVRP to work on another helicopter routing real case where helicopter capacity was constant and for each passenger left on an offshore platform there was another to go back to the continent. Good solutions were found using rounding procedures to linear programming solutions and heuristics.

Hernadvolgyi [6] used as an example another particular case of helicopter routing problem when all demands can be carried out by just one helicopter. The problem studied was the Sequential Ordering Problem, which can be seen as a version of the Asymmetric Traveling Salesman Problem with precedence constraints.

This text is organized as follows. The next section presents a MIP model with exponentially many variables and discusses column generation along with a procedure to find new profitable columns. Section 3 describes the new column generation based heuristic algorithm. The last section presents computational experiments and draws some insights on this difficult problem.

2 Model and Column Generation

The HRP can be formulated as a mixed integer program (MIP). Sets of constraints controlling demand satisfaction and offshore platforms (and airports) utilization (number of landings) can be labeled as global constraints. On the other hand, constraints regarding one single helicopter's day of work can be thought as local ones. They enforce the helicopters' sequence of flights to have flights with a limited duration, to respect weight capacity throughout the flights, and not to exceed a maximum number of total landings and landings per passenger. Also, they have a given maximum number of flights, maximum number of hours to fly and a pilot lunch break.

A multicommodity flow MIP is presented in Moreno et al.[7] providing a formulation with polynomially many variables and constraints. As should be expected, this formulation usually has a large integrality gap and is even unlikely to provide reasonable integer feasible solutions. Nevertheless, it gives a straight forward formulation with exponentially many variables by applying Dantzig-Wolfe decomposition and treating the local constraints implicitly in the construction of the helicopters' sequence of flights. This decomposition is further explored in [7] by considering the pilot lunch break requirement and the number of daily flights and hours per helicopter as global constraints and having variables associated to flights in each departure time. This last formulation was exploited in [7] to produce a heuristic algorithm. It proceeded by constructing, a priori, two large sets of variables. The first large set of variables focuses on constructing sequence of flights for each helicopter, i.e. sets of flights that can be combined to form a helicopter work day. The second set contains sets of flights that are solutions for the demands associated to each departure time. The resulting MIP is solved by an integer programming commercial package to find a good integer solution to which is subsequently applied a local search procedure.

We proceed by presenting the formulation with exponentially many variables in [7], showing its drawbacks and how to overcome them to obtain an effective column generation procedure. Next, we present the column generation subproblem and a procedure to find negative reduced cost columns.

2.1 A MIP Model with Exponentially Many Variables

Let the problem parameters be as follows. Denote by D the set of demands and by T the set of flight departure times. Let H be the set of helicopters, L be the set of all locations and P be the subset of L containing all platforms. Denote by D_t the subset of the demands in D to be attended in departure time t. Time is discretized in order to control the lifetime of each helicopter. Finally, let I denote the set of all time instants considered. The cardinality of the sets L, D, T, P, H and I is represented by nl, nd, nt, np, nh and ni, respectively. The following values are also part of the input data: q_d is the number of passengers of a demand d to be transported; c_h is the cost of each minute of flight for helicopter h; mc_h is the maximum capacity of helicopter h; lp is the maximum number of landings per passenger; lf is the maximum number of landings per flight; mL is the maximum number of landings in each departure time on the same platform; mF is the maximum number of flights of each helicopter in a day; mH is the maximum number of hours of flight of each helicopter in a day; M is the cost of leaving a passenger unattended; and lc is the cost of each landing.

This model has three sets of variables. The first one is associated with all possible flights each helicopter can perform in each of the departure times. Remark that a helicopter flight consists of a sequence of legs, i.e. one flight from the airport to the first platform, then to another platform, and so on until reaching back the airport. The second set contains variables representing unsatisfied demand. The last set represents the instants in which the pilots begin their lunch breaks. The flights are specified by their cost and row coefficients. The variables are x_{hf},

the flight f of helicopter h (binary), s_d, the number of passengers of demand d not transported (integer), and z_{hj}, the lunch break of the pilot of helicopter h starting at instant j (binary). The coefficient a_{dhf} represents the number of passengers of demand d transported by the flight f of helicopter h (integer), while df_{hf} is the duration (in minutes) of the flight f of helicopter h (integer) and pf_{hf} is the number of platform landings of flight f of helicopter h (integer).

To ease the understanding of the model, denote by F_{ih} (resp. J_{ih}) the set composed by the indices of all flights f (resp. lunch breaks j) that uses the helicopter h at instant i. Also, let K_{pt} be the set containing all flights of departure time t with landing on platform p, and let J_h be the set of all lunch breaks of helicopter h. The MIP model follows:

$$\min \sum_{h=1}^{nh} \sum_{f=1}^{nf} (c_h.df_{hf} + lc.pf_{hf}).x_{hf} + \sum_{d=1}^{nd} M.s_d \tag{0}$$

$$\sum_{h=1}^{nh} \sum_{f=1}^{nf} a_{dhf}.x_{hf} + s_d = q_d \qquad \forall d \in \{1..nd\} \tag{1}$$

$$\sum_{h=1}^{nh} \sum_{f \in K_{pt}} x_{hf} \leq mL \qquad \forall p \in \{1..np\}, \forall t \in \{1..nt\} \tag{2}$$

$$\sum_{f \in F_{ih}} x_{hf} + \sum_{j \in J_{ih}} z_{hj} \leq 1 \qquad \forall i \in \{1..ni\}, \forall h \in \{1..nh\} \tag{3}$$

$$\sum_{j \in J_h} z_{hj} = 1 \qquad \forall h \in \{1..nh\} \tag{4}$$

$$\sum_{f=1}^{nf} x_{hf} \leq mF_h \qquad \forall h \in \{1..nh\} \tag{5}$$

$$\sum_{f=1}^{nf} df_{hf}.x_{hf} \leq mH_h \qquad \forall h \in \{1..nh\} \tag{6}$$

$$x_{hf} \in \{0,1\} \qquad \forall h \in \{1..nh\}, \forall f \in \{1..nf\} \tag{7}$$

$$z_{hj} \in \{0,1\} \qquad \forall j \in \{1..ni\}, \forall h \in \{1..nh\} \tag{8}$$

$$s_d \text{ integer} \tag{9}$$

The objective function (0) minimizes the total cost, which is the weighted sum of numbers of passengers not transported, total of landings and hours flown by the helicopters. Constraints (1) control the passengers transported from each demand. Constraints (2) are used to ensure that at most mL flights with departure time t will land on platform p. Constraints (3) state that at most one flight or one lunch break of each helicopter h can occur at each instant i. The helicopters' stop for the pilot's lunch break are assured by constraints (4). The number of flights and hours of flight of the helicopters are limited by constraints (5) and (6), respectively. Finally, (7), (8) and (9) specify the domain of variables x, z and s, respectively.

The number of possible valid flights is exponential and, in this problem, it is difficult to foresee which flights are used in good solutions and to decide how some demands, which may have 2 or 3 times more passengers than the helicopter capacity, shall be split. This gives an idea of the difficulties in deriving algorithms

to implicitly consider all the possible flights. These difficulties exist because in addition to determining the flight routes, the quantities of passengers that are attended from each demand must also be specified. In fact, it seems that this partitioning aspect of the problem is much more critical than the routing aspect.

When tailoring a column generation procedure to implicitly generate columns with smallest reduced costs we can observe the following difficulty. The dual variables associated with constraints (1) can be positive or not. If they are, they give the same weight to all passengers in a same demand. This implies that in any optimal solution of the column generation subproblem the route of a helicopter will obtain smallest reduced cost by choosing the demands in decreasing order of their dual variables, picking always the maximum possible number of passengers of each demand. This suggests that column coefficients would often be either the total number of passengers of the demand or the remaining capacity in the helicopter. Therefore, the required columns would have little chance of being generated.

We overcome this problem by splitting constraints (1), which amounts to have only individual demands. Therefore, no change in the formulation is required. Now, each passenger is treated as an independent demand and, consequently, the coefficient a_{dhf} only indicates whether the corresponding passenger is in the flight or not (1 or 0). Although this enlarges the problem size, it is hard to notice any increase in the linear programming resolution time when solving the real problems in the Campos basin. The number of demands is around 150 and the number of passengers ranges from 700 to 1100.

2.2 Column Generation Subproblem

Let π_d, α_{pt}, β_{hi}, γ_h, σ_h be the dual variables associated to constraints (1), (2), (3), (5) and (6) respectively. Let also $R(hf)$, $IR(hf)$ and $D(hf)$ denote the set of platforms visited, the set of instants during which flight f occurs and set of demands flight f of helicopter h carried, respectively. The reduced cost of a variable x_{hf} is then given by the sum of $\overline{c}_R$, which depends only on the route of the helicopter, with $\overline{c}_D$ which is determined by the passengers (demands) it takes. They can be expressed as:

$$\overline{c}_R \quad = \quad c_h.df_{hf} + \sum_{p \in R(hf)} (lc - \alpha_{pt}) - \sum_{i \in IR(hf)} \beta_{hi} - \gamma_h - df_{hf}.\sigma_h$$

and

$$\overline{c}_D \quad = \quad \sum_{d \in D(hf)} -\pi_d.$$

The column generation subproblem is to find the route and the demands it attends that minimize $\overline{c}_{hf} = \overline{c}_R + \overline{c}_D$ and satisfies the local constraints, which are: (i) number of landings per passenger shall not exceed lp; (ii) the landings per flight cannot be more than lf; and (iii) given the duration of the flight, the maximum number of passengers at any moment in the flight cannot exceed mc_h.

This problem is clearly NP-hard, since the Prize Collecting TSP (Balas [2]) corresponds to the special case where the constraints are disregarded and all dual variables, except for the π_d ones, are zero. Since the focus in this work is on finding good primal feasible solution to the HRP, we next describe a heuristic procedure.

2.3 Column Generation Procedure

Our procedure is designed to take full advantage of the particular HRP we are addressing. Most of the departure times have a small number of platforms to serve, usually around 10, although there is one departure time which often has 30 or more platforms to serve. In this sense, we observe that once the route is defined, the optimal passenger assignment can be found, in polynomial time, by solving a Minimum Cost Flow (MCF) problem which has a small network. We add to that the fact that the maximum number of landings allowed in any flight (mL) is set to 6 for safety reasons at the oil company. These constraints do not change the problem complexity. Nevertheless, the small number of landings allowed and of platforms may turn acceptable to enumerate all possible routes. For time limit reasons, we used a heuristic approach.

The resulting procedure tackles the problem by separately searching for flights serving a fixed number of platforms which, in the present case, is at most 5. It proceeds by generating all possible routes with 1 and 2 offshore landings. For 3, 4, and 5 offshore landings it starts from an initial random route and performs a local search by exploring a neighborhood consisting of exchanging the platform at each position in the route with all other platforms to be served in the same departure time. The procedure stops at the local search when it finds a column with negative reduced cost. When this is not the case, a Tabu Search procedure with this same neighborhood is started. Note that a MCF problem is solved for each neighbor route that is explored, which is sometimes time consuming. Figure 1 depicts this procedure.

The procedure presented above is invoked for each helicopter at each departure time. The MCF model completes its description. The network has two distinct sets of nodes: stop nodes and demand nodes. The stop nodes are created for each point of the flight route (base, platforms and back to the base). Each flight segment between consecutive landing points is represented by an arc from its origin to destination. These arcs control the flow of passengers in the route and, for this reason, arc capacities are exactly the capacity of the helicopter in this route (which depends on the flight duration). Note that this is a single-commodity network flow problem.

Demand nodes are created for each passenger that can travel in this flight. Two arcs leave each demand node. One goes to the node corresponding to the origin of the demand on the route with cost equal to the demand reduced cost. The other arc goes to demand destination with infinite capacity and zero cost. Then, in this model, each passenger can achieve its destination either going from its demand node to his origin node traversing route segments of the flight, when served by the helicopter, or going directly from the demand node to the destination point when

```
01  Procedure Fixed Size Route Procedure (initial flight){
02      best flight ← initial flight
03      while best flight cost is improved {
04          for each iteration {
05              best neighbor ← null
06              for each neighbor {
07                  Update route
08                  Select passengers solving a MCF problem
09                  Compute neighbor cost
10                  if cost is better than best neighbor and
11                     move is not tabu
12                      best neighbor ← current neighbor
13              }
14              if the current flight is better than best neighbor {
15                  if cost is negative
16                      return current flight
17                  else
18                      set last move as tabu
19              }
20              current flight ← best neighbor
21              if current flight is better than best flight
22                  best flight ← current flight
23          }
24      }
25      return best flight
26  }
```

Fig. 1. Fixed Size Route Procedure

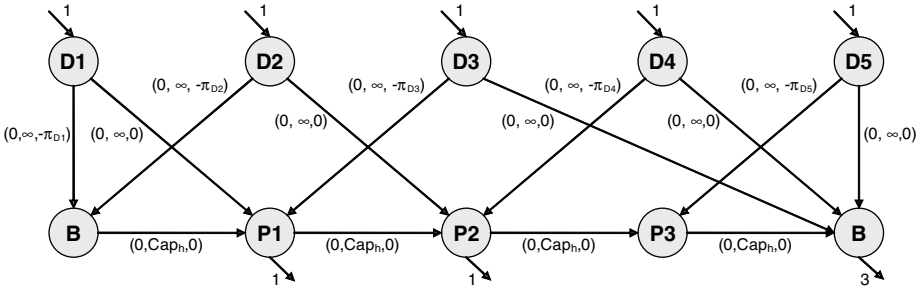


Fig. 2. Minimum Cost Flow problem network example

not. Only passengers with positive associated dual variables in (1) (negative cost) need to be considered. To obtain flights with as many passengers as possible, we consider the zero valued dual variables of (1) as slightly positive.

Figure 2 illustrates the MCF model. Each demand node (D1 to D5) has an incoming flow of one passenger. The outgoing flow of one unit is at its destination

node. Helicopter capacities are controlled by route segment arcs linking two stop nodes. The optimum flow gives the smallest $\overline{c}_D$ for a given route. The flight reduced cost is then computed by adding the previously known value $\overline{c}_R$.

3 Column Generation Based Heuristic Algorithm

The approach used to solve this HRP problem is to decompose the problem into the generation of single flights for each helicopter available and the assembly of these flights. This assembly is done by an integer programming model that constructs each helicopter's sequence of flights assuring that it meets all time related constraints while covering the transportation requests.

The algorithm starts by generating two reasonable size sets of columns as in [7]. One set is composed by sets of flights that compose helicopters' days of work and the other contains sets of flights that completely serve departure times. The restricted integer program (RIP) is initialized with these two sets of columns. At this point, the column generation phase is initiated. The linear programming relaxation of the RIP is repeatedly solved to optimality. At each iteration the dual values are obtained and used in the column generation procedure above described. One column is generated for each departure time - helicopter pair. Since these columns tend to be similar to different helicopters and also be extremal, a tailing off effect is likely to appear. Moreover, this algorithm aims at finding good integer solutions not optimal ones.

With this in mind, we add a random column generation procedure. It proceeds by randomly creating flights for randomly selected helicopters following the guidelines of the second set of columns created at the initialization, i.e. in sets that serve completely each of the departure times. A fixed number of flights is generated and the 20% with smallest reduce cost is added to the RIP. Also, to allow complementing flights to be added to the RIP, we add columns from both column generation procedures even when the reduced cost is positive.

The column generation phase is interrupted after 15 minutes. In our experiments, our heuristic column generation procedure may continue to find negative cost columns for as long as one hour or two. This would exceed the allowed computation time. We proceed by attempting to find good, or even optimal, solutions to the current RIP is made for 5 minutes. Even when we provide an initial solution to the problem, it converges very slowly and integer solutions are hard to find. To ease the solution of the MIP, we relax constraints (1) from equality (partitioning) to greater or equal inequalities (covering). In other words, we allow over satisfying the demand. However, with this change in the model, it is necessary to check if there are extra passengers in the solution.

The generation of a valid flight schedule is done by heuristic algorithms. Some post processing is necessary in order to remove exceeded demand. Finally, heuristic algorithms are also used to check if further local improvements are possible.

The post optimization removes extra passengers by efficiently solving a mixed integer program. Since helicopters and their routes are already defined, the model is used to remove each exceeded demand of the flights for each departure time.

Let D_t, and F_t be the set of passengers and the set of flights of departure time t. For each flight in F_t, consider S_f and L_f the sets of route segments and set of landing points, respectively. Let D_t^f be the subset of demands that can be transported by the flight f, D_t^{fs} be the subset of demands that traverses the segment s in flight f and D_t^{fl} be the subset of demands whose origin or destination occurs in the landing point l of flight f. This model has the following variables: k_{df} indicates how many passengers of demand d travels in flight f (integer); w_f indicates whether the flight f occurs (binary); and y_{fl} indicates if the flight f lands on platform l of its route (binary). Additional parameters are cap_f and c_f, the capacity and cost of flight f, respectively. The model can be written:

$$\min \sum_{d \in D_t} \sum_{f \in F_t} -M.k_{df} + \sum_{f \in F_t} c_f.w_f + \sum_{f \in F_t} \sum_{l \in L_f} lc.y_{fl} \tag{10}$$

s. t.

$$k_{df} - cap_f.y_{fl} \leq 0 \qquad \forall d \in D_t^{fl} \tag{11}$$

$$k_{df} - cap_f.w_f \leq 0 \qquad \forall d \in D_t^f \tag{12}$$

$$\sum_{f \in F_t} k_{df} \leq q_d \qquad \forall d \in D_t \tag{13}$$

$$\sum_{d \in D_t^{fs}} k_{df} \leq cap_f \qquad \forall s \in S_f, \forall f \in F_t \tag{14}$$

$$w_f \in \{0,1\} \qquad \forall f \in F_t \tag{15}$$

$$y_{fl} \in \{0,1\} \qquad \forall l \in L_f, \forall f \in F_t \tag{16}$$

$$k_{df} \text{ integer} \tag{17}$$

```
01  Procedure CG-HRP {
02      Generates random flights
03      while the time limit is not reached {
04          while the 15 minutes bound is not achieved {
05              Solve LP relaxation
06              Execute Column Generation Procedure
07              Execute Random Column Generation
08          }
09          Execute MIP solver algorithm
10      }
11      Execute post optimization
12  return
13  }
```

Fig. 3. Column Generation Heuristic for the HRP

The objective function maximizes the number of passengers attended and minimizes the number of flights and landings. Constraints (11) guarantee that the landing point l of the flight f will be visited if and only if there are passengers leaving or going to this point. In the same manner, constraints (12) keep or eliminate flight f. Constraints (13) assure the removal of extra passengers since

it controls the number of passengers of each demand in all flights. Constraints (14) force the number of passengers in each route segment to be lesser than helicopter capacity.

Observe that flights and landings of the solution are kept in schedule only if they are necessary. There can be lots of changes in the solution since passengers can travel in any flight that visits their origin and destination. The algorithm obtains the best possible distribution of passengers given the solution flights. Although this problem is NP-hard, it can be efficiently solved to optimality in a few seconds due to its low dimension. Figure 3 presents a pseudo-code of the complete heuristic algorithm.

4 Experiments

The algorithm was tested on 8 real instances taken from the year of 2005. The tests were executed on a Pentium IV 3.0 GHz with 1 GB of RAM. Mixed integer programs were solved using ILOG CPLEX 9.0. All data used on testing was obtained during the algorithm tests phase at the oil company. Particularly, during the period in which these instances were extracted, there were not enough helicopters available to satisfy the company demand. For this reason, some instances were very difficult to solve and there were too many unattended passengers. There are 4 instances for each of the two bases. Each base has a distinct demand and fleet profile and the respective problems are quite different. The operation in base 1 (Macaé), instances 1-4, has many demands with few passengers. Therefore, several platforms are visited in each flight departure time. Base 2 (São Tomé), instances 5-8, only has passengers' exchanges, i.e. for each person going from base to platform there is another person from platform to the base. These exchange demands usually consist of large groups of passengers. Consequently, the helicopters typically have shorter routes and higher occupancy, and fewer helicopters are necessary in base 2. The instances of base 1 have an average of 780 passengers, 16 helicopters and requires more than 70 landings, while these numbers for base 2 are 600, 6 and 25.

The objective function cost assigns 1 million units per passenger unattended, 10 thousand per landing, and roughly 10 units per minute of flight (it depends on the helicopter used and range from 4 to 20). The tests reported below compare the previous algorithm (from Moreno et al. [7]) running with a CPU time limit of 40 minutes, with the proposed algorithm CG-HRP with CPU time limits of 40 and 60 minutes. We remark that the previous algorithm calls the MIP solver only once, just after constructing all the initial columns. The CG-HRP calls to the MIP solver limits its CPU time to 5 minutes. In all tests the number of columns generated either in the initialization or in the random column generation procedure was proportional to the sum of the products of the number of helicopters, number of departure times and the number of platforms to be attended in each departure time. Table 1 presents the optimal **LP value**, the best integer solution found after the post processing **Best Int** and the total number of columns **# Cols** of the final RIP for each instance and each of three runs of the algorithms, respectively.

Table 1. Previous algorithm (40'), CG-HRP (40') and CG-HRP (60')

Inst.	*LP value*	*Best Int*	*#Cols*	*LP value*	*Best Int*	*#Cols*	*LP value*	*Best Int*	*#Cols*
1	11,936,902	45,976,604	83148	9,684,181	59,016,398	22686	9,089,961	35,056,338	28101
2	885,537	35,863,655	49987	864,796	41,945,838	25089	856,076	12,904,707	33476
3	866,562	14,886,517	70848	862,918	1,908,200	26427	861,631	908,082	32063
4	763,848	27,681,760	61794	760,726	8,814,358	28183	759,217	8,814,358	39955
5	28,037,579	381,355	13012	322,319	17,401,650	8728	322,175	381,303	15641
6	16,837,377	19,431,951	12702	13,028,145	39,391,543	10781	12,704,151	22,401,593	19305
7	46,637,814	40,339,832	12894	39,290,050	50,340,030	12062	39,289,194	42,329,967	23612
8	100,261,168	102,277,756	12959	92,657,672	102,257,841	18703	92,657,655	100,247,778	24726

It can be observed that the previous approach is quite unstable relying on the post processing to find solutions that are even better than its LP relaxation. As expected the column generation approach always improved the LP relaxation value although in 2 out of the 8 instances it failed to obtain the best integer solutions. Nevertheless the CG-HRP best integer values either beat badly the previous approach or loose by little. This suggests that investing in column generation and perhaps in branching is the way to go.

Acknowledgments. We thank three anonymous referees for their helpful comments. LM, MPA and EU were partially supported by CNPq grants 140715/2004-5, 302086/2003-0, and 304533/2002-5, respectively.

References

1. Archetti C., Mansini R., Speranza M. G.: The Split Delivery Vehicle Routing Problem with Small Capacity, Technical Report, Dipartimento Metodi Quantitativi - Universita degli studi di Brescia, 2001.
2. Balas E.: The Prize Collecting Traveling Salesman Problem, Networks 19 (1989) 621-636
3. Dror M., Laporte G., Trudeau P.: Vehicle routing with split deliveries, Discrete Applied Mathematics, 50: 239-254, 1994.
4. Dror M., Trudeau P.: Savings by Split Delivery Routing, Transportation Science, 23: 141-145, 1989.
5. Galvão R., Guimarães J.: The control of helicopter operations in the Brazilian oil industry: Issues in the design and implementation of a computerized system, European Journal of Operational Research, 49: 266-270, 1990.
6. Hernadvolgyi I.: Automatically Generated Lower Bounds for Search. PhD thesis, University of Ottawa, 2004.
7. Moreno, L., Poggi de Aragão, M., Porto, O., Reis, M.L.: Planning Offshore Helicopter Flights on the Campos Basin. XXXVII Simpósio Brasileiro de Pesquisa Operacional (SBPO), Gramado, Brazil, 2005.
8. Tijssen G. A.: Theoretical and practical aspects of linear optimization. PhD thesis, Rijksuniversiteit Groningen, 2000.

Kernels for the Vertex Cover Problem on the Preferred Attachment Model*

Josep Díaz, Jordi Petit, and Dimitrios M. Thilikos

Departament de Llenguatges i Sistemes Informàtics,
Universitat Politècnica de Catalunya,
Campus Nord Ω-228,
08034, Barcelona, Spain

Abstract. We examine the behavior of two kernelization techniques for the vertex cover problem viewed as preprocessing algorithms. Specifically, we deal with the kernelization algorithms of Buss and of Nemhauser & Trotter. Our evaluation is applied to random graphs generated under the preferred attachment model, which is usually met in real word applications such as web graphs and others. Our experiments indicate that, in this model, both kernelization algorithms (and, specially, the Nemhauser & Trotter algorithm) reduce considerably the input size of the problem and can serve as very good preprocessing algorithms for vertex cover, on the preferential attachment graphs.

1 Introduction

Given a graph G and a non-negative integer k, the Vertex Cover problem asks whether k of the vertices of G are endpoints of all of its edges. The minimum k for which a graph G has a vertex cover of size k or less is called the *vertex cover* of G and is denoted as $\mathbf{vc}(G)$.

Vertex Cover was one of the first problems proven to be NP-complete [14] and, since then, a lot of efforts have been done in theory and in practice towards coping with its computational intractability. In this direction, there were several advances on the existence of approximation algorithms [15, 13], while lower bounds to its constant factor aproximability have been proposed in [12].

More recently, Vertex Cover has been extensively studied under the viewpoint of fixed parameter complexity. According to the parameterization approach, some part of the problem is declared as the *parameter* and reflects the part of the problem that is expected to be small in most of the "real word" instances. Parameterized complexity asks whether an algorithm of complexity $\mathrm{O}(f(k) \cdot n^{\mathrm{O}(1)})$ exists, where k is the parameter and n is the problem size. Such a *parameterized algorithm* classifies the parameterized problem in the class FPT and claims that the problem is tractable at least for small values of the parameter. For more details on the recent advances and challenges of parameterized

* This research was supported by the EU 6th FP under contract 001907 (DELIS). The first author was partially supported by the *Distinció per a la Promoció de la Recerca de la GC, 2002*.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 231–240, 2006.

complexity we refer to [9, 8]. In the case of vertex cover, the parameter is k and Vertex Cover is known to be in FPT. There is a long time race towards obtaining the fastest parameterized algorithm for it; the current champion is [6] that runs in $O(kn + 1.2738^k)$ steps.

One of the main tools for the design of fast parameterized algorithms is *kernelization*. This technique consists in finding a polynomial-time reduction of a parameterized problem to itself in a way that the sizes of the instances of the new problem —we call it *kernel*— *only* depend on the parameter k. The function that bounds the size of the main part of the reduced problem determines the *size* of the kernel and is usually of polynomial (on k) size. If a kernel exists, then we can apply the fastest available exact algorithm for the problem to the (reduced size) kernel instead of the initial instance. Several kernelizations have been proposed for Vertex Cover. A simple kernel of size $O(k^2)$ was proposed by Buss [5]. So far, the smallest kernel has size $\leq 2k$ and its follows from the results of Nemhauser & Trotter in [16]. More recently a kernel of size $\leq 3k$ appeared, based on the "crown decomposition" technique in [10]. An experimental evaluation of these three kernelization techniques was done in [1]. The graphs used for the study in [1] were taken from open-source repositories of biological data. We stress that no random graph model was considered in [1].

Notice that kernelization can also be seen as a preprocessing algorithm. It essentially provides a way to preprocess the input of a problem and transform it to an equivalent one of size of provably bounded size. From a practical point of view, this approach may be useful not only when the parameter k is small, as it is presumed by the parameterized complexity. Moreover, in many cases of parameterized or exact algorithms, actual running times are much better than their theoretically proven bounds. This brings up the challenge of evaluating their performance on specific models of inputs. In this paper, we start such an evaluation with the Vertex Cover problem. A basic concern for our experiments was the model of graphs on which they should be applied.

At the end of the decade of 90's, several empirical studies showed that the degree distribution of large dynamic graphs seem to follow a power-law tail. Among those graphs studied were the WWW, Internet, metabolic networks and others (see, for example, [3] or the surveys [17, 2]). Those results gave a push to the theoretical study of new random graph models with power law tail, as the classical Erdős-Rényi-Gilbert model have a bell-shaped degree distribution, and an exponetially decreasing tail, which did not fit the experimental evidence of power law degree distribution. Recently, it has been showed that the same experimental techniques used on the WWW, namely *Traceroute sampling*, when applied to the classical $G_{n,p}$ model also follow a power law with exponent 1 [7]. In any case, the experimental work on power-law distribution graphs, gave a fruitfull research on dynamic random graph models to fit the experimental evidence, observed for power-law graphs, like the clustering coefficient and small diameter. The more basic model of heavy tailed degree distribution random graphs is the *preferential attachement model (PAM)* given implicitly in [3] and made rigorous in [4].

For the above reasons, we adopt here a model of random graphs generated using the preferred attachment criterion. In this model, a random graph is generated by adding vertices one by one, and the selection of their neighbors is done in a way that favorites vertices that have already *big degree*. In this sense, the vertices of the graph are organized around "clusters" of high degree vertices. Apart from the WWW, such models resemble graphs emerging by distinct applications such as collaboration networks (i.e. in biology, in science, or in film actors), word co-occurrences, or neural networks; see [17].

We base our experiment on the kernelizations of Buss and of Nemhauser & Trotter and we call them *Buss-* and *NT-kernelization* respectively. In both preprocessing algorithms the input is a pair (G, k) and the output is either a definite answer (`YES` or `NO`) or a new *equivalent* instance (G^*, k^*) of the problem. In case no definite answer is given by the kernelization, the returned equivalent instance (G^*, k^*) is the input for an exact algorithm for the vertex cover problem.

Let n denote the number of the vertices of the input graph. Currently, the fastest exact algorithm for Vertex Cover has running time $\mathrm{O}^*(1.19^n)$ [18]; A much simpler (and easier to implement) algorithm for the same problem was given in [11] and has running time $\mathrm{O}^*(1.221^n)$.

Clearly, a preprocessing algorithm is good if it achieves a good reduction of the input size. A measure of this reduction could be the ratio $|V(G^*)|/|V(G)|$. Instead, we suggest a more realistic measure based on the fact that $\mathbf{vc}(G) = \sum_{H \in \mathcal{G}(G)} \mathbf{vc}(H)$, where $\mathcal{G}(G)$ is the set of the connected components of a graph G. This implies that the size of the biggest component of G will dominate the running time of the (exponential time) search for a vertex cover of minimum size. Thus, it reasonable to choose as measure of the input size reduction the ratio $\max_{H \in \mathcal{G}(G^*)} |V(H)| / \max_{H \in \mathcal{G}(G)} |V(H)|$ where G^* is the output graph when we apply kernelization $\mathbf{K}$ to an input graph G. As the graphs generated by our model will always connected, we redefine the reduction measure as

$$\rho(\mathbf{K}, G) = \frac{\max_{H \in \mathcal{G}(G^*)} |V(H)|}{|V(G)|}$$

and we refer to it as the *component reduction* ratio for kernelization $\mathbf{K}$ applied to the graph G (in this paper $\mathbf{K} \in \{\mathrm{Buss}, \mathrm{NT}\}$).

In the following, we evaluate the algorithms for several values of k, with the following criteria:

1. The percentage of cases where the kernelization gives a definite answer.
2. The component reduction ratio (given that the kernelization does not give a definite answer).
3. The comparative performance of the two algorithms according to criteria 1 and 2.
4. The comparative performance of the two algorithms according to their running times.

The most remarkable result of our experiments is that both kernelization algorithms achieve a very good reduction on graphs generated under the preferential

attachment model. In particular, we observe reductions approaching the 9% in the case of Buss-kernelization and the .35% in the case of NT-kernelization (for any value of $k \leq 100$ and for graphs with 2000 vertices). This indicates that the NT-kernelization does not only provide the smallest (so far) kernel for parameterized Vertex Cover but can also serve as a very good preprocessing algorithm for the same problem when applied to graphs emerging from the preferred attachment model.

The paper is organized as follows. In Section 2 we present the two kernelization algorithms and the way we implemented them. In Section 3 we detail the model of random graphs that we use. In Section 4 we present and comment the results of our experiments. Finally, in Section 5, we give some remarks and research directions.

2 Kernelizations for Vertex Cover

We consider undirected graphs without loops nor multiple edges. Given a graph G, its vertex and edge set are denoted as $V(G)$ and $E(G)$ respectively. A connected component of a graph is called non-trivial if it contains at least 2 vertices. Given a vertex $v \in V(G)$ we denote by $\deg_G(v)$ the degree of v in G, i.e. the number of vertices adjacent to v in G. We also denote by $G - v$ the graph obtained by G after we remove vertex v and all its incident edges. If $S \subseteq V(G)$ we define the *subgraph of G induced by S* as $G[S] = (S, \{e \in E(G) \mid e \subseteq S\})$. Finally, we use the notation $I(G)$ for the set of isolated vertices in G.

2.1 The Kernelization of Buss

The first kernelization we study in this paper follows from the results in [5] and is sketched below:

def **Buss**(G, k) :
1. if $k = 0$ and $I(G) = V(G)$: return `YES`
2. if $k = 0$ and $I(G) \neq V(G)$: return `NO1`
3. if $\exists v \in V(G)$: $\deg_G(v) > k$: return **Buss**$(G - v, k - 1)$
4. if $|V(G)| > k(k + 1)$: return `NO2`
5. if G has more than k non-trivial connected components: return `NO3`
6. return $(G[V(G) - I(G)], k)$

The correctness of the algorithm is based on the three following facts: a) any vertex of degree more than k should be in any vertex cover of size $\leq k$, b) a graph with max degree $\leq k$ and a vertex cover of size $\leq k$ cannot have more than $k+k^2$ non-isolated vertices, and c) any graph with more than k non-trivial connected components has no vertex cover of size less than k. The time complexity of **Buss**(G, k) is $O(k \cdot |V(G)|)$.

In order to refine the analysis of our experiments, we distinguish between the three ways that the algorithm may return a negative answer. The first negative answer (`NO1`) appears is the case when, after the deletion of k high degree vertices,

there still remain edges in the graph. The second (NO2) appears when no high degree vertices exist and the graph has big size (more than $k+k^2$ vertices). The last one (NO3) is when the current graph has more than k non-trivial connected components.

We would like to stress that Step 6 (corresponding to NO3) is not a part of the classic kernelization of Buss. However, we added it because in our experiments it behaves rather well as a filter of NO-instances.

2.2 The Kernelization of Nemhauser & Trotter

The kernelization of Nemhauser & Trotter follows from the results in [16] and is sketched bellow:

def $\mathbf{NT}(G,k)$:
1. let $U_1 = V(G)$ and let U_2 be a new set of vertices where $|U_1| = |U_2|$
2. let $\sigma : U_1 \to U_2$ be a bijection from U_2 to U_1
3. let $H = (U_1 \cup U_2, \{\{x,y\} \mid x \in U_1, y \in U_2, \text{ and } \{x,\sigma(y)\} \in E(G)\})$
4. let S be a vertex cover of H
5. let $V_1 = \{x \mid x \in S \cap U_1 \text{ and } \sigma^{-1}(x) \in S\}$
6. let $V_0 = \{x \mid x \in U_1 - S \text{ and } \sigma^{-1}(x) \notin S\}$
7. let $V_{\frac{1}{2}} = U_1 - V_1 - V_0$
8. if $|V_1| > k$: return NO1
9. if $|V_1| = k$ and $E(G[V_{\frac{1}{2}}]) \neq \emptyset$: return NO2
10. if $|V_1| = k$ and $E(G[V_{\frac{1}{2}}]) = \emptyset$: return YES
11. if $|V_{\frac{1}{2}}| > 2(k - |V_1|)$: return NO3
12. return $(G[V_{\frac{1}{2}}], k - |V_1|)$

$\mathbf{NT}(G,k)$ first constructs a bipartite graph H where one of its parts has the vertices of G and the other copies of the vertices of G. An edge from one part to the other is placed if the corresponding vertices are adjacent to the original graph G. This construction takes place in steps **1**–**3**. In step **4**, the algorithm finds a minimal vertex cover S of H. Such a vertex cover can be computed in polynomial time because H is bipartite. Specifically, we have done so adding a source and sink vertex to H and computing a max-flow min-cut using the Edmonds-Karp algorithm (its running time is $O(|V||E|)$. In steps **5**–**7** the algorithm partitions the vertices of G into three sets V_1, V_0, and $V_{\frac{1}{2}}$: those that both themselves and their copies belong in S, those than neither themselves nor their copies belong into S, and those that either themselves or their copies belong to S (but not both). From [16], it follows that a) $\mathbf{vc}(G) = \mathbf{vc}(G[V_{\frac{1}{2}}]) + |V_1|$ and b) $|V_{\frac{1}{2}}| \leq 2 \cdot \mathbf{vc}(G[V_{\frac{1}{2}}])$. These relations justify the answers provided in steps **8**–**12**. There are three possible ways for the algorithm to return NO: the first appears when the size of V_1 is bigger than k (NO1), the second appears when $|V_1| = k$ and there are edges between vertices in $V_{\frac{1}{2}}$ (NO2), and the third appears when $|V_{\frac{1}{2}}| > 2(k - |V_1|)$ (NO3).

We did not add the filter of Step 6 of the $\mathbf{Buss}(G,k)$ algorithm as our experiments showed that, for our graph generation model, it does not offer any additional filtering for the case of NT-kernelization.

3 The Model

The graph generation algorithm is depicted bellow; we assume that m is a small constant ($m \ll n$):

def **PrefAttach**(n, m) :
1. let G be a cycle of m vertices
2. for $i = m + 1$ to n:
   add a new vertex u to G
   add m edges from u to $v_1, \ldots, v_m$,
   where each v_i is selected with probability $\frac{\deg_G(v_i)^2}{\sum_{v \in V(G-u)} \deg_G(v)^2}$
3 return G

Function **Pref-Attach**(n, m) generates graphs incrementally adding one vertex at each step. Each new vertex gets m neighbors that are picked among the vertices of the already constructed graph. The selection of a neighbor is biased in a way that vertices of high degree in G are prefered against those with low degree. This makes the generated graphs look like clusters of vertices positioned in a way that resembles the adjacency of web graphs. If G is a graph generated by **Pref-Attach**(n, m), then $|V(G)| = n$ and $|E(G)| = m(n - m + 1)$.

Our implementation of the above procedure confirms the intuition that the vast majority of the vertices in G are of low degree while a small minority has big degree. In Figure 1 one can see the degree distribution of a graph generated by **Pref-Attach**$(2000, 4)$.

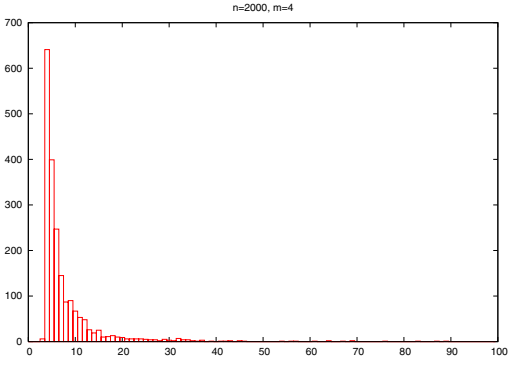


Fig. 1. Degree distribution on a **Pref-Attach**$(2000, 4)$ graph

4 Experimental Results

In this section we present some experimental results to evaluate the Buss and NT kernelization algorithms for the Vertex Cover problem on the preferred attachment model. To do so, we have implemented both algorithms as well as the generation of the random graphs. All our code has been written in C++ using the STL and compiled with GCC 3.4.1 with the `-O3` option. The experiments

have been performed on a machine with a Intel Pentium 4 CPU at 3.20GHz and with 3GB of memory running a Linux operating system.

We split the presentation of our results in three subsections. First, we present the distribution of the possible results of the two kernelization algorithms. Afterwards, we show the component reduction ratio for each one of them. Finally, we present some time measurements.

4.1 Distribution of the Possible Answers

In Figure 4.1, we present the percentages for the appearance of the possible outputs (`NO1`, `NO2`, `NO3`, and `DON'T KNOW`) of **Buss**(G, k) and **NT**(G, k) for $k \in \{1, \ldots, 100\}$. Our sample contained 500 graphs, generated by **Pref-Attach**(n, m) taking $n = 2000$ and $m = 4$. experiments. The curves for other values of n and m behave similary. We do not include the count of `YES` answers in the figures because they never appeared in our experiments.

With regard to the Buss kernelization, we can see that answers `NO1`, `NO2` and `NO3` play a complementary role as k increases: The `NO1` criterium is usefull when k is rather small and vanishes as it increases. At this point, the `NO2` criterium turns to be usefull to report negative answers but, again, vanishes as k keeps increasing. It is then the turn of the `NO3` criterium to report negative answers until a point where `DON'T KNOW` answers begin to appear, which happens in more than a half of the cases for $k > 33$ on our **Pref-Attach**$(2000, 4)$ graphs.

The NT kernelization exhibits a similar behaviour with respect to the complementary role of the different negative criteria as k increases. However, an immediate comparison of the two diagrams implies that the NT kernelization gives much more information than the Buss kernelization. For instance, we can report a negative answer on more than one half of the instances for values of k up to 65 (rather than 33 with Buss).

To have a clearear view of when certain percentages of `NO` answers can be expected, see Figure 4.1. The horizontal axis represents distinct values of n (the number of vertices of the generated graphs) and each of the lines represents the biggest k for which the percentage of the `DON'T KNOW` answer was grater than 0%, 20% or 50% for the Buss and the NT kernelizations. In this case, the experiment shows the averages computed over 200 graphs for each size.

Again it is clear that the NT-kernelization is more powerfull and scales better with the graph size than its Buss counterpart. No only that, Figure 4.1 also induces us to conjecture that if G is a random graph generated by **Pref-Attach**(m, n) then, with high probability, $\mathbf{vc}(G) \geq c_m \log^{O(1)} n$ where c_k is a constant depending on m.

4.2 Component Reduction Ratio

We consider now the reduction that is achieved in the cases where the answer is unknown and a kernel is computed. We are interested in the component reduction.

Figure 4.1 presents the percentual values of $\rho(\mathbf{Buss}, k)$ and $\rho(\mathbf{NT}, k)$. Again, our sample contained 500 graphs, generated by **Pref-Attach**(n, m) for $n = 2000$

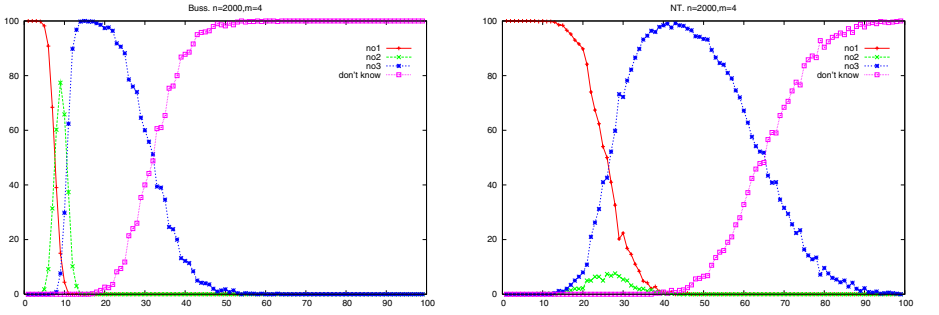


Fig. 2. Behaviour of Buss and NT kernalizations on Pref-Attach graphs with $n = 2000$ and $m = 4$ for $k \in \{1, \ldots, 100\}$

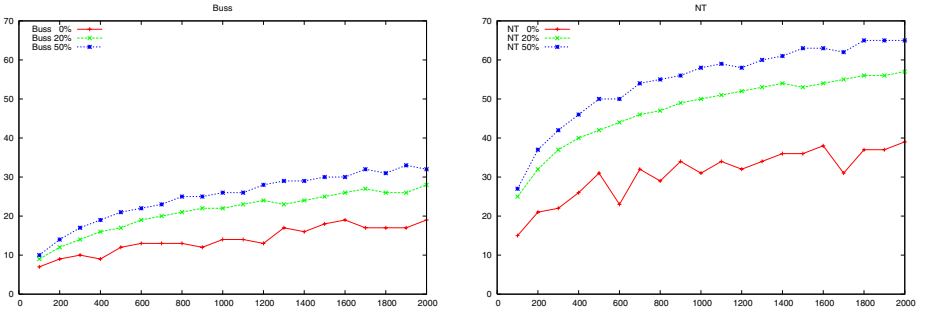


Fig. 3. Maximal value of k for which Buss and NT kernelizations can give a negative answer in $p\%$ of the cases for $p \in \{0, 20, 50\}$ on Pref-Attach graphs with $n \in \{100, \ldots, 2000\}$ and $m = 4$

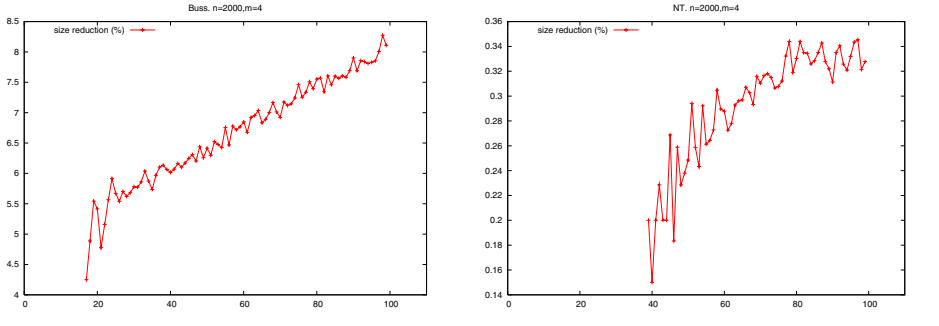


Fig. 4. Component reduction ratio

and $m = 4$. The horizontal axis represent the values of k up to 100, and the vertical axis represents the percentage of the component reduction. The curve is not shown for values of k where a definite answer was found.

The results are quite positive for the capacity of the Buss kernelization as a preprocesing algorithm: even when $k = 100$, the Buss kernelization achieves a

reduction to equivalent instances where the maximum component has size that is no bigger than the 9% of the size of the original input.

News are even better in the case of the NT kernelization, because it achieves a maximum component size reduction arround a 0.35% for any $k \leq 100$. This means that the brute force algorithm shall only have to solve vertex covers for components with an average of 7 vertices. Such a performance clearly indicates that the NT kernelization is indeed an excellent preprocessing algorithm for graphs generated by the preferred attachment model.

4.3 Running Times

So far NT kernelization beats the Buss kernelization by any means. However, we should mention that on the preferred attachment model, the opposite holds as far as the running time is concerned. Time measures for the previous experiments show that our implementation for NT is about 10 times slower than our implementation for Buss. However, the actual running times are so small (tenths of seconds for NT and hundredths of seconds for Buss) that these do not seem to matter. Moreover, we should stress that these are just preprocessing times: the main running times would still be dominated by the time required by the exact (exponential) algorithm. We thus conclude that the relative speed of both algorithms is not that big.

5 Discussion

This paper attempts a first study of the performance of kernelization algorithms when treated as preprocessing algorithms to hard problems, on the family of random graphs that we have used. Especially for the Vertex cover, it remains an open project to see whether the optimistic results of this paper can be extended for other models of random graphs generated by several proposed variations on the preferred attachment mechanism. Aa the PAM is a model for the web graph, it would be useful to confirm that the same results come up when the experiments are applied to a sufficiently large part of the main component of the web (or the internet graph). This could have a potential for a new way to identify big hubs in those graphs. Finally, we believe that the potential of the kernelization idea can offer experimental results that are really better than the formally proven bounds. It is an open project to examine this for kernelization algorithms provided by the parameterized complexity context for other problems as well.

References

1. Faisal N. Abu-Khzam, Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Christof T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics, New Orleans, LA, USA*, pages 62–69. SIAM, 2004.

2. Albert-László Barabási. Emergence of scaling in complex networks. In *Handbook of graphs and networks*, pages 69–84. Wiley-VCH, Weinheim, 2003.
3. Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
4. Béla Bollobás, Oliver Riordan, Joel Spencer, and Gábor Tusnády. The degree sequence of a scale-free random graph process. *Random Structures Algorithms*, 18(3):279–290, 2001.
5. J.F. Buss and J. Goldsmith. Nondeterminism within p. *SIAM J. Computing*, 22:560–572, 1993.
6. Jianer Chen, Iyad A. Kanj, and Ge Xia. Simplicity is beauty: Improved upper bounds for vertex cover. Technical Report 05-008, Texas A&M University, Utrecht, the Netherlands, April 2005.
7. Aaron Clauset and Cristopher Moore. Accuracy and scaling phenomena in internet mapping. *Phys. Rev. Lett.*, 94, 2005.
8. R. G. Downey and M. R. Fellows. *Parameterized complexity.* Monographs in Computer Science. Springer-Verlag, New York, 1999.
9. Michael R. Fellows. Parameterized complexity: the main ideas and some research frontiers. In *Algorithms and computation (Christchurch, 2001)*, volume 2223 of *Lecture Notes in Comput. Sci.*, pages 291–307. Springer, Berlin, 2001.
10. Michael R. Fellows. Blow-ups, win/win's, and crown rules: Some new directions in fpt. In *Graph-Theoretic Concepts in Computer Science, 29th International Workshop, WG 2003, Elspeet, The Netherlands, June 19-21*, Lecture Notes in Computer Science, pages 1–12. Springer, 2003.
11. F. V. Fomin, F. Grandoni, and D. Kratsch. Large measure and conquer: A simple $O(2^{0.288\,n})$ independent set algorithm. In *17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, page to appear. ACM and SIAM, New York, 2006.
12. Johan Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859 (electronic), 2001.
13. George Karakostas. A better approximation ratio for the vertex cover problem. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, (ICALP),*, Lecture Notes in Computer Science, pages 1043–1050. Springer, 2005.
14. Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*, pages 85–103. Plenum, New York, 1972.
15. Burkhard Monien and Ewald Speckenmeyer. Ramsey numbers and an approximation algorithm for the vertex cover problem. *Acta Inform.*, 22(1):115–123, 1985.
16. G. L. Nemhauser and L. E. Trotter, Jr. Vertex packings: structural properties and algorithms. *Math. Programming*, 8:232–248, 1975.
17. Mark E. J. Newman. Random graphs as models of networks. In *Handbook of graphs and networks*, pages 35–68. Wiley-VCH, Weinheim, 2003.
18. J. M. Robson. Finding a maximum independent set in time $O(2^{n/4})$. manuscript, `http://dept-info.labri.fr/ robson/mis/techrep.html`, 2001.

Practical Partitioning-Based Methods for the Steiner Problem

Tobias Polzin[1] and Siavash Vahdati Daneshmand[2]

[1] HaCon Ingenieurgesellschaft mbH, Hannover, Germany
[2] Theoretische Informatik, Universität Mannheim, Germany
vahdati@informatik.uni-mannheim.de

Abstract. Partitioning is one of the basic ideas for designing efficient algorithms, but on $\mathcal{NP}$-hard problems like the Steiner problem, straightforward application of the classical partitioning-based paradigms rarely leads to empirically successful algorithms. In this paper, we present two approaches to the Steiner problem based on partitioning. The first uses the fixed-parameter tractability of the problem with respect to a certain width parameter closely related to path-width. The second approach is based on vertex separators and is new in the sense that it uses partitioning to design reduction methods. Integrating these methods into our program package for the Steiner problem accelerates the solution process on many groups of instances and leads to a fast solution of some previously unsolved benchmark instances.

1 Introduction

The Steiner problem is the problem of connecting a set of terminals (vertices in a weighted graph or points in some metric space) at minimum cost. This is a classical $\mathcal{NP}$-hard problem with many important applications in network design in general and VLSI design in particular [3, 6].

For such ($\mathcal{NP}$-hard) problems, straightforward application of the classical partitioning paradigms rarely leads to empirically successful algorithms. Divide-and-conquer techniques are not generally applicable, because one usually cannot find independent subproblems. Dynamic programming techniques can indeed be applied, but they are usually practical only for a very limited range of instances.

In this paper, we present two practically helpful methods which are based on partitioning. In Sect. 2, we present an algorithm that uses the fixed-parameter tractability of the problem with respect to a certain width parameter closely related to path-width. The running time of the algorithm is linear in the number of vertices when the path-width is constant, and it is practical when the considered graph has a small width. In Sect. 3, we introduce the approach of using partitioning for reducing the size of the instance (i.e., developing partitioning-based reduction methods). We present two new reduction methods based on this approach. Methods like those described in this paper are not conceived as stand-alone solution routines for a wide range of instances; but as subroutines of more complex optimization programs, they are much more broadly applicable.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 241–252, 2006.

An additional feature is that by the cooperation of the methods presented here we can already profit from small width in subgraphs of a given instance; these interactions will be elaborated in Sects. 3 and 4. Finally, some experimental results are presented in Sect. 4.

1.1 Preliminaries

The **Steiner (tree) problem in networks** can be stated as follows: Given a (connected) network $G = (V, E, c)$ (with vertices $V = \{v_1, \ldots, v_n\}$, edges E and edge weights $c_e > 0$ for all $e \in E$) and a set R, $\emptyset \neq R \subseteq V$, of **required vertices** (or **terminals**), find a minimum weight tree in G that spans R (a **Steiner minimal tree**). For more information on this problem, see [6].

We define $r := |R|$. If we want to stress that v_i is a terminal, we will write z_i instead of v_i. A **bottleneck** of a path P is a longest edge in P. The **bottleneck distance** $b(v_i, v_j)$ or b_{ij} between two vertices v_i and v_j in G is the minimum bottleneck length taken over all paths between v_i and v_j in G. An **elementary path** is a path in which only the endpoints may be terminals. Any path between two vertices can be broken at inner terminals into one or more elementary paths. The **Steiner distance along a path** P between v_i and v_j is the length of a longest elementary path in P. The **bottleneck Steiner distance** (sometimes also called "special distance") $s(v_i, v_j)$ or s_{ij} between v_i and v_j in G is the minimum Steiner distance taken over all paths between v_i and v_j in G. A major relevance of bottleneck Steiner distances is that the cost of an optimum Steiner tree in G does not change by deleting edges (v_i, v_j) with $c_{ij} > s_{ij}$ (or, conversely, by inserting edges (v_i, v_j) of length s_{ij}) [4]. For any subset $S \subseteq R$, all b_{ij}, s_{ij} with $v_i, v_j \in S$ can be computed in time $O(|E| + |V| \log |V| + |S|^2)$ [4, 9].

2 Using (Sub-) Graphs of Small Width

In this section, we present a practical algorithm for solving the Steiner problem in graphs with a small width parameter. The width concept used here is closely related to path-width, as we will show in Sect. 2.3. For an overview of subjects concerning path-width and the more general notion of tree-width see [1]. The running time of the algorithm is linear in the number of vertices when the width is constant, thus it belongs to the category of algorithms exploiting the fixed-parameter (FP) tractability of $\mathcal{NP}$-hard problems. There are already FP-polynomial algorithms for the Steiner problem in graphs. Specifically, in [8] a linear-time algorithm for graphs with bounded tree-width is described. But this algorithm is more complicated than the one we present here, and its running time grows faster with the (tree-) width (it is given in [8] as $O(nf(d))$ with $f(d) = \Omega(d^{4d})$, where d is the tree-width of the graph). Therefore, it seems to be not as practical as our algorithm, and no experimental results are reported in [8]. In a different context (network reliability), a similar approach using path-width is described in [11], which is practical for a range of path-widths similar to the one considered here. We also adapted that approach to the Steiner problem, but the experimental results were not as good as with the one presented here.

2.1 The Basic Algorithm

We maintain a set of already visited vertices and a subset of them (the **border**) that are adjacent to some non-visited vertex. In each step, the set of visited vertices is extended by one non-visited vertex adjacent to the border. For all possible partitions in each border, we calculate (the cost of) a forest of minimum cost that contains all visited terminals with the property that each tree in the forest spans just one of the partition sets. We are finished when all vertices have been visited. The observation behind this approach is as follows: For any optimal Steiner tree T, the subgraph of T when restricted to the visited vertices is a forest, which also defines a partition in the border. The plan is to calculate these forests in a bottom-up manner, in each step using the values calculated in the previous step. If the size of all borders can be bounded by a constant, the total time can be bounded by the number of steps times another constant.

For an arbitrary ordering $v_1, \ldots, v_n$ of the vertices and any $s \in \{1, \ldots, n\}$, we define $V_s := \{v_1, \ldots, v_s\}$ and denote with G_s the subgraph of G with vertex set V_s. In the following, we assume an ordering of the vertices with the property that all G_s are connected. (For example, a depth-first-search traversal of G delivers such an ordering.) We denote with B_s the border of V_s, i.e., $B_s := \{v_i \in V_s \mid \exists (v_i, v_j) \in E : v_j \in V \setminus V_s\}$. With L_s we denote the set of vertices that leave the border after step s, i.e., $L_s := (B_{s-1} \cup \{v_s\}) \setminus B_s$. The inclusion of v_s in this definition should cover the case that v_s has no adjacent vertices in $V \setminus V_s$; this simplifies some other definitions. Consider a set Q, $B_s \cap R \subseteq Q \subseteq B_s$, and a partitioning $\mathcal{P} = \{P_1, \ldots, P_t\}$ of Q into non-empty subsets, i.e., $\dot{\bigcup}_{1 \le i \le t} P_i = Q$ and $\emptyset \notin \mathcal{P}$. The number of ways of partitioning a set of $b := |Q|$ elements into t non-empty subsets is $\left\{ {b \atop t} \right\}$, a Stirling number of the second kind, and the total number of partitions is $B(b)$, the b-th Bell number. For a partition $\mathcal{P}$ and a set $L \subseteq V$ we define $\mathcal{P} - L := \{P'_i \mid P_i \in \mathcal{P},\ P'_i = P_i \setminus L\}$. Let $F(s, \mathcal{P})$ be a forest of minimum cost in G_s containing all terminals in V_s and consisting of t (vertex-disjoint) trees $T_1, \ldots, T_t$ such that T_i spans P_i for all $i \in \{1, \ldots, t\}$. With $c(s, \mathcal{P})$ we denote the cost of $F(s, \mathcal{P})$.

Let $V_0 = B_0 = \emptyset$ and set $c(0, \emptyset) = 0$. The value $c(s, \mathcal{P})$ can be calculated recursively using a case distinction:

- $v_s \in Q$: $c(s, \mathcal{P}) = \min\{c(s-1, \mathcal{P}') + C \mid$
  $\mathcal{P}' = \{P_1, \ldots, P_y\},\ j \in \{0, \ldots, y\},\ \forall\ 1 \le l \le j : v_l \in P_l,$
  $\mathcal{P} = (\{\{v_s\} \cup \bigcup_{1 \le l \le j} P_l\} \cup \{P_{j+1}, \ldots, P_y\}) - L_s,$
  $C = \sum_{1 \le l \le j} c(v_l, v_s)\ \}$,
- $v_s \notin Q$: $c(s, \mathcal{P}) = \min\{c(s-1, \mathcal{P}') \mid \mathcal{P} = \mathcal{P}' - L_s\}$.

The cost of an optimal Steiner tree in G is: $\min\{c(s, \mathcal{P}) \mid R \subseteq V_s, |\mathcal{P}| = 1\}$. Obviously the forests $F(s, \mathcal{P})$ (and an optimal Steiner tree) can be calculated following the same pattern.

By using the recursive formula above, the necessary values can be calculated in a bottom-up manner by memorizing, for each step s, the values $c(s, \mathcal{P})$. We assume $c(s, \mathcal{P}) = \infty$ if no partition $\mathcal{P}$ is calculated at step s. This leads to the following algorithm BORDER-DP (DP stands for Dynamic Programming):

```
BORDER-DP(G, R)        (assuming an ordering of the vertices)
1   s := 0; q := 0; opt := ∞;        (q : number of visited terminals)
2   c(s, ∅) := 0;
3   while s < n :
4       s := s + 1; determine v_s, B_s and L_s;
5       if v_s ∈ R : q := q + 1;
6       forall P with c(s − 1, P) ≠ ∞ :
7           oldCost := c(s − 1, P);
8           if v_s ∉ R and ∅ ∉ P − L_s :
9               c(s, P − L_s) := oldCost;
10          Pcandidates := {P_i ∈ P | ∃v_i ∈ P_i : (v_i, v_s) ∈ E};
11          forall Pconnect ⊆ Pcandidates :
12              connectionCost := Σ_{P_i ∈ Pconnect} min_{v_i ∈ P_i, (v_i, v_s) ∈ E} c(v_i, v_s);
13              Pstay := P\Pconnect; Pnew := ({{v_s} ∪ ⋃_{P_i ∈ Pconnect} P_i} ∪ Pstay) − L_s;
14              if ∅ ∉ Pnew and c(s,Pnew) > oldCost + connectionCost :
15                  c(s,Pnew) := oldCost + connectionCost;
16                  if q = |R| and |Pnew| = 1 :        (feasible Steiner tree)
17                      opt := min(opt, c(s,Pnew));
18  return opt;
```

Let p_s denote the number of partitions at step s. We have $p_s = \sum_{R \cap B_s \subseteq Q \subseteq B_s} B(|Q|)$, where $B(b)$ is the b-th Bell number; so $p_s = O(2^{b_s} B(b_s))$ with $b_s := |B_s|$. We only maintain one global list of partitions, which is updated after each step, keeping for each valid partition a solution of minimum cost. Because of the loop in Line 11, this list can grow to at most $l_s := 2^{b_s} p_s = O(2^{2b_s} B(b_s))$ partitions. Eliminating the duplicates can be done by sorting the list: Each partition can be represented as a (lexicographically) sorted string (of length at most $2b_s$) of sorted substrings (of length at most b_s) separated by some extra symbol. Using radix sort, all the individual sortings of l_s strings can be done in total time $O(n+l_s b_s)$. Sorting the resulting list of l_s strings takes again time $O(n + l_s b_s)$. We set aside for now a total extra time of $O(|E|)$ for the operations on edges; and assume that an ordering of vertices is given (these points are explained below). The (rest of the) operations in Lines $12 - 17$ can be carried out in time $O(b_s)$. This gives the total running time $O(\sum_{s=1}^{n} b_s 2^{2b_s} B(b_s))$. Note that this bound implicitly contains the extra amortized time $O(|E|)$ by the following observation: After a vertex is visited for the first time, it remains in the border as long as it has some non-visited adjacent vertex; so each edge is accounted for by its first-visited endpoint. Now if we can guarantee an upper bound b for the size of all borders, we have an upper bound of $O(nb2^{2b} B(b))$ for the running time. By upper-bounding $B(b)$ roughly with $(2b)^b$ we get the running time $O(n2^{b \log b + 3b + \log b})$. This means that the algorithm runs in linear time for constant b and, for example, in time $O(n^2)$ for $b = \log n / \log \log n$.

For the actual implementation, some modifications are used. For example, avoiding duplicate partitions is done using hashing techniques, which reduces the amount of necessary memory. Also, some heuristics are used to recognize partitions that cannot lead to an optimal Steiner tree.

2.2 Ordering the Vertices

In Sect. 2.3, we will show that finding an ordering of vertices such that the maximum border size equals b is (up to some easy transformations) equivalent to finding a path-decomposition of path-width b. The problem of deciding whether the path-width of a given graph is at most b, and if so, finding a path-decomposition of width at most b is $\mathcal{NP}$-hard for general b, but for constant b, this problem can be solved in linear time [2]. However, already for $b > 4$ the corresponding algorithm is no longer practical [12], and it seems that no practical exact algorithm is known for more general cases. Furthermore, we have a more specific scenario (for example we differentiate between terminals and non-terminals). So for the actual implementation we use a heuristic, which has produced quite satisfactory results for our applications. The heuristic chooses in each step a vertex v_s adjacent to the border using a (ad hoc) priority function of the following parameters: size of resulting set L_s, number of visited vertices in the adjacency list of v_s, membership of v_s in R, and number of edges connecting V_s and $V \setminus V_s$. We select the starting vertex by trying a small number of terminals and performing a sweep through the graph without actually computing the partitions. In each sweep, we estimate the overall number of resulting partitions by summing up the (ad hoc) values $|B_s|^2|B_s \setminus R|$ in each step. Finally, we select the terminal that yields the smallest estimated number.

A straightforward implementation of this heuristic needs time $O(n^2)$ for all choices. This bound could be improved using advanced data structures for priority queues and additional tricks, but the ordering has not been the bottleneck in our applications; and theoretically a better (linear for constant b as in our applications) time bound for path-decomposition is available anyway.

2.3 Relation to Path-Width

In this section, we show that every path-decomposition with path-width k delivers a sequence of borders $B = (B_1, \ldots, B_s, \ldots, B_n)$ such that $\max\{|B_s| \mid 1 \leq s < n\} \leq k$ and vice versa.

A **path-decomposition** of a graph $G = (V, E)$ is a sequence of subsets of vertices $(U_1, U_2, \ldots, U_p)$, such that

1. $\bigcup_{1 \leq i \leq p} U_i = V$,
2. $\forall (v, w) \in E \quad \exists i \in \{1, \ldots, p\} : \quad v \in U_i \wedge w \in U_i$,
3. $\forall i, j, k \in \{1, \ldots, p\} : \quad i \leq j \leq k \Rightarrow U_i \cap U_k \subseteq U_j$.

The **path-width** of a path-decomposition $(U_1, U_2, \ldots, U_p)$ is $\max\{|U_i| \mid 1 \leq i \leq p\} - 1$. The **path-width** of a graph G is the minimum path-width over all possible path-decompositions of G. Note that the 3rd condition in the definition of path-decomposition can be rewritten as follows: There are functions *start*, *end*: $|V| \to \{1, \ldots, p\}$ with $v \in U_j \Leftrightarrow start(v) \leq j \leq end(v)$. We call a path-decomposition with functions *start*, *end* **bijective** if the mapping *start* is a bijection; and **minimal** if it holds: $end(v) \geq i \Rightarrow start(v) = i \vee \exists (v, w) \in E : start(w) \geq i$. We state the following two lemmas (for the proofs, see [15]).

Lemma 1. *Every path-decomposition can be transformed to a minimal and bijective path-decomposition of no larger path-width.*

Lemma 2. *Let $(U_1, \ldots, U_n)$ be a minimal, bijective path-decomposition of G with the functions start and end. Assume that the vertices are ordered according to their start values, i.e., $start(v) = s \Leftrightarrow v \in V_s \setminus V_{s-1}$. For each $s \in \{1, \ldots, n\}$ it holds: $U_s = \{v_s\} \cup B_{s-1}$.*

It follows that every path-decomposition of G can be transformed to a path-decomposition $U = (U_1, \ldots, U_n)$ of no larger path-width such that for an ordering of vertices according to the *start* function of U it holds: $U_s = \{v_s\} \cup B_{s-1}$. On the other hand, it is easy to verify that each ordering of vertices and the corresponding sequence of borders $(B_1, \ldots, B_n)$ deliver a (minimal, bijective) path-decomposition U by setting $U_s = \{v_s\} \cup B_{s-1}$. In each case, we have: $\max\{|U_s| \mid 1 \leq s \leq n\} - 1 = \max\{|B_{s-1}| \mid 1 \leq s \leq n\}$.

3 Partitioning as a Reduction Technique

In this section, we present our approach of using partitioning to design reduction methods, i.e., methods to reduce the size of a given instance without destroying an optimal solution. This approach turns out to be quite effective in the context of the Steiner problem, and it can also be useful for other problems. Furthermore, it offers a straightforward path for a distributed implementation.

The method chosen here for partitioning is based on certain separating sets (vertex separators), these are sets of vertices whose removal makes the (by assumption connected) graph disconnected. We consider here (small) separating sets that contain only terminals (**terminal separators**), although the basic ideas can be extended to general vertex separators. This choice allows us to keep the dependence between the resulting subinstances manageable.

Although one cannot assume that a typical instance of the Steiner problem has small terminal separators, the situation often changes in the process of solving an instance. This is particularly the case for geometric instances after a geometric preprocessing (FST generation phase, see [16]), but also for general instances after applying powerful reduction techniques (see [10]). In both cases, the resulting intermediate instances frequently have many small terminal separators. For geometric instances, the existence of small vertex separators was already observed in [13]; however, in that work a standard dynamic programming approach was suggested for exploiting this observation, which is not nearly as practical as the approach chosen here. The difference between the two approaches will be elaborated in Sect. 3.2.

3.1 Finding Terminal Separators

It is well known that the problem of finding vertex separators (or the vertex connectivity problem) can be solved by network flow techniques in the so-called split graph [5]. This graph is generated by splitting each vertex into two vertices

and connecting them by edges of low capacity; original edges have high (infinite) capacity. In this way, k-connectedness (finding a vertex separator of size less than k or verifying that no such separator exists) can be decided for a graph with n vertices and m edges in time $O(\min\{kmn, (k^3 + n)m\})$ [5] (this bound comes from a combination of augmenting path and preflow-push methods).

However, the application here is less general: we search for vertex separators consisting of terminals only, so only terminals need to be split. Besides, we are interested in only small separators, where k is a very small constant (usually less than 5), so we can concentrate on the augmenting flow methods. More importantly, we are not searching for a single separator of minimum size, but for many separators of small (not necessarily minimum) size. These observations have lead to the following implementation: we build the (modified) split graph (as described above), fix a random terminal as source, and try different terminals as sinks, each time solving a minimum cut problem using augmenting path methods. In this way, up to $\Theta(r)$ ($r = |R|$) terminal separators can be found in time $O(rm)$. We accelerate the process by using some heuristics. A simple observation is that vertices that are reachable from the source by paths of non-terminals need not be considered as sinks. Similar arguments can be used to discard vertices that are reachable from already considered sinks by paths of non-terminals.

Empirically, this method is quite effective (it finds enough terminal separators if they do exist) and reasonably fast, so a more stringent method (e.g., trying to find all separators of at most a given size) would not pay off. Note that the running time is within the bound given above for the k-connectedness problem, which is mainly the time for finding a single vertex separator.

3.2 Reduction by Case Differentiation

In this section, we describe a reduction method that exploits small terminal separators $S \subset R$ to reduce a given instance.

The case $|S| = 1$ corresponds to articulation points (and biconnected components). It is known [6] that the subinstances corresponding to the biconnected components can be solved independently.

The case $|S| = 2$ corresponds to separation pairs (and triconnected components). Note that the two subinstances (corresponding to two subgraphs G_1 and G_2) are no longer independent. Now, for any Steiner minimal tree T, two cases are possible:

1. The terminals in S are connected by T inside G_2. A corresponding Steiner tree can be found by solving the subinstance corresponding to G_2.
2. The terminals in S are connected by T inside G_1. Now there are two subtrees of T inside G_2, and we do not know in advance how the terminals of G_2 are divided between them. But one can observe that the problem can be solved by merging the terminals in S and solving the resulting subinstance.

Since we do not know T in advance, for a direct solution we must also consider both cases for the complement G_1. But if G_2 is relatively small, the solution of

the complementary subinstance can be almost as time-consuming as the solution of the original instance, meaning that not much is gained (or time may even be lost, because now we have to solve it twice). A classical approach would search for components of almost equal size, but we choose a different approach. The main idea is to solve only the small component twice, and then take edges that are common to both solutions and discard edges that are included in neither. After these modifications, we have a smaller instance with the same optimum solution value, and we can proceed with this reduced instance.

For the general case ($|S| = k$), the basic approach is the same as for the case $|S| = 2$; but a larger number of cases must be considered now. Remember from Sect. 2.1 that the number of cases is $B(k)$, the k-th Bell number. So this method can be used profitably only for small k (usually for $k \leq 4$).

Actually, not all these cases must always be considered explicitly, because many of them can be ruled out at little extra cost using some heuristics. A basic idea for such heuristics is the following:

Lemma 3. *Let z_i and z_j be two terminals in the separator S and let b^1_{ij} and s^2_{ij} be the bottleneck distance in G_1 and bottleneck Steiner distance in G_2 between z_i and z_j, respectively. Then the cases in which z_i and z_j are connected in G_1 can be discarded if $b^1_{ij} \geq s^2_{ij}$.*

Proof. Consider a Steiner tree T connecting z_i and z_j in G_1. A bottleneck on the fundamental path between z_i and z_j has at least cost b^1_{ij}. Removing such a bottleneck and reconnecting the two resulting subtrees of T with the subpath corresponding to s^2_{ij}, we get again a feasible solution of no larger cost in which z_i and z_j are connected in G_2. □

For the cases in which we assume that z_i and z_j are connected in G_1, we do not merge z_i and z_j while solving the subinstance corresponding to G_2, but connect them with an edge of weight b^1_{ij}. In case this edge is not used in the solution of the subinstance, this can lead to more reductions.

Such observations can be used to rule out many cases in advance. Nevertheless, a question arises: Can we find an alternative method that does not need explicit case differentiation? We introduce such an alternative in the following.

3.3 Reduction by Local Bounds

The general principle of bound-based reduction methods is to compute an upper bound *upper* and a lower bound under some constraint $lower_{constrained}$. The constraint cannot be satisfied by any optimal solution if $lower_{constrained} > upper$. The constraint is usually that the solution must contain some pattern (e.g., an edge or more complex patterns like trees, see [10]). Once it is established that the test condition (the inequality above) is valid, the corresponding pattern (e.g., the edge) can be excluded, yielding a smaller (reduced) instance with the same optimal solution. But it is usually too costly to recompute a (strong) lower bound from scratch for each constraint. Here one can use an approach based on linear programming. Any linear relaxation can provide a dual feasible solution of value

lower and reduced costs $\tilde{c}$. We can use a fast method to compute a constrained lower bound with respect to $\tilde{c}$. The sum of the two bounds is a lower bound for the value of any solution satisfying the constraint. For details, see [4, 10].

In the following, we show how to develop a reduction test condition based on local bounds, the application is analog to the usage of globally computed bounds for reductions, see [10]. This approach has two main advantages: The bounds can be computed faster; and there is less chance that the deviations between the original instance and its linear relaxation can accumulate (and thus deteriorate the computed bounds and methods using them). The main difficulty is that the bounds must somehow take the dependence on the rest of the graph into account.

Let S be a terminal separator in G and G_1 and G_2 the corresponding subgraphs. The bounds will be computed locally in supplemented versions of G_2. Let C be a clique over S. We denote with (C, b) the weighted version of C with weights equal to bottleneck distances in G_1; similarly for (C, s) with weights equal to bottleneck Steiner distances in G. Let G_2' and G_2'' be the instances of the Steiner problem created by supplementing G_2 with (C, s) and (C, b), respectively. We compute a lower bound $lower_{constrained}(G_2'')$ for any Steiner tree satisfying a given constraint in G_2'' and an upper bound $upper(G_2')$ corresponding to an (unrestricted) Steiner tree in G_2'. The test condition is: $upper(G_2') < lower_{constrained}(G_2'')$.

Lemma 4. *The test condition is valid, i.e., no Steiner minimal tree in G satisfies the constraint if $upper(G_2') < lower_{constrained}(G_2'')$.*

Proof. Consider $T_{con}^{opt}(G)$, an optimum Steiner tree of cost $opt_{con}(G)$ satisfying the constraint. The subtrees of this tree restricted to subgraphs G_1 and G_2 build two forests F_1 (with connected components T_i) and F_2 (Fig. 1, left). Removing F_2 and reconnecting F_1 with $T^{upper}(G_2')$ we get a feasible solution again, which is not necessarily a tree (Fig. 1, middle). Let S_i be the subset of S in T_i. Consider two terminals of S_i: Removing a bottleneck on the corresponding fundamental path disconnects T_i into two connected components. Repeating this step until all terminals in S_i are disconnected in T_i, we have removed $|S_i| - 1$ bottlenecks, which together build a spanning tree $spanT_i$ for S_i (Fig. 1, right). Repeating this for all T_i, we get again a feasible Steiner tree $T^{upper}(G')$ for the graph G', which is created by adding the edges of (C, s) to G.

Remember from Sect. 1.1 that the optimum solution value does not change by inserting any edges (v_i, v_j) of length s_{ij} into G, so the optimum solution values in G' and G are the same. Let $upper(G')$ be the weight of $T^{upper}(G')$. By construction of $T^{upper}(G')$, we have: $upper(G') = opt_{con}(G) + upper(G_2') - c(F_2) - \sum_i c(spanT_i)$. The edge weights of the trees $spanT_i$ correspond to bottlenecks in F_1, so by definition they cannot be smaller than the corresponding bottleneck distances in G_1. By construction of G_2'', all these edges (with the latter weights) are available in G_2''. Since the trees $spanT_i$ reconnect the forest F_2, together with F_2 they build a feasible solution for G_2'', which even satisfies the constraint (because F_2 did), so it has at least the cost $opt_{con}(G_2'')$. This means:

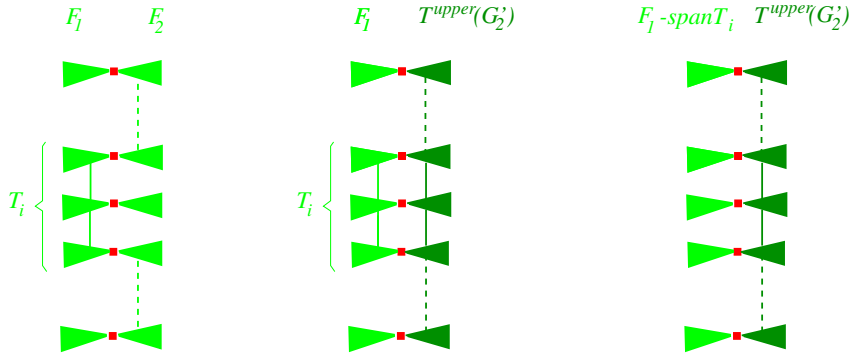


Fig. 1. Construction of $T^{upper}(G')$ from $T^{opt}_{con}(G)$

$$\begin{aligned} upper(G') &\leq opt_{con}(G) + upper(G_2') - opt_{con}(G_2'') \\ &< opt_{con}(G) + lower_{constrained}(G_2'') - opt_{con}(G_2'') \quad \text{(test condition)} \\ &\leq opt_{con}(G). \end{aligned}$$

Thus $opt_{con}(G) > upper(G') \geq opt(G') = opt(G)$, meaning that the constraint cannot be satisfied without deteriorating the optimum solution value. □

4 Experimental Results

In this section, we study the empirical impact of the presented methods. Methods like those in this paper cannot be expected to be usable as stand-alone solution methods for a wide range of instances; however, they are very helpful as subroutines in many cases. By integrating these methods in our program package for the Steiner problem [15], we could already solve several previously unsolved benchmark instances from SteinLib [7], which otherwise could not be tackled in reasonable times. For the experiments in this paper, we concentrate on instances from a current real-world application (the LOFAR radio telescope project, which is described below). An additional advantage of these instances (beside their interesting practical background) is that we are already able to solve all of them without the techniques described in this paper, so we can present concrete running times for different solution methods, which also demonstrate the improvements gained by the techniques presented here.

The LOFAR (LOw Frequency ARray) project is concerned with the construction of the largest radio telescope of the world, which is currently being built by ASTRON in the Netherlands. It consists of many sensor stations that have to be located along five spiral arms, with each arm stretching over hundreds of kilometers. The distance between adjacent sensor stations along each arm should increase in a logarithmic progression. The LOFAR sensor stations must be placed while avoiding obstacles where stations cannot be placed geographically (e.g., the North Sea and population centers). The sensor stations should be

connected by (expensive) optical fibers in order to send the data collected by the sensors to a central computer for processing and analysis. Since cheaper existing optical fibers with unused capacity can be purchased from cable providers, they should be utilized to set up the optical network in the most economical way. Note that the cost function is quite general, so these instances are not (pure) geometric instances. A research branch in the LOFAR project is working on a simulation-optimization framework [14], where different topological designs and cost functions are developed. The Steiner problem is used to model the routing part of the problem, in order to find a low-cost cabling for each of the scenarios considered. The large amounts of money involved justify the wish for optimal (or at least provably near-optimal) solutions. On the other hand, changes in the scenario give rise to a large number (hundreds) of new candidate instances, so excessively long runs for single instances are not tolerable.

The latest collection of instances we received from the LOFAR project [14] consists of more than one hundred instances, divided in 13 groups (with different settings of parameters). All these instances have 887 vertices, thereof 101 terminals, and 163205 edges; but with various cost functions. For the experiments here, we (randomly) chose one instance from each group (the results inside each group were similar). We compare three (exact) solution methods:

(I) As a basis for comparisons, we use a somehow standard branch-and-cut approach based on the classical (directed) cut formulation of the Steiner problem, using a cut generating routine as described in [9]. However, in contrast to [9], here we use no reductions at all. Since the program in [9] heavily exploits the reduction-based methods (e.g. for computing sharp upper bounds), here as a substitute we utilize the MIP optimizer of CPLEX 8.0 after solving the LP-relaxation to get a provably optimal integer solution (the additional times for the MIP optimizer were relatively marginal for the considered instances).

(II) In the second set of experiments, we use exactly the same cut-based algorithm as under (I), but this time after performing our strong reduction techniques from [10] as preprocessing.

(III) Finally, in the third set of experiments we use exactly the same reduction techniques as under (II), but additionally we use the partitioning-based techniques described in this paper. The cut-based routine is dropped, so no LP-solver is used at all.

The results are summarized in Table 1 (all computations were performed on a machine with an INTEL Pentium-4 3.4 GHz processor). We observe:

- The reduction techniques can heavily accelerate the solution process, a fact that is meanwhile well established. For the considered set of instances, our previous reduction techniques already improve the solution times of the branch-and-cut algorithm by more than one order of magnitude.
- The partitioning-based techniques presented in this paper improve the (exact) solution times by one additional order of magnitude, thereby eliminating the need for an LP-solver like CPLEX altogether for all considered instances.

Table 1. Summarized solution times for the LOFAR instances using different methods

Solution Method	(I) (B&C)	(II) (Preprocessing + B&C)	(III) (Preprocessing + Partitioning)
Average Solution Time (seconds)	396	11	1.2

References

1. H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.
2. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.
3. X. Cheng and D.-Z. Du, editors. *Steiner Trees in Industry*, volume 11 of *Combinatorial Optimization*. Kluwer Academic Publishers, Dordrecht, 2001.
4. C. W. Duin. Preprocessing the Steiner problem in graphs. In D. Du, J. Smith, and J. Rubinstein, editors, *Advances in Steiner Trees*, pages 173–233. Kluwer, 2000.
5. M. R. Henzinger, S. Rao, and H. N. Gabow. Computing vertex connectivity: New bounds from old techniques. *J. Algorithms*, 34(2):222–250, 2000.
6. F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. North-Holland, Amsterdam, 1992.
7. T. Koch and A. Martin. SteinLib. `http://elib.zib.de/steinlib`, 2001.
8. E. Korach and N. Solel. Linear time algorithm for minimum weight Steiner tree in graphs with bounded tree-width. Technical Report 632, Technicon - Israel Institute of Technology, Computer Science Department, Haifa, Israel, 1990.
9. T. Polzin and S. Vahdati Daneshmand. Improved algorithms for the Steiner problem in networks. *Discrete Applied Mathematics*, 112:263–300, 2001.
10. T. Polzin and S. Vahdati Daneshmand. Extending reduction techniques for the Steiner tree problem. In R. Möhring and R. Raman, editors, *ESA 2002*, volume 2461 of *Lecture Notes in Computer Science*, pages 795–807, 2002. Springer.
11. A. Pönitz and P. Tittmann. Computing network reliability in graphs of restricted pathwidth. Technical report, Hochschule Mittweida, 2001.
12. H. Röhrig. Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, 1998.
13. J. S. Salowe and D. M. Warme. Thirty-five point rectilinear Steiner minimal trees in a day. *Networks*, 25:69–87, 1995.
14. L. P. Schakel. Personal communication, 2005. Faculty of Economics, University of Groningen, and `http://www.lofar.org/`.
15. S. Vahdati Daneshmand. *Algorithmic Approaches to the Steiner Problem in Networks*. PhD thesis, University of Mannheim, 2004. `http://bibserv7.bib.uni-mannheim.de/madoc/volltexte/2004/176`.
16. D. M. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plane Steiner tree problems: A computational study. In D-Z. Du, J. M. Smith, and J. H. Rubinstein, editors, *Advances in Steiner Trees*, pages 81–116. Kluwer, 2000.

Algorithmic and Complexity Results for Decompositions of Biological Networks into Monotone Subsystems

Bhaskar DasGupta[1,5], German A. Enciso[2,6], Eduardo Sontag[3,7], and Yi Zhang[4,8]

[1] Department of Computer Science, University of IL at Chicago, Chicago, IL 60607
dasgupta@cs.uic.edu
[2] Mathematical Biosciences Institute, 250 Mathematics Building, 231 W 18th Ave, Columbus, OH 43210
genciso@mbi.osu.edu
[3] Department of Mathematics, Rutgers University, New Brunswick, NJ 08903
sontag@math.rutgers.edu
[4] Department of Computer Science, University of IL at Chicago, Chicago, IL 60607
yzhang3@cs.uic.edu

Abstract. A useful approach to the mathematical analysis of large-scale biological networks is based upon their decompositions into monotone dynamical systems. This paper deals with two computational problems associated to finding decompositions which are optimal in an appropriate sense. In graph-theoretic language, the problems can be recast in terms of maximal sign-consistent subgraphs. The theoretical results include polynomial-time approximation algorithms as well as constant-ratio in-approximability results. One of the algorithms, which has a worst-case guarantee of 87.9% from optimality, is based on the semidefinite programming relaxation approach of Goemans-Williamson [14]. The algorithm was implemented and tested on a Drosophila segmentation network [7] and an Epidermal Growth Factor Receptor pathway model [25], and it was found to perform close to optimally.

1 Introduction

In living cells, networks of proteins, RNA, DNA, metabolites, and other species process environmental signals, control internal events such as gene expression, and produce appropriate cellular responses. The field of systems (molecular) biology is largely concerned with the study of such networks, viewed as dynamical systems. One approach to their mathematical analysis relies upon viewing them

[5] Supported by NSF grants CCR-0206795, CCR-0208749 and IIS-0346973.
[6] Work done while the author was with the Mathematics Department of Rutgers University and partly supported by NSF grant CCR-0206789.
[7] Partly supported by NSF grants EIA 0205116 and DMS-0504557.
[8] Partly supported by NSF grants CCR-0206795 and CCR-0208749.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 253–264, 2006.

as made up of subsystems whose behavior is simpler and easier to understand. Coupled with appropriate interconnection rules, the hope is that emergent properties of the complete system can be deduced from the understanding of these subsystems.

A particularly appealing class of candidates for "simpler behaved" subsystems are *monotone systems*, as in [16, 15, 29]. Monotone systems are a class of dynamical systems for which pathological behavior ("chaos") is ruled out. Even though they may have arbitrarily large dimensionality, monotone systems behave in many ways like one-dimensional systems. For instance, in monotone systems, bounded trajectories generically converge to steady states, and there are no stable oscillatory behaviors. Monotonicity is closely related to positive and feedback loops in systems. The topic of analyzing the behaviors of such feedback loops is a long-standing one in biology in the context of regulation, metabolism, and development; a classical reference in that regard is the work [23] of Monod and Jacob in 1961. See also, for example, [20, 22, 34, 26, 31, 6, 4, 1, 27].

An interconnection of monotone subsystems, that is to say, an entire system made up of monotone components, may or may not be monotone: "positive feedback" (in a sense that can be made precise) preserves monotonicity, while "negative feedback" destroys it. Thus, oscillators such as circadian rhythm generators require negative feedback loops in order for periodic orbits to arise, and hence are not themselves monotone systems, although they can be decomposed into monotone subsystems (cf. [5]). A rich theory is beginning to arise, characterizing the behavior of non-monotone interconnections. For example, [3] shows how to preserve convergence to equilibria; see also the follow-up papers [2, 18, 12, 9, 13]. Even for monotone interconnections, the decomposition approach is very useful, as it permits locating and characterizing the stability of steady states based upon input/output behaviors of components, as described in [4]; see also the follow-up papers [1, 11, 19]. Moreover, a key point brought up in [32] is that new techniques for monotone systems in many situations allow one to characterize the behavior of an entire system, based upon the "qualitative" knowledge represented by general network topology and the inhibitory or activating character of interconnections, combined with only a relatively *small amount of quantitative* data. The latter data may consist of steady-state responses of components (dose-response curves and so forth), and there is no need to know the precise form of dynamics or parameters such as kinetic constants in order to obtain global stability conclusions.

Generally, a graph, whose edges are labeled by "+" or "−" signs (sometimes one writes $+1, -1$ instead of $+, -$, or uses respectively activating "→" or inhibiting "⊣" arrows), is said to be *sign-consistent* if all paths between any two nodes have the same net sign, or equivalently, all closed loops have positive parity, i.e. an even number, possibly zero, of negative edges. (For technical reasons, one ignores the direction of arrows, looking only at undirected graphs; see more details in Section 2.)

When applying decomposition theorems such as those described in [3, 4, 1, 32, 2, 18, 11, 19, 12, 9, 13], it tends to be the case that *the fewer the number*

of interconnections among components, the easier it is to obtain useful conclusions. One may view a decomposition into interconnections of monotone subsystems as the "pulling out" of "inconsistent" connections among monotone components, the original system being a "negative feedback" loop around an otherwise consistent system. In this interpretation, the number of interconnections among monotone components corresponds to the number of variables being fed-back. In addition, and independently from the theory developed in the above references, one might speculate that nature tends to favor systems that are decomposable into small monotone interconnections, since "negative" feedback loops, although required for homeostasis and for periodic behavior, have potentially destabilizing effects, especially if there are signal propagation delays. Some evidence is provided by work in progress such as [21], where the authors compare certain biological networks and appropriately randomized versions of them and show that the original networks are closer to being consistent, and by [28], where the authors show that, in a Boolean setting, and using a mean-field calculation of sensitivity, networks of Boolean functions behave in a more and more "orderly" fashion the closer that the components are to being monotone.

Thus, we are led to the subject of this paper, namely computing the smallest number of edges that have to be removed so that there remains a consistent graph. In this paper, we study the computational complexity of the question of how many edges must be removed in order to obtain consistency, and we provide a polynomial-time approximation algorithm guaranteed to solve the problem to about 87.9% of the optimum solution, which is based on the semidefinite programming relaxation approach of Goemans-Williamson [14] (A variant of the problem is discussed as well). We also observe that it is not possible to have a polynomial-time algorithm with performance too close to the optimal. While our emphasis is on theory, one of the algorithms was implemented, and we show results of its application to a Drosophila segmentation network and to an Epidermal Growth Factor Receptor pathway model. It turns out that, when applying the algorithm, often the solution is much closer to optimal than the worst-case guarantee of 87.9%, and indeed often gives an optimal solution.

2 Monotone Systems and Consistency

We will illustrate the motivation for the problem studied here using systems of ordinary differential equations

$$\dot{x} = F(x) \tag{1}$$

(the dot indicates time derivative, and $x = x(t)$ is a vector), although the discussion applies as well to more general types of dynamical systems such as delay-differential systems or certain systems of reaction-diffusion partial differential equations. In applications to biological networks, the component $x_i(t)$ of the vector $x = x(t)$ indicates the concentration of the ith species in the model at time t. We will restrict attention to models in which the direct effect that one given variable in the model has over another is either consistently inhibitory or consistently promoting. Thus, if protein A binds to the promoter region of

gene B, we assume that it does so either to consistently prevent the transcription of the gene or to consistently facilitate it. (Of course, this condition does not prevent protein A from having an indirect influence, through other molecules, perhaps dimmers of A itself, that can ultimately lead to the opposite effect on gene B.) Mathematically, we require that for every $i, j = 1 \ldots n$, $i \neq j$, the partial derivative $\partial F_i / \partial x_j$ be either ≥ 0 at all states or ≤ 0 at all states.

Given any partial order $\leq$ defined on $\mathbb{R}^n$, a system (1) is said to be *monotone with respect to* $\leq$ if $x_0 \leq y_0$ implies $x(t) \leq y(t)$ for every $t \geq 0$. Here $x(t)$, $y(t)$ are the solutions of (1) with initial conditions x_0, y_0, respectively. Of course, whether a system is monotone or not depends on the partial order being considered, but we one says simply that a system is *monotone* if the order is clear from the context. Monotonicity with respect to nontrivial orders rules out chaotic attractors and even stable periodic orbits; see [16, 15, 29], and is, as discussed in the introduction, a useful property for components when analyzing larger systems in terms of subsystems.

A useful way to define partial orders in $\mathbb{R}^n$, and the only one to be further considered in this paper, is as follows. Given a tuple $s = (s_1, \ldots s_n)$, where $s_i \in \{1, -1\}$ for every i, we say that $x \leq_s y$ if $s_i x_i \leq s_i y_i$ for every i. For instance, the "cooperative order" is the orthant order $\leq_s$ generated by $s = (1, \ldots 1)$. This is the order $\leq$ defined by $x \leq y$ if and only if $x_i \leq y_i$ for all $i = 1, \ldots, n$. It is not difficult to verify if a system is cooperative with respect to an orthant order; the following lemma, known as "Kamke's condition," is not hard to prove, see [29] for details (also [3] in the more general context of monotone systems with input and output channels).

Lemma 1. *Consider an orthant order $\leq_s$ generated by $s = (s_1, \ldots, s_n)$. A system (1) is monotone with respect to $\leq_s$ if and only if*

$$s_i s_j \frac{\partial F_j}{\partial x_i} \geq 0, \ \ i, j = 1 \ldots n, \ \ i \neq j. \tag{2}$$

An equivalent way to phrase this condition is to ask that $\partial F_i / \partial x_j \geq 0$ at all states for every $i, j, i \neq j$, which is the Kamke condition for the special case of the cooperative order. The name of the order arises because in a monotone system with respect to that order each species promotes or "cooperates" with each other.

A rephrasing of this characterization of monotonicity with respect to orthant orders can be given by looking at the *signed digraph* associated to (1) and defined as follows. Let $V(G) = \{1, \ldots, n\}$. Given vertices i, j, let $(i, j) \in E(G)$ and $f_E(i, j) = 1$ if both $\partial F_j / \partial x_i \geq 0$ and the strict inequality holds at least at one state. Similarly let $(i, j) \in E(G)$ and $f_E(i, j) = -1$ if both $\partial F_j / \partial x_i \leq 0$ and the strict inequality holds at least at one state. Finally, let $(i, j) \notin E(G)$ if $\partial F_j / \partial x_i \equiv 0$. Recall that we are assuming that one of the three cases must hold. Now we can define an orthant cone using any function $f_V : V(G) \to \{-1, 1\}$, by letting $x \leq_{f_V} y$ if and only if $f_V(i) x_i \leq f_V(i) y_i$ for all i. Given f_V, we define the consistency function $g : E(G) \to \{\text{true, false}\}$ by $g(i, j) = f_V(i) f_V(j) f_E(i, j)$. Then, the following analog of Lemma 1 holds.

Lemma 2. *Consider a system (1) and an orthant cone* $\leq_{f_V}$. *Then (1) is monotone with respect to* $\leq_{f_V}$ *if and only if* $g(i,j) \equiv 1$ *on* $E(G)$.

For the next lemma, let the *parity* of a chain in G be the product of the signs $(+1,-1)$ of its individual edges. We will consider in the next result closed *undirected chains*, that is, sequences $x_{i_1} \dots x_{i_r}$ such that $x_{i_1} = x_{i_r}$, and such that for every $\lambda = 1, \dots, r-1$ either $(x_{i_\lambda}, x_{i_{\lambda+1}}) \in E(G)$ or $(x_{i_{\lambda+1}}, x_{i_\lambda}) \in E(G)$.

Lemma 3. *Consider a dynamical system (1) with associated directed graph* G. *Then (1) is monotone with respect to some orthant order if and only if all closed undirected chains of* G *have parity* 1.

2.1 Systems with Inputs and Outputs

As we discussed in the introduction, a useful approach to the analysis of biological networks consists of decomposing a given system into an interconnection of monotone subsystems. The formulation of the notion of interconnection requires subsystems to be endowed with "input and output channels" through which information is to be exchanged. In order to address this we consider *controlled* dynamical systems ([33], which are systems with an additional parameter $u \in \mathbb{R}^m$, and which have the form

$$\dot{x} = g(x,u). \tag{3}$$

The values of u over time are specified by means of a function $t \to u(t) \in \mathbb{R}^m$, $t \geq 0$, called an *input* or *control*. Thus each input defines a time-dependent dynamical system in the usual sense. To system (3) there is associated a *feedback function* $h : \mathbb{R}^n \to \mathbb{R}^m$, which is usually used to create the closed loop system $\dot{x} = g(x, h(x))$. Finally, if $\mathbb{R}^n, \mathbb{R}^m$ are ordered by orthant orders $\leq_{f_V}$, $\leq_q$ respectively, we say that the system is monotone if it satisfies (2) for every u, and also

$$q_k f_V(j) \frac{\partial g_j}{\partial u_k} \geq 0, \text{ for every } k, j \tag{4}$$

(see also [3].) As an example, let us consider the following biological model of testosterone dynamics [10, 24]:

$$\begin{aligned} \dot{x}_1 &= \frac{A}{K + x_3} - b_1 x_1 \\ \dot{x}_2 &= c_1 x_1 - b_2 x_2 \\ \dot{x}_3 &= c_2 x_2 - b_3 x_3. \end{aligned} \tag{5}$$

Drawing the digraph of this system, it is easy to see that it is not monotone with respect to any orthant order, as follows by application of Lemma 3. On the other hand, replacing x_3 in the first equation by u, we obtain a system that is monotone with respect to the orders $\leq_{(1,1,1)}$, $\leq_{(-1)}$ for state and input respectively. Defining $h(x) = x_3$, the closed loop system of this controlled system is none other than (5). The paper [10] shows how, using this decomposition

together with the "small gain theorem" from monotone input/output theory ([3]) leads one to a proof that the system does not have oscillatory behavior, even under arbitrary delays in the feedback loop, contrary to the assertion made in [24]. We can carry out this procedure on an arbitrary system (1) with a directed graph G as follows: given a set E of edges in G, enumerate the edges in E^C as $(i_1, j_1), \ldots (i_m, j_m)$. For every $1 \leq k \leq m$, replace all appearances of x_{i_k} in the function F_{j_k} by the variable u_k to form the function $g(x, u)$. Define $h(x) = (x_{i_1}, \ldots x_{i_m})$. It is easy to see that this controlled system (3) has closed loop (1).

Let the set E be called *consistent* if the undirected subgraph of G generated by E has no closed chains with parity -1. Note that this is equivalent to the existence of f_V such that $g \equiv 1$ on E, by Lemma 4 applied to the open loop system (3). If E is consistent, then the associated system (3) itself can also be shown to be monotone: to verify condition (4), simply define each q_k so that (4) is satisfied for k, j_k. Since $\partial g_{j_k}/\partial u_k = \partial F_{j_k}/\partial x_{i_k} \not\equiv 0$, this choice is in fact unambiguous. Conversely, if (3) is monotone with respect to the orthant orders $\leq_{f_V}$, $\leq_q$, then in particular it is monotone for every fixed constant u, so that E is consistent by Lemma 3. We thus have the following result.

Lemma 4. *The set of edges E of the digraph G is consistent iff the corresponding controlled system* (3) *is monotone with respect to some orthant orders.*

3 Statement of Problem

A natural problem is therefore the following. Given a dynamical system (1) that admits a digraph G, use the procedure above to decompose it as the closed loop of a monotone controlled system (3), while minimizing the number $\|E^C\|$ of inputs. Equivalently, *find f_V such that $P(E_+) = \| E_+ \|$ is maximized and $P(E_-) = \| E_- \| = \|E_+^C\|$ minimized*. This produces the following problem formulation.

Problem 1 (Undirected Labeling Problem(ULP))
An instance of this problem is (G, h), where $G = (V, E)$ is an undirected graph and $h: E \mapsto \{0, 1\}$. A valid solution is a vertex labeling function $f: V \to \{0, 1\}$. Define an edge $\{u, v\} \in E$ to be consistent iff $h(u, v) \equiv (f(u) + f(v)) \pmod 2$. The objective is then to find a valid solution maximizing $|F|$ *where F is the set of consistent edges.*

There is a second, slightly more sophisticated way of writing a system (1) as the feedback loop of a system (3) using an arbitrary set of edges E. Given any such E, define $S(E^c) = \{i \mid \text{there is some } j \text{ such that } (i, j) \in E^c\}$. Now enumerate $S(E^c)$ as $\{i_1, \ldots i_m\}$, and for each k label the set $\{j \mid (i_k, j) \in E^c\}$ as $j_{k1}, j_{k2}, \ldots$. Then for each k, l, one can replace each appearance of x_{i_k} in $F_{j_{kl}}$ by u_k, to form the function $g(x, u)$. Then one lets $h(x) = (x_{i_1}, \ldots, x_{i_m})$ as above. The closed loop of this system (3) is also (1) as before but with the advantage that there are $|S(E^c)|$ inputs, and of course $|S(E^c)| \leq |E^c|$.

If E is a consistent and *maximal* set, then one can make (3) into a monotone system as follows. By letting f_V be such that $g \equiv 1$ on E, we define the order $\leq_{f_V}$ on $\mathbb{R}^n$. For every i_k, j_{kl} such that $(i_k, j_{kl}) \in E^C$, it must hold that $f_V(i_k) f_V(j_{kl}) f_E(i_k, j_{kl}) = -1$. Otherwise $E \cup \{(i_k, j_{kl})\}$ would be consistent, thus violating maximality. By choosing $q_k = -f_V(i_k)$, equation (4) is therefore satisfied. Conversely, if the system generated by E using this second algorithm is monotone with respect to orthant orders, and if h is a negative function, then it is easy to verify that E must be both consistent and maximal.

Thus the problem of finding E consistent and such that $P(E_-) = \|S(E_-)\| = \|S(E^C)\|$ is smallest, when restricted to those sets that are maximal and consistent (this does not change the minimum $\|S(E^C)\|$), is equivalent to the following problem: decompose (1) into the negative feedback loop of an orthant monotone control system, using the second algorithm above, and using as few inputs as possible. This produces the following problem formulation.

Problem 2 (Directed Labeling Problem(DLP))
An instance of this problem is (G, h) *where* $G = (V, E)$ *is a directed graph and* $h: E \to \{0, 1\}$. *A valid solution is a vertex labeling function* $f: V \to \{0, 1\}$. *Define an edge* $(u, v) \in E$ *to be consistent iff* $h(u, v) \equiv (f(u) + f(v)) \pmod 2$. *The objective is then to find a valid solution* minimizing $|g(E - F)|$ *where* $g(C) = \{u \in V \mid \exists y \in V, (u, y) \in C\}$ *for any* $C \subseteq E$ *and* F *is the set of consistent edges.*

4 Theoretical Results

Theorem 5

(a) For some constant $\varepsilon > 0$, *it is not possible to approximate in polynomial time the ULP and the DLP problems to within an approximation ratio of* $1 - \varepsilon$ *and* $1 + \varepsilon$, *respectively, unless P=NP.*
(b) For ULP, we provide a polynomial time α*-approximation algorithm where* $\alpha \approx 0.87856$ *is the approximation factor for the MAX-CUT problem obtained in [14] via semidefinite programming.*
(c) For DLP, if d_{in}^{max} *denotes the maximum in-degree of any vertex in the graph, then we give a polynomial-time approximation algorithm with an approximation ratio of at most* $d_{in}^{max} \cdot O(\log |V|)$.

5 Two Examples of Applications of the ULP Algorithm

5.1 Drosophila Segment Polarity

An important part of the development of the early Drosophila (fruit fly) embryo is the differentiation of cells into several stripes (or *segments*), each of which eventually gives rise to an identifiable part of the body such as the head, the wings, the abdomen, etc. Each segment then differentiates into a posterior and an anterior part, in which case the segment is said to be *polarized*. (This differentiation process continues up to the point when all identifiable tissues of the fruit

fly have developed.) Differentiation at this level starts with differing concentrations of certain key proteins in the cells; these proteins form striped patterns by reacting with each other and by diffusion through the cell membranes.

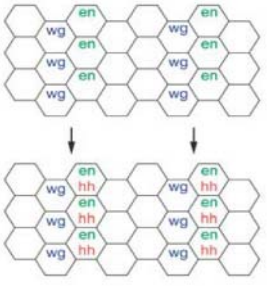


Fig. 1. A digram of the Drosophila embryo during early development. A part of the segment polarization process is displayed. Courtesy of N. Ingolia and PLoS [17].

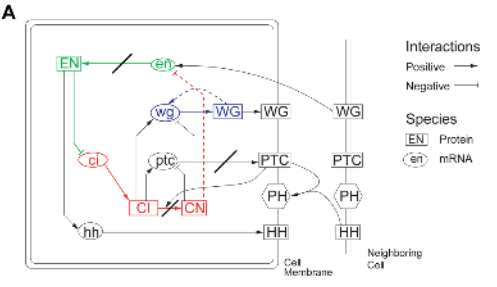


Fig. 2. The network associated to the Drosophila segment polarity, as proposed in [7], Courtesy of N. Ingolia and PLoS. The three edges that have been crossed have been chosen in order to let the remaining edges form an orthant monotone system.

A model for the network that is responsible for segment polarity [7] is illustrated in Figure 2. As explained above, this model is best studied when multiple cells are present interacting with each other. But it is interesting at the one-cell level in its own right — and difficult enough to study that analytic tools seem mostly unavailable. The arrows with a blunt end are interpreted as having a negative sign in our notation. Furthermore, the concentrations of the membrane-bound and inter-cell traveling compounds PTC, PH, HH and WG(membrane) on all cells have been identified in the one-cell model (so that, say, HH→ PH is now in the digraph). Finally, PTC acts on the reaction CI→ CN itself by promoting it without being itself affected, which in our notation means PTC $\stackrel{+}{\rightarrow}$ CN *and* PTC $\stackrel{-}{\rightarrow}$ CI.

The Implementation. The Matlab implementation of the algorithm on this digraph with 13 nodes and 20 edges produced several partitions with as many as 17 consistent edges. One of these possible partitions simply consists of placing the three nodes ci, CI and CN in one set and all other nodes in the other set, whereby the only inconsistent edges are CL $\stackrel{+}{\rightarrow}$ wg, CL $\stackrel{+}{\rightarrow}$ ptc, and PTC $\stackrel{+}{\rightarrow}$ CN. But note that it is desirable for the resulting open loop system to have as simple remaining loops as possible after eliminating all inconsistent edges. In this case, the remaining directed loops $EN \stackrel{-}{\rightarrow} ci \stackrel{+}{\rightarrow} CI \stackrel{+}{\rightarrow} CN \stackrel{-}{\rightarrow} en \stackrel{+}{\rightarrow} EN$ and $EN \stackrel{-}{\rightarrow} ci \stackrel{+}{\rightarrow} CI \stackrel{+}{\rightarrow} CN \stackrel{-}{\rightarrow} wg \stackrel{+}{\rightarrow} WG \stackrel{+}{\rightarrow} WG(membrane) \stackrel{+}{\rightarrow} en \stackrel{+}{\rightarrow} EN$ can still cause difficulties.

A second partition which generated 17 consistent edges is that in which EN, hh, CN, and the membrane compounds PTC, PH, HH are on one set, and the remaining compounds on the other. The edges cut are ptc $\xrightarrow{+}$ PTC, CI $\xrightarrow{+}$ CN and en $\xrightarrow{+}$ EN, each of which eliminates one or several positive loops. By writing the remaining consistent digraph in the form of a cascade, it is easy to see that the only loop whatsoever remaining is wg $\leftrightarrow$ WG; this makes the analysis proposed in [12] easier. In this relatively low dimensional case we can prove that in fact $OPT = 17$ as stated below.

Lemma 6. *Any partition of the nodes in the digraph in Figure 2 generates at most* 17 *consistent edges.*

It is surprising that a realistic biological system with as many as 13 variables and 20 edges can be transformed into a monotone system after the deletion of only three nodes. It is conceivable that this restricts the possible dynamics of the system. This is especially the case given that the open loop digraph has almost no closed oriented paths (except for $WG \leftrightarrow wg$), which is evidence that the dynamics of the control system under constant inputs may be especially simple, e.g. such that all solutions converge towards a unique equilibrium.

Multiple Copies. It was mentioned above that the purpose of this network is to create striped patterns of protein concentrations along multiple cells. In this sense, it is most meaningful to consider a *coupled* collection of networks as it is given originally in Figures 1 and 2. Consider a row of k cells, each of which has independent concentration variables for each of the compounds, and let the cell-to-cell interactions be as in Figure 2 with cyclic boundary conditions (that is, the k-th cell is coupled with the first in the natural way). We show that the results can be extended in a very similar manner as before.

Lemma 7. *For the k-cell linearly coupled network described in Figure 2, OPT=17k.*

5.2 EGFR Signaling

In their May 2005 paper [25], Oda et al. integrate the information that has become available about the *epidermal growth factor receptor* (EGFR) signalling process from multiple sources, and they define a network with 330 known molecules under 211 chemical reactions. The network itself is available from the supplementary material in SBML format (*Systems Biology Markup Language, www.sbml.org*), and will most likely be subject to continuous updates.

The Implementation. Each reaction in the network classifies the molecules as reactants, products, and/or modifiers (enzymes). This information was imported into Matlab using the Systems Biology Toolbox. The digraph G that is used for this analysis has many more edges than the digraph considered in the digraph displayed in [25]. The reason for this is as follows: if molecules A and B are both reactants in the same reaction, then the presence of A will have an indirect inhibiting effect on the concentration of B, since it will accelerate the reaction

which consumes B (assuming B is not also a product). Therefore a negative edge must also appear from A to B, and vice versa. Similarly, modifiers have an inhibiting effect on reactants. We thus define G by letting $\text{sign}(i,j) = 1$ if there exists a reaction in which j is a product and i is either a reactant or a modifier. We let $\text{sign}(i,j) = -1$ if there exists a reaction in which j is a reactant, and i is also either a reactant or a modifier. Similarly $\text{sign}(i,j) = 0$ if the nodes i,j are not simultaneously involved in any given reaction, and $\text{sign}(i,j)$ is undefined (NaN) if the first two conditions above are both satisfied. An undefined edge can be thought of as an edge that is *both* positive and negative, and it can be dealt with, given an arbitrary partition, by deleting exactly one of the two signed edges so that the remaining edge is consistent. Thus, in practice, one can consider undefined edges as edges with sign 0, and simply add the number of undefined edges to the number of inconsistent edges in the end of each procedure, in order to form the total number of inputs. This is the approach followed here; there are exactly 7 such entries in the digraph G.

The Results. After running the algorithm 100 times for this problem, and choosing that partition which produced the highest number of consistent edges, the induced consistent set contained 633 out of 852 edges (ignoring the edges on the diagonal and the 7 undefined edges). See the supplementary material for the relevant Matlab functions that carry out this algorithm. A procedure analogous to that carried out for system (5) allows to decompose the system as the feedback loop of a controlled monotone system using $852 - 633 = 219$ inputs. Since the induced consistent set is maximal by definition, we are guaranteed that the function h is a negative feedback. Contrary to the previous application, many of the reactions involve several reactants and products in a single reaction. This induces a denser amount of negative and positive edges: even though there are 211 reactions, there are 852 (directed) edges in the 330×330 graph G. It is very likely that this substantially decreases OPT for this system. The approximation ratio of the SDP algorithm is guaranteed to be at least 0.87 for some r, which gives the estimate $\text{OPT} \leq \approx 633/0.87 \approx 728$ (valid to the extent that r has sampled the right areas of the 330-dimensional sphere, but reasonably accurate in practice).

One procedure that can be carried out to lower the number of inputs is a hybrid algorithm involving *out-hubs*, that is, nodes with an abnormally high out-degree. Recall from the description of the DLP algorithm that all the out-edges of a node x_i can be potentially cut at the expense of only one input u, by replacing all the appearances of x_i in $f_j(x)$, $j \neq i$, by u. We considered the k nodes with the highest out-degrees, and eliminated all the out-edges associated to these hubs from the reaction digraph to form the graph G_1. Then we run the ULP algorithm on G_1 to find a partition f_V of the nodes and a set of m edges that can be cut to eliminate all remaining negative closed chains. Finally, we put back on the digraph those edges that were taken in the first step, and which are consistent with respect to the partition f_V. The result is a decomposition of the system as the negative feedback loop of a controlled monotone system, using at most $k + m$ edges.

An implementation of this algorithm with $k = 60$ yielded a total maximum number of inputs $k + m = 137$. This is a significant improvement over the 226 inputs in the original algorithm. Clearly, it would be worthwhile to investigate further the problem of designing efficient algorithms for the DLP problem to generate improved hybrid algorithmic approaches. The approximation ratios in Theorem 5(c) are not very satisfactory since d_{in}^{max} and $\log |V|$ could be large factors; hence future research work may be carried out in designing better approximation algorithms.

5.3 Supplementary Material: MATLAB Implementation Files

A set of MATLAB programs have been written to implement the algorithms described in this paper. They can be accessed from the URL
`http://www.math.rutgers.edu/~sontag/desz_README.html`.

References

1. D. Angeli, J. E. Ferrell Jr. and E. D. Sontag. *Detection of multi-stability, bifurcations, and hysteresis in a large class of biological positive-feedback systems*, Proc. Nat. Acad. Sci. USA, 101: 1822-1827, 2004.
2. D. Angeli, P. De Leenheer and E. D. Sontag. *A small-gain theorem for almost global convergence of monotone systems*, Systems and Control Letters, 51: 185-202, 2004.
3. D. Angeli and E.D. Sontag. *Monotone control systems*, IEEE Trans. Autom. Control, 48: 1684-1698, 2003.
4. D. Angeli and E. D. Sontag. *Multistability in monotone I/O systems*, Systems and Control Letters, 51: 185-202, 2004.
5. D. Angeli and E. D. Sontag. *An analysis of a circadian model using the small-gain approach to monotone systems*, proceedings of the IEEE Conf. Decision and Control, Paradise Island, Bahamas, Dec. 2004, IEEE Publications, 575-578, 2004.
6. O. Cinquin and J. Demongeot. *Positive and negative feedback: striking a balance between necessary antagonists*, J. Theor. Biol., 216: 229-241, 2002,
7. G. von Dassaw, E. Meir, E.M. Munro, and G.M. Odell. *The segment polarity network is a robust developmental module*, Nature 406: 188-192, 2000.
8. D. L. DeAngelis, W. M. Post and C. C. Travis. *Positive Feedback in Natural Systems*, Springer-Verlag, New York, 1986.
9. G.A. Enciso, H.L. Smith, and E. D. Sontag. *Non-monotone systems decomposable into monotone systems with negative feedback*, J. of Differential Equations, 2005, to appear.
10. G. Enciso and E. Sontag. *On the stability of a model of testosterone dynamics*, Journal of Mathematical Biology 49: 627-634, 2004.
11. G. Enciso and E. D. Sontag. *Monotone systems under positive feedback: multistability and a reduction theorem*, Systems and Control Letters, 54: 159-168, 2005.
12. G. Enciso and E. Sontag. *Global attractivity, I/O monotone small-gain theorems, and biological delay systems*, Discrete and Continuous Dynamical Systems, to appear.
13. T. Gedeon and E. D. Sontag. *Oscillation in multi-stable monotone system with slowly varying positive feedback*, submitted for publication (abstract in Sixth SIAM Conf. on Control and its Applications, New Orleans, July 2005).

14. M. Goemans and D. Williamson. *Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming*, Journal of the ACM, 42 (6): 1115-1145, 1995.
15. M. Hirsch. *Systems of differential equations that are competitive or cooperative II: Convergence almost everywhere*, SIAM J. Mathematical Analysis, 16: 423-439, 1985.
16. M. Hirsch. *Differential equations and convergence almost everywhere in strongly monotone flows*, Contemporary Mathematics, 17: 267-285, 1983.
17. N. Ingolia. *Topology and robustness in the drosophila segment polarity network*, Public Library of Science, 2 (6): 0805–0815, 2004.
18. P. De Leenheer, D. Angeli and E.D. Sontag. *On predator-prey systems and small-gain theorems*, J. Mathematical Biosciences and Engineering, 2: 25-42, 2005.
19. P. De Leenheer and M. Malisoff. *Remarks on monotone control systems with multi-valued input-state characteristics*, proceedings of the 44th IEEE Conference on Decision and Control and European Control Conference ECC 05 (Seville, Spain, December 2005), to appear.
20. J. Lewis, J.M. Slack and L. Wolpert. *Thresholds in development*, J. Theor. Biol., 65: 579-590, 1977.
21. A. Maayan, R. Iyengar and E. D. Sontag. *A study of almost-sign consistency on biological networks*, in preparation.
22. H. Meinhardt. *Space-dependent cell determination under the control of morphogen gradient*, J. Theor. Biol., 74: 307-321, 1978.
23. J. Monod and F. Jacob. *General conclusions: telenomic mechanisms in cellular metabolism, growth, and differentiation*, Cold Spring Harbor Symp. Quant. Biol., 26: 389-401, 1961.
24. J.D. Murray. *Mathematical Biology, I: An introduction*, New York, Springer, 2002.
25. K. Oda, Y. Matsuoka, A. Funahashi, H. Kitano. *A comprehensive pathway map of epidermal growth factor receptor signaling*, Molecular Systems Biology doi:10.1038/msb4100014, 2005.
26. E. Plathe, T. Mestl and S.W. Omholt. *Feedback loops, stability and multistationarity in dynamical systems*, J. Biol. Syst., 3: 409-413, 1995.
27. E. Remy, B. Mosse, C. Chaouiya and D. Thieffry. *A description of dynamical graphs associated to elementary regulatory circuits*, Bioinformatics, 19 (Suppl. 2): ii172-ii178, 2003.
28. I. Shmulevich and E.D. Sontag. *Order and Distance to Monotonicity in Boolean Networks*, in preparation.
29. H. L. Smith. *Monotone Dynamical Systems*, Providence, R.I., AMS 1995.
30. H. L. Smith. *Systems of ordinary differential equations wich generate an order-preserving flow: A survey of results*, SIAM Revews, 30: 87-111, 1988.
31. E. H. Snoussi. *Necessary conditions for multistationarity and stable periodicity*, J. Biol. Syst., 6: 39, 1998.
32. E. D. Sontag. *Some new directions in control theory inspired by systems biology*, Systems Biology, 1: 9-18, 2004.
33. E. D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems*, Springer, New York, 1990 (Second Edition, 1998).
34. R. Thomas. *Logical analysis of systems comprising feedback loops*, J. Theor. Biol., 73: 631656, 1978.
35. A. I. Volpert, V. A. Volpert, and V. A. Volpert. *Traveling Wave Solutions of Parabolic Systems*, volume 140 of Translations of Mathematical Monographs, AMS, 2000.

A Maximum Profit Coverage Algorithm with Application to Small Molecules Cluster Identification

Refael Hassin and Einat Or

Department of Statistics and Operations Resea rch, Tel Aviv University,
Tel Aviv, 69978, Israel
{hassin, eior}@post.tau.ac.il

Abstract. In this paper we present the CLUSTER IDENTIFICATION OF MOLECULES (CIM), which is a clustering problem in a finite metric space. We model the problem as a PARAMETER ESTIMATION VIA LIKELIHOOD MAXIMIZATION and as a novel clustering problem, the MAXIMUM PROFIT COVERAGE PROBLEM (MPCP). We present a numerical study in which we compare a greedy heuristic and a random heuristic for MPCP, to the known Expectation Minimization approach for the likelihood maximization model. We present a polynomial time approximation scheme for MPCP in Euclidean space.

1 Introduction

1.1 Problem Definition

In this article we model, and analyze the CLUSTER IDENTIFICATION OF MOLECULES (CIM), which is a clustering problem in a finite metric space. CIM[1] has the following characteristics which separate it from other clustering models: 1. In most models outliers are a small portion of the data set, whereas in CIM they may be the vast majority of the objects. 2. The clusters identified by CIM are compact and their diameter is bounded. 3. There is a lower bound on the number of objects in a cluster. 4. Clusters may be very close to one another, as a result of the bound on the diameter. What may be considered as one cluster in other clustering models is considered as several clusters in CIM. 5. The number of clusters is not known a-priori to the clustering procedure.

In this paper we present CIM and model it as a MAXIMUM PROFIT COVERAGE PROBLEM (MPCP). The model is a measure to be optimized, rather then a heuristic.

Consider a finite set S in a metric space M with a distance function d. A ball with center t and radius r is the subset $B(t,r) = \{x \in M | d(t,x) \leq r\}$. We say that the ball *covers* the points of S that it contains. Given a set of balls $\mathcal{B}$ of radius r, a *coverage* $P = \{S'_1, \ldots, S'_l\}$ is a set of clusters such that each of them consists of points covered by a single ball of $\mathcal{B}$. Let $S'_P = \cup_{i=1}^{l} S'_i$, and define

[1] The problem originated in *COMPUGEN* LTD. in the field of "in silico" drug design.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 265–276, 2006.

the *profit* of P as $\sum_{q \in S'_P} w_q - c|P|$, where c is the cost of a ball used by P, and w_q is a revenue obtained by covering $q \in S$. The MAXIMUM PROFIT COVERAGE PROBLEM (MPCP) is the problem of finding a coverage with maximum profit.

1.2 Related Work

The $MPCP$ is related to the BUDGETED MAXIMUM COVERAGE PROBLEM, which has a $\left(1 - \frac{1}{e}\right)$-approximation using a greedy algorithm [13]. This approximation bound cannot be used to approximate $MPCP$.

Clustering. There is a large variety of clustering models, as reflected for example in the survey by Du and Paradalos [4], and they have numerous applications in many different areas. Even simple variants are known to be NP-hard and therefore research has been focused on approximation algorithms and heuristics. The heuristics used in practice are often based on empirical experience (e.g [14]). Most measures (i.e. objective functions to be optimized) are based on a given number of clusters and the relation of the between-clusters weight and the in-cluster weights.

Most heuristics are based on two main methods. One is the *hierarchical* method, which re-partitions the set until a stopping condition is met, or an aggregation presses which begins by considering each point as a cluster and then merging close clusters until a stopping condition is met [10, 16, 21, 22]. The second is the k-means heuristic, where a *mean* of a cluster is the average of the clusters points. This non hierarchical method initially takes k point of the set which are mutually farthest apart, at this point each cluster has one point. Next, it examines each point in the set and assigns it to the nearest cluster. This continues until all the points are grouped into k clusters [10]. Both heuristics take the number of clusters as input, and do not return compact clusters of bounded diameter.

An important class of clustering problems is probabilistic clustering. It arises when the data consists of a set of points generated by an unknown *mixture* of distributions, and the problem is to estimate the parameters of each of the distributions creating the mixture, and its weight in the mixture. This task is commonly done by likelihood maximization (for example [17]). The mixture models solved by likelihood maximization can also be applied to cases where the number of distributions is unknown. There are methods for estimating the number of distributions [20].

Clustering with outliers. In statistics, *outliers* are defined as data objects which originated from a different probabilistic mechanism [1]. When clustering of data is considered, the intuitive definition of outliers becomes "points which do not belong to any of the clusters". A more specific definition is derived from the clustering target or the clustering process [3, 5, 6, 9, 14, 25, 26]. In the case of the CIM, the outliers are objects in a neighborhood not dense enough, according to the given definition of density.

The natural problem of clustering a data set containing outliers, when the number of clusters is not predetermined, is usually solved by heuristics. Models for *k-median* and *k-center* with outliers are introduced in [2]. For *k-median*

outliers were considered by demanding payment on unclustered points and for *k-center* the number of points which are considered as outliers was added to the model. Heuristics for clustering data containing outliers were introduced, among others, in [3, 5, 6, 9, 14, 25, 26]. In hierarchical methods clusters that "grow" slowly are considered as outliers, whereas in the k-means method points that are far from all means are considered as outliers.

All algorithms presented in these articles are not suitable for CIM. The clusters found by the algorithms are not bounded in diameter, and/or have no lower bound on their density. Since the nature of clusters found is different from the one defined in CIM, the outliers definition is also different.

1.3 Our Contribution

We present two models for CIM, one as a PARAMETER ESTIMATION VIA LIKELIHOOD MAXIMIZATION and the other as MPCP. We introduce a polynomial time approximation scheme (PTAS) to MPCP in Euclidean space using the shifting strategy [11, 8]. We present two practical heuristics for MPCP, one greedy and the other random, which introduce good results in numerical studies of CIM.

This paper is organized as follows: In Section 2 we introduce CIM and model it as PARAMETER ESTIMATION VIA LIKELIHOOD MAXIMIZATION. In Section 3 we model the problem as MPCP. In Section 4 we introduce a random heuristic and a greedy heuristic for MPCP, and in Section 5 we introduce numerical results. In Section 6 we give some theoretical results, We also analyze a greedy algorithm.

2 Cluster Identification of Molecules

Finding a drug to an illness is a problem of a lock and a key. The lock is a protein (or, more precise, its active site), that should be inhibited or exhibited in the body. The key is a small molecule that binds to the protein and inhibits or exhibits its action in the body. There is no doubt that the key should have a structure that fits the lock, but the biochemical system in which this lock and key function is dynamic, and hence the Structure Activity Relation (SAR), of the small molecule and the given protein, was not yet unfolded.

One of the approaches to this problem is to investigate the relation between a secondary structure of the small molecule to the biochemical activity, rather than the regular structure model of atoms and bonds. The secondary structure views the small molecule as a set of chemical attributes such as base, acid, hydrophobic, hydrophilic, hydrogen bonds etc. Given a set of small molecules that bind to the same protein, we wish to check whether there is a similarity in their secondary structure. Since the protein binding site is big, different small molecules may bind in different parts of it, and several secondary structures may explain the binding. Still it is natural to assume that a secondary structure that appeared in many of the small molecules that bind to the protein, characterizes the protein, and hence explains the binding. The CLUSTER IDENTIFICATION OF MOLECULES is the problem of identifying recurring secondary structures (clusters of secondary structures) in a set of small molecules that bind to the same protein.

Formally, we consider the secondary structure as a set of colored points, called *nodes*, in $\mathbb{R}^3$. The coordinates represent the location of a chemical functionality and the color represents the nature of the functionality. We will use this representation throughout this section and refer to it as a *molecule structure*.

Denote by V_t the set of nodes of a molecule structure t, and let $n = |V_t|$. The $3-D$ structure of t can be expressed as a vector of $\binom{n}{2}$ distances $d^t(i,j)$ between the pairs $v_i, v_j \in V_t$. The distance measuring process has a normally distributed error with a constant variance.

Thus, t can be viewed as a $\binom{n}{2}$-dimensional multi-normal random variable. The variance of the distances is constant, and caused due to measurement errors.

The distance between two molecule structures with the same number of nodes and the same multi-set of colors of the nodes, is defined as follows. Let P denote the set of mappings $p : V_t \to V_s$ such that the color of the source and the objective is the same. The distance between the molecule structures t and s is

$$D(s,t) = min_{p \in P} \sqrt{\sum_{i,j \in V_t} [d^t(i,j) - d^s(p(i),p(j))]^2}.$$

If the number of nodes is different or the multi-set of node colors is different, then the distance is infinity.

The set of all molecule structures with a given number of nodes, a given multi-set of node colors and a distance function D is a metric space. In the remaining of this paper we denote such a metric space by M.

Given a set of molecule structures, the CLUSTER IDENTIFICATION OF MOLECULES can be viewed as the problem of estimating the parameters of a mixture of distributions, since each molecule is represented by the vector of its distances, which is a multi-normal random variable. We will now build the likelihood function L of this probabilistic clustering.

Let t denote a molecule structure in M. t has a positive probability to be generated by any of k multi-normal distributions considered by the mixture model. Consider the possibility that molecule structure t has originated from the j-th distribution. Let V_j denote the set of nodes of the molecule structure which is the mean of the j-th distribution. Let $L_{(j,p)}(t)$ denote the likelihood that t has originated from the j-th distribution under the mapping p of their nodes.

$$L_{(j,p)}(t) = \frac{1}{\sqrt{2\pi|\Sigma_j|}} \exp^{-\frac{1}{2}([x(p)-m_j])^T \Sigma_j^{-1} [x(p)-m_j]},$$

where (m_j, Σ_j) are the mean vector and covariance matrix of the j-th distribution, and $x(p)$ is x when adapted to m_j under the permutation p. The normality is a result of the error in the measure of the distance, as mentioned above. Denote by $\beta_{(j,t)}(p)$ the probability that p was the mapping by which t originated from the j-th distribution. Clearly, $\sum_{p \in P} \beta_{(j,t)}(p) = 1$. The likelihood of the molecule structure t given that it is obtained from the j-th distribution is:

$$L_j(t) = \sum_{p \in P} \beta_{(j,t)}(p) L_{(j,p)}(t).$$

No information exists on the distribution $\beta_{(j,t)}$, but it could be estimated by solving $\max_{p \in P} L_{(j,p)}(t)$. Even if we assume $\beta_{(j,t)}(p)$ is known, since $|P| = O(|V_t|!)$ this step is exponential in the dimension of M, this likelihood function is expensive to calculate for one point, let alone to maximize.

3 Cluster Identification of Molecules as a MPCP

Solving the likelihood maximization associated with the cluster identification of molecules in a general metric space is computationally difficult even for small sized instances, as demonstrated in Section 2. We now show how the problem can be approximated via the MAXIMUM PROFIT COVERAGE PROBLEM.

A general finite metric space can be described by a graph $G = (V, E)$, and a distance function D on its edges. In such a case a ball can be defined only as centered at a vertex. In the CIM the balls are defined by the subset of molecule structures which they cover, i.e for each subset that can be covered by a ball of radius r we define one ball in the input set of balls $\mathcal{B}$. We compute the set of balls exhaustively, by checking for each subset whether it is coverable by a ball of radius r. This step is theoretically exponential i the size of S, but in practice the number of balls is very small due to the distribution of points as presented in Section 2. In simulations the number of balls defined was $O(n^2)$, and the process of defining them was efficient.

Consider a set S of molecule structures. The CLUSTER IDENTIFICATION OF MOLECULES can be viewed as the search for such dense balls, using the average of the points covered by each ball as estimate for the mean value of the distribution that generated the points of S in the ball[2]. We use a ball since the variance is constant and equal in all dimensions, i.e. for all the distances in the molecule structure. The intuition behind this approach relies on the fact that a point close to t is likely to be generated by t, in terms of the value of the likelihood function.[3]

A ball $B(t, r)$, covers the molecule structures which are contained in it. There is a revenue of 1 from covering a molecule structure $t \in S$, and every ball used by the solution costs c. The cost c represents a lower bound on the number of molecule structures in the cluster, whereas $2r$ represents an upper bound on the diameter of the cluster. Let S_F denote the molecule structures covered by a coverage F. The profit of a coverage F is $|S_F| - c|F|$. The maximum profit

[2] This point is a molecule structure which is not in S, and can be reconstructed to a set of nodes in $\mathbb{R}^3$ from the vector of its distances, with an error negligible in relation to the variance of the measuring process.

[3] This view of density ignores some aspects of the normal behavior, such as greater density in proximity to the expected value. A change in the objective function may better integrate this characteristic of the problem into the model. If a value of a ball will include not only the number of points it contains but also an indication on their variance in the ball, another aspect of the normality of the data origin will be integrated into the solution, but other difficulties will airs. We will show that even the simple definition of density used in MPCP gives very good results.

coverage of S can be used to estimate the large clusters of S. Since many covers may have the same profit, the center of the cluster will be defined by the average of the molecule structures it covers.

4 Heuristics

4.1 Heuristics for MPCP

In this section we describe two heuristics for computing a coverage with high profit in a general metric space. The first is a greedy heuristics, and the second is based on a random selection of subsets. We denote the set of possible balls defined in Section 3 by $\mathcal{B}$. The number of such balls is $O(|S|^d)$, where $d = \binom{n}{2}$, where n is the number of nodes in a molecule structure of S.

Consider the following "greedy" algorithm denoted as **GR**:

Let $B_j \subset \mathcal{B}$ be the set of balls already chosen before the j-th iteration, $S'_j \subseteq S$ the set of points that are not covered by B_j, and n^j_B the number of points of S'_j covered by $B \in \mathcal{B}$. Recall that c denotes the cost of a ball. Let $R^j_B = n^j_B - c$ denote the profit from a ball B. A *profitable ball* is a ball that covers at least $c+1$ points of S'_j, i.e. $R^j_B > 0$. Let $B^*_j = \arg\max_{B\in\mathcal{B}}\{R^j_B\}$ (break ties arbitrarily). **GR** is performed as follow:
Let $\mathcal{B}_1 = \emptyset$. As long as there is a profitable ball, let $\mathcal{B}_{j+1} = B_j \cup \{B^*_j\}$.

The greedy algorithm is performed in $O(|S|^{d+1})$ time. The number of iterations is bounded by $\frac{|S|}{c+1}$, since at least $c+1$ new points are covered at each iteration since, and the number of operations at each iteration is $|\mathcal{B}| = O(|S|^d)$.

The randomized heuristic **RA** repeatedly generates random solutions to the problem, and eventually chooses the one with the maximal profit. A solution is generated by randomly choosing a profitable ball from the list of profitable balls, then updating the list and choosing again, until there is no profitable ball and the list is empty. The generated solution is compared with the previous best solution and saved if it gives a higher profit. The randomized algorithm is performed in $O(|S|^d)$ iterations, since the number of balls dominates the number of iterations[4].

4.2 A Heuristic for CIM in the Euclidean Space

If each molecule in S has nodes of distinct colors, then the metric space M is Euclidean. The distribution $\beta_{(k,t)}$ becomes deterministic and the problem becomes the known PARAMETER ESTIMATION FOR GAUSSIAN MIXTURE (PEGM) [15]. It requires to estimate the parameters of a set of multi-normal distributions $\{(m_k, \sum_k)\}_{k=1,\ldots,K}$, and their mixture proportions $\alpha_j \geq 0$, $\sum_{k=1}^K \alpha_k = 1$, when a set of points generated from these distributions is given. Note that the

[4] In practice the execution time of both the greedy and the random heuristics is much lower since $\mathcal{B}$ is smaller.

number of distributions is assumed to be predetermined. There are methods for estimating the number of distributions. The parameter estimation is usually done by likelihood maximization. In this case the likelihood function can be maximized numerically. We will elaborate on this case since we will use this model for comparison to the results obtained by modeling CIM as MPCP and using GR and RA. We use algorithm **MEM**. **MEM** is based on the common method of likelihood maximization via Expectation Maximization, i.e. the **EM** algorithm [15, 19, 23]. The input to the **EM** algorithm includes the number of distributions k and will return a maximum likelihood estimator of the parameters of k distributions that best explain S. In order to find the optimal k we use the rule given by [20] to estimate the number of generating distributions, i.e k that maximizes $\max \log\_\text{like}(k) + k \log(N)$, where N is the number of samples, and $\max \log\_\text{like}(k)$ is the maximal value of the likelihood function for k distributions. Since we are only interested in the distributions which generated a large number of points, i.e. the dense sets of S, we define a dense subset of S as one generated from a distribution with a mixture parameter α greater then $\frac{c}{|S|}$.

5 Numerical Comparison of the Heuristics

We described algorithms **GR** and **RA** for MPCP in a general metric space. However, in the Euclidean space CIM becomes the well-known PEGM where the number of distributions is unknown. Since PEGM has a well-known method of solution, presented in **MEM**, we conduct our numerical analysis in the Euclidean space, where we can compare **GR** and **RA** to **MEM**. Our numerical analysis is hence a comparison of the three algorithms on simulated input for the problem in the Euclidean space.

The algorithms were applied with the following parameters:

1. **MEM** was applied with $num\_iterations = 10000$ and $c = 3, 5, \ldots, 23$.
2. **GR** was applied with $c = 3, 5, \ldots, 23$. The algorithm returns the mean of the molecule structures covered by each ball in the solution as the estimate of the expectation, and the number of molecule structures covered by the ball. We choose the mean since it is a natural estimate of the expectation.
3. **RA** was applied with $c = 3, 5, 7, \ldots, 23$. For each value of c **RA** was run 10000 times and the best solution was chosen. The algorithm returns the mean of the molecule structures covered by each ball in the solution as the estimate of the expectation, and the number of molecule structures covered by the ball.

Remark 1

1. Although c is part of the input, we ran the algorithms on all possible values of c in order to check the sensitivity of the algorithms to the value of c.
2. The running time of all algorithms was a couple of minutes.

The data was simulated in the following way:

1. Randomly choose $|V| = 4$ points in the cube $[0, 10]^3 \subset I\!R^3$. The points represent the nodes of a molecule structure.[5]
2. Compute the vector $m = (m_1, \ldots, m_6)$ of $\binom{|V|}{2} = \binom{4}{2} = 6$ distances between the $|V| = 4$ points.
3. For every $i = 1, \ldots, 6$, choose a distance from a normal distribution with mean m_i and a known constant variance.

Steps 1−2 create a molecule structure and represent it as a vector of the distances between its nodes. Step 3 generates a sample from a distribution with mean value created by Steps $1 - 2$.

We present the results of the following cases:

- In a simulation of type a we simulate a CIM instance by generating 20 samples from each of 5 mean molecule structures. A total of 100 molecule structures. The set A of simulations includes 100 CIM instances of type a. In simulations of type a there is no noise, there are 5 clusters that we wish to identify.
- In a simulation of type b we simulate a CIM instance by generating 20 samples from each of 5 mean molecule structures. In addition we generated noise by generating 27 molecule structures from which we generated one sample, 20 molecule structures from which we generated two samples, and 6 molecule structures from which we generated three samples. Type b simulations include 5 clusters that we wish to identify and another 85 points that are considered as noise. The set B of simulations includes 100 cases of type b.

For each of the three algorithms, in each set A, B we calculated the following measures:

$ABN :=$ the average number of balls (distributions) used in the solution.
$MSE :=$ the average distance between a center of a cluster defined by the algorithm and the closest mean value. The closest mean value, is the closest vector m used in Step 2 of the data simulation process.
$GMSE :=$ the average distance between a mean value (only of the 5 clusters we wish to identify, i.e. with number of samples $> c$) and the closest center of a cluster defined by the algorithm.

MSE associates for each cluster representative, the nearest mean value, so some mean values may not be associated with any cluster, while $GMSE$ associates each mean value with the nearest cluster representative, so if a representative is not the closest representative to any mean value, it would not be considered.

The results are presented in Table (1). In general **RA** and **MEM** give accurate estimations of the mean values, while **GR** has slightly less accurate results.

[5] We applied this simulation to molecule structures of all sizes between $|V| = 4$ and $|V| = 10$ nodes, and the results were similar. We therefore present the results of molecule structures of size 4 as representative results.

Table 1. results for the set A and B

		Set A			**Set** B	
c	ABN	MSE	GMSE	ABN	MSE	GMSE
		MEM			**MEM**	
3	5.313	0.981	1.193	10.029	3.534	1.736
5	5.059	0.983	1.195	8.926	3.400	1.736
7	4.876	0.950	1.164	7.925	3.192	1.721
9	4.692	0.964	1.189	7.000	2.837	1.704
11	4.673	0.966	1.227	6.000	2.461	1.708
13	4.681	0.960	1.216	5.442	2.086	1.729
15	4.562	0.986	1.365	5.271	1.987	1.746
17	4.428	1.011	1.507	5.087	1.823	1.717
19	3.000	1.534	1.598	4.938	1.725	1.708
21	1.114	6.322	30.736	3.454	2.492	12.724
23	1.107	6.514	34.363	2.614	2.722	19.243
		GR			**GR**	
3	5.360	1.453	1.405	5.680	1.441	1.382
5	5.223	1.448	1.426	5.286	1.416	1.398
7	5.1322	1.446	1.428	5.202	1.411	1.402
9	5.031	1.442	1.478	5.052	1.400	1.486
11	4.950	1.434	2.088	4.940	1.392	2.349
13	4.859	1.431	2.502	4.829	1.388	3.117
15	4.708	1.425	3.775	4.748	1.376	3.790
17	4.507	1.407	5.577	4.627	1.370	5.068
19	3.755	1.411	10.918	4.938	1.725	1.708
21	2.251	1.176	37.567	1.622	1.523	48.630
23	1.750	1.097	46.450	2.622	2.255	106.512
		RA			**RA**	
3	6.070	0.857	0.813	6.540	0.852	0.794
5	5.340	0.855	0.844	5.465	0.836	0.818
7	5.153	0.850	0.846	5.224	0.832	0.822
9	5.041	0.847	0.879	5.072	0.817	0.896
11	4.950	0.831	1.483	4.950	0.810	1.775
13	4.864	0.833	1.958	4.837	0.798	2.527
15	4.708	0.822	3.187	4.748	0.791	3.195
17	4.507	0.801	4.956	4.627	0.785	4.474
19	3.755	0.788	10.332	3.736	0.752	10.959
21	2.251	1.064	38.414	1.622	0.872	46.169
23	1.750	1.132	47.785	2.622	1.259	101.040

The types of errors in estimation presented by the algorithms are the following:

1. Joining close clusters, i.e. returning only one expectation instead of two close expectations. This error is common with **MEM** [19] and is expected from **GR**. It was less common with **RA**. Such an error is reflected by the number of clusters estimated, and by the value of **GMSE** in Table 1 set A.

2. Cutting a cluster in two. This error is mainly characteristic to **RA**. If c is small enough then one cluster may be split into several profitable balls, so that one cluster is estimated by several close expectations.
3. Omitting a cluster. If c is large then it may happen that there is no ball of radius r containing $c+1$ points, although there is a mean value which generated more then $c+1$ points. For example for $c > 15$ in **GR** and **RA**, the $GMSE$ is much greater than the MSE because the clusters found are fairly accurate, but there are clusters omitted from the solution. This is less common with **MEM**.
4. **MEM** is sensitive to outliers, since the likelihood maximization includes the explaining of all data points, and the size of the model is not known. This explains the large MSE in Table 1 set B, that diminishes as c grows since less small clusters are included.
5. Since **MEM** is sensitive to outliers it clusters many of these points for small values of c, while **GR** and **RA** hardly cluster outliers.

6 Theoretical Analysis of MPCP

In this section we present a PTAS for MPCP in the Euclidean space. We adapt the shifting method of Hochbaum and Maass [11] for MPCP. We introduce the method in the plane, which could be generalized to a fixed dimension d as in [11].

Let the set S of n given points in the plane be enclosed in a region I.

The goal is to cover a subset of S with disks of diameter D such that the profit is maximized. Let the shifting parameter be l. In the first phase the area I is subdivided into vertical strips of width D, where each strip is left closed and right open. Groups of l consecutive strips, resulting in strips of width lD each, are considered. For any fixed subdivision of I into strips of width D, there are l different ways of partitioning I into strips of width lD. These partitions can be ordered such that each can be derived from the previous one by shifting it to the right over distance D. Repeating the shift l times we return to the initial partition. We denote the l distinct shift partitions by $P_1, P_2, \ldots, P_l$. Let A be any algorithm that delivers a solution of width lD in any strip of width lD (or less). For a given partition P_i let $A(P_i)$ be the algorithm that applies algorithm A to each strip in the partition P_i and outputs the union of all disks used. This process is repeated for each partition P_i, $i = 1, 2, \ldots, l$. The shifting algorithm A_P, defined for a given local algorithm A, delivers the set of disks of maximum profit among the l sets delivered by $A(P_1), A(P_2), \ldots, A(P_l)$. We begin by introducing two properties of the problem.

Property 1: Denote by $opt(S)$ the value of the optimal solution for the set S. If $S' \subset S$, $opt(S') \leq opt(S)$.

Property 2: Let OPT be the set of unit balls in the optimal solution for the set S, and $OPT' = OPT \setminus \{B\}$ where $B \in OPT$. Let $S \setminus S'$ be the subset of points which B covers uniquely then OPT' is optimal for S'.

Let the performance ratio of an algorithm A be denoted by r_A.

Lemma 1. *$r_{S_A} \leq r_A(1 - \frac{1}{l})$, where A is a local algorithm and l is the shifting parameter.*

Given a coverage of S, an equivalent coverage using unit balls could be defined by shifting the unit balls such that a maximum number of the points they cover will be on their sphere, and the set of points covered by each unit ball will not change. Denote the set of balls with a maximum number of the points they cover will be on their sphere by B_S. In the following we consider B_S as the set of possible unit balls in any coverage of S.

Remark 2. If the set of balls $\mathcal{B}$ is a given set, as in the general definition of $MPCP$, then it can be shown using the bound on the volume of the cube, that the following local algorithm still holds.

THE MAXIMUM PROFIT COVERAGE PROBLEM could be solved in a small cube in R^d. Let A denote a cube of volume $(2l)^d$ and $S' \subseteq S$ denote the set of points in it. The set of unit balls $B_{S'}$ that could be defined by S' is bounded by $2(|S'|_d)$ as demonstrated in Section 3. Since A could be covered by $\sqrt{2}l^d$ balls, checking all subsets of at most $\sqrt{2}l^d$ balls yields the optimal solution. The complexity is $O(|S'|^{d\sqrt{2}l^d})$ time.

References

1. V. Barnett and T. Lewis, "Outliers in statistical data" *Wiley* 1984.
2. M. Charikar, S. Khuller, D.M. Mount and G. Narasimhan, "Algorithms for facility location problems with outliers", *SODA*,2001,642-651.
3. R.N. Dave and R. Krishnapuram, "Robust Clustering Methods: A Unified View", *IEEE Transactions on Fuzzy Systems* **5** 1997, 270-293.
4. D.-Z. Du and P.M. Paradalos, "Handbook of Combinatorial Optimization" *Kluwer Academic Publishers* 1998, 261-329.
5. M. Ester, H. Kreigel, J. Sander and X. Xu, "A density based Algorithm for Discovering Clusters in Large Spatial Databases with Noise", *KDD-96* 1996, 226-231.
6. M. Ester, Hans-Peter Kriegel and Xiaowei Xu "A Database Interface for Clustering in Large Spatial Databases", *KDD-95* (1995).
7. R.J Fowler, M. S. Paterson and S. L. Tanimoto, "Optimal packing and covering in the plane are NP-complete," *Information Processing Letters* **12** (1981), 290-308.
8. T. F. Gonzalez, "Covering a set of points in multidimensional space," *Information Processing Letters* **40** (1991), 181-188.
9. S. Guha, R. Rastogi and K. Shim, "CURE: A Efficient Clustering Algorithm for large Databases", *Proc. of the ACM SIGMOND Conference on Management of Data* (1998).
10. J.W.Hanand M.Kamber, "Data Mining: Concepts And Techniques", *San Francisco: Morgan Kaufmann Publishers*, (2001).
11. D.S.Hochbaum and W.Maass, "Approximation schemes for covering and packing problems in image processing and VLSI," *Journal of ACM*, **32** (1985), 130-136.
12. M.F. Jiang, S.S Tseng and C.M. Su, "Two-phase clustering process for outliers detection", *Pattern Recognition Letters* **22** (2001), 691-700.
13. S. Khuller, A. Moss and J. Naor, "The budgeted maximum coverage problem," *Information Processing Letters* **70** (1999), 290-308.

14. R. Nag and J. Han , "Efficient and Effective Clustering Methods for Spatial Data Mining", *Proceedings of the 20th VLDB conference* (1994) 145-155.
15. G. J. McLachlan and T. Krishnan, "The EM Algorithm and Extensions", *Wiley-Interscience* (1996) .
16. C. F. Olson "Parallel Algorithms for Hierarchical Clustering", Technical report, Computer Science Division, Univ. of California at Berkley, (1993).
17. Y. Pawitan, "In All Likelihood: Statistical Modelling and Inference Using Likelihood",*Oxford University Press* (2000).
18. S. Richardson and P.J. Green, "On Bayesian Analysis of mixtures with an Unknown number of components," *J. R. Stat. Soc. B* **59** (1997), 731-792.
19. R.A. Redner and H.F. Walker, "Mixture densities, maximum likelihood and the EM algorithm," *SIAM Review* **26** 1984, 195-239.
20. G. Schwarz, "Estimating the dimension of a model," *The Annals of Statistics* **6** (1978), 461-464.
21. D. Spielman and S-H Teng, "Spectral partitioning works: planar graphs and finite element meshes," *Proc. of 37th FOCS* (1996), 96-105.
22. J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **22** (2000), 888-905.
23. L. Xu and M. Jordan, "On the convergence properties of the EM Algorithm for Gaussian Mixtures", *Neural Computation* **8** (1996), 129-151.
24. X. Zhuang, Yan Huang, K. Palaniappan and Yunxin Zhao "Gaussian Mixture Density Modelling, and Applications", *IEEE Transactions on Image Processing* **5** (1996), 1293-1301.
25. J. Zhang and Y. Leung, "Robust Clustering by Pruning Outliers", *IEEE Transactions on Systems, Man and Cybernetics-Part B: Cybernetics* **33** (2003), 983-998.
26. T. Zhang, R. Ramakrishnan and M. Livny, "BRITCH: An Efficient Data Clustering Method for Very Large Databases", *Proc. of the ACM SIGMOND Conference on Management of Data* (1996), 103-114.

Algorithmic Challenges in Web Search Engines

Ricardo Baeza-Yates

Yahoo! Research,
Barcelona, Spain & Santiago, Chile
ricardo@baeza.cl

Abstract. We present the main algorithmic challenges that large Web search engines face today. These challenges are present in all the modules of a Web retrieval system, ranging from the gathering of the data to be indexed (crawling) to the selection and ordering of the answers to a query (searching and ranking). Most of the challenges are ultimately related to the quality of the answer or the efficiency in obtaining it, although some are relevant even to the existence of current search engines: context based advertising.

As the Web grows and changes at a fast pace, the algorithms behind these challenges must rely in large scale experimentation, both in data volume and computation time, to understand the main issues that affect them. We show examples of our own research and of the state of the art. The full version of this paper appears in [1].

References

1. Ricardo Baeza-Yates. Algorithmic Challenges in Web Search Engines. In *LATIN 2006*, Jos R. Correa, Alejandro Hevia, Marcos A. Kiwi (Eds.), Valdivia, Chile. Lecture Notes in Computer Science 3887, 2006, 1-7.
2. R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, Addison-Wesley, England, 513 pages, 1999.
3. R. Baeza-Yates. Information Retrieval in the Web: beyond current search engines, Int. Journal of Approximate Reasoning 34 (2-3), 97-104, 2003.
4. R. Baeza-Yates, C. Castillo, M. Marin, A. Rodriguez. Crawling a Country: Better Strategies than Breadth-First for Page Ordering. In *WWW 2005, Industrial Track*, ACM Press, Chiba, Japan, May 2005.
5. Ricardo A. Baeza-Yates, Carlos A. Hurtado, Marcelo Mendoza. Query Recommendation Using Query Logs in Search Engines, Current Trends in Database Technology - EDBT 2004 Workshops, Workshop on Clustering Information over the Web, Heraklion, Crete, Greece, March 14-18, 2004, Revised Selected Papers. LNCS 3268, Springer, p. 588-596.
6. R. Baeza-Yates, A Fast Set Intersection Algorithm for Sorted Sequences, In *15th Combinatorial Pattern Matching 2004*, LNCS, Springer, Istanbul, Turkey, July 2004.
7. Ricardo Baeza-Yates. Applications of Web Query Mining. In *European Conference on Information Retrieval (ECIR'05)*, D. Losada, J. Fernández-Luna (editors), Springer LNCS 3408, Santiago de Compostela, Spain, March 2005, 7–22.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 277–278, 2006.

8. Ricardo Baeza-Yates, Barbara Poblete, A Website Mining Model Centered on User Queries, European Web Mining Forum, B. Berendt *et al*, editors. October 2005, Oporto, Portugal, p. 3-15.
9. R. Baeza-Yates, A. Pereira and N. Ziviani, WIM: A Web Information Mining Model for the Web, In *LA-WEB 2005*, IEEE CS Press, Oct 2005, 233-241.
10. H. K. Bhargava and J. Feng. Paid placement strategies for internet search engines. In Proceedings of the eleventh international conference on World Wide Web, pages 117-123. ACM Press, 2002.
11. S. Chakrabarti. Mining the Web: Discovering knowledge from hypertext data, Morgan Kaufmann, 2003.
12. B. Davison, editor. Workshop on Adversarial Information Retrieval on the Web. URL: http://airweb.cse.lehigh.edu/2005/. Chiba, Japan, May 2005.
13. J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999. Preliminary version presented at SODA 1998.
14. J. Kleinberg and P. Raghavan. Query Incentive Networks. Proc. 46th IEEE Symposium on Foundations of Computer Science, 2005.
15. M. Koster. A standard for robot exclusion. http://www.robotstxt.org/wc/exclusion.html, 1996.
16. V. Makinen and G. Navarro. Compressed Full Text Indexes. Technical Report TR/DCC-2005-7, Dept. of Computer Science, University of Chile, June 2005. Available at http://pizzachili.dcc.uchile.cl/biblio.html
17. S. Nicholson, T. Sierra, U.Y. Eseryel, J.H. Park, P. Barkow, E.J. Pozo, J. Ward. How Much of It is Real? Analysis of Paid Placement in Web Search Engine Results, JASIST, 2005.
18. L. Page, S. Brin, R. Motwani, and T. Winograd. The Pagerank citation algorithm: bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.
19. B. Ribeiro-Neto, M. Cristo, P. Golgher, and E. Silva de Moura. Impedance coupling in content-targeted advertising. In Proceedings of the 28th Annual international ACM SIGIR Conference on Research and Development in information Retrieval (Salvador, Brazil, August 15 - 19, 2005). SIGIR '05. ACM Press, New York, NY, 496-503, 2005.
20. Barry Wellman. Computer Networks As Social Networks, *Science* 293 (5537), p. 2031-2034, 2001.
21. A.C-C. Yao, editor. First Workshop on Internet and Networks Economy, WINE 2005. URL: http://www.cs.cityu.edu.hk/ wine2005/, Hong-Kong, 2005.

On the Least Cost for Proximity Searching in Metric Spaces*

Karina Figueroa[1,2], Edgar Chávez[1], Gonzalo Navarro[2], and Rodrigo Paredes[2]

[1] Universidad Michoacana, México
{karina, elchavez}@fismat.umich.mx
[2] Center for Web Research, Dept. of Computer Science, Universidad de Chile
{gnavarro, raparede}@dcc.uchile.cl

Abstract. Proximity searching consists in retrieving from a database those elements that are similar to a query. As the distance is usually expensive to compute, the goal is to use as few distance computations as possible to satisfy queries. Indexes use precomputed distances among database elements to speed up queries. As such, a baseline is AESA, which stores all the distances among database objects, but has been unbeaten in query performance for 20 years. In this paper we show that it is possible to improve upon AESA by using a radically different method to select promising database elements to compare against the query. Our experiments show improvements of up to 75% in document databases. We also explore the usage of our method as a probabilistic algorithm that may lose relevant answers. On a database of faces where any exact algorithm must examine virtually all elements, our probabilistic version obtains 85% of the correct answers by scanning only 10% of the database.

1 Introduction

Proximity or similarity searching is nowadays an essential tool in a number of practical tasks such as vector quantization of signals, pattern recognition, retrieval of multimedia information, etc. In these applications there is a database (for example, a set of documents) and a similarity measure among its objects (for example, the cosine distance). The similarity is modeled by a distance function defined by experts in each application domain, which tells how similar two objects are. The distance function is normally considered quite expensive to compute, so that even I/O operations or the CPU cost of side computations are not considered. That is, the search complexity is taken as just the number of distance evaluations needed to answer a query, and thus the goal is to answer the queries by performing the minimum number of distance evaluations.

To reduce the query cost, an index is built on the database before searching it. The index is a data structure that stores information on some distances among database elements. This information is used later to discard some elements without comparing them directly with the query.

* Supported by CONACyT (Mexico) and Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, (Chile).

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 279–290, 2006.

Different indexes store different information on the database distances [7]. Some store a subset of the distances, e.g. all the distances between k chosen pivots and all the rest, or all the distances between an element and its subtree, in a tree-structured index. Some store just a range of distance values, and so on. In general, the more information an index stores, the lower query cost it achieves (although some use memory better than others). In this view, in a database of n objects the most information an index could store is the $n(n-1)/2$ distances among all element pairs. This is usually avoided because it requires $O(n^2)$ space, but it is applicable in some areas such as pattern recognition, as well as to index database subsets. In particular, using all the available information establishes a *baseline* on how good an index could be. Actually, all the development on metric space indexing can be regarded as the quest for maintaining good efficiency while reducing the amount of information stored [7].

The canonical algorithm that uses all the data is AESA [17]. For 20 years AESA has been the indexing technique requiring, by far, the least number of distance computations among all other indexes (which require much less space).

In this paper we show, for the first time, that it is possible to establish a new baseline on the number of distance evaluations for proximity searching. More specifically, AESA works by choosing a "pivot" from the remaining set of candidates and using it to prune more candidates. The closer the pivot to the query, the more effective the pruning is. We introduce a new technique called *iAESA* to choose the next pivot, which guesses better a close candidate and yields reductions in the number of distance evaluations of up to 75% in document databases.

In very high dimensions, even AESA and iAESA boil down to a sequential database scan. We explore the usage of iAESA as a probabilistic scheme that may lose some relevant answers, but could quickly find most of them. We show that, for example, on a database of face images where no exact algorithm can obtain any significant savings over a sequential scan, iAESA retrieves 85% of the correct answers by scanning just 10% of the database. This is 80% less than what would be needed to obtain the same result with probabilistic AESA.

2 Related Work

2.1 Notation and Basic Concepts

Let $(\mathbb{X}, d)$ be a metric space, where $\mathbb{X}$ is the universe of objects and d the distance function among the objects in $\mathbb{X}$. The distance function $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$ is defined by experts in the application domain and expresses the dissimilarity between objects in $\mathbb{X}$. The distance function must satisfy the following properties: strict positiveness ($d(x, y) > 0 \iff x \neq y$), symmetry ($d(x, y) = d(y, x)$) and triangle inequality ($d(x, z) \leq d(x, y) + d(y, z)$).

Let $\mathbb{U} \subseteq \mathbb{X}$ be our database of size n, $q \in \mathbb{X}$ the query, and $r \geq 0$. The similarity queries can be classified into two basic types:

- Range query, $(q, r)_d = \{u \in \mathbb{U} \mid d(u, q) \leq r\}$
- k-nearest neighbor query, $k\text{NN}(q)_d = A$ such that $\forall u \in A, v \in \mathbb{U} - A$, $d(u, q) \leq d(v, q)$, and $|A| = k$.

The naive approach to these kind of queries is to compare the whole database against the query. This solution, however, requires n distance computations. An index is a data structure on $\mathbb{U}$ that solves queries of either type trying to use less than n distance evaluations. As the objects are black-boxes, the search always proceeds by comparing q against some element of $\mathbb{U}$, discarding candidates using that distance and the help of the index, and so on until every element is either discarded or reported.

The performance of the algorithms in metric spaces is affected by the intrinsic dimension of data [7]. When the dimension grows, the mean of a random distance increases and the variance diminishes. In high dimensions, there are no algorithms that can avoid sequential scan. AESA is also affected by dimension in spite of being the best proximity search algorithm in metric spaces.

2.2 AESA

The Approximating and Eliminating Search Algorithm (AESA) was introduced by E. Vidal in 1986 [17]. AESA needs to compute and store a matrix as an index, recording every distance $d(u, v)$, $\forall u, v \in \mathbb{U}$, that is $O(n^2)$ distances. During the search process, an element from the remaining candidates, called a "pivot", is chosen and compared against the query. AESA uses the matrix of distances to discard remaining candidates using the triangle inequality. The algorithm is described in Section 2.3.

Although $O(n^2)$ space can be a large amount of memory, there are applications with small enough databases (up to few thousand objects) where managing all the $O(n^2)$ distances is possible. For this kind of applications, AESA is still a practical solution and the one performing least distance computations.

In the case of larger databases, where $O(n^2)$ distances cannot be stored, it is still possible to partition the database with another technique and apply AESA on each partition [11].

AESA has been for 20 years the algorithm that computes the least number of distance evaluations to answer proximity queries. There have been some algorithms aimed at reducing its preprocessing time or space used. LAESA [13] chooses k elements of $\mathbb{U}$ as potential pivots, then reducing the space to $O(kn)$. An improved version of LAESA is Tree LAESA (TLAESA) [12] which achieves sublinear side computations at query time at the expense of doubling the number of distance computations on average. Reduced Overhead AESA (ROAESA) [18] strictly calculates the same distances as AESA but reduces the query processing time. Recently, graph t-spanner indexes [15] were used to simulate AESA, obtaining almost the same number of distance calculations and using much less memory. In fact, all the development on indexes for metric spaces can be seen as attempts to simulate the performance of AESA using less memory [7].

2.3 Searching Using AESA

Like most indexing algorithms, AESA solves nearest neighbor queries by choosing a *pivot* $u \in \mathbb{U}$ to compare against q, then filtering out as many candidates of $\mathbb{U}$ as possible, and repeating until all candidates are compared or discarded. AESA proposes a specific method to select the pivots: The next pivot to compare against q is chosen as the candidate u minimizing

$$D(u) \;=\; \sum_{p \in |\mathbb{P}|} |d(u,p) - d(p,q)|, \tag{1}$$

where $\mathbb{P}$ are those pivots already compared against q (thus the $d(p,q)$ distances are known, whereas the $d(u,p)$ distances are stored in the matrix). The goal of minimizing $D(u)$ is to find a pivot as close as possible to q. The algorithm to answer a closest neighbor query 1-NN$(q)_d$ is summarized in five steps.

AESA

```
1.  Let ℙ ← ∅ set of pivots
2.  Let 𝔽 ← ∅ set of filtered elements
3.  r ← ∞
4.  For u ∈ 𝕌, D(u) ← 0,  D_max_u ← 0
5.  while 𝕌 ≠ ℙ ∪ 𝔽 do
6.    p ← argmin_{u∈𝕌−ℙ−𝔽} D(u)
7.    ℙ ← ℙ ∪ {p}
8.    if d(q,p) < r then
9.       r ← d(p,q)
10.      p* ← p
11.   for u ∈ 𝕌 − ℙ − 𝔽 do
12.     D_max_u ← max(D_max_u, |d(q,p) − d(u,p)|)
13.    if  D_max_u > r then
14.      𝔽 ← 𝔽 ∪ {u}
15.    else
16.      D(u) ← D(u) + |d(q,p) − d(u,p)|
17. return p*
```

iAESA

```
1.  Let ℙ ← ∅ set of pivots
2.  Let 𝔽 ← ∅ set of filtered elements
3.  r ← ∞,  Π_q ←<>
4.  For u ∈ 𝕌, F(u) ← 0, Π_u ←<>
5.  For u ∈ 𝕌, D_max_u ← 0
6.  while 𝕌 ≠ ℙ ∪ 𝔽 do
7.    p ← argmin_{u∈𝕌−ℙ−𝔽} F(u)
8.    ℙ ← ℙ ∪ {p}
9.    insert p in Π_q
10.   if d(q,p) < r then
11.      r ← d(p,q)
12.      p* ← p
13.   for u ∈ 𝕌 − ℙ − 𝔽 do
14.     D_max_u ← max(D_max_u, |d(q,p) − d(u,p)|)
15.    if  D_max_u > r then
16.      𝔽 ← 𝔽 ∪ {u}
17.    else
18.      insert p in Π_u
19.      F(u) ← F(Π_q, Π_u)
20. return p*
```

Fig. 1. AESA and iAESA algorithms to retrieve the nearest neighbor (iAESA is described in Section 3)

1. **Initialization.** The sets of pivots $\mathbb{P}$ and filtered elements $\mathbb{F}$ are empty. Let $D(u) \leftarrow 0$ for u, $D_{max_u} \leftarrow 0$ and $r \leftarrow \infty$. Steps 2-5 are repeated until $\mathbb{U} = \mathbb{P} \cup \mathbb{F}$.
2. **Approximating.** In this step a new pivot p is selected according to Equation (1). That is $p \leftarrow \text{argmin}_{u \in \mathbb{U}-\mathbb{F}-\mathbb{P}}\ D(u)$.

3. **Distance computation.** Element p is compared against the query q by computing $d(p, q)$. The new p will be added to the set of used pivots $\mathbb{P}$.
4. **Updating the NN.** If $d(q, p) < r$, the current nearest neighbor and r are updated. Every object in $\mathbb{U} - \mathbb{F} - \mathbb{P}$ updates its approximation criterion according to Equation (1), that is $D(u) \leftarrow D(u) + |d(u, p) - d(p, q)|$ and $D_{max_u} \leftarrow \max(D_{max_u}, |d(u, p) - d(p, q)|)$.
5. **Eliminating.** Those $u \in \mathbb{U} - \mathbb{F} - \mathbb{P}$ such that $D_{max_u} > r$ are discarded using the triangle inequality. The elements filtered in this step are added to $\mathbb{F}$. The process continues at step 2.

The nearest neighbor query process of AESA is presented in Fig. 1 (left). Range query process $(q, r)_d$ can be implemented similarly by keeping r fixed and reporting every p that $d(p, q) \leq r$.

These algorithms generalize to k-NN queries, where $k > 1$, by maintaining a pool with the k closest elements p^* found until now, so that r is the distance to the current k-th nearest neighbor.

2.4 Proximity Preserving Order

We introduce some terminology needed to explain our technique [5].

Let $\mathbb{P} \subseteq \mathbb{U}$. Every element $u \in \mathbb{U}$ defines a *preorder* $\leq_u$ in $\mathbb{P}$ given by the distance to u. It is defined for $y, z \in \mathbb{P}$, as $y \leq_u z \Leftrightarrow d(u, y) \leq d(u, z)$. The relation $\leq_u$ is a preorder and not an order because some elements can be at the same distance of u, and then $\exists y \neq z$ such that $y \leq_u z \ \wedge \ z \leq_u y$.

Every object u can compute its preorder of $\mathbb{P}$ and associate it to a permutation, because the preorder induces a total order in the quotient set. Let us define $\Pi_u = p_1, p_2, \ldots, p_{|\mathbb{P}|}$ where $p_i \leq_u p_{i+1}$ the permutation of u. The elements at the same distance take an arbitrary but consistent order. We use $\Pi_u^{-1}(p_i)$ to identify the position of element p_i in the permutation Π_u.

It is important to notice that two equal elements must have the same permutation, while two similar objects will hopefully have a similar permutation. So if Π_u is similar to Π_q we expect u to be close to q.

Similarity between the permutations of q and u can be measured by Kendall Tau, Spearman Rho, or Spearman Footrule metric [10], among others. As all of these have a comparable predictive power [5], we choose Spearman Footrule because it is not expensive to compute. This measure is defined as follows:

$$F(u) = F(\Pi_u, \Pi_q) = \sum_{i=1}^{|\mathbb{P}|} |\Pi_u^{-1}(p_i) - \Pi_q^{-1}(p_i)|, \tag{2}$$

where $\mathbb{P}$ is the current set of pivots. For example, let $\Pi_q = p_1, p_2, p_3, p_4, p_5$ be the permutation of the query, and $\Pi_u = p_3, p_2, p_1, p_5, p_4$ the permutation of an element $u \in \mathbb{U}$. According to Equation (2), we have $F(\Pi_q, \Pi_u) = |1 - 3| + |2 - 2| + |3 - 1| + |4 - 5| + |5 - 4| = 6$.

3 Our Proposal: iAESA

From Section 2.2, we notice that the only way to improve the performance of AESA seems to be by modifying the approximation criterion, that is, by proposing a different method to select the next pivot.

We propose to select as the new pivot the element whose permutation is the most similar to the permutation of the query. We describe this process next.

3.1 Searching Using iAESA

The algorithm consists basically in modifying the approximation criterion, which will be replaced by the similarity between permutations. The permutations will be formed by the pivots already used.

Instead of $D(u)$, we will use $F(u)$, which is initialized at 0 and updated upon choosing a new pivot p according to Equation (2). The new pivot will be the one with smallest $F(u)$. Fig. 1 (right) gives the algorithm.

3.2 Comparing AESA with iAESA

In Fig. 2 we compare iAESA with AESA, using the same example shown in [17]. The example retrieves the nearest neighbor. In the figure (left side) the objects are $p_1, \ldots, p_7$ and q is the query; the solid lines are the terms of Equation (1); the dashed lines indicate the process to select the next pivot (labeled step-1, step-2 and step-3); the circle and the semicircles are the distances from a pivot to the query, and the semicircles are labeled by the order of execution, step-1, step-2 and step-3. They help viewing which elements are to be chosen given the approximation criterion.

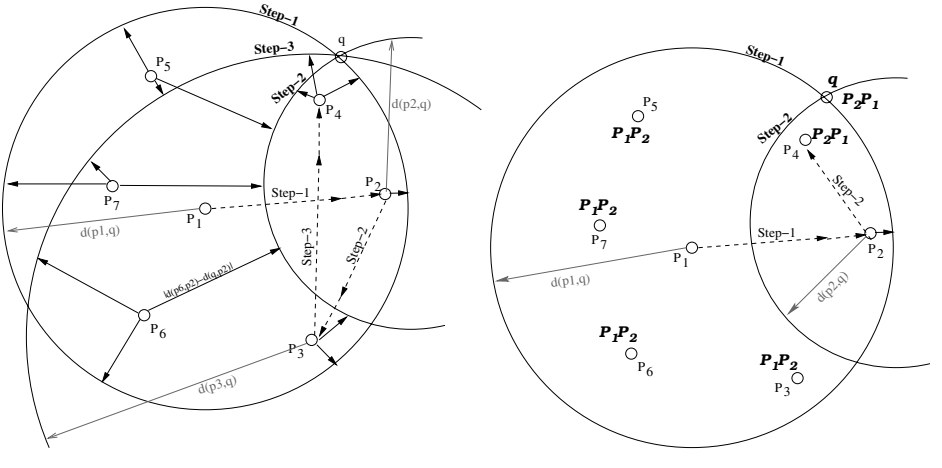


Fig. 2. On the left, the example shown in [17] to explain AESA. On the right, iAESA process for the same set of elements. The order of selection is step-1, step-2 and step-3. Note that iAESA uses one pivot less than AESA in this example.

AESA initially selected p_1. The next pivot was p_2 because it minimizes Equation (1) (step-1). The next pivot is p_3 (step-2) and finally p_4 (step-4) according to the approximation criterion.

On the other hand, in the process of iAESA, p_1 and p_2 are selected in the same way as AESA. We have drawn the permutation of the elements at this point. Note that p_4 has the same permutation as $\Pi_q = \{p_2, p_1\}$, therefore p_4 is the next (and last) pivot. In this example iAESA uses the pivots (p_1, p_2, p_4), one less than AESA.

The CPU time complexity of AESA is $O(|\mathbb{P}| \cdot n)$, as $|\mathbb{P}|$ is the number of iterations over the elements not yet discarded and $D(u)$ is updated in constant time. iAESA complexity is higher because we need $O(|\mathbb{P}|)$ time to update Π_u and $F(u)$. This yields a total complexity of $O(|\mathbb{P}|^2 \cdot n)$.

3.3 Combining AESA and iAESA

AESA and iAESA criteria can be combined into an algorithm that we call *iAESA2*. The idea is to modify the approximation criterion of iAESA (i.e., the similarity between permutations) using AESA approximation criterion $D(u)$ to break ties in $F(u)$. These ties are common when there are few pivots. The CPU time complexity of iAESA2 is also $O(|\mathbb{P}|^2 \cdot n)$.

In other words, iAESA2 uses two approximation criteria: a primary one given by the least value of Spearman Footrule metric (i.e. the most similar permutation) and a secondary one given by the smallest $D(u)$.

4 Probabilistic iAESA

A serious problem of all algorithms in metric spaces, even for AESA, is that when the dimension of the space grows [7], the whole database needs to be reviewed. In this case a probabilistic algorithm (which can miss some relevant answers) is a practical tool. Any exact algorithm can be turned into probabilistic, by letting it work until some predefined work threshold and measuring how many relevant answers did it find.

Probabilistic algorithms have been proposed both for vector spaces [1, 19] and for general metric spaces [9, 8, 6, 4]. In [4] they use a technique to obtain probabilistic algorithms that is relevant to this work. They use different techniques to sort the database according to some *promise value*. As they traverse the database in such order, they obtain more and more relevant answers to the query. A good database ordering is one that obtains most of the relevant answers by traversing a small fraction of the database. In other words, given a limited amount of work, the algorithm finds each correct answer with some probability, and it can refine the answer incrementally if more work is allowed. Thus, the problem of finding good probabilistic search algorithms translates into finding a good ordering of the database given a query q.

Under this model, a probabilistic version of k-NN AESA, iAESA and iAESA2 consists in reviewing objects up to some fraction of the database and reporting

the k closest object found until then. In Fig. 1, we should replace the **while** condition (line 5) by **while** $|\mathbb{P}| < percentage\_of\_database$. For range queries we would simply report any relevant element found until the scanning is stopped.

5 Experimental Results

We conducted experiments on different synthetic and real-life metric databases. The real-life metric spaces are TREC-3 documents under cosine distance [2], and a database of feature vectors of face images under Euclidean distance [14]. The synthetic metric spaces are random vectors in the unitary cube.

5.1 Exact iAESA: Unitary Cube

The performance of the existing algorithms, to answer both range and k-nearest neighbor queries, worsens as the dimension of the space grows [3]. Therefore, it is interesting to experiment with spaces with different dimensions.

A way to control the dimension of the space is to generate synthetic sets uniformly distributed in the unitary cube, and use this set as an abstract metric space. We experimented with 4 to 14 dimensions, for databases of size from 5,000 to 20,000 elements. The performance of our technique can be seen in Fig. 3. Notice that, as we increase the dimension of the data, the problem becomes more difficult. Nevertheless, iAESA retains its (slight) advantage over AESA when the dimension grows. In the best case, iAESA requires 17% less distance evaluations than AESA. iAESA2 had the same performance as iAESA, so we omitted it in this experiment. On the other hand, we note that iAESA loses its advantage over AESA as the number k of nearest neighbors sought grows. For example, in dimension 14 AESA takes over for $k > 5$.

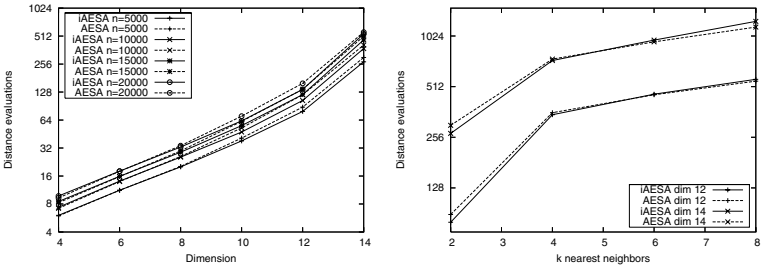


Fig. 3. Performance of our technique against AESA for different dimensions and $k = 2$ (left side). On the right, we retrieve different numbers k of nearest neighbors, on dimensions 12 and 14 and $n = 5,000$. Note the logscale.

5.2 Exact iAESA: Documents

A set of 1265 English documents obtained from the Wall Street Journal 87-89 collection from TREC-3 was indexed. We compare the documents under the vector space, using the cosine distance [2].

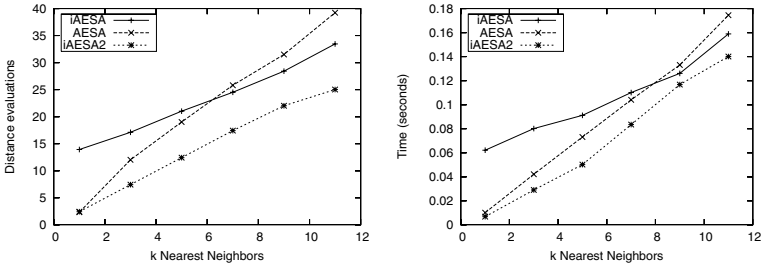


Fig. 4. Comparing the performance of our technique against AESA on a document database (1265 documents)

Fig. 4 shows the results on this space. This time iAESA improves upon AESA as the number k of nearest neighbors retrieved grows over 8. On the left we show distance computations. It can be seen that iAESA2 is clearly better than both AESA and iAESA in all cases, improving upon AESA by up to 75%, when $k = 1$. Fig. 4 (right) displays overall CPU time. It can be seen that, even though iAESA and iAESA2 suffer from a higher number of side CPU computations, they are still preferable over AESA.

As a sanity check, we compared these results against choosing the next pivot at random. This turned out to make four times more evaluations than iAESA.

5.3 Probabilistic iAESA: Unitary Cube

We experimented with 3,000 synthetic random vectors in the unitary cube of 128 dimensions. Any exact algorithm is forced to compare every element in such a high-dimensional space. Fig. 5 (left) shows the percentage of successful queries when the number of objects compared against the query is limited. That is, we plot the percentage of queries that retrieved all their correct k nearest neighbors after scanning a fraction of the database. We can see that iAESA finds the k nearest neighbors faster than AESA, and that iAESA2 is the fastest for large

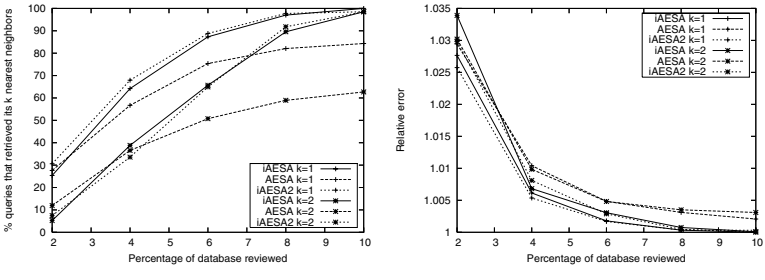


Fig. 5. Searching for k nearest neighbors on 3,000 random vectors in 128 dimensions. On the left, fraction of correct queries. On the right, distance approximation ratio for the unsuccessful queries.

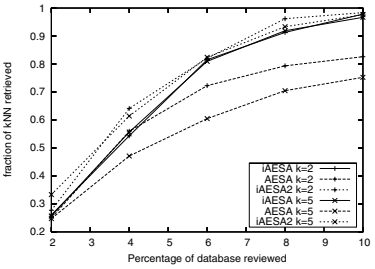


Fig. 6. Fraction of the answer retrieved as the scanning progresses. Random vectors on dimension 128 and $n = 3000$.

k. AESA, on the other hand, needs to compare almost 80% of the database in order to retrieve 95% of the answer.

On the right we show the ratio among the distance to the k-th NN found by the algorithm versus the distance to the true k-th NN. In this computation we exclude the queries where the algorithm finds all the k correct neighbors, that is, on queries considered unsuccessful on the left plot.

Note that we have defined a query as unsuccessful even if it finds $k-1$ out of the k correct elements. Fig. 6 plots the fraction of the correct nearest neighbors found as we scan the database. For example, for with iAESA2 we need to scan about 7% of the database to find 90% of the answers.

5.4 Probabilistic iAESA: Face Images

Many real databases are composed of few objects, each of very high intrinsic dimension. This is the case the FERET database of face images [16]. We use a target set with 762 images of 254 different classes (3 similar images per class), and a query set of 254 images (1 image per class). The intrinsic dimension of the

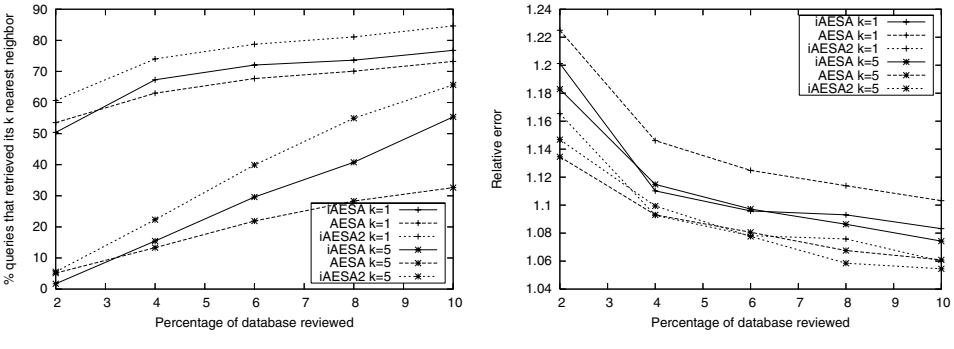


Fig. 7. Searching for the k nearest neighbors in a real-life image database of 762 faces. On the left, fraction of correct queries. On the right, ratio of distances to the k-th NN found versus the real k-th NN.

database is around 40, which is considered intractable: exact AESA and iAESA must scan 90% of the database in order to answer k-NN queries on this space.

The performance of the probabilistic algorithms is compared in Fig. 7. It can be seen that larger fractions of the database must be scanned in order to satisfactorily solve queries with larger k. Again, iAESA2 is faster than the others.

6 Conclusions and Future Work

Proximity searching in metric spaces consists in retrieving the elements from the database that are relevant to a given query. The similarity between objects is measured by a distance function that is usually expensive to compute. AESA [17] has been without question, for 20 years, the most successful algorithm to solve similarity queries, because it computes the least number of distance evaluations to answer them. We present a new technique, called iAESA, able to improve upon AESA by up to 75% over different metric databases.

In very high dimensions there are no exact algorithms able to avoid sequential scan. We propose a new probabilistic algorithm based on iAESA, which is able to solve a large fraction of queries by scanning a small fraction of the database. For example, on a faces image database, iAESA solves 85% of the queries by checking just 10% of the database.

The only weak point of our approach is the extra CPU time required to reduce the distance computations. This can be significant if the distances are not very expensive to compute. We plan to address this issue in two ways. One is to avoid, upon the insertion of a new pivot, the full recomputation of the permutation of each element as well as its distance to the query permutation. We are exploring a scheme that reduces this work by about 50%, and further reductions could be possible by using smarter data structures. Another idea is based on the fact that the distances to the query permutation can only grow as more pivots are inserted. Thus we can delay the updating of the permutation of every element until it would become the next pivot. At this point the pivot insertions delayed are carried out on the candidate's permutation and its distance to the query permutation is updated. This may push the candidate behind on the priority queue and forces us to choose the next best candidate, until we get an up-to-date next candidate. Thus many elements could be removed from the candidate set without ever having updated their permutations, thus saving CPU time.

References

1. S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimension. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 573–583, 1994.
2. R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley, 1999.
3. C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces-index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

4. B. Bustos and G. Navarro. Probabilistic proximity search algorithms based on compact partitions. *Journal of Discrete Algorithms (JDA)*, 2(1):115–134, 2003.
5. E. Chávez, K. Figueroa, and G. Navarro. Proximity searching in high dimensional spaces with a proximity preserving order. In *MICAI 2005: Advances in Artificial Intelligence*, volume 3789 of *LNCS*, pages 405–414, 2005.
6. E. Chávez and G. Navarro. Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Information Processing Letters*, 85(1):39–46, 2003.
7. E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
8. P. Ciaccia and M. Patella. Searching in metric spaces with user-defined and approximate distances. *ACM Trans. on Database Systems*, 27(4):398–437, 2002.
9. K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.
10. R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math*, 17(1):134–160, 2003.
11. K. Fredriksson. Parallel and memory adaptive metric indexes. *Pattern Recognition Letters*, to appear.
12. L. Micó, J. Oncina, and R. Carrasco. A fast branch and bound nearest neighbour classifier in metric spaces. *Pattern Recognition Letters*, 17:731–739, 1996.
13. L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.
14. P. Navarrete and J. Ruiz-Del-Solar. Analysis and comparison of eigenspace-based face recognition approaches. *Int. Journal of Pattern Recognition and Artificial Intelligence*, 16(7):817–830, 2002.
15. G. Navarro, R. Paredes, and E. Chávez. t-spanners as a data structure for metric space searching. In *SPIRE 2002: 9th International Symposium on String Processing and Information Retrieval*, LNCS 2476, pages 298–309, 2002.
16. P. Phillips, H. Wechsler, J. Huang, and P. Rauss. The FERET database and evaluation procedure for face recognition algorithms. *Image and Vision Computing Journal*, 16(5):295–306, 1998.
17. E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.
18. J. Vilar. Reducing the overhead of the AESA metric-space nearest neighbor searching algorithm. *Information Processing Letters*, 56:256–271, 1995.
19. D. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, U. of California, 1996.

Updating Directed Minimum Cost Spanning Trees

Gerasimos G. Pollatos*, Orestis A. Telelis**, and Vassilis Zissimopoulos

Dept. of Informatics and Telecommunications,
University of Athens, Hellas, Greece
{gpol, telelis, vassilis}@di.uoa.gr

Abstract. We consider the problem of updating a directed minimum cost spanning tree (DMST), when edges are deleted from or inserted to a weighted directed graph. This problem apart from being a classic for directed graphs, is to the best of our knowledge a wide open aspect for the field of dynamic graph algorithms. Our contributions include results on the hardness of updates, a dynamic algorithm for updating a DMST, and detailed experimental analysis of the proposed algorithm exhibiting a speedup factor of at least 2 in comparison with the static practice.

Keywords: branchings, dynamic graph algorithms, data structures.

1 Introduction

We study the problem of updating a *directed minimum spanning tree* (DMST) efficiently when a directed edge is inserted to or deleted from a weighted digraph. On a digraph $G(V, E)$, of $|V| = n$ vertices and $|E| = m$ edges, each associated with a non-negative cost $c(e)$, a DMST is defined as *a maximal acyclic subset of edges, such that no vertex of the digraph has more than one incoming edge in this set, and the total edge cost is minimum.* If G is strongly connected this definition implies indeed a directed tree (also called arborescence) blossoming out of its root, otherwise it may be a collection of trees (also called a *branching* [1]). Since G can always be made strongly connected by the addition of at most $O(n)$ edges, we can assume a directed tree. Applications of DMST updates range from wireless networks [2, 3] to hardware design [4, 5].

An identical polynomial time algorithm was described for this problem in [1, 6, 7]. For the rest of the discussion we refer to this algorithm as Edmonds' algorithm [1]. Tarjan [8] gave an implementation of $O(\min\{m \log n, n^2\})$ time. Gabow et al. [9] improved the running time to $O(m + n \log n)$ by using a special implementation of Fibonacci heaps. Improved heaps in [10] yielded deterministic $O(m \log \log n)$ and randomized $O(m\sqrt{\log \log n})$ time.

* Author partially supported by the programme $\Pi ENE\Delta 2003$ of the Greek General Secretariat of Research and Technology.

** Author partially supported by the Greek Ministry of Education under the project PYTHAGORAS II.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 291–302, 2006.

To the best of our knowledge, the *dynamic* DMST problem is a wide open aspect for the area of *dynamic graph algorithms* [11], in contrast to the near optimal achievements seen for the minimum spanning tree in undirected graphs [12]. A *fully* dynamic graph algorithm maintains efficiently a solution to a graph problem when edges are deleted from or inserted to the underlying graph in time less than the time required for re-evaluating a solution from scratch.

Our contributions include a hardness result regarding the complexity of dynamic DMST updates (section 3), the design of a fully dynamic algorithm and its analysis in the *output complexity* model (sections 4-5), and extended experimental investigation of the proposed algorithm (section 6), revealing a speedup factor of at least 2 in comparison with the static re-evaluation practice. In the output complexity model the complexity of a dynamic algorithm is measured with respect to a minimal subset of the previous output that needs to be updated [13, 14, 15, 16].

2 Preliminaries

From now on we assume as input a strongly connected digraph $G(V, E)$, with edge costs $c(e) \geq 0$. If the input digraph is not strongly connected, we add a vertex v_∞ and $2n$ edges of infinite (very large) weight, (v_∞, v_i) and (v_i, v_∞) for each $v_i \in V$, so as to make it strongly connected. These edges will never be affected by dynamic edge operations, so that strong connectivity of the underlying digraph is always assured. For each directed edge $e = (u, v) \in E$, we refer to $t(e) = u$ as the tail vertex of e, and $h(e) = v$ as its head vertex. For $S \subset V$, let $\delta_E(S) = \{e \in E | h(e) \in S, t(e) \in V - S\}$ be the "in" cut-set of S w.r.t. E. The algorithm of Edmonds greedily produces an edge set $H \subseteq E$ and prunes it to obtain the DMST T:

1. set $H = \emptyset$
2. set $\hat{c}(e) = c(e)$ for every edge e
3. while there are more than one vertices, pick a vertex v
   (a) let $e^\star$ be the incoming edge of v with the minimum $\hat{c}(e)$
   (b) set $\hat{c}(e) = \hat{c}(e) - \hat{c}(e^\star)$ for every incoming edge of v
   (c) insert edge $e^\star$ in H
   (d) if a directed cycle occurred, *contract* the cycle into a single vertex.
4. create T from H by removing redundant edges.

The loop (lines (a)-(d)) creates an edge set $H \subseteq E$, and the final DMST T is produced from H, by removal of redundant edges. This removal can be performed in $O(n)$ time [8], thus making the loop a complexity bottleneck for the algorithm. In a strongly connected digraph the algorithm will eventually contract the vertex set into a single vertex. At most $n - 1$ contractions will take place, since each contraction absorbs at least one of the original digraph's vertices. In the sequel we refer to vertices emerged by contraction as *c-vertices* and to vertices of the original digraph as *simple* vertices.

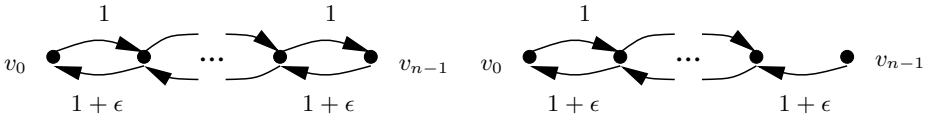


Fig. 1. Edges of cost 1 form the DMST for the left digraph. Deletion of (v_{n-2}, v_{n-1}) causes the DMST to change entirely (right) into a new one consisting of edges of cost $1+\epsilon$ because inclusion of any of the 1-weighted edges cannot yield a maximal edge set with DMST properties.

3 On the Hardness of Updates

We consider the hardness of DMST updates when the only information retained and used is the DMST itself. We use the framework presented in [17] which assumes that the unit operation of an algorithm is evaluation and positivity testing of an analytic function over the edge weights of the underlying digraph. Such an algorithm is called an *analytic tree program*. A lower bound on the verification complexity of a DMST is obtained:

Lemma 1. *Given a directed acyclic graph G of m edges with positive edge costs and a subset T of edges, an analytic tree program verifying that T is a DMST of G incurs $\Omega(m)$ complexity.*

Proof. A feasible tree (or a collection of trees - a branching) in a DAG, is any assignment of a unique incoming edge to each vertex. This can be checked in $O(n)$ time. The cost of T is minimum if and only if for every $e \notin T$ there is $e' \in T$ with $h(e) = h(e')$ and $c(e') \leq c(e)$, which translates to testing that each vertex is assigned its minimum cost incoming edge. This implies testing a set of $\Theta(m)$ inequalities for analytic functions of edge weights. A classic result of Rabin [18] states that *all* these inequalities must be evaluated in the worst case. □

The $\Omega(m)$ lower bound for verification holds for general digraphs in the worst case. This leads to the following result:

Theorem 1. *Dynamic maintenance of a DMST under edge deletions and/or insertions is as hard as recomputing a DMST from scratch if only the DMST information is retained and used between updates.*

Proof. Consider a digraph G of n vertices $v_0, \ldots, v_{n-1}$. Let edges (v_i, v_{i+1}), for $i = 0, \ldots, n-2$ have cost 1 and edges (v_i, v_{i-1}), $i = 1, \ldots, n-1$ have cost $1+\epsilon$ for some $\epsilon > 0$. Set all other edges to some cost $M > 1+\epsilon$. Then a DMST of this digraph is the *directed line* $\{(v_i, v_{i+1}) | i = 0 \ldots n-2\}$. Removal of edge (v_{n-2}, v_{n-1}) from this set, causes the DMST to change completely to another optimal set of edges $\{(v_i, v_{i-1}) | i = 1, \ldots, n-1\}$. Re-insertion of the removed edge causes the DMST to change entirely to its former state (see fig. 1 for an example). Every algorithm using only DMST information to update the DMST per edge operation requires at least the time given by lemma 1, which is $\Omega(n^2)$ for dense digraphs. □

A similar result was derived in [17] for shortest paths tree updates. In the next section we take the approach of maintaining intermediate information related to construction of a solution (also suggested in [17] and investigated later in [14] for shortest paths tree updates). Note that when the underlying digraph is restricted to remain a DAG in between edge operations, a simple application of Fibonacci heaps yields an $O(\log n)$ update time dynamic algorithm.

4 Dynamic Algorithm

The algorithm maintains as many contractions as possible per edge operation, along with the selected edges (edges of H). The purpose of this practice is to efficiently initialize and execute the implementation of Edmonds' algorithm known from [9] on a maintained partially contracted digraph, so as to process less vertices and edges per edge operation. We show that such a partially contracted digraph can be recognized in $O(n)$ time by using simple operations over an appropriate data structure, and a modified version of the implementation of [9].

4.1 An Augmented Structure

We present a data structure, namely the *Augmented Tree* (ATree), which appropriately encodes the redundant edge set H along with all vertices (c-vertices and simple ones) processed during execution of Edmonds' algorithm. Simple vertices are represented in the ATree by *simple nodes* while c-vertices are represented by *c-nodes*. For the rest we denote simple nodes with N_i^s, where $v_i \in V$ and c-nodes with N_j^c. We use unsuperscripted N to refer to ATree nodes regardless of their type. Six records are maintained at each node N of the ATree:

1. $e(N)$ is the edge selected by the algorithm for the represented vertex. If no edge was selected we set $e(N) = null$ and call N a *root node*. The root node will be unique as discussed below.
2. $y_N = \hat{c}(e)$ is the cost of edge $e(N)$ at the time it was selected for the vertex represented by N.
3. $children(N)$ is a list holding the children of N in the ATree.
4. $parent(N)$ is the parent node of N in the ATree (which equals to $null$ if N is the root node).
5. $contracted\text{-}edges(N^c)$ is a list holding all edges contracted during creation of the corresponding c-vertex represented by N^c (that is, edges having both their end-vertices on the contracted cycle).
6. $kind(N)$ is the kind of node N (simple node or c-node).

Since the digraph is strongly connected, all vertices will be eventually contracted to a single c-vertex by the end of the algorithm's execution. This c-vertex is represented by the root node of the ATree. The parent of each other node N is the intermediate c-node N^c to which it was contracted. Since the parent of each node is unique, the described structure is indeed a tree.

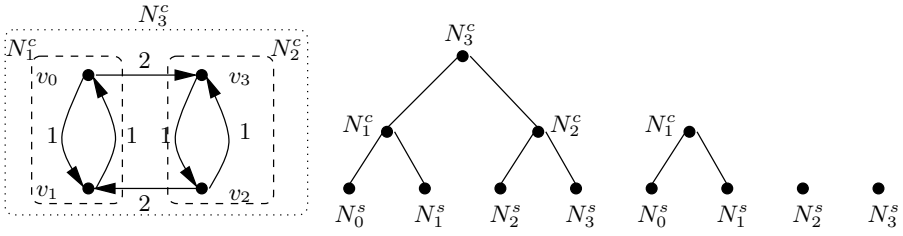


Fig. 2. Execution of Edmonds' algorithm on the digraph on the left performs contractions marked with dashed lines. The representative ATree appears in the middle. The decomposed ATree after deletion of edge (v_2, v_3) represents a partially contracted digraph of three vertices (on the right).

The ATree has at most $O(n)$ nodes because the algorithm handles $O(n)$ contractions. Construction of an ATree can be embedded into the implementation of [9], without affecting its complexity. However, maintenance of *contracted-edges* lists requires special manipulation with respect to the implementation of [9], and we defer this discussion to paragraph 4.5. Fig. 2 depicts an ATree example (middle) with respect to execution of Edmonds' algorithm on a digraph (left).

4.2 Deleting Edges

We discuss how to handle edge deletions using the ATree structure. Let $e_{out} \in E$ be an edge we want to remove from the digraph. Two cases must be considered:

1. $e_{out} \notin H$: we only need to remove e_{out} from the digraph and from the *contracted-edges* list to which e_{out} belongs. This can be achieved in $O(1)$ time, if we use an *endogenous* list implementation [9]: each edge has associated pointers in the digraph representation, pointing to the next and previous elements in the list.
2. $e_{out} \in H$, in which case we proceed by *decomposing* the ATree, initializing Edmonds' algorithm w.r.t. the remainders of the ATree and execute it.

Decomposition. The decomposition of the ATree begins from node N such that $e(N) = e_{out}$ and proceeds by following a path from N towards the ATree root and removing all c-nodes on this path except N. Each of the children of a removed c-node is made the root of its own subtree. By the end of this procedure, the initial structure has been decomposed into smaller ATrees, each corresponding to a contracted subset of the original digraph's vertices. Observe that all these ATrees remain intact after decomposition, because e_{out} was not part of their formation. An example of ATree decomposition is shown on the right of fig. 2.

4.3 Recognizing a Partially Contracted Digraph

Having performed the decomposition of the ATree, we proceed by recognizing the partially contracted digraph $G(V', E')$ represented by the remainders (namely

smaller ATrees). Let $V' = \{N_1 \dots N_k\}$ be the roots of ATrees after decomposition. These will constitute the vertex set of the digraph. A BFS on each tree suffices to assign each original digraph vertex v_i to some ATree root in V' in $O(n)$ time. Now we need to identify E' without scanning all edges of the original digraph. E' consists of edges having their end-vertices in different remaining ATrees. Let $R = \{N_1^c \dots N_r^c\}$ be the set of removed c-nodes during decomposition of the ATree. Note that the union of *contracted-edges*(N_i^c) lists, $N_i^c \in R$, is precisely the correct edge set E' and it can be found in $O(n)$ time.

Given the partially contracted digraph $G(V', E')$, each $N \in V'$ associated with a set of incoming edges $\delta_{E'}(N)$, a second aspect concerns consideration of the proper reduced costs $\hat{c}$ for these edges. Let $v_i \in V$ be a vertex of the original digraph represented as a leaf N_i^s of an ATree with root $N \in V'$ (it may occur that N_i^s is the root N itself). Let $e = (u, v_i)$ with $e \in \delta_E(v_i) \cap \delta_{E'}(N)$. Let $\mathcal{P} = [N_i, N_1^c, \dots N_l^c, N]$ denote the path from N_i to N in the ATree. Then by definition of the ATree and by functionality of Edmonds' algorithm described in section 2, we can determine the reduced cost of e:

$$\hat{c}(e) = c(e) - \sum_{N \in [N_i, N_1^c, \dots N_l^c]} y_N$$

As an example in the decomposed ATree of fig 2 we obtain $\hat{c}(e) = c(e) - y_{N_0}$ for all edges $e \in \delta_{E'}(N_1^c) \cap \delta_E(v_0)$. Our practice is to compute a reduction quantity r_i (i.e. the subtracted sum) for each simple node N_i^s of the remaining ATrees with a single BFS on each remaining ATree in a total of $O(n)$ time. Then, we can scan once the edges $e = (u, v_i) \in E'$ and assign them the proper reduced cost $\hat{c}(e) = c(e) - r_i$.

4.4 Inserting Edges

Edge insertion is handled by reduction to edge deletion. Let e_{in} be the edge we want to insert, at cost $c(e_{in})$. We have to check whether e_{in} should replace some edge encoded in the ATree. This check involves *only* c-nodes of the ATree that are ancestors of $N_{h(e_{in})}^s$ and is performed as follows: starting from the node $N_{h(e_{in})}^s$ we follow the path towards the ATree root. For each visited node N, we check whether $c(e(N)) > c(e_{in})$. If this is not the case, we proceed to the parent node. Otherwise, we have found a candidate node N which should have e_{in} as its selected edge, because it is of lower cost. It may be the case that the root node of the ATree is reached: then e_{in} cannot replace any edge of H. In this case we insert it in the digraph and in the *contracted-edges* list associated with the least common ancestor of $N_{t(e_{in})}^s$ and $N_{h(e_{in})}^s$.

Given that we have found a candidate node N which should replace its $e(N)$ with e_{in}, we have to determine whether e_{in} should or should not belong in the "in" cut-set of N. To do so we examine whether the $N_{t(e_{in})}^s$ is hanged in the subtree rooted at N, by engaging a BFS on this subtree. If $N_{t(e_{in})}^s$ is found, it is implied that e_{in} should *not* belong in the "in" cut-set of N, so we simply insert the edge in the digraph and in the *contracted-edges* list of the least common

ancestor of $N^s_{t(e_{in})}$ and $N^s_{h(e_{in})}$. Otherwise, we insert e_{in} in the digraph and engage a *virtual* deletion of $e(N)$, i.e. without actually removing $e(N)$ from the digraph. After this virtual edge deletion recognition of $G(V', E')$ takes place as described in the previous paragraph, and the algorithm of Edmonds is executed over $G(V', E' \cup \{e_{in}\})$.

4.5 Maintaining Contracted Edges

We describe here how to maintain a *contracted-edges*(N^c) list for each c-node N^c of the ATree structure, by introducing a simple modification on the $O(m+n \log n)$ time implementation of Gabow et al. [9], without burdening the complexity. A brief description of the implementation follows.

The loop of Edmonds' algorithm is executed by growing a path, referred to as the *growth path* in [9]. The growth path is constructed as follows: initially, an arbitrary vertex s, called *current root vertex*, is considered and an incoming edge $e = (u, s)$ of minimum cost $\hat{c}(e)$ is selected. Vertex u gets marked, edge e is added in the growth path and the process is repeated by considering vertex u as the current root vertex. If the insertion of e causes a directed cycle (i.e. its tail $t(e)$ is already marked), a contraction of the cycle happens and a new c-vertex replaces all cycle vertices in the growth path. This c-vertex becomes the current root vertex of the growth path.

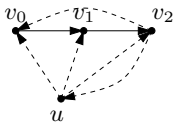
Each vertex $u \in V$ is associated with an *exit list*, which holds outgoing edges of u, incoming to some vertex on the current growth path. If we let $v_0, \ldots, v_l$ be the current growth path, with v_0 its current root vertex, such a list has the following contents:

1. *If u is not on the growth path:* its associated exit list contains only the edges e with $t(e) = u$ and $h(e) = v_j$ such that v_j is on the growth path.
2. *If $u = v_i$ is on the growth path:* only edges e with $t(e) = u$ and $h(e) = v_j$ such that v_j is on the growth path and $j < i$, are contained.

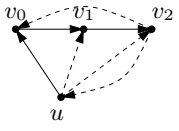
Furthermore, in both cases, the edges are sorted in increasing order of j. When a vertex is either added to the growth path, or takes place in a contraction, its exit list is scanned once (for purposes related to details of [9]) and cleared. The following modified manipulation of exit lists is adopted:

1. When a vertex u is added to the growth path, its exit list is scanned once and cleared as in [9], but each edge (u, v_i) (v_i belonging on the growth path) is added in a list *deprecated*(v_i).
2. When a contraction of vertices $v_0, \ldots, v_k$ happens, the new c-vertex is given an explicit name, say c. The exit lists of the contracted vertices are scanned once and cleared as required in [9], but their contents are merged into a *contracted-edges*(c) list initialized for the new c-vertex c. All *deprecated*(v_i), $i = 0 \ldots k$, are merged into *contracted-edges*(c).

By these modifications all edges contracted due to the emergence of a new c-vertex c (having both their end-vertices in the cycle) are stored in its associated



$exit(u)$:$\{(u, v_0), (u, v_1), (u, v_2)\}$
$exit(v_0)$:$\{\ \}$
$exit(v_1)$:$\{\ \}$
$exit(v_2)$:$\{(v_2, v_0)\}$
$deprecated(v_1)$:$\{(v_0, v_1)\}$
$deprecated(v_2)$:$\{(v_1, v_2)\}$



$exit(u)$:$\{\ \}$ $deprecated(v_0)$:$\{(u, v_0)\}$
$exit(v_0)$:$\{\ \}$ $deprecated(v_1)$:$\{(v_0, v_1), (u, v_1)\}$
$exit(v_1)$:$\{\ \}$ $deprecated(v_2)$:$\{(v_1, v_2), (u, v_2)\}$
$exit(v_2)$:$\{(v_2, v_0), (v_2, u)\}$

$contracted\text{-}edges(c)$:$\{(v_2, v_0), (v_2, u), (u, v_0), (v_0, v_1),$
$(u, v_1), (v_1, v_2), (u, v_2)\}$

Fig. 3. In the upper part, exit lists and deprecated lists are shown for the current growth path v_0, v_1, v_2. When edge (u, v_0) is added on the growth path the updated exit lists and the deprecated lists are as shown in the center part. Augmentation of the growth path with edge (v_2, u) causes contraction of all vertices. The list of contracted edges for the new c-vertex c is the union of exit and deprecated lists, shown in the lower part.

list *contracted-edges*(c). Merging of the lists can be done in k steps and since the algorithm performs k steps anyway for identifying the cycle, its complexity is not burdened. An example of the described manipulation appears in fig. 3.

5 Complexity

In order to study the output complexity of the proposed dynamic scheme, we have to identify the minimal portion of the maintained output that is affected by each edge operation. As mentioned previously, the output consists of all processed vertices (simple and c-vertices). A vertex v (whether a simple or c-vertex) is affected if it takes part in a *different* contraction in the new output after an edge operation. A contraction is defined exactly by the vertices and edges that comprise the directed cycle which caused it. A different contraction is one which was not present in the previous output. We denote the set of affected vertices with ρ, $|\rho|$ being its size. The *extended* set of affected elements (namely vertices and edges incoming to affected vertices) is denoted as $||\rho||$: this definition was introduced in [13] also used in [14, 16]. First we show that:

Lemma 2. *Root nodes of ATrees which emerged after decomposition of the initial ATree represent affected vertices.*

Proof. Let e_{out} be the removed edge, and N be the corresponding ATree node with $e(N) = e_{out}$. Clearly N is affected by definition, since it will change its selected incoming edge. Hence any contraction to which it takes part differs

from previous contractions at least by the new edge. For the rest roots of ATrees one of the following happens: either they take part in at least one contraction not present in the previous output, or they take part in a contraction also present in the previous output. In the latter case however, this contraction also included N, hence in the new output it differs at least by $e(N)$. □

Notice that $|\rho| \leq n$. During edge insertion/deletion all supplementary operations incur $O(n)$ complexity, while re-execution of Edmonds' algorithm processes only affected vertices, given by lemma 2. Hence:

Theorem 2. *Fully dynamic maintenance of a directed minimum cost spanning tree can be done in* $O(n + ||\rho|| + |\rho| \log |\rho|)$ *output complexity per edge operation.*

By the previous discussion and the results of [10]:

Corollary 1. *Fully dynamic maintenance of a directed minimum cost spanning tree can be done in deterministic* $O(n+||\rho|| \log\log |\rho|)$ *and* $O(n+||\rho|| \sqrt{\log\log |\rho|})$ *randomized output complexity per edge operation in sparse digraphs.*

6 Experimental Evaluation

We evaluated the proposed method on sequences of edge operations on digraphs of varying order and density. Implementation was grown in `C++` under version `3.1` of the `gcc` compiler with optimization level 3. Experiments were carried out on an Intel P4 `3.2 GHz` PC with `1 GB` of memory, running `Linux Kernel 2.6`. `CPU` time was measured using the `getrusage()` system call. We implemented the description of [9] for dense digraphs of $O(n^2)$ edges and the deterministic heaps of [10] for sparse digraphs of $O(n)$ edges. Both implementations discourage usage of pre-existing data structure libraries due to the heaps used and due to the *endogenous* nature of other supplementary structures.

6.1 Experimental Setup

A large set of random digraphs divided into three categories was used:

Dense Digraphs: $n = 500i$, $i = 1 \ldots 10$, densities $p = 0.2i$, $i = 1 \ldots 5$.
Sparse Digraphs: $n = 500i$, $i = 1 \ldots 10$, densities $p = \frac{c}{n-1}$, with c taking values in $\{10, 20, 30, 40, 50\}$.
Embedded Cliques: We generated 10 digraphs of order $n = 5000$ and density $\frac{10}{n-1}$, and embedded on each of them a clique of increasing order $500i$, $i = 1 \ldots 10$.

Edge costs were chosen uniformly at random from the range $1 \ldots 1000$. The dynamic algorithm was compared against re-executing Edmonds' algorithm on the whole digraph instance per edge operation (static practice). An edge operation was chosen to be insertion or deletion with probability 0.5. Average `CPU` time and number of iterations performed by each algorithm were derived over 10^4 operations per digraph instance. % Gain for both measures, defined as the relative savings of the dynamic over the static practice, is discussed.

6.2 Discussion

The proposed practice achieves substantial CPU time savings over the static DMST re-evaluation, as shown in fig. 4, for both complete and very sparse digraphs. Fig. 5 shows that for general dense digraphs, the savings are stable across orders and densities over 60% on average. For sparse digraphs the gain in CPU time increases from about 65% to near 95% when $c = 10$ (very sparse digraphs) as n increases, while the increase becomes more modest when c becomes larger.

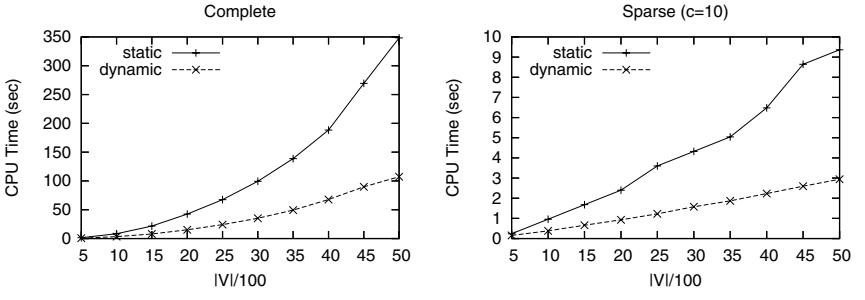


Fig. 4. Performance comparison per edge operation in complete and very sparse digraphs

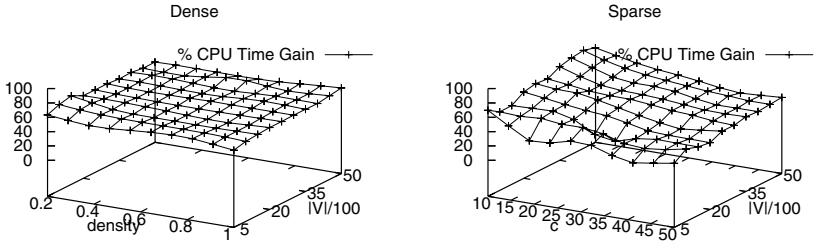


Fig. 5. Almost stable over 60% CPU Time gain of dynamic over static per edge operation in dense digraphs and increasing gain in sparse digraphs

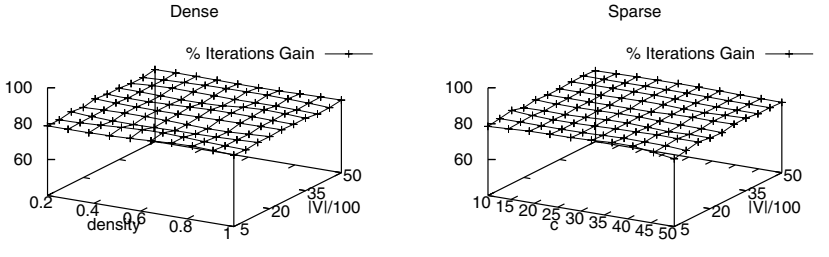


Fig. 6. Stable 80% iterations gain of dynamic over static per edge operation in dense and sparse digraphs

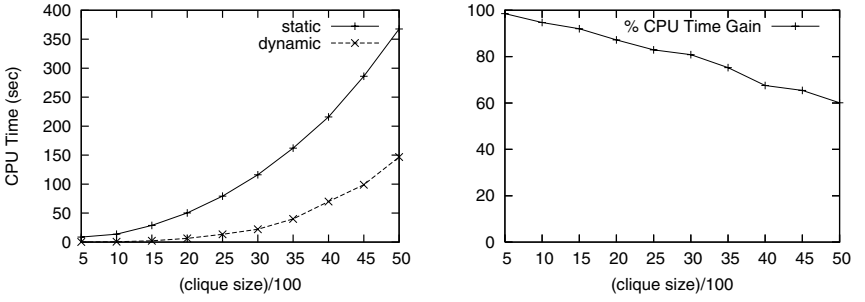


Fig. 7. Performance and % CPU Time gain for embedded cliques of increasing size

A stability of the dynamic practice is observed in fig. 6 in terms of iterations savings to a 80% gain across all dense and sparse graphs. This result combined with the 60% time gain for dense digraphs and the increasing gain observed for large sparse instances, implies a dependance of the overall performance on the density of edges. This dependance was further examined on very sparse instances ($c = 10$) having an embedded clique of increasing size, i.e. on digraphs of increasing non-uniformly distributed density.

The results we obtained confirm this dependance. Initially, when the embedded clique is very small and thus the instance is sparse, the overall gain is approximately 95% as shown in Fig. 7. As the embedded clique grows and the density of the considered digraphs increases, the gain decreases, resulting in a still significant 60% when the whole digraph has become complete.

Conclusively, the proposed dynamic algorithm achieves an update time reduced by a factor of more than 2 as opposed to solving the problem statically on dense digraphs. We believe that the case of sparse digraphs merits theoretical investigation from an average case complexity perspective, since there appears to be an asymptotic improvement on average.

7 Conclusions

We have studied the problem of updating the DMST of a weighted digraph changing by edge insertions and deletions. We provided an $\Omega(n^2)$ complexity lower bound when the only information retained is the DMST itself, and designed a dynamic algorithm of $O(n + ||\rho|| + |\rho| \log |\rho|)$ output complexity, where ρ is a minimal subset of the output that needs to be updated per edge operation. Experimental evaluation of the proposed technique establishes its practical value, and raises an open question regarding average case analysis for sparse digraphs.

Acknowledgements. We thank three anonymous reviewers for comments that helped improve the quality of the paper.

References

1. Edmonds, J.: Optimum branchings. Journal of Research of the National Bureau for Standards **69B** (1967) 125–130
2. Kang, I., Poovendran, R.: Maximizing network lifetime of broadcasting over wireless stationary adhoc networks (to appear). Mobile Networks **11**(2) (2006)
3. Li, N., Hou, J.: Topology Control in Heterogeneous Wireless Networks: Problems and Solutions. In: Proceedings of the 23rd IEEE INFOCOM. (2004)
4. Li, Z., Hauck, S.: Configuration compression for virtex fpgas. In: Proceedings of the 9th IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'01. (2001) 147–159
5. He, L., Mitra, T., Wong, W.: Configuration bitstream compression for dynamically reconfigurable FPGAs. In: Proceedings of the 2004 International Conference on Computer-Aided Design, ICCAD'04. (2004) 766–773
6. Bock, F. In: An algorithm to construct a minimum spanning tree in a directed network. In: Developments in Operations Research. Gordon and Breach (1971) 29–44
7. Chu, Y.J., Liu, T.H.: On the shortest arborescence of a directed graph. Scientia Sinica **14** (1965) 1396–1400
8. Tarjan, R.E.: Finding optimum branchings. Networks **7** (1977) 25–35
9. Gabow, H.N., Galil, Z., Spencer, T.H., Tarjan, R.E.: Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. Combinatorica **6** (1986) 109–122
10. Mendelson, R., Tarjan, R.E., Thorup, M., Zwick, U.: Melding Priority Queues. In: Proceedings of SWAT'04, Springer LNCS 3111. (2004) 223–235
11. Eppstein, D., Galil, Z., Italiano, G.F.: 8: Dynamic graph algorithms. In: Algorithms and Theory of Computation Handbook. CRC Presss (1999)
12. Holm, J., de Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. Journal of the ACM **48** (2001) 723 – 760
13. Alpern, B., Hoover, R., Rosen, B.K., Sweeney, P.F., Zadeck, F.K.: Incremental evaluation of computational circuits. In: Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, SODA'90. (1990) 32–42
14. Ramalingam, G., Reps, T.: On the complexity of dynamic graph problems. Theoretical Computer Science **158**(1&2) (1996) 233–277
15. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully dynamic algorithms for maintaining shortest paths trees. Journal of Algorithms **34** (2000) 251–281
16. Pearce, D.J., Kelly, P.H.J.: A Dynamic Algorithm for Topologically Sorting Directed Acyclic Graphs. In: Proceedings of the 3rd Workshop on Efficient and Experimental Algorithms, WEA'04, Springer LNCS 3059. (2004) 383–398
17. Spira, P.M., Pan, A.: On Finding and Updating Spanning Trees and Shortest Paths. SIAM Journal on Computing **4**(3) (1975) 364–380
18. Rabin, M.O.: Proving simultaneous positivity of linear forms. Journal of Computers and Systems Sciences **6** (1972) 639–650

Experiments on Exact Crossing Minimization Using Column Generation

Markus Chimani, Carsten Gutwenger, and Petra Mutzel

Department of Computer Science, University of Dortmund, Germany
{markus.chimani, carsten.gutwenger, petra.mutzel}@cs.uni-dortmund.de

Abstract. The crossing number of a graph G is the smallest number of edge crossings in any drawing of G into the plane. Recently, the first branch-and-cut approach for solving the crossing number problem has been presented in [3]. Its major drawback was the huge number of variables out of which only very few were actually used in the optimal solution. This restricted the algorithm to rather small graphs with low crossing number.

In this paper we discuss two column generation schemes; the first is based on traditional algebraic pricing, and the second uses combinatorial arguments to decide whether and which variables need to be added. The main focus of this paper is the experimental comparison between the original approach, and these two schemes. We also compare these new results to the solutions of the best known crossing number heuristic.

1 Introduction

Crossing minimization is among the oldest and most fundamental problems arising in the area of automatic graph drawing and VLSI design and has been studied in graph theory for many decades.

A drawing of a graph $G = (V, E)$ is a mapping of each node $v \in V$ to a distinct point and each edge $e = (v, w) \in E$ to a curve connecting the incident nodes v and w without passing through any other node. Common points of two edges that are not incident nodes are called *crossings*. The *crossing number*, denoted by $\mathrm{cr}(G)$, is the minimum number of crossings among all drawings of G.

The crossing minimization problem is a central problem in the area of automatic graph drawing, since a low crossing number is among the most important design criteria for drawings [12]. Yet its roots are much older: P. Turán described in his "Note of Welcome" in the first issue of *Journal of Graph Theory* [13] that he started to examine this topic during the second world war. Even though much research has been conducted – see, e.g., [14] for an overview – even the crossing numbers for complete and complete bipartite graphs can only be conjectured.

In 2005, Buchheim et al. [3] introduced the first algorithm to compute the exact crossing number, based on a branch-and-cut approach, which we outline in Section 2. The problem with this approach was the huge amount of variables required. This restricted the algorithm to relatively small graphs with few crossings, since its running time heavily depends on the latter parameter.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 303–315, 2006.

To overcome this drawback, we devised two different column generation schemes, i.e., methods starting with a small subset of variables and dynamically adding variables during the computation; see Section 3. The first approach, called *algebraic pricing*, is based on the standard pricing technique first introduced by Dantzig and Wolfe [4]. Starting from a small subset of the variables, we can compute *reduced costs* for the variables not yet in the ILP and decide on their addition based on these values. The second scheme is called *combinatorial column generation*. Its main idea is to interpret the absence of certain variables in a combinatorial way, such that we can quite intuitively decide, whether these variables have to be added or not.

Section 4.1 presents the results of our experimental study, comparing the original method to the two column generation schemes. Note that even the original method is already an improved version of the one presented in [3], due to the use of primal heuristics during the cutting phase, as well as the recently developed preprocessing method called *non-planar core* [7]. The dataset used for the study is the well known *Rome library* [5], which has, e.g., also been used for a comparison between crossing minimization heuristics [8]. Section 4.2 gives an experimental analysis of the quality achieved by the currently best known heuristics compared to these values.

2 The Branch-and-Cut Approach

ILP and Cutting Planes. In order to solve the crossing minimization problem for a graph $G = (V, E)$, it is necessary to determine which edges cross in an optimal solution. However, it is even NP-complete to decide if there exists a drawing realizing a given set of edge crossings [10]. Therefore, the ILP solution needs to contain information on the order of these crossings along an edge. This is achieved by restraining the solution to *crossing restricted drawings* (or *CR-drawings*), i.e., drawings in which each edge is involved in at most one crossing. To compute an optimal solution for G, we represent each edge of G by a sufficiently long chain of *segments*. An optimal crossing restricted solution for this modified graph clearly induces a crossing minimal solution for G. Obviously, $|E| - \deg(v) - \deg(w) - 1$ segments are sufficient for an edge (v, w), but also any upper bound for $\mathrm{cr}(G)$ bounds the number of required segments. This expansion leads to the problematically high number of variables.

If D is a set of crossings and we place a new node on every crossing $(e, f) \in D$, i.e., we split both e and f and identify the created dummy nodes, we obtain a *(partial) planarization* G_D of G with respect to D. Then, D is realizable if and only if G_D is planar.

The ILP-formulation for finding a crossing minimal CR-drawing has a 0/1 variable $x[e, f]$ for each unordered pair (e, f) of segments, which is 1 if e and f cross and 0 otherwise. The optimization goal is to minimize the sum of all the $x[e, f]$ according to the following constraints – please refer to [3, 2] for details:

- *CR-constraints* assuring that each segment crosses at most one other edge.
- *Kuratowski constraints* that guarantee the realizability of the set of segment crossings given by the variables that are set to 1.

Our formulation contains a Kuratowski constraint for each subdivision of K_5 or $K_{3,3}$ in every planarization with respect to a crossing restricted set of crossings. It is clearly impractical to generate all these constraints in advance, hence they are incorporated into a branch-and-cut framework, and Kuratowski constraints are only generated when necessary. For integer solutions, the separation problem for Kuratowski constraints can be solved in linear time by finding a Kuratowski subdivision in the partial planarization of G [15], for fractional solutions the separation problem is solved heuristically using rounding techniques.

Primal Heuristic. The state-of-the-art crossing minimization heuristic is the *planarization approach* which has been proposed by Batini et al. [1]. It first computes a planar subgraph of G, and then inserts the remaining edges one after the other. During the edge insertion step, crossings are replaced by dummy vertices, such that the final result is a planarization of G. Gutwenger and Mutzel compared various variants of the planarization approach in an experimental study [8], including further permutation and postprocessing strategies. In our implementation, we use the heuristic which achieved the best quality in this study.

Preprocessing. The traditional method of preprocessing for crossing minimization is to run the algorithm on each *block*, i.e., 2-connected component, of G separately, and to merge *chains* of edges, i.e., edges connected by nodes with degree two, into single edges.

In [7], Gutwenger and Chimani presented a linear-time reduction scheme based on SPQR-trees [6], which can further simplify these blocks. The resulting *non-planar core* may contain edges with integer weights. It is easy to see, that we can extend the ILP given above to support these weights by simply modifying the edges' coefficients in the objective function. This reduction method is known to roughly halve the graphs' size on average for the Rome library which we use in our experimental study. This is an improvement of about 2/3 over the traditional preprocessing and was therefore used in all experiments given below.

3 Column Generation Schemes

The main idea of column generation is to start solving a problem only with a relatively small subset of the ILP's variables. During the branch-and-cut computation we can decide which variables might have to be added in order to improve the solution. This idea is based on the observation, that in many problems there are a lot of "unused" variables which are 0 during the whole computation. Identifying and excluding superfluous variables leads to smaller ILPs and LP relaxations, thus improving the overall running time. Furthermore, the branch tree will be smaller, since there are less variables to branch on.

When analyzing the ILP presented above, we can clearly see that the full expansion of the graph, in order to guarantee the existence of a crossing restricted drawing that realizes the crossing number, will be unnecessary in most cases. Hence we can try to start with variables only for the first segment of each chain, adding additional segment-variables later when necessary.

3.1 Algebraic Pricing

The standard idea of column generation is called *pricing*. Thereby we test all variables not yet in the ILP by computing their *reduced costs*. Let c be the vector of coefficients the variable v would have in the ILP's constraints (i.e., rows), and let d be the vector of the dual variables of the current solution. The reduced cost of v is simply defined as the scalar product of c and d. Based on the sign of this value we can decide whether it could be necessary to add this variable to the ILP or if the addition of this variable cannot lead to better solutions. This operation may give false positives, hence we try to add the variables with the smallest reduced costs first. Their addition can lead to rejecting other variables which had a negative reduced cost before.

Due to the structure of our specific problem, we actually do not have to test all variables not introduced yet. It suffices to test variables which "extend" the currently introduced edge segments. More technically, this means that we start with a quadratic number of variables: for all pairwise disjoint and non-adjacent edges e and f, we label one of their expanded segments with e_0 and f_0, and add a variable representing a crossing $x[e_0, f_0]$ between these segments. In the pricing step we have only to consider variables that are a *convex extension* of the existing variables. We can add variables for a segment left or right to the segment for which a variable already exists, leading to positive and negative indices. In the first iteration we will have to test the variables $x[e_{\pm 1}, f_0]$ and $x[e_0, f_{\pm 1}]$. Figure 1 shows the situation at some later iteration step: The indices of segments of e are plotted on the horizontal axis, the indices of the segments of f on the vertical. The gray rectangles mark included variables; the circles denote potential new variables, whereby the crosses mark variables which, although adjacent to the current set of realized variables, are not tested, since they would extend the

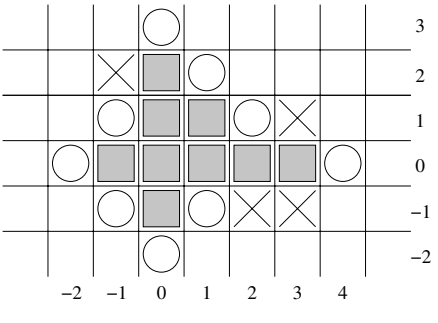


Fig. 1. Algebraic Pricing. Convex extension of the variable set included in the ILP

variable set in a way that is not orthogonally convex. Note that the orthogonally convex extensions are sufficient, and crossed out variables can be realized by first extending via the variables "in between".

3.2 Combinatorial Column Generation

Our second column generation approach, which turns out to be much more effective, works as follows: As for algebraic pricing, we start with a variable set of quadratic size, representing possible crossings between the first segments of the edges. Based on the fact that the graph extension was necessary only for efficient realizability testing, we will drop the CR-constraints for all these variables. For all additional segments associated to variables added throughout the calculation, the CR-constraints will be included into the ILP.

Now we will modify the standard framework of calculation in a node in the branch-and-bound tree: the traditional order is to first solve the LP relaxation of the current (sub)problem, apply cutting planes afterwards, and finally perform the pricing step. In our scheme, we flip the cutting-plane and the column generation steps for reasons explained in the following – see [2] for details.

When solving the initial ILP, we may find solutions that would violate the CR-constraints which we did not add. Such solutions cannot be interpreted as a (partial) planarization, hence our cutting plane approach would fail. Our column generation scheme simply checks all these CR-constraints not in the ILP. Let e_0 be a segment whose CR-constraint would be violated by the variables $x[e_0, f_{i_1}^{(1)}]$, $x[e_0, f_{i_2}^{(2)}]$, ..., $x[e_0, f_{i_k}^{(k)}]$. Intuitively this means that the segment e_0 is crossed by k different segments, namely $f_{i_1}^{(1)}$, ..., $f_{i_k}^{(k)}$, but the order of these crossings in unknown. For all $y = 1, \ldots, k$, let j_y be the largest index for which the variable $x[e_{j_y}, f_{i_y}^{(y)}]$ is already in the ILP. We will then simply add the variables $x[e_{j_y+1}, f_{i_y}^{(y)}]$, for all y.

The trick is to reduce the objective function coefficient by some small ϵ for each variable which does not represent a crossing between two segments with index 0. Thereby we ensure that instead of putting all these edges on the first segment, at least one of these segments will be routed via the newly added segment since it is cheaper. If necessary, the LP relaxation is then also able to shift the already existing crossings on the other segment of e_0's edge to higher indices. If, at some point, $j_y + 1 \geq s$ – with s being the maximum number of segments of the corresponding edge – we will not introduce a new variable but add the CR-constraint corresponding to e_0 to the ILP instead. Note that this situation can happen, due to the fact that the ILP only tries to find a solution strictly better then the one given by the upper bound, and hence the maximum expansion of an edge might be one segment less than actually necessary for the realization of a crossing minimal drawing.

See Figure 2 for an example of the presented column generation scheme: consider an integer solution by the LP relaxation to be interpreted as shown on the left: There are two segments crossing the first segment of the horizontal

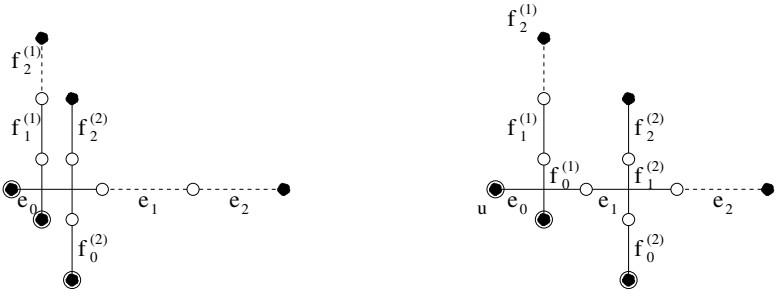


Fig. 2. Combinatorial Column Generation. Left: implicit CR-constraint is violated; Right: by adding additional variables and reducing the cost on the additional segment, the situation is resolved.

edge e. Since this violates the CR-constraint of that segment, we add the variables $x[e_0, f_0^{(1)}]$ and $x[e_0, f_1^{(2)}]$, such that a solution as given on the right can be computed. While the first solution had an objective value of $1 + 1 = 2$, the new solution is $1 + 1 - \epsilon = 2 - \epsilon$, and will therefore be preferred by the LP solver.

Whenever the column generation terminates without adding new variables – and therefore not forcing us to resolve the LP relaxation – we can interpret the solution for our traditional cutting scheme. Note that whenever no implicit CR-constraint is violated by the LP-relaxation, it is obvious that the variables currently not in the ILP are not necessary for the optimal solution. Hence the optimal objective value on the restricted variable set is at least as small as on the fully expanded ILP, which suffices to proof the optimality of the induced solutions.

4 Experimental Comparison

We implemented the described algorithms as part of the open-source C++ library *Open Graph Drawing Framework* OGDF [11]. We use the free branch-and-cut-and-price framework ABACUS [9], in conjunction with the commercial optimization library CPLEX (version 9). The tests were performed on a single AMD Opteron CPU with 2.4 GHz and 32 GB RAM shared between 4 CPUs. As it turned out, the memory consumption seldomly exceeded 1 GB.

For the following experiments, we used the *Rome library*, introduced in [5], as a benchmark set of graphs. It has already been used for various experimental studies, including [8] where different crossing heuristics have been compared. The set contains 11,389 graphs that consist of 10 to 100 nodes and 9 to 158 edges. These graphs were generated from a core set of 112 "real life" graphs used in database design and software engineering applications.

Due to the complexity of the crossing minimization problem, we restricted ourselves to the graphs with up to 75 nodes for the column generation comparison; combinatorial column generation *(CCG)* was then used to compute the graphs with up to 90 nodes. For each computation scheme, we applied a 30

minute time limit per instance. In the following diagrams, we will use AP as a shorthand for algebraic pricing, and F as a shorthand for the full ILP, without any column generation scheme.

4.1 Comparison Between Column Generation Schemes

Figure 3 shows the percentage of graphs for which the exact crossing number was computed. The size of the circles denotes the number of graphs per node count. While the full ILP could only solve a third of the large graphs, CCG could still solve about 50%. Note that even after only 5 minutes, CCG could already solve more graphs than F and AP after 30. Furthermore, there was no instance which could be solved by either F or AP, but remained unsolved by CCG.

As it turns out, the number of edges in the non-planar core is much more influential than the number of nodes in the original graph. Figure 4 shows the relationship between those two properties: while we can clearly see a correlation, the variance is quite high. When we look at Figure 5 we can see the clear dependence of a successful computation on the number of core edges.

Based on the primal heuristic, we know that the Rome library consists of many graphs with a crossing number of at most 1; we call these graphs *trivial*; in fact, nearly all graphs with up to 35 nodes are trivial. Since they are of no interest for our branch-and-cut algorithm, we restrict ourselves to the non-trivial graphs. Also note that for the following observations on the number of required variables, we only consider the graphs which were solved to provable optimality. Since CCG was able solve many more instances than AP, we will also consider the *common set*, which is the set of instances solved by AP and therefore also by CCG.

Figure 6 shows the number of variables used by AP and CCG, relative to the full (potential) variable set. While CCG needed more variables on average for all the graphs it could solve, it used less variables than AP, when compared

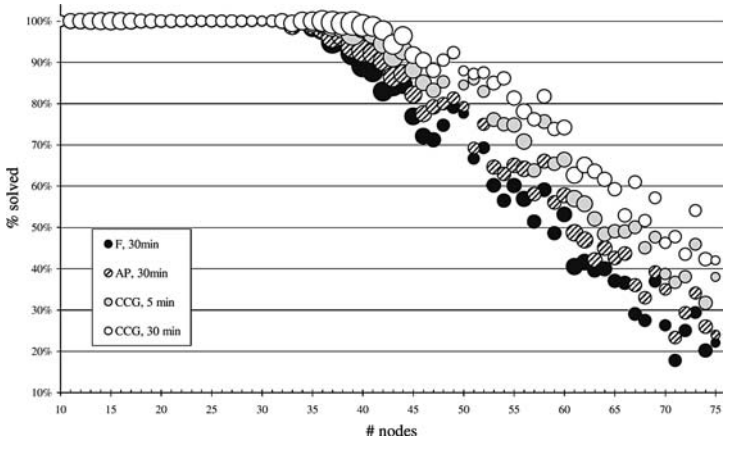


Fig. 3. Percentage of graphs solved

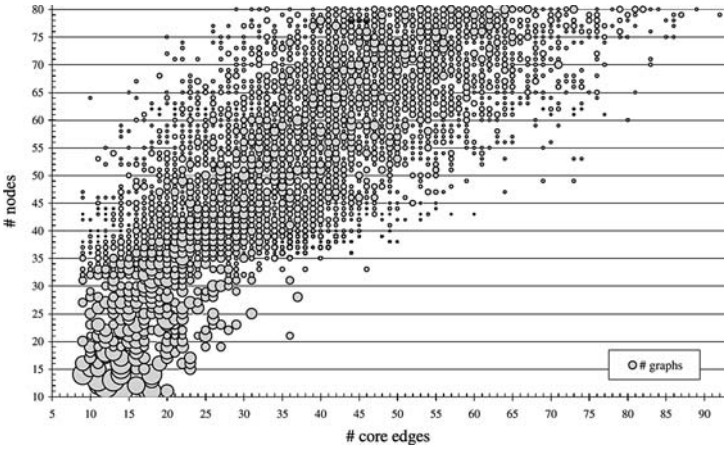


Fig. 4. Correlation between number of nodes and number of core edges

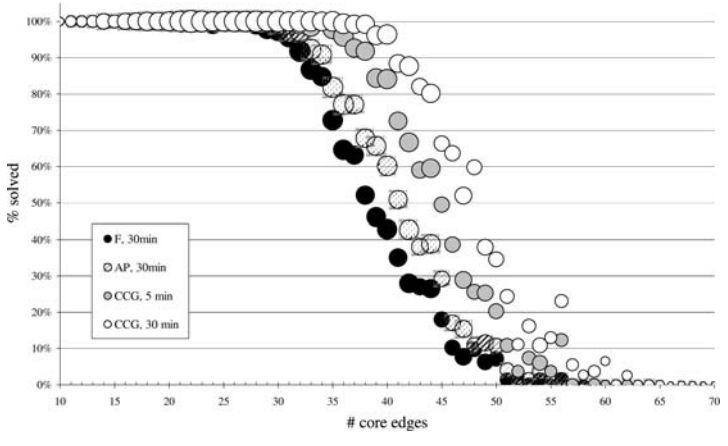


Fig. 5. Percentage of graphs solved

on the common set. This shows that CCG was able to solve graphs which required a larger variable set, whereby AP was too slow to tackle such graphs successfully.

Figure 7 shows the actual numbers of generated variables (compared on the common set). Note that the number of variables generated by CCG stays very close to the number of the initially generated variables. While AP could only solve graphs with roughly 10.000 potential variables on average, a similar statistic for all the graphs solved by CCG would show an average of between 30.000 and 40.000 potential variables.

According to the above statistics, it is obvious that the running time of CCG is superior to AP's, and that both are more efficient than no column generation

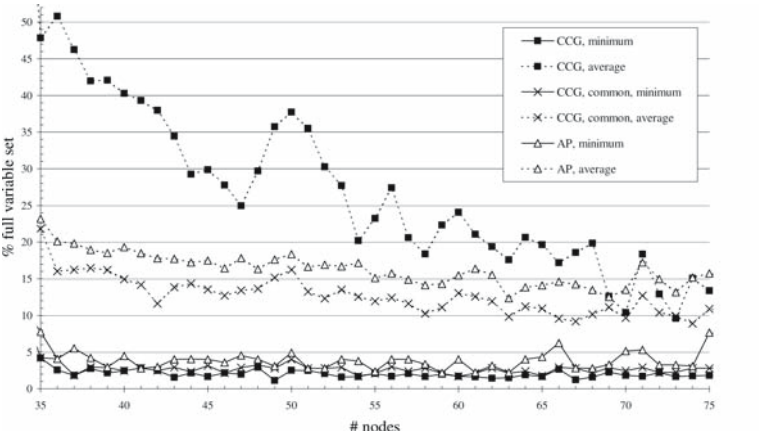


Fig. 6. Number of variables generated, relative to the full variable set

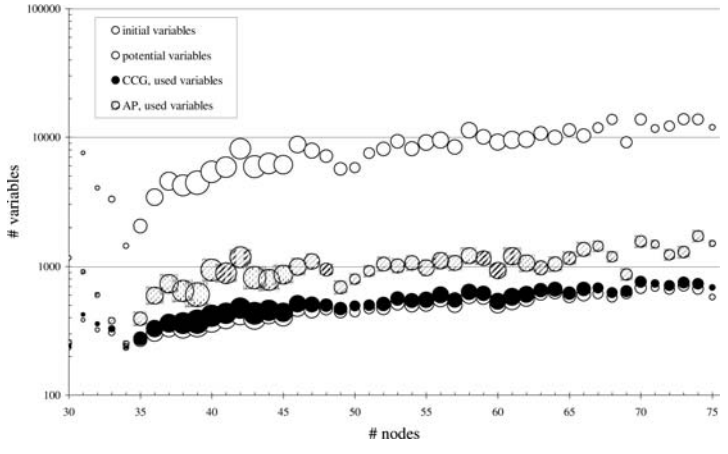


Fig. 7. Number of variables generated

at all. Yet, it is impressive that – over the set of instances solved by F – even the maximum running time of CCG is always far below F's average. When comparing AP to CCG on the common set, we see that AP's average is about equal to CCG's maximum time. The average running time of CCG over all successfully solved graphs is under 5 minutes.

4.2 Comparison with Heuristics

In order to evaluate the quality of crossing minimization heuristics, we compared the heuristic to the final upper bound of CCG after the 30 minutes time limit for each graph in the benchmark set. Figure 8 evaluates the quality of this upper bound by comparing it to the corresponding lower bound. The size of the circles

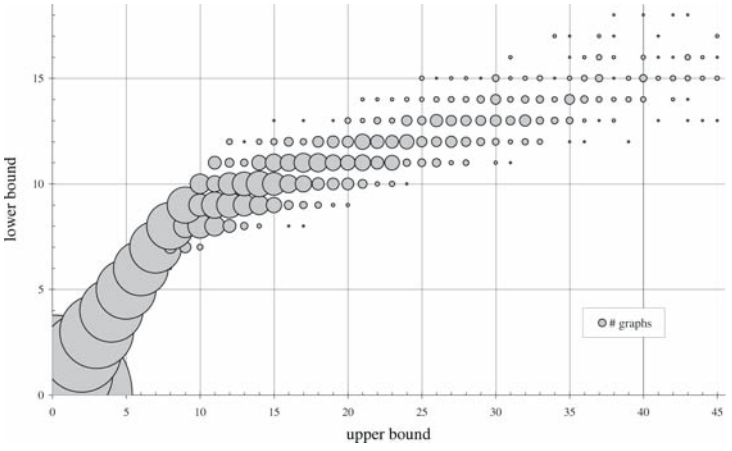


Fig. 8. Final lower bound compared to final upper bound

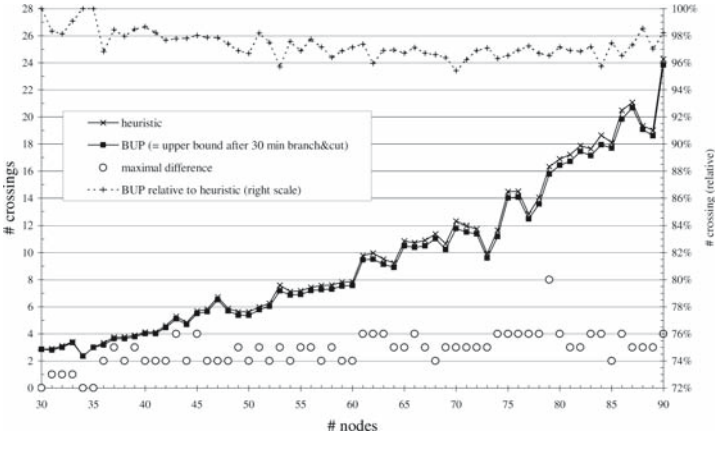


Fig. 9. Average number of crossings achieved by heuristic and CCG

reflects the number of graphs at the data point. The main diagonal shows the graphs solved to proven optimality; virtually all graphs with crossing number up to 7 are solved optimally, and even 5 graphs with crossing number 12 could be solved. In the following, we compare the heuristic solution to this upper bound, which we denote by BUP.

Figure 9 shows the average number of crossings with respect to the number of nodes for both the heuristic and BUP. It turns out that the heuristic solution is very close to BUP. For graphs with up to 45 nodes, we know that CCG could solve almost all instances to optimality, so we can conclude that the heuristic performs very well on these instances. The dashed line shows the relative improvement

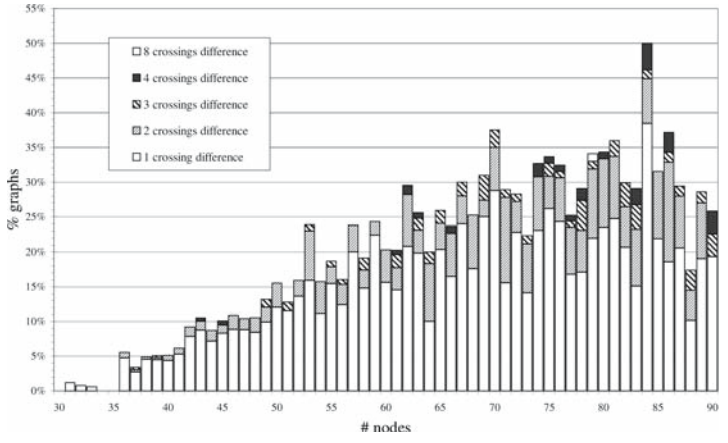


Fig. 10. Absolute improvement of CCG by number of nodes

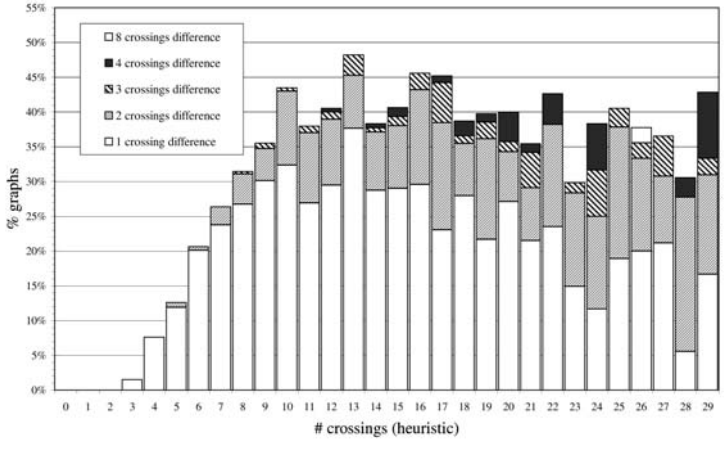


Fig. 11. Absolute improvement of CCG by number of crossings

achieved by CCG; for graphs with up to 45 nodes, this is about 2%. The small circles in the lower part of the diagram give the maximal absolute improvement over the heuristic; the largest improvement was 8 crossings achieved for a graph with 79 nodes. We further observed that the heuristic solution is optimal for virtually all instances with up to 3 crossings, and that the absolute deviation from the upper bound of CCG grows linear for graphs with up to 12 crossings by about 1/12 per crossings.

We consider the absolute improvement achieved by CCG in more detail. Figure 10 shows the percentage of graphs that deviate by 1,2,... crossings from the heuristic solution with respect to the number of nodes. It is interesting to consider

the same statistic with respect to the value of the heuristic solution (see Figure 11). For graphs with up to 3 crossings, the heuristic solves nearly all instances to optimality. Between 10 and 16 crossings, the heuristic solution can be improved for about half of the graphs. For larger crossing numbers, the statistic is not useful, since CCG timed out early.

5 Conclusion

As our experiments show, the algebraic pricing approach is clearly inferior to the combinatorial column generation. The former suffers from the high degree of redundancy in the full ILP and needs to add much too many variables in order to proof optimality. Combinatorial column generation on the other hand, gains a vast improvement compared to solving the ILP over the full variable set.

The second main finding is that the currently best known heuristics are very good in practice and solve many instances to their optimal value. Yet we can see that there is still room for improvement when the crossing numbers get higher.

Acknowledgement

We want to thank Michael Jünger and Christoph Buchheim for many useful discussions on branch-and-cut-and-price theory.

References

1. C. Batini, M. Talamo, and R. Tamassia. Computer aided layout of entity-relationship diagrams. *Journal of Systems and Software*, 4:163–173, 1984.
2. C. Buchheim, M. Chimani, D. Ebner, C. Gutwenger, M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. A branch-and-cut approach to the crossing number problem. *Discrete Optimization*. Submitted for publication.
3. C. Buchheim, D. Ebner, M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. Exact crossing minimization. In P. Eades and P. Healy, editors, *Graph Drawing 2005*, volume 3843 of *LNCS*. Springer-Verlag.
4. G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.
5. G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Computational Geometry: Theory and Applications*, 7(5-6):303–325, 1997.
6. G. Di Battista and R. Tamassia. On-line maintanance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318, 1996.
7. C. Gutwenger and M. Chimani. Non-planar core reduction of graphs. In P. Eades and P. Healy, editors, *Graph Drawing 2005*, volume 3843 of *LNCS*. Springer-Verlag.
8. C. Gutwenger and P. Mutzel. An experimental study of crossing minimization heuristics. In G. Liotta, editor, *Graph Drawing 2003*, volume 2912 of *LNCS*, pages 13–24. Springer-Verlag, 2004.
9. M. Jünger and S. Thienel. The ABACUS system for branch-and-cut-and-price-algorithms in integer programming and combinatorial optimization. *Software: Practice & Experience*, 30(11):1325–1352, 2000.

10. J. Kratochvíl. String graphs. II.: Recognizing string graphs is NP-hard. *Journal of Combinatorial Theory, Series B*, 52(1):67–78, 1991.
11. OGDF – Open Graph Drawing Framework. University of Dortmund, Chair of Algorithm Engineering and Systems Analysis. Web site under construction.
12. H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battista, editor, *Graph Drawing '97*, volume 1353 of *LNCS*, pages 248–261. Springer-Verlag, 1997.
13. P. Turán. A note of welcome. *Journal of Graph Theory*, 1:7–9, 1977.
14. Imrich Vrto. Crossing numbers of graphs: A bibliography. `http://www.ifi.savba.sk/~imrich/crobib.pdf`.
15. S. G. Williamson. Depth-first search and Kuratowski subgraphs. *Journal of the ACM*, 31(4):681–693, 1984.

Goal Directed Shortest Path Queries Using Precomputed Cluster Distances*

Jens Maue[1], Peter Sanders[2], and Domagoj Matijevic[1]

[1] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany
{jensmaue, dmatijev}@mpi-inf.mpg.de
[2] Universität Karlsruhe, 76128 Karlsruhe, Germany
sanders@ira.uka.de

Abstract. We demonstrate how Dijkstra's algorithm for shortest path queries can be accelerated by using precomputed shortest path distances. Our approach allows a completely flexible tradeoff between query time and space consumption for precomputed distances. In particular, sublinear space is sufficient to give the search a strong "sense of direction". We evaluate our approach experimentally using large, real-world road networks.

1 Introduction

Computing shortest paths in graphs (networks) with nonnegative edge weights is a classical problem of computer science. From a worst case perspective, the problem has largely been solved by Dijkstra in 1959 [1] who gave an algorithm that finds all shortest paths from a starting node s using at most $m + n$ priority queue operations for a graph $G = (V, E)$ with n nodes and m edges.

However, motivated by important applications (e.g., in transportation networks), there has recently been considerable interest in the problem of accelerating *shortest path queries*, i.e., the problem to find a shortest path between a source node s and a target node t. In this case, Dijkstra's algorithm can stop as soon as the shortest path to t is found. Furthermore, if the underlying graph does not change too often, it is possible to store some precomputed information that helps to accelerate the queries.

An extreme way to accelerate queries is to precompute *all* shortest path distances $d(s, t)$. However, this is impractical for large graphs since it requires quadratic space and preprocessing time. We explore the question whether it is possible to speed up queries by precomputing and storing only *some* shortest path distances.

Our approach, **P**recomputing **C**luster **D**istances (PCD), is very simple. We partition the graph into k disjoint clusters $\mathcal{V} = V_1 \dot\cup \cdots \dot\cup V_k$ and store the pair of starting and end point as well as the length of the shortest connection between each pair of clusters. This information needs space $\mathcal{O}(k^2)$ and can easily be computed using k executions of Dijkstra's algorithm. Refer to Section 3 for

* Partially supported by DFG grant SA 933/1-2.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 316–327, 2006.

details. In Section 4 we explain how this information can be used to compute upper and lower bounds for $d(s,t)$ at query time. These bounds can be used to prune the search. Section 5 completes the description of our algorithmic approach by presenting some fast partitioning heuristics. Most of them require no geometric information.

We then evaluate PCD in Section 6 using large, real-world road networks as they are used in car navigation systems. It is both geometrically plausible, and supported by our experiments that the speedup compared to Dijkstra's basic algorithm scales with $\sqrt{k}$, i.e., we get a flexible tradeoff between space consumption and query time that already yields useful acceleration for very small space consumption. To the best of our knowledge this is the first sublinear space acceleration technique that yields speedups $\gg 2$. Perhaps the most interesting application of PCD would be a combination with previous techniques that use linear space and deliver very high speedups but have no sense of goal direction [2, 3, 4]. Section 7 explores further future perspectives on PCD.

2 Related Work

Like most approaches to shortest paths with nonnegative edge weights, our method is based on Dijkstra's algorithm [1]. This algorithm maintains *tentative distances* $d[v]$ that are initially ∞ for *unreached nodes* and 0 for the source node s. A priority queue stores *reached* nodes ($d[v] < \infty$) using $d[v]$ as the priority. In each iteration, the algorithm removes the closest node u, *settles* it because $d[u]$ now is the shortest path distance $d(s,u)$, and *relaxes* the edges leaving u — if $d[u] + c((u,v)) < d[v]$, then $d[v]$ is decreased to $d[u] + c((u,v))$. When Dijkstra's algorithm is used for s–t shortest path queries, it can stop as soon as t is settled. When we talk about *speedup*, we mean the ratio between the complexity of this algorithm and the complexity of the accelerated algorithm.

A classical technique that gives a speedup of around two for road networks is *bidirectional search* which simultaneously searches forward from s and backwards from t until the search frontiers meet. Our implementation optionally combines PCD with bidirectional search.

Another classical approach is goal direction via *A* search*: a lower bound $\underline{d}(v,t)$ is used to direct the search towards the goal. This method can be interpreted as defining *vertex potentials* $p(v) := \underline{d}(v,t)$ and corresponding *reduced edge weights* $c^*((u,v)) := c((u,v)) + p(v) - p(u)$. Originally, lower bounds were based on the Euclidean distance from u to t and the "fastest street" in the network. Besides requiring geometric information, this very conservative bound is not very effective when searching *fastest* routes in road networks. In this respect, a landmark result was recently obtained by Goldberg and Harrelson [5, 6]. *Landmark A** uses precomputed distances to k *landmarks* to obtain lower bounds. Already around $k = 16$ carefully selected landmarks are reported to yield speedups around 70. Combined with a sophisticated implementation of *reach based routing* [2] this

Table 1. Tradeoff between space, preprocessing time and query time depending on the choice of the parameter k for different speedup techniques, k is the number of clusters except for the landmark method. We assume road networks with n nodes and $\Theta(n)$ edges. $D(n)$ is the execution time of Dijkstra's algorithm, B is the number of border nodes. Preprocessing time does not include the time for partitioning the graph. (For the landmark method this is currently dominating the preprocessing time.)

Method		space	preprocessing	query
Edge Flags	[10, 9]	$\Theta(n \cdot k)$ Bits	$\Theta(B \cdot D(n))$	? ($\approx D(n)/2000@k = 128$)
Separator Hierarchy	[11]	$\Omega\left(B^2/k\right)$	$\Theta(B \cdot D(n/k))$	$\Omega\left(D(n/k) + B^2/k\right)$
Landmarks	[5]	$\Theta(n \cdot k)$	$\Theta(k \cdot D(n))$	? ($\approx D(n)/70@k = 16$)
PCD		$\Theta\left(k^2 + B\right)$	$\Theta(k \cdot D(n))$	$\Omega\left(D(n/\sqrt{k})\right)$

method currently yields the fastest query times for large road networks [4]. The main drawback of landmarks is the space consumption for storing $\Theta(kn)$ distances. This is where PCD comes into play, which already yields a very strong sense of direction using much less space than landmarks. Still, the story is a bit more complicated than it sounds. We first considered PCD in March 2004 during a workshop on shortest paths in Karlsruhe. However, it turned out the lower bounds obtainable from PCD are *not* usable for A* search because they can lead to negative reduced edge weights.[1] Apparently, Goldberg et al. independently made similar observations [7]. The present paper describes a solution: use PCD to obtain *both* upper and lower bounds to prune search. The basic idea for PCD was presented in a Dagstuhl Workshop [8] (slides only).

There are several other speedup techniques that are based on partitioning the network into k clusters [9, 10, 11]. However, the preprocessing time required by these methods not only depends on k and the network size but also on the number B of border nodes, i.e., the number of nodes that have nodes from other clusters as neighbors. Furthermore, all of these methods need more space than PCD. Table 1 summarizes the tradeoff between space, preprocessing time and query time of these methods. Note that usually there is no worst case bound on the query time known. The given functions make additional assumptions that seem to be true for road network.

An interesting way to classify speedup techniques is to look at two major ways to prune the search space of Dijkstra's algorithm. A* search and PCD direct the search towards the goal. Other algorithms skip nodes or edges that are only relevant for short distance paths. In particular, reach based routing [2, 4] and highway hierarchies [3] achieve very high speedups without any sense of goal direction. Other techniques like edge flags [10, 9], geometric containers [12], and to some extent the landmark method show both effects. Therefore, we expect a major future application of PCD to augment highway hierarchies and reach based routing with a space efficient way to achieve goal direction.

[1] This was pointed out by Rolf Möhring.

3 Preprocessing

Suppose, the input graph has been partitioned into clusters $\mathcal{V} = V_1 \dot\cup \cdots \dot\cup V_k$. We want to compute a complete distance table that allows to look up

$$d(V_i, V_j) := \min_{s \in V_i, t \in V_j} d(s, t) \tag{1}$$

in constant time. We can compute $d(S, V_i)$ for a fixed cluster S and $i = 1, \ldots, k$ using just one single source shortest path computation: add a new node s' connected to all border nodes of S using zero weight edges. Perform a single source shortest path search starting from s'. Fig. 1 illustrates this approach.

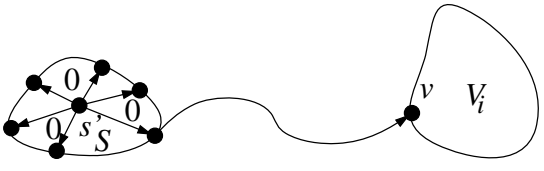


Fig. 1. Preprocessing connections from cluster S

The following simple lemma shows that this suffices to find all connections from S to other clusters.

Lemma 1. $d(S, V_i) = \min_{v \in V_i} d(s', v)$.

The proof is almost obvious. We include it nevertheless since several other speedup techniques require shortest path computations from all *border nodes* of all partitions.

Proof. We have $d(S, V_i) \leq \min_{v \in V_i} d(s', v)$ as any shortest path $(s', s, \ldots, v \in V_i)$ found during the search from s' contains a path $(s, \ldots, v)$ connecting the clusters S and V_i.

On the other hand, there cannot be a shorter connection from S to V_i. Assume the contrary, i.e., there is a path $(s \in S, \ldots, u \in S, u' \notin S, \ldots, v' \in V_i)$ with $d(s, v') < \min_{v \in V_i} d(s', v)$. Then $(s', u, \ldots, v')$ would constitute a shorter connection from s' to v', which is a contradiction. □

Repeating this procedure for every cluster yields the complete distance table. In addition, for each pair V_i, V_j we store a start point $v_i \in V_i$ and an end point $v_j \in V_j$ such that $d(v_i, v_j) = d(V_i, V_j)$.

4 Queries

We describe the query algorithm for bidirectional search between a source s and a target t. To allow sublinear execution time, the algorithm assumes that the

distance values and predecessor information used by Dijkstra's algorithm have been initialized properly during preprocessing. Let S and T denote the clusters containing s and t respectively. The search works in two phases.

In the first phase, we perform ordinary bidirectional search from s and t until the search frontiers meet, or until $d(s, s')$ and $d(t', t)$ are known, where s' is the first border node of S settled in the forward search, and t' the first border node of T settled in the backward search.

For the second phase we only describe forward search—backward search works completely analogously. The forward search grows a shortest path tree using Dijkstra's algorithm, additionally maintaining an upper bound $\hat{d}(s,t)$ for $d(s,t)$, and computing lower bounds $\underline{d}(s,w,t)$ for the length of any path from s via w to t. The search is pruned using the observation that the edges out of w need not be considered if $\underline{d}(s,w,t) > \hat{d}(s,t)$. Phase two ends when the search frontiers of forward and backward search meet. In a cleanup phase, the distance values and predecessor values changed during the search are reset to allow the proper initialization for the next query. This can be done efficiently by maintaining a stack of all nodes ever reached during the search. It remains to explain how $\hat{d}(s,t)$ and $\underline{d}(s,w,t)$ are computed.

The upper bound is updated whenever a shortest path to a node $u \in U$ is found such that u is the starting point of the shortest connection between clusters U and T. Let t_{UT} denote the stored end point of the precomputed shortest connection from U to T. Then we have

$$d(s,t) \leq d(s,u) + \underbrace{d(u,t_{UT})}_{=d(U,T)} + d(t_{UT},t) \quad . \tag{2}$$

The value of $d(s,u)$ has just been found by the forward search, and $d(U,T)$ and t_{UT} have been precomputed; thus, the sum in Equation (2) is defined if $d(t_{UT},t)$ is known, i.e. if t_{UT} has already been found by the backward search. Otherwise, we use an upper bound of the diameter of T instead of $d(t_{UT},t)$ (see Section 5). $\hat{d}(s,t)$ is the smallest of the bounds from Equation (2) encountered during forward or backward search. The following lemma establishes the lower bound.

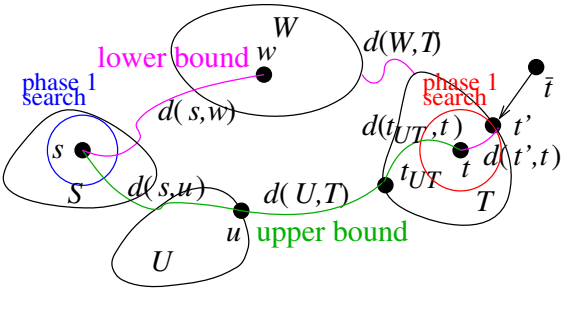


Fig. 2. Constituents of upper and lower bounds for $d(s,t)$

Lemma 2. *Consider any node $w \notin T$. Let W denote the cluster containing w, and let* $\text{Border}(T) := \{t' \in T : \exists \bar{t} \notin T : (\bar{t}, t') \in E\}$ *denote the* border *of T, then any shortest path from s via w to t has a length of at least*

$$\underline{d}(s, w, t) := d(s, w) + d(W, T) + \min_{t' \in \text{Border}(T)} d(t', t) \ . \tag{3}$$

Proof. We show that $\underline{d}(s, w, t) \leq d(s, w) + d(w, t)$ or, equivalently, $d(W, T) + \min_{t' \in \text{Border}(T)} d(t', t) \leq d(w, t)$. Consider a shortest path $P = (w, \ldots, t'', \ldots t)$ from w to t where t'' denotes the first node on this path that is in cluster T. We have $d(w, t) = d(w, t'') + d(t'', t)$. Since $(w, \ldots, t'')$ is a connection from W to T we have $d(w, t'') \geq d(W, T)$. Furthermore, since t'' is a border node of T, we have $d(t'', t) \geq \min_{t' \in \text{Border}(T)} d(t', t)$. □

$\underline{d}(s, w, t)$ can be computed efficiently as $d(s, w)$ has been found by forward search, W can be found by storing a cluster identifier with each node, $d(W, T)$ has been precomputed, and $\min_{t' \in T} d(t', t)$ has been determined by the end of the first phase. Fig. 2 depicts the situation for computing upper and lower bounds.

Space Efficient Implementation

The algorithm described above is straight forward to implement using space $\mathcal{O}(k^2 + n)$. This can be reduced to $\mathcal{O}(k^2 + B)$ where B is the number of *border nodes* that have neighbors in other cluster. The problem is that when settling a node u, we need to know its cluster id. The key observation is that clusters only change at border nodes so that it suffices to store the cluster ids of all B border nodes in a hash table.

5 Partitioning

For any set $C = \{c_1, \ldots, c_k\} \subset V$ of k distinct *centers*, assigning each node $v \in V$ to the center closest to it results in a *k-center clustering*. Here, the *radius* $r(C_i)$ of a cluster C_i denotes the distance from its center c_i to the furthest member.[2] A k-center clustering can be obtained using *k'-oversampling*: a sample set C' of k' centers is chosen randomly from V for some $k' \geq k$, and a k'-center clustering is computed for it by running one single source shortest path search from a dummy node connected with each center by a zero-weight edge. Then, clusters are deleted successively until k clusters are left. A cluster C_i is deleted by removing the corresponding center c_i from C' and reassigning each member of C_i to the center now closest to it. This amounts to a shortest path search from the neighboring clusters which now grow into the deleted cluster. This process terminates with a $(k'-1)$-clustering. There are several ways to choose a cluster for deletion: in Section 6 results are shown for the MinSize and the MinRad heuristics, which

[2] Note that for undirected graphs, $2\,r(C_i)$ is an upper bound of the diameter of C_i since $d(u, v) \leq d(u, c_i) + d(c_i, v) \leq 2\,r(C_i)$ for any $u, v \in C_i$. This bound can be used in the query as shown in Section 4. For directed graphs we can use $r(C_i) + \max_{c \in C_i} d(c, c_i)$ for the same purpose.

choose the cluster of minimum size and minimum radius respectively, and the MinSizeRad heuristic, which alternates between the former two. It can be shown that partitioning using the MinSize heuristic searches $\mathcal{O}(n \log \frac{k'}{k})$ nodes using Dijkstra's algorithm and hence has negligible cost compared to computing cluster distances which requires searching $\mathcal{O}(nk)$ nodes.[3] The radius of a cluster affects the lower bounds of its members, and it seems that a good partitioning for PCD has clusters of similar size and a low average radius. Oversampling indeed keeps a low average radius since deleted clusters tend to be distributed to neighbors of lower radius. However, a higher radius is acceptable for smaller clusters since the lower bound is not worsened for too many nodes then, whereas a low radius allows a bigger size. Both values can be combined into the *weighted average radius*, in which the single radii are weighted with their clusters' sizes.

Our k-center heuristics are compared with a simple partitioning based on a rectangular grid and with Metis [13]. Metis was originally intended for parallel processing where partitions should have close to equal size and small boundaries in order to reduce communication volume.

6 Experiments

The PCD algorithms were implemented in C++ using the static graph data structure from the C++ library LEDA 5.1 [14] and compiled with the GNU C++ compiler 3.4 using optimization level `-O3`. All tests were performed on a 2.4 GHz AMD opteron with 8 GB of main memory running Linux (kernel 2.6.11). We use the same input instances as in [3]—industrial data for the road network of Western Europe and freely available data for the US [15]. Table 2 gives more details. In order to make experiments with a wide range of parameter choices, we used subgraphs of these inputs. Unless otherwise noted, the experiments make the following common assumptions: we use undirected graphs in order to be able to use simple implementations of the partitioning heuristics. (Our PCD implementation works for general directed graphs.) Edge weights are estimated travel

Table 2. The graph instances used for experiments

Instance	n	m	Description
DEU	4 375 849	5 483 579	Germany
SCA	2 453 610	2 731 129	Sweden & Norway
IBE	872 083	1 177 734	Spain & Portugal
SUI	630 962	771 694	Switzerland
MID	5 246 822	6 494 670	Midwest (IL,IN,IA,KS,MI,MN,NE,ND,OH,SD,WI)
WES	4 429 488	5 296 150	West (CA, CO, ID, MT, NV, OR, UT, WA, WY)
MAT	2 226 138	2 771 948	Middle Atlantic (DC, DE, MD, NJ, NY, PA)
NEN	896 115	1 058 481	New England (CT, ME, MA, NH, RI, VT)

[3] Throughout this paper $\log x$ stands for $\log_2 x$.

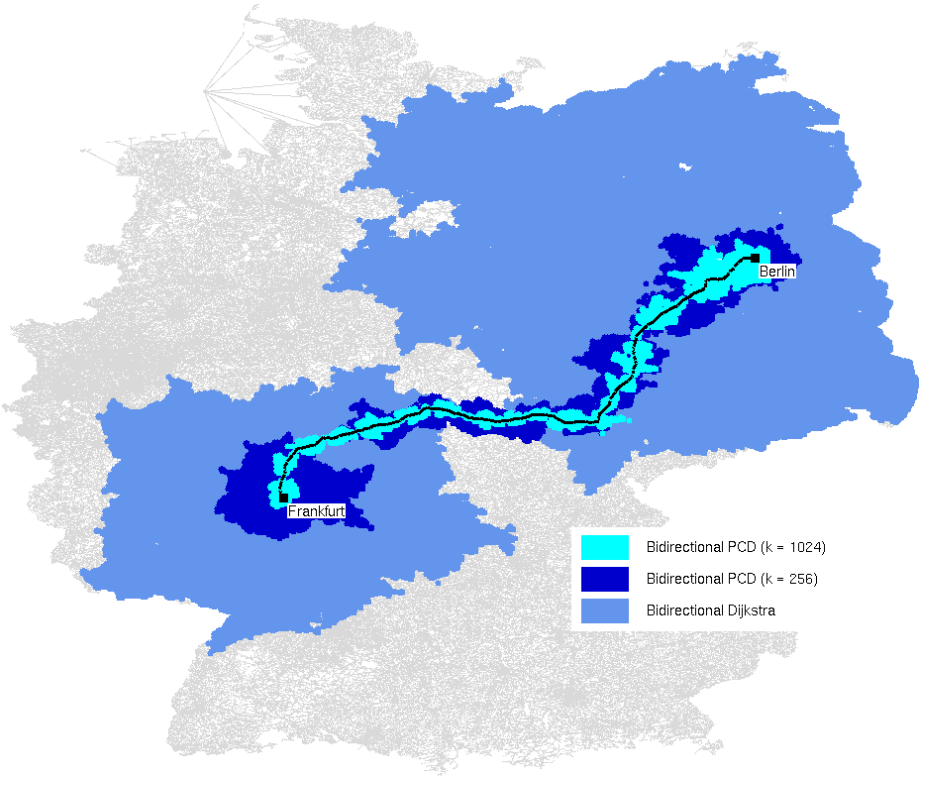


Fig. 3. The search space for a sample query from Frankfurt to Berlin

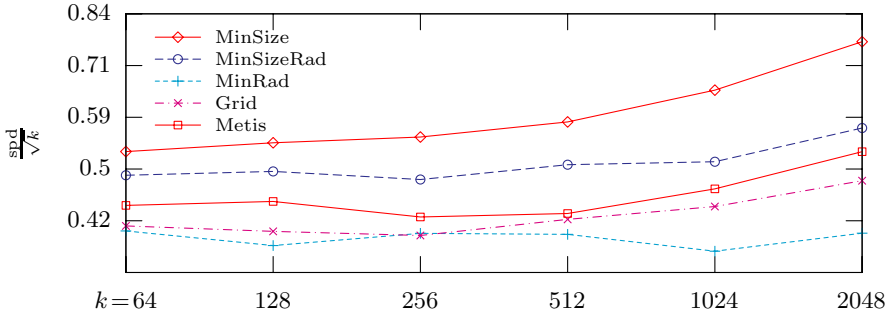


Fig. 4. Scaled speedups for bidirectional PCD depending on the method of clustering

times. The default instance is the road network of Germany (`DEU`). Partitioning is done using $k \log k$-oversampling with the MinSize heuristic. The speedup is the ratio between the number of nodes settled by Dijkstra's unidirectional algorithm and by the accelerated algorithm. The given values for queries are averages over 1000 random query pairs.

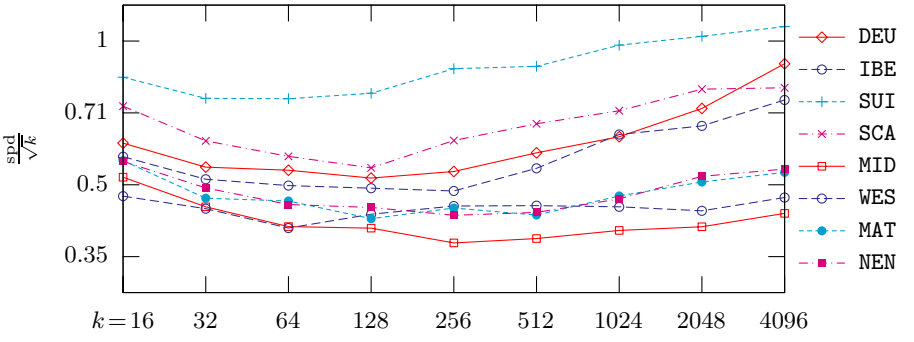


Fig. 5. Scaled speedups for bidirectional PCD and different inputs

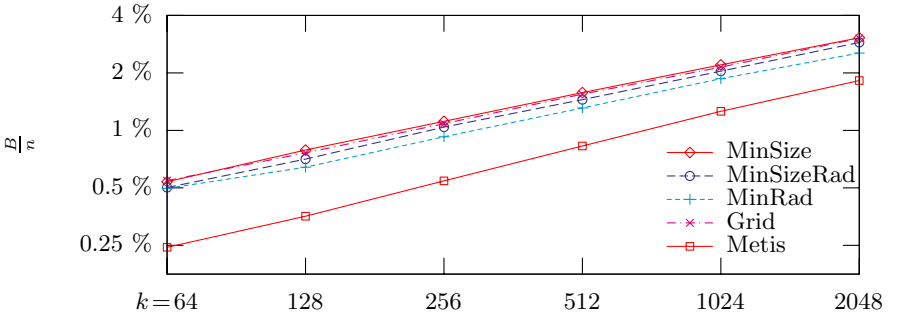


Fig. 6. Relative number of border nodes B for DEU depending on the clustering method The k-center heuristics all use $k \log k$-oversampling.

Fig. 3 gives an example for the shape of the search space explored by PCD. As to be expected, the search space is a kind of halo around the shortest path that gets narrower when k is increased. A somewhat optimistic approximation of the observed behavior is that the clusters near the shortest path are explored. The shortest path will intersect $\mathcal{O}(\sqrt{k})$ clusters of size $\mathcal{O}(n/k)$ on the average, i.e., the intersected clusters contain $\mathcal{O}(n/\sqrt{k})$ nodes. Since Dijkstra's algorithm visits $\Theta(n)$ nodes on the average, we expect a speedup of $\mathcal{O}(\sqrt{k})$.

This hypothesis is verified in Fig. 4, which compares the different partitioning methods from Section 5. Scaled by $\sqrt{k}$, the speedups describe fairly flat lines as our hypothesis suggests. The MinSize heuristic yields the highest speedups, while the other heuristics perform worse though they also yield fairly small average radii as mentioned before. Since MinRad keeps deleting clusters in urban areas, it ends up with clusters of similar radii but differing sizes, wheras MinSize deletes clusters regardless of their size yielding a good ratio of radius and size. Interestingly, for the minimum size rule the speedups appear to scale even better than $\Theta(\sqrt{k})$. This observation is confirmed in Fig. 5 for further instances.

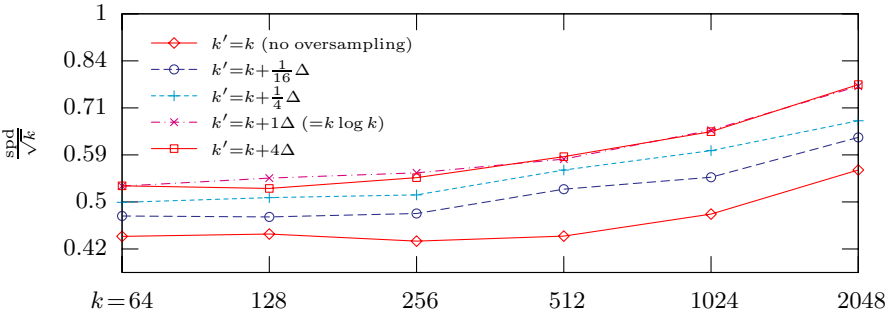


Fig. 7. Speedups for bidirectional PCD with k'-oversampling using different values for k', tested on DEU. $\Delta = k \log k - k$ denotes the difference between $k \log k$ and k.

Table 3. Performance of PCD for several graph instances and selected values of k. prep.= preprocessing time. B = total number of border nodes. spd = speedup. settled = number of settled nodes. t = query time.

graph	k	$\frac{B+k^2}{n}$	prep.	PCD unidirectional				PCD bidirectional			
			[min]	t[ms]	spd	settled	spd	t[ms]	spd	settled	spd
DEU	2^4	< 0.01	2.6	2491	2.2	1 171 110	2.2	2114	2.5	1 028 720	2.4
	2^6	0.01	11.1	1410	3.6	749 870	3.3	971	5.2	553 863	4.3
	2^8	0.03	35.0	677	7.7	443 832	5.9	422	12.3	295 525	8.5
	2^{10}	0.26	123.0	256	21.2	199 663	13.2	157	35.0	127 604	20.2
	2^{12}	3.88	558.2	110	70.5	86 401	37.1	62	114.9	50 417	57.4
SCA	2^{10}	0.44	60.7	173	21.3	136 365	14.2	81	36.5	70 766	22.9
IBE	2^{10}	1.25	11.7	43	16.5	51 716	13.7	20	24.7	26 591	20.4
SUI	2^{10}	1.71	11.1	20	20.8	25 070	19.1	9	31.1	12 848	31.4
MID	2^{10}	0.22	89.2	287	15.4	326 112	10.3	223	18.5	242 153	12.8
WES	2^{10}	0.25	80.8	238	15.2	227 768	11.3	169	19.0	159 409	14.4
MAT	2^{10}	0.50	35.5	101	17.1	114 702	12.0	76	21.1	83 577	15.2
NEN	2^{10}	1.21	11.0	39	15.1	49 259	11.6	28	19.3	34 625	15.0

As expected, Metis finds clusters with smaller borders as can be seen in Fig. 6. However, since the percentage of border nodes is very small even for oversampling, this appears to be less relevant.

Oversampling using the MinSize deletion heuristic is tested for several values of k' in Fig. 7. Starting from $k' = k$, which means just choosing k centers randomly, increasing k' up to $k \log k$ increases the speedup significantly. However, no considerable further improvement is visible for $k' > k \log k$.

Table 3 summarizes our results for several instances and different numbers of clusters k calculated by $k \log k$-oversampling.The highest speedup in our tests is 114.9, while speedups of more than 30 are still achieved while the number of cluster pairs k^2 remains significantly below n and the total number of border nodes B is negligible. The speedups for the US instances are smaller probably because the available data provides less information on speed differences

of roads, while using travel distances rather than travel times seems to yield even smaller speedups. The reason is that edges off the fastest path often have high travel time values compared to edges on the path, so that pruning happens earlier.

The speedups in terms of query time are higher than those in terms of settled nodes. The main reason for this lies in the size of the priority queue in PCD, which affects the average time for queue operations: after a small ball around the source is searched, most of the nodes on its boundary are pruned, and the search frontier turns into a small corridor around the shortest path. Then, the queue holds a number of nodes which remains roughly constant until finishing. The queue size corresponds to the width of the corridor, so the average queue size is in $\mathcal{O}(\sqrt{\frac{n}{k}})$, while in Dijkstra's algorithm the queue keeps growing and holds $\mathcal{O}(n)$ nodes on the average. Since the average time of an operation is logarithmic in the size and the number of operations is linear in the number of settled nodes, the relation between the speedup in terms of query time and that in terms of settled nodes is roughly $\frac{\log n}{\log \frac{n}{k}}$.

7 Conclusion

We have demonstrated that PCD can give route planning in road networks a strong sense of goal direction leading to significant speedups compared to Dijkstra's algorithm using only sublinear space. The most obvious task for future work is to combine this with speedup techniques that have no sense of goal direction [2, 3, 4]. There are good reasons to believe that one would get a better tradeoff between speedup and space consumption than any previous method.

As a standalone method, PCD is interesting because its unidirectional variant also works for networks with time dependent edge weights such as public transportation networks or road networks with information when roads are likely to be congested: simply use an optimistic estimate for lower bounds and a pessimistic estimate for the upper bounds. Most other speedup techniques do not have such an obvious generalization.

PCD itself could be improved by giving better upper and lower bounds. Upper bounds are already very good and can be made even better by splitting edges crossing a cluster border such that the new node has equal distance from both cluster centers. For example, this avoids cluster connections that use a small road just because the next entrance from a motorway is far away from the cluster border. While this is good if we want to approximate distances, initial experiments indicate that it does not give additional speedup for exact queries. The reason is that *lower bounds* have a quite big error related to the cluster diameters. Hence, better lower bounds could lead to significant improvements. For example, can one effectively use all the information available from precomputed distances between clusters explored during bidirectional search?

It seems that a good partitioning algorithm should look for clusters of about equal size and low diameter; perhaps, these might be two of the main parameters for an easily computable objective function. In the literature there is a lot of

work on approximation algorithms for various k-center problems. It might be interesting to adapt some of the proposed algorithms to our situation.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
2. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: 6th Workshop on Algorithm Engineering and Experiments. (2004)
3. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: 13th European Symposium on Algorithms. Volume 3669 of LNCS., Springer (2005) 568–579
4. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering & Experiments, Miami (2006)
5. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. (2005) 156–165
6. Goldberg, A.V., Werneck, R.F.: An efficient external memory shortest path algorithm. In: Workshop on Algorithm Engineering & Experiments. (2005) 26–40
7. Goldberg, A.: personal communication. (2005)
8. Sanders, P.: Speeding up shortest path queries using a sample of precomputed distances. Workshop on Algorithmic Methods for Railway Optimization, Dagstuhl, June 2004, slides at `http://www.dagstuhl.de/04261/` (2004)
9. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up Dijkstra's algorithm. In: 4th International Workshop on Efficient and Experimental Algorithms. (2005)
10. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In: Münster GI-Days. (2004)
11. Schulz, F., Wagner, D., Zaroliagis, C.D.: Using multi-level graphs for timetable information. In: 4th Workshop on Algorithm Engineering and Experiments. Volume 2409 of LNCS., Springer (2002) 43–59
12. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: 11th European Symposium on Algorithms. Volume 2832 of LNCS., Springer (2003) 776–787
13. Karypis, G., Kumar, V.: Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. `http://www-users.cs.umn.edu/~karypis/metis/` (1995)
14. Mehlhorn, K., Näher, S.: The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press (1999)
15. U.S. Census Bureau: UA Census 2000 TIGER/Line files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html` (2002)

Author Index