LNCS 7659

Guy Even
Dror Rawitz (Eds.)

Design and Analysis of Algorithms

First Mediterranean Conference on Algorithms, MedAlg 2012
Kibbutz Ein Gedi, Israel, December 2012
Proceedings

Springer

Guy Even   Dror Rawitz (Eds.)

Design and Analysis of Algorithms

First Mediterranean Conference
on Algorithms, MedAlg 2012
Kibbutz Ein Gedi, Israel, December 3-5, 2012
Proceedings

Springer

Volume Editors

Guy Even
Dror Rawitz

Tel-Aviv University
School of Electrical Engineering
Tel-Aviv 67789, Israel
E-mail: {guy, rawitz}@eng.tau.ac.il

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-34861-7 e-ISBN 978-3-642-34862-4
DOI 10.1007/978-3-642-34862-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012951175

CR Subject Classification (1998): F.2.1-2, G.2.2-3, E.1, G.1.1-2, G.1.6, G.1.10, I.2.8, I.6.1, I.6.3, G.4

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

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

Printed on acid-free paper

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

Preface

This volume contains the papers presented at MedAlg 2012: First Mediterranean Conference on Algorithms held during December 3–5, 2012, in Kibbutz Ein Gedi.

There were 44 submissions. Each submission was reviewed by at least two, and on average three, Program Committee members. The committee decided to accept 18 papers.

Four invited speakers volunteered to present talks at the conference: Yosi Azar, Artur Czumaj, Yishay Mansour, and David Peleg.

We would like to thank the Program Committee members and the reviewers for their help in selecting the papers in these proceedings. We thank the authors for submitting their papers. We thank Moti Medina for his devoted service as our Publicity Chair. Gabriel Scalosub served as the Local Chair and took close care of organizing the meeting.

We would like to thank the financial support of I-CORE ALGO: The Israeli Center of Research Excellence in Algorithms headed by Yishay Mansour. Special thanks to Sarit Cohen Shalev from I-CORE ALGO for the assitance in organizing the conference.

The EasyChair system was used for the submission of the papers, their reviewing, and the generation of these proceedings.

September 2012

Guy Even
Dror Rawitz

Organization

Program Committee

Hannah Bast	University of Freiburg, Germany
Niv Buchbinder	Open University of Israel
Matthias Englert	University of Warwick, UK
Guy Even	Tel Aviv University, Israel
Pierre Fraigniaud	CNRS and University of Paris 7, France
Chien-Chung Huang	Humboldt-Universität zu Berlin, Germany
Giuseppe Italiano	Rome University "Tor Vergata", Italy
Ilan Newman	Haifa University, Israel
Dror Rawitz	Tel Aviv University, Israel
Liam Roditty	Bar-Ilan University, Israel
Christian Scheideler	University of Paderborn, Germany
Baruch Schieber	IBM T.J. Watson Research Center, USA
Tami Tamir	The Interdisciplinary Center, Israel
Anke van Zuylen	Max Planck Institute for Informatics, Germany
Berthold Vöcking	RWTH Aachen University, Germany
Peter Widmayer	ETH Zurich, Switzerland
Christos Zaroliagis	Computer Technology Institute and University of Patras, Greece

Steering Committee

Pankaj Agarwal	Duke University, USA
Imrich Chlamtac	University of Trento, Italy
Alberto Marchetti-Spaccamela	Sapienza University of Rome, Italy
David Peleg	Weizmann Institute, Israel
Michael Segal	BGU, Israel
Paul Spirakis	University of Patras, Greece
Roger Wattenhofer	ETH, Switzerland

Local Arrangements Chair

Gabriel Scalosub	Ben-Gurion University of the Negev, Israel

Publicity Chair

Moti Medina	Tel Aviv University, Israel

Additional Reviewers

Abed, Fidaa
Aharoni, Ron
Antoniadis, Antonios
Bansal, Nikhil
Brandes, Ulrik
Cacchiani, Valentina
Chatzigiannakis, Ioannis
Ediger, David
Epstein, Leah
Foschini, Luca
Giakkoupis, George
Grigni, Michelangelo
Haeupler, Bernhard
Hassidim, Avinatan
Hermelin, Danny
Hruz, Tomas
Hüllmann, Martina
Keller, Orgad
Kniesburges, Sebastian
Kontogiannis, Spyros
Lovett, Shachar
Maheshwari, Anil
Marathe, Madhav
Medina, Moti
Michail, Panagiotis
Mihalak, Matus
Nielsen, Frank
Nussbaum, Yahav
Ogierman, Adrian
Ott, Sebastian
Patt-Shamir, Boaz
Rabinovich, Yuri
Regnier, Mireille
Sach, Benjamin
Saket, Rishi
Schmid, Stefan
Schwartz, Roy
Shende, Sunil
Smorodinsky, Shakhar
Solomon, Shay
Tsichlas, Kostas
Viola, Emanuele
Wahlström, Magnus
Wong, Prudence W.H.

Table of Contents

Detecting Approximate Periodic Patterns

Amihood Amir[1,2,⋆], Alberto Apostolico[3,4,⋆⋆], Estrella Eisenberg[1],
Gad M. Landau[5,6,⋆⋆⋆], Avivit Levy[7,8,†], and Noa Lewenstein[9]

[1] Department of Computer Science, Bar-Ilan University,
Ramat-Gan 52900, Israel
amir@cs.biu.ac.il
[2] Department of Computer Science, Johns Hopkins University,
Baltimore, MD 21218
[3] College of Computing, Georgia Institute of Technology,
801 Atlantic Drive, Atlanta, GA 30318, USA
axa@cc.gatech.edu
[4] Dipartimento di Ingegneria dell' Informazione, Università diPadova,
Via Gradenigo 6/A, 35131 Padova, Italy
[5] Department of Computer Science, University of Haifa,
Mount Carmel, Haifa 31905, Israel
landau@cs.haifa.ac.il
[6] Department of Computer Science and Engineering,
Polytechnic Institute of New York University,
6 Metrotech Center, Brooklyn, NY 11201
[7] Department of Software Engineering, Shenkar College,
12 Anna Frank, Ramat-Gan, Israel
avivitlevy@shenkar.ac.il
[8] CRI, Haifa University, Mount Carmel, Haifa 31905, Israel
[9] Netanya College, Netanya, Israel
noa.lewenstein@gmail.com

Abstract. Given $\epsilon \in [0, 1)$, the *ϵ-Relative Error Periodic Pattern Problem (REPP)* is the following:

INPUT: An n-long sequence S of numbers $s_i \in \mathbb{N}$ in increasing order.
OUTPUT: The longest ϵ-relative error periodic pattern, i.e., the longest subsequence $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ of S, for which there exists a number p such that the absolute difference between any two consecutive numbers in the subsequence is at least p and at most $p(1 + \epsilon)$.

The best known algorithm for this problem has $O(n^3)$ time complexity. This bound is too high for large inputs in practice. In this paper we give a new algorithm for finding the longest ϵ-relative error periodic pattern (the REPP problem). Our method is based on a transformation

⋆ Partly supported by NSF grant CCR-09-04581, ISF grant 347/09, and BSF grant 2008217.
⋆⋆ Partly supported by BSF grant 2008217.
⋆⋆⋆ Partly supported by the National Science Foundation Award 0904246, Israel Science Foundation grant 347/09, Yahoo, Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG.
† Partly supported by ISF grant 347/09.

of the input sequence into a different representation: the ϵ-*active maximal intervals list* L, defined in this paper. We show that the transformation of S to the list L can be done efficiently (quadratic in n and linear in the size of L) and prove that our algorithm is linear in the size of L. This enables us to prove that our algorithm works in sub-cubic time on inputs for which the best known algorithm works in $O(n^3)$ time. Moreover, though it may happen that our algorithm would still be cubic, it is never worse than the known $O(n^3)$-algorithm and in many situations its complexity is $O(n^2)$ time.

1 Introduction

Many real world phenomena have a particular type of event that repeats periodically during a certain period of time. Examples of highly periodic events include road traffic peaks [5], load peaks on web servers [6], monitoring events in computer networks [1] and many others. Finding periodicity in real-world data often leads to useful insights by shedding light on the structure of the data, and giving a basis to predicting future events. Moreover, in some applications periodic patterns can point out a problem. In a computer network, for example, repeating error messages can indicate a misconfiguration, or even a security intrusion such as a port scan [4].

However, such periodic patterns typically occur temporarily, and need not persist throughout the whole period of time covered by the event log. Since short (approximate) periodic patterns will appear in any sequence of random events that is long enough, a periodic pattern is more interesting the more repetitions it contains. Therefore, finding the longest periodic pattern, which contains the largest number of repetitions of the same event, is of interest.

The input data we are given in such cases can be modeled as a sequence of events, each associated with a timestamp. Since we study repetition of events of the same type, we can treat each type of event separately. Thus, the input consists of a sequence S of n distinct numbers $s_1, s_2, \ldots, s_n$ in increasing order, which are the times at which an event of a particular type has occurred. A periodic pattern then corresponds to an (approximate) arithmetic progression in this sequence. For a given sequence of numbers $s_1, s_2, \ldots, s_k$, $s_i \in S$, the differences $s_{i+1} - s_i$ of consecutive numbers in the sequence are called *periods*. In most real-world applications, the timestamps which are given as input are imprecise. Hence, no exact arithmetic progression may be present in the input, and it is necessary to allow some slack in the periodic patterns.

Though finding approximate periodic patterns is a widely studied subject in data mining (e.g. [3, 7–9]), bioinformatics (for example tandem repeats in genomic sequences), or astronomy (Lomb-Scargle periodograms), the input is rather different from the one considered in this paper and applying existing methods in these fields directly or through data conversion is inadequate or inefficient. In this paper, we follow the recent framework of [2] to the study of approximate periodic arithmetic progressions.

If the approximate period of the patterns of interest are known, then the periodic patterns that we want to find can be an *absolute error periodic pattern.* In this problem we are given $p_{min} > 0$ and $p_{max} \geq p_{min}$ and we want to find a subsequence $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ of the input sequence S, such that the absolute difference between any two consecutive numbers s_{i_j}, s_{i_j+1} is at least p_{min} and at most p_{max}. This type of pattern is useful if one is interested in a few particular periods: For example, in a sequence of log file entries, one might be interested only in events which occur (approximately) every hour, every day, every week, or every month. In this case, one can define an interval of acceptable distances for each period of interest. For a given interval $[p_{min}, p_{max}]$, it is not difficult to compute the longest absolute error periodic pattern in linear time. Such an algorithm is described in [2].

Finding periodic patterns with arbitrary periods, where no period (or range of periods) of interest is given in advance, is more challenging. In this case, there are several choices for defining a valid pattern. The definition of [2] bounds the ratio between the longest and the shortest distance between consecutive events in the sequence. In contrast to an absolute error bound, this formalization has the natural property that it is scale invariant. The definition is as follows:

Definition 1. [relative error periodic pattern]
Given[1] $\epsilon \in [0, 1)$ *and a sequence* S *of numbers in increasing order, an* ϵ-relative error periodic pattern *is a subsequence* $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ *of* S*, for which there exists a number* p *such that the absolute difference between any two consecutive numbers in the subsequence is at least* p *and at most* $p(1 + \epsilon)$*.*

The Problem Definition. The problem is formally defined below.

Definition 2. *Given* $\epsilon \in [0, 1)$*, the* ϵ-Relative Error Periodic Pattern Problem (REPP) *is the following:*

INPUT: An n*-long sequence* S *of numbers* $s_i \in \mathbb{N}$ *in increasing order. OUTPUT: The longest* ϵ*-relative error periodic pattern, i.e., the longest subsequence* $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ *of* S*, for which there exists a number* p *such that the absolute difference between any two consecutive numbers in the subsequence is at least* p *and at most* $p(1 + \epsilon)$*.*

We call p *an* approximate period of the REPP*.*

Note that we may assume, w.l.o.g., that the times of the events are discrete.

Example: Let $\epsilon = 0.5$ and $S = 1, 6, 9, 10, 16, 21$. The longest exact periodic pattern has length 2. However, consider $p = 4$. $p(1 + \epsilon) = 4 \cdot 1.5 = 6$. For the subsequence $1, 6, 10, 16, 21$ it holds that:

$$4 \leq |6 - 1| = 5 \leq 6,$$

$$4 \leq |10 - 6| = 4 \leq 6,$$

[1] Unlike [2], we assume that ϵ is strictly less than 1. This assumption is used in our algorithm.

$$4 \leq |16 - 10| = 6 \leq 6,$$
$$4 \leq |21 - 16| = 5 \leq 6.$$

[2] gives a simple $O(n^3)$ time algorithm to obtain, for a given ϵ, the longest relative error pattern in a sequence of n numbers. This bound is too high for large inputs in practice. Therefore, [2] explores approximate solutions, which approximate the longest periodic pattern that is present in the input. For this relaxed version of the problem, [2] is able to greatly reduce the run time to $O(n^{1+\gamma})$, for any constant $\gamma > 0$. This approximation algorithm is indeed faster and can be used in practical situation where the goal is to detect whether there are significant periodic patterns in the data. However, it has the following two drawbacks:

1. The length of the periodic pattern returned by the approximation algorithm may be larger than the actual REPP.
2. The period of the periodic pattern returned by the approximation algorithm may be larger than the actual REPP.

Thus, when the goal is to analyze the given data and its periodic patterns, the approximation algorithm may not be adequate. In such cases we need to solve the REPP problem exactly. Yet, no worst-case sub-cubic time algorithm is known for the REPP problem.

1.1 Paper Contribution

In this paper we give a new algorithm for finding the *exact* longest ϵ-relative error periodic pattern (the REPP problem). Our algorithm is based on a different method than that of [2]. We use the new concept of an *active maximal interval* in the input sequence, formally defined in Sect. 2. Informally, an interval is a segment in the sequence S and we call it *active* for some integer d, if there exists an element in the sequence preceding the interval, called an *activating element*, such that all the elements in the interval are within difference at least d and at most$(1 + \epsilon)d$ from the activating element. The difference d is then called an *activating difference* for this interval. We call an active interval *maximal* for a specific activating difference d, if it is not contained in another active interval for d. Given the input sequence S, let

$$D(S) = \{d \in \mathbb{N} \mid s, t \in S, s < t, \text{ such that } d = t - s\}.$$

We transform the input sequence into a different representation: the list L, defined as follows.

The Definition of the ϵ-Active Maximal Intervals List L. Given $d \in D(S)$, let $L_d(S)$ be the list of all active maximal intervals I such that d is an activating difference for I, sorted by their first element. Let $L(S)$ be the concatenation list of the lists $L_d(S)$, for all $d \in D(S)$, sorted by the activating difference. For simplicity, we omit S from the notation and use the notation

L and L_d throughout the paper. We call L, the *ϵ-active maximal intervals list* of S.

In Sect 3, we show that L can be constructed from the input sequence S in time quadratic in n, the size of S, and linear in the size of L. We, therefore, describe a new algorithm to the problem that is given L as input, and prove that it is linear in the size of L. This enables us to show that our algorithm works in sub-cubic time on inputs for which the best known algorithm works in $O(n^3)$ time. Moreover, though it may happen that our algorithm would still be cubic, it is never worse than the known $O(n^3)$-algorithm and in many situations its complexity is $O(n^2)$ time (see the remark below).

Notations. For a given set A, we denote by $|A|$ the size of A, i.e., the number of elements A contains.

Results. We prove the following theorem and corollary.

Theorem 1. *Let (ϵ, S) be an instance of the REPP problem and let L be the ϵ-active maximal intervals list of S. Then, there exists an algorithm for the REPP problem with complexity $O(|L|)$.*

Corollary 1. *Let (ϵ, S) be an instance of the REPP problem and let L be the ϵ-active maximal intervals list of S. Then, if $|L| = o(n^3)$, then the REPP problem on S can be solved in sub-cubic time.*

Remark. The algorithm of [2] can also be analyzed in terms of an input parameter $D(S)$. Specifically, the $O(n^3)$-algorithm of [2] runs in $O(|D(S)|n)$ time. Thus, whenever $|D(S)| = o(n^2)$, this algorithm would be sub-cubic. To compare with our algorithm, note that the list L contains $O(|D(S)|)$ lists each of length $O(n)$, therefore, its $O(|L|)$-worst case performance is never worse than that of [2]. Moreover, as demonstrated in Subsection 3.3, the size of each sub-list in L can be significantly smaller, even $O(1)$, leading to a substantial improvement over the algorithm of [2].

2 Preliminaries

In this section we give a basic lemma and some formal definitions needed for the description of our algorithm and its analysis in Section 3.

Lemma 1 is a basic key property of the problem that we use. It enables the algorithm we describe to search solutions with approximate periods in the set $D(S)$ only. It was also implicitly used in [2].

Lemma 1. *Let (ϵ, S) be an instance of the REPP problem, and let $P = s_{i_1}, \ldots,$ s_{i_k} be a subsequence of S which is a solution to this instance. Let p be the approximate period of this REPP. Then, there exists $p' \in D(S)$ such that p' is an approximate period of this REPP.*

Proof. Let p' be the smallest difference between any two consecutive numbers of P. Note that $p' \in D(S)$. We claim that p' is an approximate period of P. Let d be any absolute difference between consecutive numbers of P. Obviously, $d \geq p'$. We now need to show that $d \leq (1+\epsilon)p'$. Assume to the contrary that $d > (1+\epsilon)p'$. Since p is an approximate period of P, we have that $p \leq d \leq (1+\epsilon)p$. This holds for any absolute difference d. Since, p' is also an absolute difference between two consecutive numbers in P, it holds that $p \leq p'$. Therefore, $(1+\epsilon)p \leq (1+\epsilon)p' < d$. Contradiction. $\square$

We make use of the following definitions.

Definition 3. [Interval]
An interval *is a subsequence of S in which every pair of subsequent elements in the interval is a pair of subsequent elements in S.*

Let s_i and s_j, $1 \leq i, j \leq n$, be the first and last element of S in the interval, respectively, then the interval is denoted by $[s_i, s_j]$.

Definition 4. [Active Interval and Virtual Interval]
An interval $[s_i, s_j]$ such that there exists $d \in D$ and $t \in S$, $t < s_i$, for which for every $s \in [s_i, s_j]$ it holds that $d \leq s - t \leq d(1+\epsilon)$, is called an active interval.

If an interval is not active it is called a virtual interval.

Definition 5. [Activating Difference and Activating Element]
Let $[s_i, s_j]$ be an active interval, and let $d \in D(S)$ and $t \in S$, $t < s_i$, such that for every $s \in [s_i, s_j]$ it holds that $d \leq s - t \leq d(1+\epsilon)$. We call such a number $d \in D(S)$ an activating difference *for the interval $[s_j, s_j]$. The element $t \in S$ is called an* activating element *for the interval $[s_i, s_j]$.*

Definition 6. [Active Maximal Interval]
Let $[s_i, s_j]$ be an active interval with activating difference d. Then, $[s_j, s_j]$ is called a maximal interval for the activating difference *d, if neither $[s_{i-1}, s_j]$ nor $[s_i, s_{j+1}]$ are active intervals with activating difference d.*

Definition 7. [Nested Intervals]
Let $I_1 = [s_i, s_j]$ and $I_2 = [s_k, s_\ell]$ be two different intervals. If $i \leq k \leq \ell \leq j$ or $k \leq i \leq j \leq \ell$, then I_1 and I_2 are called nested.

3 The REPP Intervals Algorithm

In this section we describe our new method for solving the REPP problem and prove Theorem 1.

The $O(n^3)$ algorithm of [2] processes the n-length sequence S, for each $d \in D(S)$. Since $|D(S)|$ can be $O(n^2)$ for some input sequences, this gives the bound on this algorithm's complexity. Our method still loops for every $d \in D(S)$, however, instead of processing the whole sequence S, we only process the pre-constructed list L.

For clarity of exposition, we first introduce the algorithm with an inefficient implementation in Subsection 3.1 and prove its correctness in Subsection 3.2.

In order to analyze the complexity of the algorithm, we describe an efficient implementation in Subsection 3.3. The described algorithm finds only the length of the REPP. Therefore, we explain how to find the REPP itself in Subsection 3.4. We conclude by showing an efficient algorithm for constructing the list L in Subsection 3.5.

3.1 The Algorithm

The algorithm's main loop, given $d \in D(S)$, processes L_d to find the longest relative error periodic pattern with period d. The algorithm maintains a record for each interval with the field $\ell(I)$, which holds the length of the longest periodic pattern starting in this interval until the right end of the sequence S. The detailed algorithm is given in Fig. 1.

```
THE REPP INTERVALS ALGORITHM
Input: $L$, $D(S)$
1  $max \leftarrow 1$
2  for each $d \in D(S)$ do
3        if $L_d = \phi$ then
4              $max_d \leftarrow 1$
5        else
6              $max_d \leftarrow 2$
7              for each $I \in L_d$ do
8                    $\ell(I) \leftarrow 2$
9              for each $I \in L_d$ from right to left do
10                   for each $I' \in L_d$ containing an activating element for $I$
                           with activating difference $d$ do
11                         $\ell(I') \leftarrow \max\{\ell(I'), \ell(I)+1\}$
12                         if $\ell(I') > max_d$ then
13                               $max_d \leftarrow \ell(I')$
14       if $max_d > max$ then
15             $max \leftarrow max_d$
Output: $max$
```

Fig. 1. A REPP algorithm using the ϵ-active maximal intervals list L

3.2 The Correctness of the REPP Intervals Algorithm

The correctness of the REPP intervals algorithm in Fig. 1 follows from Lemma 2 and Corollary 2 below. The proof of Lemma 2 is omitted due to space limitations and will appear in the full version of the paper.

Lemma 2. *Let (ϵ, S) be an instance of the REPP problem then the* **for** *loop in lines 2–15 of the REPP intervals algorithm finds the longest ϵ-relative error periodic pattern in S with period d.*

Corollary 2 immediately follows from Lemma 2 and lines 14–15 of the REPP intervals algorithm.

Corollary 2. *Let (ϵ, S) be an instance of the REPP problem then the REPP intervals algorithm returns the longest ϵ-relative error periodic pattern in S.*

3.3 The Complexity of the REPP Intervals Algorithm

Our goal in this subsection is to prove the following lemma.

Lemma 3. *Let (ϵ, S) be an instance of the REPP problem and let L be the ϵ-active maximal intervals list of S. Then, the REPP intervals algorithm works in time $O(|L|)$.*

The main loop of the intervals algorithm in lines 2–15 (see Fig. 1) processes every L_d in L. The inner loop in lines 7–8 clearly takes $O(|L_d|)$ time. It remains to show that the inner loop in lines 9–15 takes $O(|L_d|)$ time. This is not apparent from Fig. 1, since this loop contains another inner for-loop that may take more than $O(1)$. We, therefore, describe an efficient implementation of the algorithm for which the total time complexity of the loop in lines 9–15 is indeed $O(|L_d|)$.

Efficient Implementation of the REPP Intervals Algorithm. We show that it is possible to implement a sliding window technique on the list L_d. First note that, by the definition of an activating element t for an interval $I = [s_i, s_j]$, we know that for every $s \in [s_i, s_j]$ it holds that $d \leq s - t \leq d(1+\epsilon)$, thus, $t \leq s - d$ and $s - d(1+\epsilon) \leq t$. Therefore, an interval I' that contains an activating element for I must have an element in the window $[s_j - d(1+\epsilon), s_i - d]$ that contains all the possible activating elements for I (there must be at least one such element if I is active). In order to efficiently find a window in L_d containing all the intervals I', we need to find in the list L_d the interval that starts closest from the left to the element $s_i - d$ and the interval that ends closest from the right to the element $s_j - (1+\epsilon)d$. This can be easily done in $O(1)$ time by sliding the borders, if we have L_d sorted both from left to right by starting elements of the intervals and from right to left by ending elements of the intervals. However, in general such different sorts may result in different lists, and the borders of the window we get will not properly define a window in L_d. Note, however, that sorting L_d by start points of the intervals or by end points of the intervals gives different lists if and only if L_d contains nested intervals. Fortunately, we have the following lemma.

Lemma 4. *For every $d \in D(S)$, the list L_d does not contain nested intervals.*

Proof. The lemma follows immediately from the maximality of the intervals in the list L_d. Recall that by definition intervals that share an end point or a start point are nested. Only the greater of such a pair of intervals can be maximal. Since by definition the list L_d contains only maximal intervals, the lemma follows. □

Lemma 4 assures that sorting L_d by the starting elements from left to right gives the same list of intervals as sorting it from right to left by the ending elements. The two borders search, thus, properly defines a window of all intervals I' that contain an activating element for the interval I. Note also, that since we use a sliding window, we do not have to search the borders in the full list L_d and we only linearly scan the start (or end, according to the need of the search) point of the next intervals in the list L_d from the current window borders until we reach the closest value to the searched value, as explained above. We now have to show how to efficiently make the computation and update for all the intervals in this window. In order to avoid re-update of intervals in the window, we only make the computation in line 11 of the algorithm in Fig 1 when an interval going out of the scope of the window when the next interval in L_d is processed and the new borders of the window are found. To this end, we keep for the window the value of $\ell(I) + 1$ in a separate variable. Therefore, each interval in the list L_d is entering the window once, going out of it once and updated once while going out. The value max_d can be checked, and updated if needed, each time the ℓ-field of an interval is updated.

This concludes the proof of Lemma 3.

A Sub-Cubic Time Example. We now describe an example of input for which the REPP intervals algorithm achieves sub-cubic time, whereas the algorithm of [2] has $O(n^3)$ time complexity. In fact, in this example the complexity of our algorithm is $O(n^2)$, which is the actual complexity of our algorithm for many practical situations. The input sequence is as follows: $S = 1, 2, 4, 7, 11, 16, 22, \ldots$. In this example the differences between consecutive numbers are: $1, 2, 3, 4, 5, 6 \ldots$, and $D(S) = \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15, 18, 20, 21, \ldots\}$. In general, for $|S| = n$ we have $|D(S)| = \Theta(n^2)$, and the algorithm of [2] has $\Theta(n^3)$ time complexity. However, each activating difference in $D(S)$ has very few active maximal intervals and $|L| = O(n^2)$. Thus, our interval algorithm time complexity is $O(n^2)$.

In practice, it is rather rare to have a list L of bigger size. This is possible, however, as the next example demonstrates.

A Worst Case Example. The following example shows that the intervals algorithm does not improve over the $O(n^3)$ algorithm on every input sequence.

Let the first $\frac{n}{2}$ elements of S be:

$$s_1 = 1$$

$$s_{i+1} = s_i + i, \quad i = 1, \ldots, \frac{n}{2} - 1.$$

The next $\frac{n}{2}$ elements are:

$$s_{i+1} = s_i + \frac{n^2}{4}, \ i = \frac{n}{2}, \ldots, n.$$

For example, if $n = 10$ then $S = 1, 2, 4, 7, 11, 36, 61, 86, 111, 136$.

Note that $1, 2, \ldots, \frac{n^2}{4} \in D(S)$, so $|D(S)| = \Theta(n^2)$.

Let $\epsilon = 1 - \frac{1}{n^2}$.

Each of the last $\frac{n}{2}$ elements defines an interval. Each of these intervals is an active maximal interval for $d = \frac{n^2}{8} + 1, \ldots, \frac{n^2}{4} - 1$, i.e. our algorithm's running time will be $\Theta(n^3)$.

3.4 Constructing the REPP

The REPP intervals algorithm described in Fig. 1 gives the *length* of the REPP solution, however, the definition of the problem requires returning the REPP itself. We, therefore, describe how the REPP solution can also be found. To this end the algorithm maintains another field, $next(I)$, for each interval I. In this field we keep the interval I_{next} for which the length of the REPP recorded in the field $\ell(I)$ is $\ell(I_{next}) + 1$. This field can be easily updated while updating the field $\ell(I)$. In addition, we also keep the interval that gives the maximum value, when updating max_d and max.

Given this additional information we can trace the actual REPP beginning from the leftmost interval I in which it starts, which is given with the value max. We know that there exists an element t in the interval I that activates the interval recorded in $next(I)$, i.e., all the elements can be used in a REPP with difference d that uses the element s. Such an element s can be easily found in $O(1)$-time computation using the border elements values of the interval $next(I)$. This trace proceeds, then, to the interval $next(I)$. When the last interval (the one for which the value of $next(I)$ is the initial value *Null*) is reached the actual REPP is found. Obviously, the process takes $O(|L|)$ time.

We have, therefore, proven Theorem 1.

3.5 The Construction of the ϵ-Active Maximal Intervals List L

Since the algorithm in Fig 1 gets as input the list L, however, the input of the problem is the sequence S, we need to show how to efficiently construct L from S. We now show how to construct it in time linear in the size of its output L. The construction algorithm works in four phases.

Phase 1: Given an element $s \in S$, we find for each element $s' \in S$ such that $s' > s$, the range of differences d for which $d \leq s' - s \leq (1 + \epsilon)d$. The borders $[b_1, b_2]$ of this range can be computed in $O(1)$ time, as follows: $b_1 = (s' - s)/(1 + \epsilon)$ and $b_2 = s' - s$.

Phase 2: Using the ranges of differences found in phase 1, we find the (maybe virtual) intervals of elements in S that the element s activates by grouping the elements s' that are all within the same range of activating differences, possibly splitting the ranges of differences we got in phase 1 .

Phases 1 and 2 are repeated for each $s \in S$.

Phase 3: In order to find the active maximal intervals we search the borders of the range of activating differences of each interval in the sorted list of differences $D(S)$. When an interval is found to be active for an actual range of differences in the list $D(S)$, it is put in the list L_d for each d in this range. An interval is put in the list L_d only if it is maximal for this d.

Note that, we process the intervals in the order of their starting elements and, therefore, the output lists L_d are sorted by the starting element of the intervals it contains with no need to explicitly sort them.

Phase 4: The list L is constructed by concatenating the sorted lists L_d for each $d \in D(S)$ in increasing order of d.

The Time Complexity of the L-Construction Algorithm. Phase 1, clearly, takes $O(n)$ time for each of the $O(n)$ repetitions, for a total of $O(n^2)$ time. Note that, we write the intervals in phase 2 such that in each interval in the output list an element of S is added or removed. Therefore, each activating element $s \in S$ has at most $2n$ intervals and the total length of the list of intervals constructed in phase 1 and 2 is $O(n^2)$. Therefore, phases 1 and 2 take $O(n^2)$ time and construct a list of intervals of size at most $O(n^2)$. This list of intervals is then processed in phases 3 and 4 to construct the lists L_d, for each $d \in D(S)$, and then the list L. Since the intervals we get in phase 2 are of increasing difference from the activating elements, we need not actually search their borders in the whole sorted list $D(S)$, but rather use a sliding window in $D(S)$ in order to find whether an interval is active. Thus, phase 3 takes total time $O(n^2 + |L|)$ rather than $O(n^2 \log n + |L|)$. The overall construction time is, therefore, $O(n^2 + |L|)$.

4 Conclusion and Open Problems

In this paper we described a new algorithm to find the longest relative error periodic pattern in a sequence. This algorithm runs in sub-cubic time for many practical situations even when the best known algorithm has cubic running time. Moreover, it never performs worse than the best known algorithm. We have shown, though, that there are pathological cases where this algorithm requires cubic running time. It is still an open question whether there exists a sub-cubic time algorithm for all possible input sequences.

References

1. Bagchi, S., Hung, E., Iyengar, A., Vogl, N.G., Wadia, N.: Capacity planning tools for web and grid environments. In: Proc. 1st International Conference on Performance Evaluation Methodolgies and Tools, VALUETOOLS (2006) ISBN = 1-59593-504-5, article number 25, http://doi.acm.org/10.1145/1190095
2. Gfeller, B.: Finding Longest Approximate Periodic Patterns. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 463–474. Springer, Heidelberg (2011)

3. Han, J., Dong, G., Yin, Y.: Efficient mining of partial periodic patterns in time series database. In: Proc. of the 15th International Conference on Data Engineering (ICDE 1999), pp. 106–115. IEEE Computer Society (1999)
4. Ma, S., Hellerstein, J.L.: Mining partially periodic event patterns with unknown periods. In: The 17th International Conference on Data Engineering (ICDE), pp. 205–214. IEEE Computer Society (2001)
5. Federal Highway Administration U.S. Department of Transportation, Conjestion: a national issue (August 2011), http://www.ops.fhwa.dot.gov/aboutus/opstory.htm
6. Panteleenko, V.V.: Instantaneous offloading of web server loads, Ph.D. thesis. University of Notre Dame (2002)
7. Rasheed, F., Alshalalfa, M., Alhajj, R.: Efficient periodicity mining in time series databases using suffix trees. IEEE Transactions on Knowledge and Data Engineering 99(preprints) (2010)
8. Tanbeer, S., Ahmed, C., Jeong, B.-S., Lee, Y.-K.: Discovering Periodic-Frequent Patterns in Transactional Databases. In: Theeramunkong, T., Kijsirikul, B., Cercone, N., Ho, T.-B. (eds.) PAKDD 2009. LNCS, vol. 5476, pp. 242–253. Springer, Heidelberg (2009)
9. Yang, J., Wang, W., Yu, P.S.: Mining asynchronous periodic patterns in time series data. IEEE Trans. on Knowl. and Data Eng. (15), 613–628 (2003)

Graph Expansion Analysis for Communication Costs of Fast Rectangular Matrix Multiplication

Grey Ballard[1,⋆], James Demmel[2,⋆,⋆⋆], Olga Holtz[3,⋆⋆⋆], Benjamin Lipshitz[1,⋆], and Oded Schwartz[1,†]

[1] EECS Department, University of California, Berkeley, CA 94720
{ballard,lipshitz,odedsc}@eecs.berkeley.edu
[2] Mathematics Department and CS Division, University of California, Berkeley, CA 94720
demmel@cs.berkeley.edu
[3] Departments of Mathematics, University of California, Berkeley and Technische Universität Berlin
holtz@math.berkeley.edu

Abstract. Graph expansion analysis of computational DAGs is useful for obtaining communication cost lower bounds where previous methods, such as geometric embedding, are not applicable. This has recently been demonstrated for Strassen's and Strassen-like fast square matrix multiplication algorithms. Here we extend the expansion analysis approach to fast algorithms for rectangular matrix multiplication, obtaining a new class of communication cost lower bounds. These apply, for example to the algorithms of Bini et al. (1979) and the algorithms of Hopcroft and Kerr (1971). Some of our bounds are proved to be optimal.

1 Introduction

The time cost of an algorithm, sequential or parallel, depends not only on how many computational operations it executes but also on how much data it moves. In fact, the cost of data movement, or *communication*, is often much more expensive than the cost of computation. Architectural trends predict that computation cost will continue to decrease exponentially faster than communication cost, leading to ever more algorithms that are dominated by the communication

⋆ Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

⋆⋆ Research is also supported by DOE grants DE-SC0003959, DE- SC0004938, and DE-AC02-05CH11231.

⋆⋆⋆ Research supported by the Sofja Kovalevskaja programme of Alexander von Humboldt Foundation and by the National Science Foundation under agreement DMS-0635607.

† Research supported by U.S. Department of Energy grants under Grant Numbers DE-SC0003959.

costs. Thus, in order to minimize running times, algorithms should be designed with careful consideration of their communication costs. To that end, we discuss asymptotic costs of algorithms in terms of both number of computations performed (flops in the case of numerical algorithms) and units of communication: *words moved*.

For a sequential algorithm, we determine the communication cost incurred on a simple machine model which consists of two levels of memory hierarchy, as described in Section 1.3. In many cases, naïve implementations of algorithms incur communication costs much higher than necessary; reformulating the algorithm to performing the same arithmetic in a different order can drastically decrease the communication costs and therefore the total running time. In order to determine the possible improvements and identify whether an algorithm is optimal with respect to communication costs, one seeks communication lower bounds.

Hong and Kung [17] were the first to prove communication lower bounds for matrix multiplication algorithms. They show that on a two-level machine model, any algorithm which performs the $\Theta(n^3)$ flops of classical matrix multiplication must move at least $\Omega(n^3/\sqrt{M})$ words between fast and slow memory, where M is the number of words that can fit simultaneously in fast memory. Irony, Toledo, and Tiskin [22] generalized their classical matrix multiplication result to a distributed-memory parallel machine model using a *geometric embedding* argument. Ballard, Demmel, Holtz and Schwartz [4] showed this proof technique is applicable to a more general set of computations, including one-sided matrix factorizations such as LU, Cholesky, and QR and two-sided matrix factorizations which are used in eigenvalue and singular value computations, most of which perform $\Theta(n^3)$ computations in the dense matrix case. Many of these bounds on $\Theta(n^3)$ algorithms have been shown to be optimal.

However, the geometric embedding approach does not seem to apply to computations which do not map to a simple geometric computation space. In the case of classical matrix multiplication and other $O(n^3)$ algorithms, the computation corresponds to a three-dimensional lattice. In particular, the geometric embedding approach does not readily apply to Strassen's algorithm for matrix multiplication that requires $O(n^{\log_2 7})$ flops. Instead, Ballard, Demmel, Holtz, and Schwartz [5] show that a different proof technique based on analysis of the expansion properties of the computational directed acyclic graph *(CDAG)* can be used to obtain communication lower bounds for both sequential and parallel models for these algorithms. The proof technique can also be used to bound how well the corresponding parallel algorithms can strongly-scale [2]. We use this same approach here to prove bounds on fast rectangular matrix multiplication algorithms, which introduce some extra technical challenges.

1.1 Expansion and Communication

The CDAG of a recursive algorithm has a recursive structure, and thus its expansion can be analyzed combinatorially (similarly to what is done for expander graphs in [30,1,26]) or by spectral analysis (in the spirit of what was done for the Zig-Zag expanders [31]). Analyzing the CDAG for communication cost bounds was first suggested by Hong and Kung [17]. They use the red-blue pebble game to obtain tight lower bounds on the communication costs of many algorithms, including classical $\Theta(n^3)$ matrix multiplication, matrix-vector multiplication, and FFT. Their proof is obtained by considering dominator sets of the CDAG.

Other papers study connections between bounded space computation and combinatorial expansion-related properties of the corresponding CDAG (see e.g., [32,9,8] and references therein). The study of expansion properties of a CDAG was also suggested as one of the main motivations of Lev and Valiant [28] in their work on superconcentrators and lower bounds on the arithmetic complexity of various problems.

1.2 Fast Rectangular Matrix Multiplication

Following Strassen's algorithm for fast multiplication of square matrices [33], the arithmetic complexity of multiplying rectangular matrices has been extensively studied (see [19,11,13,29,20,21,14] and further details in [12]). When there is an algorithm for multiplying an $m \times n$ matrix A with an $n \times p$ matrix B to obtain an $m \times p$ matrix C using only q scalar multiplications, we use the notation $\langle m, n, p \rangle = q$.[1] The above studies try to minimize the number of multiplications q (as a function of m, n, and p). A particular focus of interest is maximizing α so that $\langle n, n, n^\alpha \rangle = O(n^2 \log n)$ namely maximizing the size of a rectangular matrix, so that it can be multiplied (from right) with a square matrix, in time which is only slightly more than what is needed to read the input.[2] Recall that $\langle m, n, p \rangle = \langle n, p, m \rangle = \langle p, m, n \rangle = \langle m, p, n \rangle = \langle p, n, m \rangle = \langle n, m, p \rangle$ for all m, n, p [18].

Rectangular matrix multiplication is used in many algorithms, for solving problems in linear algebra, in combinatorial optimization, and other areas. Utilizing fast algorithms for rectangular matrix multiplication has proved to be quite useful for improving the complexity of solving many of those problems (a very partial list includes [16,25,7,36,27,34,35,23,24]).

1.3 Communication Model

We model communication costs on a sequential machine as follows. Assume the machine has a fast memory of size M words and a slow memory of infinite size. Further assume that computation can be performed only on data stored in the fast memory. On a real computer, this model may have several interpretations and may be applied to anywhere in the memory hierarchy. For example the slow memory might be the hard drive and the fast memory the DRAM; or the slow memory might be the DRAM and the fast memory the cache.

[1] Recall that $\langle m, n, p \rangle = q$ implies that for all integers t, $\langle m^t, n^t, p^t \rangle = q^t$ by recursion (tensor powering), and also that the arithmetic complexity of $\langle m^t, n^t, p^t \rangle$ is $O(q^t)$ regardless of the number of additions in $\langle m, n, p \rangle$.

[2] Note that our approach may not apply to algorithms of the form $\langle n, n, n^\alpha \rangle = O(n^2 \log n)$. It only applies to algorithms that are a recursive application of a base-case algorithm.

The goal is to minimize the number of words W transferred between fast and slow memory, which we call the communication cost of an algorithm. Note that we minimize with respect to an algorithm, not with respect to a problem, and so the only optimization allowed is re-ordering the computation in a way that is consistent with the CDAG of the algorithm. The sequential communication cost is closely related to communication costs in the various parallel models. We discuss this relationship briefly in Section 6.

1.4 The Communication Costs of Rectangular Matrix Multiplication

The communication costs lower bounds of rectangular matrix multiplication algorithms are determined by properties of the underlying CDAGs. Consider $\langle m^t, n^t, p^t \rangle = q^t$ matrix multiplication that is generated from t tensor powers of $\langle m, n, p \rangle = q$. Denote the former by the *algorithm* and the latter by the *base case*, and consider their CDAGs. They both consist of four parts: the encoding graphs of A and B, the scalar multiplications, and the decoding graph of C. The encoding graphs correspond to computing linear combinations of entries of A or B, and the decoding graph to computing linear combinations of the scalar products. See Figure 1 in Section 4 for a diagram of the algorithm CDAG, and Figure 2 in Section 5 for an example of a base-case CDAG. Let us state the communication cost lower bounds of the two main cases.

Theorem 1. *Let $\langle m^t, n^t, p^t \rangle = q^t$ be the algorithm obtained from a base case $\langle m, n, p \rangle = q$. If the decoding graph of the base case is connected, then the communication cost lower bound is*

$$W = \Omega\left(\frac{q^t}{M^{\log_{mp} q - 1}}\right).$$

Further, in the case that $n \leq m$ and $n \leq p$ this bound is tight.

Note that in the case $m = n = p$, this result reproduces the lower bound for Strassen-like square matrix multiplication algorithms in [5]. In this case, for $\omega_0 = \log_n q$, we obtain $W = \Omega\left(\frac{(n^t)^{\omega_0}}{M^{\omega_0/2-1}}\right)$.

Theorem 2. *Let $\langle m^t, n^t, p^t \rangle = q^t$ be the algorithm obtained from a base case $\langle m, n, p \rangle = q$. If an encoding graph of the base case is connected and has no multiply-copied inputs*[3]*, then*

$$W = \Omega\left(\frac{q^t}{t^{\log_N q} M^{\log_N q - 1}}\right),$$

where $N = mn$ or $N = np$ is the size of the input to the encoding graph. Further, this bound is tight if $N = \max\{mn, np, mp\}$, up to a factor of $t^{\log_N q}$, which is a polylogarithmic factor in the input size.

[3] See Section 2 for a formal definition.

We also treat the cases of disconnected encoding and decoding graphs and obtain similar bounds with restrictions on the fast memory size M. See Corollaries 1 and 2 in Section 4.

These theorems and corollaries apply in particular to the algorithms of Bini et al. [11] and Hopcroft and Kerr [19], which we detail in Section 5.

1.5 Paper Organization

In Section 2 we state some preliminary facts about the computational graph and edge expansion. Section 3 explains the connection between communication cost and edge expansion. The proofs of the lower bound theorems stated in Section 1.4, as well as some extensions, appear in Section 4. In Section 5 we apply our new lower bounds to two example algorithms: Bini's algorithm and the Hopcroft-Kerr algorithm. Appendix A gives further details of Bini's algorithm and the Hopcroft-Kerr algorithm.

2 Preliminaries

2.1 The Computational Graph

For a given algorithm, we consider the CDAG $G = (V, E)$, where there is a vertex for each arithmetic operation *(AO)* performed, and for every input element. G contains a directed edge (u, v), if the output operand of the AO corresponding to u (or the input element corresponding to u), is an input operand to the AO corresponding to v. The in-degree of any vertex of G is, therefore, at most 2 (as the arithmetic operations are binary). The out-degree is, in general, unbounded, i.e., it may be a function of $|V|$.

The Relaxed Computational Graph. For a given recursive algorithm, the *relaxed* computational graph is almost identical to the computational DAG with the following change: when a vertex corresponds to re-using data across recursive levels, we replace it with several connected "copy vertices," each of which exists in one recursive level. While the CDAG of a recursive algorithm may have vertices of degree that depend on $|V|$, this relaxed CDAG has constant bounded degree. We use the relaxed graph to handle such cases in Section 4.2.

Multiply-Copied Vertices. We say that a base-case encoding subgraph has *no multiply-copied vertices* if each input vertex appears at most once as an output vertex. An output vertex v is *copied* from an input vertex if the in-degree of v is exactly one. See, for example, Figure 2. The vertex a_{11} is copied to the third output of $Enc_1 A$ but is not copied to any other outputs. Since all other inputs are also copied at most once, there are no multiply-copied vertices in Figure 2.

This condition is necessary for the degree of the entire algorithm's encoding subgraph to be at most logarithmic in the size of the input. We are not aware of any fast matrix multiplication algorithm that has multiply-copied vertices, although the recursive formulation of classical matrix multiplication does.

2.2 Edge Expansion

The edge expansion $h(G)$ of a d-regular undirected graph $G = (V, E)$ is:

$$h(G) \equiv \min_{U \subseteq V, |U| \leq |V|/2} \frac{|E(U, V \setminus U)|}{d \cdot |U|}$$

where $E(A, B) \equiv E_G(A, B)$ is the set of edges connecting the vertex sets A and B. We omit the subscript G when the context makes it clear. Treating a CDAG as undirected simplifies the analysis and does not affect the asymptotic communication cost. For many graphs, small sets expand more than larger sets. Let $h_s(G)$ denote the edge expansion for sets of size at most s in G:

$$h_s(G) \equiv \min_{U \subseteq V, |U| \leq s} \frac{|E(U, V \setminus U)|}{d \cdot |U|} \ .$$

Note that CDAGs are typically not regular. If a graph $G = (V, E)$ is not regular but has a bounded maximal degree d, then we can add ($< d$) loops to vertices of degree $< d$, obtaining a regular graph G'. We use the convention that a loop adds 1 to the degree of a vertex. Note that for any $S \subseteq V$, we have $|E_G(S, V \setminus S)| = |E_{G'}(S, V \setminus S)|$, as none of the added loops contributes to the edge expansion of G'.

2.3 Matching Sequential Algorithm

In many cases, the communication cost lower bounds are matched by the naïve recursive algorithm. The cost of the recursive algorithm applied to $\langle m^t, n^t, p^t \rangle = q^t$, taking $N^* = \max\{mn, np, mp\}$ is

$$W(t) = \begin{cases} q \cdot W(t-1) + \Theta\left((N^*)^{t-1}\right) & \text{if } (N^*)^t > M/3 \\ 3(N^*)^t & \text{otherwise} \end{cases} ,$$

since the algorithm does not communicate once the three matrices fit into fast memory. The solution to this recurrence is given by

$$W = \Theta\left(\frac{q^t}{M^{\log_{N^*} q - 1}}\right).$$

3 Communication Cost and Edge Expansion

In this section we recall the partition argument and how to combine it with edge expansion analysis to obtain communication cost lower bounds. This follows our approach in [5,2]. A similar partition argument previously appeared in [17,22,4], where other techniques (geometric or combinatorial) are used to connect the number of flops to the amount of data in a segment.

3.1 The Partition Argument

Let M be the size of the fast memory. Let O be any total ordering of the vertices that respects the partial ordering of the CDAG G. This total ordering can be thought of as the actual order in which the computations are performed. Let $\mathcal{P}$ be any partition of V into segments $S_1, S_2, ...$, so that a segment $S_i \in \mathcal{P}$ is a subset of the vertices that are contiguous in the total ordering O.

Let R_S and W_S be the set of read and write operands, respectively. Namely, R_S is the set of vertices outside S that have an edge going into S, and W_S is the set of vertices in S that have an edge going outside of S. Then the total communication costs due to reads of AOs in S is at least $|R_S| - M$, as at most M of the needed $|R_S|$ operands are already in fast memory when the execution of the segment's AOs starts. Similarly, S causes at least $|W_S| - M$ actual write operations, as at most M of the operands needed by other segments are left in the fast memory when the execution of the segment's AOs ends. The total communication cost is therefore bounded below by

$$W \geq \min_{\mathcal{P}} \sum_{S \in \mathcal{P}} \left(|R_S| + |W_S| - 2M\right) \ . \tag{1}$$

3.2 Edge Expansion and Communication Cost

Consider a segment S and its read and write operands R_S and W_S.

Proposition 1. *If the graph G containing S has $h_s(G)$ edge expansion*[4] *for sets of size $s = |S|$, maximum (constant) degree d, and at least $2|S|$ vertices, then $|R_S| + |W_S| \geq \frac{1}{2} \cdot h_s(G) \cdot |S|$.*

Proof. We have $|E(S, V \setminus S)| \geq h_s(G) \cdot d \cdot |S|$. Either (at least) half of the edges $E(S, V \setminus S)$ touch R_S or half of them touch W_S. As every vertex is of degree d, we have $|R_S| + |W_S| \geq \max\{|R_S|, |W_S|\} \geq \frac{1}{d} \cdot \frac{1}{2} \cdot |E(S, V \setminus S)| \geq h_s(G) \cdot |S|/2$. $\square$

Combining this with (1) and choosing to partition V into $|V|/s$ segments of equal size s, we obtain: $W \geq \max_s \frac{|V|}{s} \cdot \left(\frac{h_s(G) \cdot s}{2} - 2M\right)$. Choosing the minimal s so that

$$\frac{h_s(G) \cdot s}{2} \geq 3M \tag{2}$$

we obtain

$$W \geq \frac{|V|}{s} \cdot M \ . \tag{3}$$

In some cases, as in fast square and rectangular matrix multiplication, the computational graph G does not fit this analysis: it may not be regular, it may have

[4] For many algorithms, the edge expansion $h(G)$ deteriorates with $|G|$, whereas $h_s(G)$ is constant with respect to $|G|$, which allows for better communication lower bounds.

vertices of unbounded degree, or its edge expansion may be hard to analyze. In such cases, we may then consider some subgraph G' of G instead to obtain a lower bound on the communication cost. The natural subgraph to select in fast (square and rectangular) matrix multiplication algorithms is the decoding graph or one of the two encoding graphs.

4 Expansion Properties of Fast Rectangular Matrix Multiplication Algorithms

There are several technical challenges that we deal with in the rectangular case, on top of the analysis in [5] (where we deal with the difference between addition and multiplication vertices in the recursive construction of the CDAG). These additional challenges arise from the differences between the CDAG of rectangular algorithms, such as Bini's algorithm and the Hopcroft-Kerr algorithm on the one hand, and of Strassen's algorithm on the other hand. The three subgraphs, two encoding and one decoding, are of the same size in Strassen's and of unequal size in rectangular algorithms. The largest expansion guarantee is given by the subgraph corresponding to the largest of the three matrices. One consequence is that it is necessary to consider the case of unbounded degree vertices that may appear in the encoding subgraphs. Additionally, in some cases the encoding or decoding graphs consist of several disconnected components.

4.1 The Computational Graph for $\langle m^t, n^t, p^t \rangle = q^t$

Consider the computational graph H_t associated with multiplying a matrix A of dimension $m^t \times n^t$ by a matrix B of dimension $n^t \times p^t$. Denote by Enc_tA the part of H_t that corresponds to the encoding of matrix A. Similarly, Enc_tB, and Dec_tC correspond to the parts of H_t that compute the encoding of B and the decoding of C, respectively (see Figure 1).

A Top-Down Construction of the Computational Graph. We next construct the computational graph H_{i+1} by constructing $Dec_{i+1}C$ from Dec_iC and Dec_1C and similarly constructing $Enc_{i+1}A$ and $Enc_{i+1}B$, then composing the three parts together.

1. Duplicate Dec_1C q^i times.
2. Duplicate Dec_iC mp times.
3. Identify the $mp \cdot q^i$ output vertices of the copies of Dec_1C with the $mp \cdot q^i$ input vertices of the copies of Dec_iC:
   - Recall that each Dec_1C has mp output vertices.
   - The first output vertex of the q^i Dec_1C graphs are identified with the q^i input vertices of the first copy of Dec_iC.
   - The second output vertex of the q^i Dec_1C graphs are identified with the q^i input vertices of the second copy of Dec_iC. And so on.
   - We make sure that the jth input vertex of a copy of Dec_iC is identified with an output vertex of the jth copy of Dec_1C.

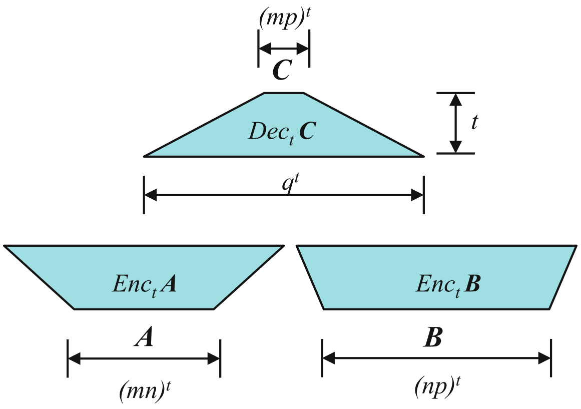


Fig. 1. Computational graph for $\langle m^t, n^t, p^t \rangle = q^t$ rectangular matrix multiplication generated from t recursive levels with base graph given by $\langle m, n, p \rangle = q$. In this figure $m < p < n$.

4. We similarly obtain $Enc_{i+1}A$ from Enc_iA and Enc_1A,
5. and $Enc_{i+1}B$ from Enc_iB and Enc_1B.
6. For every i, H_i is obtained by connecting edges from the jth output vertices of Enc_iA and Enc_iB to the jth input vertex of Dec_iC.

This completes the construction. Let us note some properties of this graphs.

As all out-degrees are at most mp and all in degree are at most 2 we have:

Proposition 2. *All vertices of Dec_tC are of degree at most $mp + 2$, as long as $n > 1$ (that is, as long as the base case is not an outer product).*

Proof. If the set of input vertices of Dec_1C and the set of its output vertices are disjoint, then the proposition follows.. Assume (towards contradiction) that the base graph Dec_1C has an input vertex which is also an output vertex. An output vertex represents the inner product of two n-vectors, i.e., the corresponding row-vector of A and column vector of B. The corresponding bilinear polynomial is irreducible. This is a contradiction, since $n > 1$ an input vertex represents the multiplication of a (weighted) sum of elements of A with a (weighted) sum of elements of B. $\square$

Note, however, that Enc_1A and Enc_1B may have vertices which are both inputs and outputs, therefore Enc_tA and Enc_tB may have vertices of out-degree which is a function of t. In [5,2], it was enough to analyze Dec_tC and lose only a constant factor in the lower bound. However in several rectangular matrix multiplication algorithms, it is necessary to consider the encoding graphs as well, since they may provide a better expansion than the decoding graph.

Lemma 1. *If Dec_1C is connected, then the edge expansion of Dec_tC is*

$$h(Dec_tC) = \Omega\left(\left(\frac{mp}{q}\right)^t\right).$$

Proof. The proof follows that of Lemma 4.9 in [5] adapting the corresponding parameters. We provide it here for completeness. Let $G_t = (V, E)$ be Dec_tC, and let $S \subseteq V, |S| \leq |V|/2$. We next show that $|E(S, V \setminus S)| \geq c \cdot d \cdot |S| \cdot \left(\frac{mp}{q}\right)^t$, where c is some universal constant, and d is the constant degree of Dec_tC (after adding loops to make it regular).

The proof works as follows. Recall that G_t is a layered graph (with layers corresponding to recursion steps), so all edges (excluding loops) connect between consecutive levels of vertices. We argue (in Proposition 4) that each level of G_t contains about the same fraction of S vertices, or else we have many edges leaving S. We also observe (in Fact 5) that such homogeneity (of a fraction of S vertices) does not hold between distinct parts of the lowest level, or, again, we have many edges leaving S. We then show that the homogeneity between levels, combined with the heterogeneity of the lowest level, guarantees that there are many edges leaving S.

Let l_i be the ith level of vertices of G_t, so $(mp)^t = |l_1| < |l_2| < \cdots < |l_i| = (mp)^{t-i+1}q^{i-1} < \cdots < |l_{t+1}| = q^t$. Let $S_i \equiv S \cap l_i$. Let $\sigma = \frac{|S|}{|V|}$ be the fractional size of S and $\sigma_i = \frac{|S_i|}{|l_i|}$ be the fractional size of S at level i. Let $\delta_i = \sigma_i - \sigma_{i+1}$. Due to averaging, we observe the following:

Fact 3. *There exist i and i' such that $\sigma_i \leq \sigma \leq \sigma_{i'}$.*

Fact 4

$$\begin{aligned}|V| = \sum_{i=1}^{t+1} |l_i| &= \sum_{i=1}^{t+1} |l_{t+1}| \cdot \left(\frac{mp}{q}\right)^i \\ &= |l_{t+1}| \cdot \left(1 - \left(\frac{mp}{q}\right)^{t+2}\right) \cdot \frac{q}{q - mp} \\ &= \left(\frac{mp}{q}\right)^t \cdot |l_1| \cdot \left(1 - \left(\frac{mp}{q}\right)^{t+2}\right) \cdot \frac{q}{q - mp}.\end{aligned}$$

so $\frac{q-mp}{q} \leq \frac{|l_{t+1}|}{|V|} \leq \frac{q-mp}{q} \cdot \frac{1}{1-\left(\frac{mp}{q}\right)^{t+2}}$, *and* $\frac{q-mp}{q} \cdot \left(\frac{mp}{q}\right)^t \leq \frac{|l_1|}{|V|} \leq \frac{q-mp}{q} \cdot \left(\frac{mp}{q}\right)^t \cdot \frac{1}{1-\left(\frac{mp}{q}\right)^{t+2}}$.

Proposition 3. *There exists $c' = c'(G_1)$ so that $|E(S, V \setminus S) \cap E(l_i, l_{i+1})| \geq c' \cdot d \cdot |\delta_i| \cdot |l_i|$.*

Proof (of Proposition 3). Let G' be a G_1 component connecting l_i with l_{i+1} (so it has mp vertices in l_i and q in l_{i+1}). G' has no edges in $E(S, V \setminus S)$ if all or none of its vertices are in S. Otherwise, as G' is connected, it contributes at least one edge to $E(S, V \setminus S)$. The number of such G_1 components with all their vertices in S is at most $\min\{\sigma_i, \sigma_{i+1}\} \cdot \frac{|l_i|}{mp}$. Therefore, there are at least $|\sigma_i - \sigma_{i+1}| \cdot \frac{|l_i|}{mp}$ G_1 components with at least one vertex in S and one vertex that is not. □

Proposition 4 (Homogeneity between levels). *If there exists i so that $\frac{|\sigma-\sigma_i|}{\sigma} \geq \frac{1}{10}$, then*

$$|E(S, V \setminus S)| \geq c \cdot d \cdot |S| \cdot \left(\frac{mp}{q}\right)^t$$

where $c > 0$ is some constant depending on G_1 only.

Proof (of Proposition 4). Assume that there exists j so that $\frac{|\sigma-\sigma_j|}{\sigma} \geq \frac{1}{10}$. By Proposition 3, we have

$$\begin{aligned}
|E(S, V \setminus S)| &\geq \sum_{i \in [t]} |E(S, V \setminus S) \cap E(l_i, l_{i+1})| \\
&\geq \sum_{i \in [t]} c' \cdot d \cdot |\delta_i| \cdot |l_i| \\
&\geq c' \cdot d \cdot |l_1| \sum_{i \in [t]} |\delta_i| \\
&\geq c' \cdot d \cdot |l_1| \cdot \left(\max_{i \in [t+1]} \sigma_i - \min_{i \in [t+1]} \sigma_i \right).
\end{aligned}$$

By the initial assumption, there exists j so that $\frac{|\sigma-\sigma_j|}{\sigma} \geq \frac{1}{10}$, therefore $\max_i \sigma_i - \min_i \sigma_i \geq \frac{\sigma}{10}$. By Fact 4, $|l_1| \geq \frac{q-mp}{q} \cdot \left(\frac{mp}{q}\right)^t \cdot |V|$. As $|S| = \sigma \cdot |V|$, we have

$$\begin{aligned}
|E(S, V \setminus S)| &\geq c' \cdot d \cdot |l_1| \cdot \frac{\sigma}{10} \\
&\geq c' \cdot d \cdot \frac{q-mp}{q} \cdot \left(\frac{mp}{q}\right)^t \cdot |V| \cdot \frac{\sigma}{10} \\
&\geq c \cdot d \cdot |S| \cdot \left(\frac{mp}{q}\right)^t
\end{aligned}$$

for any $c \leq \frac{c'}{10} \cdot \frac{q-mp}{q}$. □

Let T_t be a tree corresponding to the recursive construction of G_t in the following way: T_t is a tree of height $t+1$, where each internal node has mp children. The root r of T_t corresponds to l_{t+1} (the largest level of G_t). The mp children of r correspond to the largest levels of the mp graphs that one can obtain by removing the level of vertices l_{t+1} from G_t. And so on. For every node u of T_t, denote by V_u the set of vertices in G_t corresponding to u. We thus have $|V_r| = q^t$ where r is the root of T_t, $|V_u| = q^{t-1}$ for each node u that is a child of r; and in general we have $(mp)^i$ tree nodes u corresponding to a set of size $|V_u| = q^{t-i+1}$. Each leaf l corresponds to a set of size 1.

For a tree node u, let us define $\rho_u = \frac{|S \cap V_u|}{|V_u|}$ to be the fraction of S nodes in V_u, and $\delta_u = |\rho_u - \rho_{p(u)}|$, where $p(u)$ is the parent of u (for the root r we let $p(r) = r$). We let t_i be the ith level of T_t, counting from the bottom, so t_{t+1} is the root and t_1 are the leaves.

Fact 5. *As $V_r = l_{t+1}$ we have $\rho_r = \sigma_{t+1}$. For a tree leaf $u \in t_1$, we have $|V_u| = 1$. Therefore $\rho_u \in \{0,1\}$. The number of vertices u in t_1 with $\rho_u = 1$ is $\sigma_1 \cdot |l_1|$.*

Proposition 5. *Let u_0 be an internal tree node, and let $u_1, u_2, \ldots, u_{mp}$ be its mp children. Then*

$$\sum_i |E(S, V \setminus S) \cap E(V_{u_i}, V_{u_0})| \geq c'' \cdot d \cdot \sum_i |\rho_{u_i} - \rho_{u_0}| \cdot |V_{u_i}|$$

where $c'' = c''(G_1)$.

Proof (Proposition 5). The proof follows that of Proposition 3. Let G' be a G_1 component connecting V_{u_0} with $\bigcup_{i\in[mp]} V_{u_i}$ (so it has q vertices in V_{u_0} and one in each of V_{u_1},V_{u_2},…,$V_{u_{mp}}$). G' has no edges in $E(S, V \setminus S)$ if all or none of its vertices are in S. Otherwise, as G' is connected, it contributes at least one edge to $E(S, V \setminus S)$. The number of G_1 components with all their vertices in S is at most $\min\{\rho_{u_0}, \rho_{u_1}, \rho_{u_2}, \ldots, \rho_{u_{mp}}\} \cdot \frac{|V_{u_1}|}{mp}$. Therefore, there are at least $\max_{i\in[mp]}\{|\rho_{u_0} - \rho_{u_i}|\} \cdot \frac{|V_{u_1}|}{mp} \geq \frac{1}{(mp)^2} \cdot \sum_{i\in[mp]} |\rho_{u_i} - \rho_{u_0}| \cdot |V_{u_i}|$ G_1 components with at least one vertex in S and one vertex that is not. □

By Proposition 5, we have

$$\begin{aligned}
|E(S, V \setminus S)| &= \sum_{u\in T_t} |E(S, V \setminus S) \cap E(V_u, V_{p(u)})| \\
&\geq \sum_{u\in T_t} c'' \cdot d \cdot |\rho_u - \rho_{p(u)}| \cdot |V_u| \\
&= c'' \cdot d \cdot \sum_{i\in[t]} \sum_{u\in t_i} |\rho_u - \rho_{p(u)}| \cdot q^{i-1} \\
&\geq c'' \cdot d \cdot \sum_{i\in[t]} \sum_{u\in t_i} |\rho_u - \rho_{p(u)}| \cdot (mp)^{i-1} \\
&= c'' \cdot d \cdot \sum_{v\in t_1} \sum_{u\in v\sim r} |\rho_u - \rho_{p(u)}|
\end{aligned}$$

as each internal node has mp children, and $v \sim r$ is the path from v to the root r. By the triangle inequality for the function $|\ \cdot\ |$ and Fact 5,

$$\begin{aligned}
&\geq c'' \cdot d \cdot \sum_{v\in t_1} |\rho_u - \rho_r| \\
&\geq c'' \cdot d \cdot |l_1| \cdot ((1 - \sigma_1) \cdot \rho_r + \sigma_1 \cdot (1 - \rho_r))
\end{aligned}$$

By Proposition 4, w.l.o.g., $|\sigma_{t+1} - \sigma|/\sigma \le \frac{1}{10}$ and $|\sigma_1 - \sigma|/\sigma \le \frac{1}{10}$. As $\rho_r = \sigma_{t+1}$, and by Fact 4,

$$\begin{aligned} &\ge \frac{3}{4} \cdot c'' \cdot d \cdot |l_1| \cdot \sigma \\ &\ge c \cdot d \cdot |S| \cdot \left(\frac{mp}{q}\right)^t \end{aligned}$$

for any $c \le \frac{3}{4} \cdot c''$. This completes the proof of Lemma 1. □

Using Lemma 2.1 of [5] (decomposition into edge disjoint small subgraphs) we deduce that for sufficiently large t,

$$h_s(Dec_tC) = \Omega\left(\left(\frac{mp}{q}\right)^{\log_q s}\right).$$

Thus there exists a constant c such that for $s = cM^{\log_{mp} q}$, $s \cdot h_s(Dec_tC) \ge 3M$. Plugging this into inequality (3) we obtain Theorem 1.

4.2 Stretching a Segment

We next consider the case where all vertices have a degree bounded by $O(t)$. We analyze the edge expansion of the relaxed computational graph,[5] which corresponds to the same set of computations but has a constant degree bound. We then show that an augmented partition argument (similar to that in Section 3.1) results in a communication cost lower bound which is optimal up to at most a polylogarithmic factor.

Since a relaxed encoding graph has a constant degree bound we can analyze the expansion of the Enc_tA and Enc_tB parts of the computational graph by exactly the same technique used for Dec_tC above. Plugging in the corresponding parameters, we thus obtain:

Lemma 2. *Let G'_t be the relaxed computational graph of computing $\langle m^t, n^t, p^t\rangle = q^t$ based on $\langle m, n, p\rangle = q$. Let Enc'_tA and Enc'_tB be the subgraphs corresponding to the encoding of A and B in G'_t. Then*

$$h_s(Enc'_tA) = \Omega\left(\left(\frac{mn}{q}\right)^{\log_q s}\right) \quad \textit{and} \quad h_s(Enc'_tB) = \Omega\left(\left(\frac{np}{q}\right)^{\log_q s}\right).$$

Consider a CDAG G with maximum degree $O(t)$ and its corresponding relaxed CDAG G' of constant degree. Given the expansion of G' we would like to deduce the communication cost incurred by computing G. To this end we need amended versions of inequalities (2) and (3); since by transforming G' back to G $|R_s| + |W_s|$ may contract by a factor of $O(t)$, we need to compensate for that by increasing the segment size s. To be precise, we want $\frac{|R_s|+|W_s|}{ct} - 2M = M$. Following

[5] See Section 2 for a formal definition.

inequality (2), we thus choose the minimal s such that $h_s(Enc_t A) \cdot s \geq c' tM$, where c' is some universal constant. By inequality (3) and Lemma 2, $\left(\frac{mn}{q}\right)^{\log_q s} \cdot s = \Theta(tM)$, so

$$W = \Omega\left(\frac{q^t}{(tM)^{\log_{mn} q}} M\right)$$

and Theorem 2 follows.

4.3 Disconnected Encoding or Decoding Graphs

The CDAG of any fast (rectangular or square) matrix multiplication algorithm must be connected, due to the dependencies of the output entries on the input entries. The encoding and decoding graphs, however, are not always connected (see e.g., Bini's algorithm, in Section 5.1 and Appendix A). Consider a case where each connected components of $Dec_t C$ is small enough to fit into the fast memory. Then our proof technique cannot provide a nontrivial lower bound. Even if a connected component is larger than M, but has $\leq M$ inputs and $\leq M$ outputs, the partition into segments approach provides no communication cost lower bound (see inequality (1) and its proof). In the case that the inputs of an encoding graph or the output of the decoding graph do not fit into fast memory, and the disconnected components all have the same number of input and output vertices, the lower bound technique still applies. Formally,

Corollary 1. *If the base-case decoding graph is disconnected and consists of X connected components of equal input and output size, then* $W = \Omega\left(\frac{q^t}{M^{\log_{mp/X}(q/X)-1}}\right)$.

Proof. Since $Dec_t C$ is disconnected $h(Dec_t C) = 0$. However it consists of X^t connected components, each of which has nonzero expansion, therefore the entire graph does have expansion for small sets. Each connected component is recursively constructed from a base graph with q/X inputs and mp/X outputs. By Lemma 1, each connected component CC_t of $Dec_t C$ has expansion

$$h(CC_t) = \Omega\left(\left(\frac{mp}{q}\right)^t\right).$$

In order to apply Lemma 2.1 of [5] (decomposition into edge disjoint small subgraphs), we decompose $Dec_t C$ into connected components of size s, where s needs to satisfy two conditions. First, s must be smaller than the size of the connected components of $Dec_t C$ (otherwise we cannot claim any expansion), namely $s = O\left(\left(\frac{q}{X}\right)^t\right)$. Second, s must be large enough so that the output of one component does not fit into fast memory (otherwise the expansion guarantee does not translate into a communication lower bound):

$$\left(\frac{mp}{X}\right)^k = \Omega(M),$$

where $k = \log_{q/X} s$ is the number of recursive steps inside one component. We then deduce that

$$h_s(Dec_t C) = \Omega\left(\left(\frac{mp}{q}\right)^{\log_{q/X} s}\right).$$

Thus there exists a constant c such that for $s = cM^{\log_{mp/X}(q/X)}$, $s \cdot h_s(Dec_t C) \geq 3M$. Plugging this into inequality (3) we obtain Corollary 1. Note that in the case that $M = \Omega\left(\left(\frac{mp}{X}\right)^t\right)$, the argument above does not apply, but the result still holds because it is weaker than the trivial bound that the entire output must be written: $W = \Omega\left((mp)^t\right)$. □

Corollary 2. *If a base-case encoding graph is disconnected and consists of X connected components of equal input and output size, has N inputs, where $N = mn$ or $N = np$, and has no multiply-copied inputs, then* $W = \Omega\left(\frac{q^t}{t^{\log_{N/x}(q/X)} M^{\log_{N/X}(q/X)-1}}\right)$.

Proof. Let G'_t be the relaxed computational graph of computing $\langle m^t, n^t, p^t \rangle = q^t$ based on $\langle m, n, p \rangle = q$. Let Enc'_t be the subgraph corresponding to the encoding of A or B in G'_t, and N be mn (for the encoding of A) or np (for the encoding of B). Then by the same argument as above,

$$h_s(Enc'_t) = \Omega\left(\left(\frac{N}{q}\right)^{\log_{q/X} s}\right).$$

Since by transforming G' back to G the sum $|R_s| + |W_s|$ may contract by a factor of $O(t)$ (recall Section 4.2), we need to compensate for that by increasing the segment size s. Thus the above only holds for $\left(\frac{N}{X}\right)^k = \Omega(Mt)$, where $k = \log_{q/X} s$. It follows that there exists a constant c such that for $s = c(tM)^{\log_{mp/X}(q/X)}$, $s \cdot h_s(Enc'_t) \geq 3tM$. Plugging this into inequality (3) we obtain Corollary 2. Note that in the case that $M = \Omega\left(\left(\frac{N}{X}\right)^t\right)$, the argument above does not apply, but the result still holds because it is weaker than the trivial bound that the entire input must be read: $W = \Omega\left(N^t\right)$. □

5 The Communication Costs of Some Rectangular Matrix Multiplication Algorithms

In this section we apply our main results to get new lower bounds for rectangular algorithms based on Bini's algorithm [11] and the Hopcroft-Kerr algorithm [19]. All rectangular algorithms yield a square algorithm. In the case of Bini the exponent is $\omega_0 \approx 2.779$, slightly better than Strassen's algorithm ($\omega_0 \approx 2.807$), and in the case of Hopcroft-Kerr the exponent is $\omega_0 \approx 2.811$, slightly worse than Strassen's algorithm. These algorithms are stated explicitly, which is not true of most of the recent results that significantly improve ω_0. See Table 1 for an enumeration of several algorithms based on [11,19] and their lower bounds.

5.1 Bini's Algorithm

Bini et al. [11] obtained the first approximate matrix multiplication algorithm. They introduce a parameter λ into the computation and give an algorithm that computes matrix multiplication up to terms of order λ. It was later shown how to convert such approximate algorithms into exact algorithms without changing the asymptotic arithmetic complexity, ignoring logarithmic factors [10].[6]

Bini et al. show how to compute $2 \times 2 \times 2$ matrix multiplication approximately where one of the off-diagonal entries of an input matrix is zero using 5 scalar multiplications. This can be used twice to give an algorithm for $\langle 3, 2, 2 \rangle = 10$ matrix multiplication. Notably this algorithm has disconnected $Enc_1 A$ (see Figure 2).

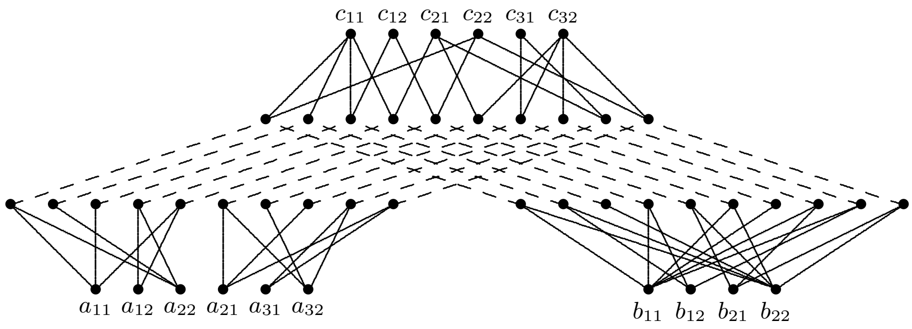

c_{11} c_{12} c_{21} c_{22} c_{31} c_{32}
a_{11} a_{12} a_{22} a_{21} a_{31} a_{32}
b_{11} b_{12} b_{21} b_{22}

Fig. 2. Computational graph for 1 level of Bini's $\langle 3, 2, 2 \rangle = 10$ algorithm. Solid lines indicate dependencies of additions and make up $Enc_1 A$, $Enc_1 B$, and $Dec_1 C$. Dashed lines indicate dependencies of multiplications and connect these three subgraphs. Note that $Enc_1 A$, the bottom-left part of the graph, is disconnected and has two connected components of equal size and equal input/output ratio. Note that the base-case graph of Bini's algorithm is presented, for simplicity, with vertices of in-degree larger than two. A vertex of degree larger than two, in fact, represents a full binary (not necessarily balanced) tree. The expansion arguments hold for any way of drawing the binary trees.

From this $\langle 3, 2, 2 \rangle = 10$ algorithm one immediately obtains 5 more algorithms by transposition and interchanging the encoding and decoding graphs [18]. Other algorithms can be constructed by taking tensor products of these base cases. When taking tensor products, the number of connected components of each encoding and decoding graph is the product of the number of connected components in the base cases. For example there are 4 ways to construct algorithms for $\langle 6, 6, 4 \rangle = 100$: one where $Enc_1 A$ and $Enc_1 B$ each have two components, one where $Enc_1 A$ and $Dec_1 C$ each have two components, one where $Enc_1 B$ and $Dec_1 C$ each have two components, and one where $Enc_1 A$ has four components. Similarly there are 8 ways to construct algorithms for the square multiplication $\langle 12, 12, 12 \rangle = 1000$.

[6] We treat here the original, approximate algorithm, not any of the exact algorithms that can be derived from it.

Table 1. Asymptotic lower bounds for several variants of the algorithms by Bini et al. and Hopcroft-Kerr. Many more with different shapes and with different disconnected subgraphs can be given for Bini's algorithm, and analyzed by similar means; we list only a representative sample. Recall that the base case $\langle m,n,p\rangle = q$ is used for the computation of $\langle m^t,n^t,p^t\rangle = q^t$.

	Algorithm	Disconnected	Communication Cost Lower Bound	by	Tight?
Bini et. al. [11]	$\langle 3,2,2\rangle = 10$	*EncA*	$10^t/M^{\log_6 10-1}$	Thm 1	Yes
	$\langle 3,2,2\rangle = 10$	*DecC*	$10^t/(t^{\log_6 10}M^{\log_6 10-1})$ $10^t/(M^{\log_3 5-1})$	Thm 2 Cor 1	Up to polylog factor No
	$\langle 2,3,2\rangle = 10$	*EncA*	$10^t/M^{\log_4 10-1}$ $10^t/(t^{\log_6 10}M^{\log_6 10-1})$	Thm 1 Thm 2	No Up to polylog factor
	$\langle 2,3,2\rangle = 10$	*EncB*	$10^t/M^{\log_4 10-1}$ $10^t/(t^{\log_6 10}M^{\log_6 10-1})$	Thm 1 Thm 2	No Up to polylog factor
	$\langle 2,2,3\rangle = 10$	*EncB*	$10^t/M^{\log_6 10-1}$	Thm 1	Yes
	$\langle 2,2,3\rangle = 10$	*DecC*	$10^t/(t^{\log_6 10}M^{\log_6 10-1})$ $10^t/(M^{\log_3 5-1})$	Thm 2 Cor 1	Up to polylog factor No
	$\langle 6,6,4\rangle = 100$	*EncA*, *EncB*	$100^t/M^{\log_{24} 100-1}$ $100^t/(t^{\log_{18} 50}M^{\log_{18} 50-1})$	Thm 1 Cor 2	No No
	$\langle 12,12,12\rangle = 1000$	*EncA*, *EncB*	$1000^t/M^{\log_{144} 1000-1}$	[5]	Yes
Hopcroft-Kerr [19]	$\langle 3,2,3\rangle = 15$	None	$15^t/M^{\log_9 15-1}$	Thm 1	Yes
	$\langle 3,3,2\rangle = 15$	None	$15^t/M^{\log_6 15-1}$ $15^t/(t^{\log_9 15}M^{\log_9 15-1})$	Thm 1 Thm 2	No Up to polylog factor
	$\langle 2,3,3\rangle = 15$	None	$15^t/M^{\log_6 15-1}$ $15^t/(t^{\log_9 15}M^{\log_9 15-1})$	Thm 1 Thm 2	No Up to polylog factor
	$\langle 9,6,6\rangle = 225$	None	$225^t/M^{\log_{54} 225-1}$	Thm 1	Yes
	$\langle 6,6,9\rangle = 225$	None	$225^t/M^{\log_{54} 225-1}$	Thm 1	Yes
	$\langle 6,9,6\rangle = 225$	None	$225^t/M^{\log_{36} 225-1}$ $225^t/(t^{\log_{54} 225}M^{\log_{54} 225-1})$	Thm 1 Thm 2	No Up to polylog factor
	$\langle 18,18,18\rangle = 3375$	None	$3375^t/M^{\log_{324} 3375-1}$	[5]	Yes

5.2 The Hopcroft-Kerr Algorithm

Hopcroft and Kerr [19] provide an algorithm for $\langle 3,2,3\rangle = 15$, and prove that fewer than 15 scalar multiplications is not possible. In their algorithm, all the encoding and decoding graphs are connected. Thus, only Theorems 1 and 2 are necessary for proving the lower bounds. For the square case $\langle 18,18,18\rangle = 3375$, Theorem 1 reproduces the result of [5].

6 Discussion and Open Problems

Using graph expansion analysis we obtain tight lower bounds on recursive rectangular matrix multiplication algorithms in the case that the output matrix is at least as large as the input matrices, and the decoding graph is connected. We also obtain a similar bound in the case that the encoding graph of the largest matrix is connected, which is tight up to a factor that is polylogarithmic in the input, assuming no multiply copied inputs. Finally we extend these bounds to some disconnected cases, with restrictions on the fast memory size. Whenever the decoding graph is not the largest of the three subgraphs (equivalently, whenever the output matrix is smaller than one of the input matrices), or when the largest graph is disconnected, our bounds are not tight.

6.1 Limitations of the Lower Bounds

There are several cases when our lower bounds do not apply. These are cases where the full algorithm is a hybrid of several base algorithms combined in an arbitrary sequence. Consider the case where two base algorithms are applied recursively. If the recursion alternates between them, our lower bounds apply to the tensor product of the two base cases, which can be thought of as taking two recursive steps at once. However, for cases of arbitrary choice of which base case to apply at each recursive step, we do not provide communication cost lower bounds. The technical difficulty in extending our results in this case lies in generalizing the recursive construction of the decoding graph given in Section 4.1. Similarly, if the base-case decoding (or encoding) graph is disconnected and contains several connected components of different sizes, our bounds do not apply. In this case the connected components of the entire decoding (or encoding) graph are constructed out of all possible interleavings of the different connected components. Finally, the lower bounds do not apply to algorithms that are not recursive, including approximate algorithms that are not bilinear.

6.2 Parallel Case

Although our main focus is on the sequential case, we note that the sequential communication bounds presented here can be generalized to communication bounds in the distributed-memory parallel model of [3]. The lower bound proof technique here can be extended to obtain both memory-dependent and memory-independent parallel bounds as in [2]. Further, the Communication Avoiding Parallel Strassen *(CAPS)* algorithm presented in [3] is shown to be communication-optimal and faster (both theoretically and empirically) than previous attempts to parallelize Strassen's algorithm [6]. The parallelization approach of CAPS is general, and in particular it can be applied to rectangular matrix multiplication, giving a communication upper bound which matches the lower bounds in the same circumstances as in the sequential case.

6.3 Blackbox Use of Fast Square Matrix Multiplication Algorithms

Instead of using a fast rectangular matrix multiplication algorithm, one can perform rectangular matrix multiplication of the form $\langle m^t, n^t, p^t \rangle$ with fewer than the naïve number of $(mnp)^t$ multiplications by blackbox use of a square matrix multiplication algorithm with exponent ω_0 (that is, an algorithm for multiplying $n \times n$ matrices with $O(n^{\omega_0})$ flops). The idea is to break up the original problem into $\left(\frac{m^t}{n^t}\right) \cdot \left(\frac{p^t}{n^t}\right)$ square matrix multiplication problems of size

$(n^t) \times (n^t)$.[7] The arithmetic cost of such a blackbox algorithm is $\Theta((mpn^{\omega_0-2})^t)$. Using the upper and lower bounds in [5], the communication cost is $\Theta\left(\frac{(mpn^{\omega_0-2})^t}{M^{\omega_0/2-1}}\right)$.

We note that, in some cases, blackbox use of a square algorithm may give a lower communication cost than a rectangular algorithm, even if it has a higher arithmetic cost. In particular, if $q < mpn^{\omega_0-2}$, then the rectangular algorithm performs asymptotically fewer flops. It is possible to have simultaneously $\omega_0/2 > \log_{mp} q$, meaning that for certain values of M and t the communication cost of the rectangular algorithm is higher. On some machines, the arithmetically slower algorithm may require less total time if the communication cost dominates.

References

1. Alon, N., Schwartz, O., Shapira, A.: An elementary construction of constant-degree expanders. Combinatorics, Probability & Computing 17(3), 319–327 (2008)
2. Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., Schwartz, O.: Brief announcement: strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. In: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2012, pp. 77–79. ACM, New York (2012)
3. Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., Schwartz, O.: Communication-optimal parallel algorithm for Strassen's matrix multiplication. In: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2012, pp. 193–204. ACM, New York (2012)
4. Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Minimizing communication in numerical linear algebra. SIAM J. Matrix Analysis Applications 32(3), 866–901 (2011)
5. Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Graph expansion and communication costs of fast matrix multiplication. J. ACM (accepted, 2012)
6. Ballard, G., Demmel, J., Lipshitz, B., Schwartz, O.: Communication-avoiding parallel Strassen: Implementation and performance. In: Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2012, ACM, New York (2012)
7. Beling, P., Megiddo, N.: Using fast matrix multiplication to find basic solutions. Theoretical Computer Science 205(1-2), 307–316 (1998)
8. Bilardi, G., Pietracaprina, A., D'Alberto, P.: On the Space and Access Complexity of Computation DAGs. In: Brandes, U., Wagner, D. (eds.) WG 2000. LNCS, vol. 1928, pp. 47–58. Springer, Heidelberg (2000)
9. Bilardi, G., Preparata, F.: Processor-time tradeoffs under bounded-speed message propagation: Part II, lower boundes. Theory of Computing Systems 32(5), 1432–4350 (1999)
10. Bini, D.: Relations between exact and approximate bilinear algorithms. applications. Calcolo 17, 87–97 (1980), doi:10.1007/BF02575865
11. Bini, D., Capovani, M., Romani, F., Lotti, G.: $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. Information Processing Letters 8(5), 234–235 (1979)

[7] Assume, for simplicity, that $n < m, p$.

12. Bűrgisser, P., Clausen, M., Shokrollahi, M.A.: Algebraic Complexity Theory. Grundlehren der mathematischen Wissenschaften, vol. 315. Springer (1997)
13. Coppersmith, D.: Rapid multiplication of rectangular matrices. SIAM Journal on Computing 11(3), 467–471 (1982)
14. Coppersmith, D.: Rectangular matrix multiplication revisited. J. Complex. 13, 42–49 (1997)
15. Fischer, P., Probert, R.: Efficient Procedures for Using Matrix Algorithms. In: Loeckx, J. (ed.) ICALP 1974. LNCS, vol. 14, pp. 413–427. Springer, Heidelberg (1974)
16. Galil, Z., Pan, V.: Parallel evaluation of the determinant and of the inverse of a matrix. Information Processing Letters 30(1), 41–45 (1989)
17. Hong, J.W., Kung, H.T.: I/O complexity: The red-blue pebble game. In: STOC 1981: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, pp. 326–333. ACM, New York (1981)
18. Hopcroft, J., Musinski, J.: Duality applied to the complexity of matrix multiplications and other bilinear forms. In: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, STOC 1973, pp. 73–87. ACM, New York (1973)
19. Hopcroft, J.E., Kerr, L.R.: On minimizing the number of multiplications necessary for matrix multiplication. SIAM Journal on Applied Mathematics 20(1), 30–36 (1971)
20. Huang, X., Pan, V.Y.: Fast rectangular matrix multiplications and improving parallel matrix computations. In: Proceedings of the Second International Symposium on Parallel Symbolic Computation, PASCO 1997, pp. 11–23. ACM, New York (1997)
21. Huang, X., Pan, V.Y.: Fast rectangular matrix multiplication and applications. J. Complex. 14, 257–299 (1998)
22. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. J. Parallel Distrib. Comput. 64(9), 1017–1026 (2004)
23. Kaplan, H., Sharir, M., Verbin, E.: Colored intersection searching via sparse rectangular matrix multiplication. In: Proceedings of the Twenty-Second Annual Symposium on Computational Geometry, SCG 2006, pp. 52–60. ACM, New York (2006)
24. Ke, S., Zeng, B., Han, W., Pan, V.: Fast rectangular matrix multiplication and some applications. Science in China Series A: Mathematics 51, 389–406 (2008), doi:10.1007/s11425-007-0169-2
25. Knight, P.: Fast rectangular matrix multiplication and QR decomposition. Linear Algebra and its Applications 221, 69–81 (1995)
26. Koucky, M., Kabanets, V., Kolokolova, A.: Expanders made elementary (2007) (in preparation), http://www.cs.sfu.ca/~kabanets/papers/expanders.pdf
27. Kratsch, D., Spinrad, J.: Between $O(nm)$ and $O(n)$?. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2003, pp. 709–716. Society for Industrial and Applied Mathematics, Philadelphia (2003)
28. Lev, G., Valiant, L.G.: Size bounds for superconcentrators. Theoretical Computer Science 22(3), 233–251 (1983)
29. Lotti, G., Romani, F.: On the asymptotic complexity of rectangular matrix multiplication. Theoretical Computer Science 23(2), 171–185 (1983)
30. Mihail, M.: Conductance and convergence of Markov chains: A combinatorial treatment of expanders. In: Proceedings of the Thirtieth Annual IEEE Symposium on Foundations of Computer Science, pp. 526–531 (1989)

31. Reingold, O., Vadhan, S., Wigderson, A.: Entropy waves, the zig-zag graph product, and new constant-degree expanders. Annals of Mathematics 155(1), 157–187 (2002)
32. Savage, J.: Space-time tradeoffs in memory hierarchies. Technical report, Brown University, Providence, RI, USA (1994)
33. Strassen, V.: Gaussian elimination is not optimal. Numer. Math. 13, 354–356 (1969)
34. Yuster, R., Zwick, U.: Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, pp. 254–260. Society for Industrial and Applied Mathematics, Philadelphia (2004)
35. Yuster, R., Zwick, U.: Fast sparse matrix multiplication. ACM Trans. Algorithms 1(1), 2–13 (2005)
36. Zwick, U.: All pairs shortest paths using bridging sets and rectangular matrix multiplication. J. ACM 49, 289–317 (2002)

A Details of Bini's and the Hopcroft-Kerr Algorithm

In this appendix we give the details of Bini's algorithm [11] and the Hopcroft-Kerr algorithm [19].

We express an algorithm for $\langle m, n, p\rangle = q$ matrix multiplication by giving the three adjacency matrices of the encoding and decoding graphs: U of dimension $mn \times q$, V of dimension $np \times q$, and W of dimension $mp \times q$. The rows of U, V, and W, correspond to the entries of A, B, and C, respectively, in row-major order. The columns correspond to the q multiplications. To be precise, each column of U specifies a linear combination of entries of A; and each column of V specifies a linear combination of entries of B. These two linear combinations are to be multiplied together, and then the corresponding column of W specifies to which entries of C that product contributes, and with what coefficient.[8]

A.1 Bini's Algorithm

We provide all 6 base cases for Bini's algorithm that appear is Section 5.1. They are labeled by the shape of the multiplication and which graph is disconnected. The first algorithm is:

$$U^{\langle 3,2,2\rangle, EncA} = \begin{bmatrix} 1&0&1&0&1&0&0&0&0&0 \\ 0&0&0&\lambda&\lambda&0&0&0&0&0 \\ 0&0&0&0&0&1&0&1&0&1 \\ 1&1&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&\lambda&\lambda \\ 0&0&0&0&0&1&1&0&1&0 \end{bmatrix} \equiv \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix}$$

[8] The sparsity of the matrices in this notation correspond loosely to the number of additions and subtractions, but this notation is not sufficient to specify the leading constant hidden in the computational costs. In particular, this notation does not show the advantage of Winograd's variant of Strassen's algorithm [15] over Strassen's original formulation [33].

$$V^{\langle 3,2,2\rangle,EncA} = \begin{bmatrix} \lambda & 0 & 0 & -\lambda & 0 & 1 & 1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & \lambda & 0 & 0 & -1 & 0 & 1 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & \lambda & 0 \\ 1 & -1 & 1 & 0 & 1 & \lambda & 0 & 0 & 0 & -\lambda \end{bmatrix} \equiv \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix}$$

$$W^{\langle 3,2,2\rangle,EncA} = \begin{bmatrix} \lambda^{-1} & \lambda^{-1} & -\lambda^{-1} & \lambda^{-1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\lambda^{-1} & 0 & \lambda^{-1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\lambda^{-1} & 0 & \lambda^{-1} & 0 \\ 0 & 0 & 0 & 0 & 0 & \lambda^{-1} & -\lambda^{-1} & \lambda^{-1} & 0 & \lambda^{-1} \end{bmatrix} \equiv \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \end{bmatrix}$$

The remaining 5 algorithms can be concisely expressed in terms of the rows of the first algorithm:

$$U^{\langle 3,2,2\rangle,DecC} = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \end{bmatrix} \qquad V^{\langle 3,2,2\rangle,DecC} = \begin{bmatrix} V_1 \\ V_3 \\ V_2 \\ V_4 \end{bmatrix} \qquad W^{\langle 3,2,2\rangle,DecC} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix}$$

$$U^{\langle 2,3,2\rangle,EncA} = \begin{bmatrix} U_1 \\ U_3 \\ U_5 \\ U_2 \\ U_4 \\ U_6 \end{bmatrix} \qquad V^{\langle 2,3,2\rangle,EncA} = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \end{bmatrix} \qquad W^{\langle 2,3,2\rangle,EncA} = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix}$$

$$U^{\langle 2,3,2\rangle,EncB} = \begin{bmatrix} W_1 \\ W_3 \\ W_5 \\ W_2 \\ W_4 \\ W_6 \end{bmatrix} \qquad V^{\langle 2,3,2\rangle,EncB} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix} \qquad W^{\langle 2,3,2\rangle,EncB} = \begin{bmatrix} V_1 \\ V_3 \\ V_2 \\ V_4 \end{bmatrix}$$

$$U^{\langle 2,2,3\rangle,EncB} = \begin{bmatrix} V_1 \\ V_3 \\ V_2 \\ V_4 \end{bmatrix} \qquad V^{\langle 2,2,3\rangle,EncB} = \begin{bmatrix} U_1 \\ U_3 \\ U_5 \\ U_2 \\ U_4 \\ U_6 \end{bmatrix} \qquad W^{\langle 2,2,3\rangle,EncB} = \begin{bmatrix} W_1 \\ W_3 \\ W_5 \\ W_2 \\ W_4 \\ W_6 \end{bmatrix}$$

$$U^{\langle 2,2,3\rangle,DecC} = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix} \qquad V^{\langle 2,2,3\rangle,DecC} = \begin{bmatrix} W_1 \\ W_3 \\ W_5 \\ W_2 \\ W_4 \\ W_6 \end{bmatrix} \qquad W^{\langle 2,2,3\rangle,DecC} = \begin{bmatrix} U_1 \\ U_3 \\ U_5 \\ U_2 \\ U_4 \\ U_6 \end{bmatrix}$$

A.2 The Hopcroft-Kerr Algorithm

For the Hopcroft-Kerr algorithm we give only 3 of the 6 base cases, since all the graphs are connected.

$$U^{\langle 3,2,3\rangle} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & 1 & -1 \end{bmatrix} \equiv \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix}$$

$$V^{\langle 3,2,3\rangle} = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \equiv \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ V_6 \end{bmatrix}$$

$$W^{\langle 3,2,3\rangle} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \equiv \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \\ W_7 \\ W_8 \\ W_9 \end{bmatrix}$$

$$U^{\langle 2,3,3\rangle} = \begin{bmatrix} U_1 \\ U_3 \\ U_5 \\ U_2 \\ U_4 \\ U_6 \end{bmatrix} \qquad V^{\langle 2,3,3\rangle} = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \\ W_7 \\ W_8 \\ W_9 \end{bmatrix} \qquad W^{\langle 2,3,3\rangle} = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ V_6 \end{bmatrix}$$

$$U^{\langle 3,3,2\rangle} = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \\ W_7 \\ W_8 \\ W_9 \end{bmatrix} \qquad V^{\langle 3,3,2\rangle} = \begin{bmatrix} V_1 \\ V_4 \\ V_2 \\ V_5 \\ V_3 \\ V_6 \end{bmatrix} \qquad W^{\langle 3,3,2\rangle} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix}$$

Multicast Routing for Energy Minimization Using Speed Scaling

Nikhil Bansal[1,⋆], Anupam Gupta[2,⋆⋆], Ravishankar Krishnaswamy[3,⋆⋆⋆], Viswanath Nagarajan[4], Kirk Pruhs[5,†], and Cliff Stein[6,‡]

[1] Mathematics and Computer Science, Eindhoven University of Technology
[2] Computer Science, Carnegie Mellon University
[3] Computer Science, Princeton University
[4] IBM T.J. Watson Research Center
[5] Computer Science, University of Pittsburgh
[6] Industrial Engineering and Operations Research, Columbia University

Abstract. We consider virtual circuit multicast routing in a network of links that are speed scalable. We assume that a link with load f uses power $\sigma + f^\alpha$, where σ is the static power, and $\alpha > 1$ is some constant. We assume that a link may be shutdown if not in use. In response to the arrival of client i at vertex t_i a routing path (the virtual circuit) P_i connecting a fixed source s to sink t_i must be established. The objective is to minimize the aggregate power used by all links.

We give a polylog-competitive online algorithm, and a polynomial-time $O(\alpha)$-approximation offline algorithm if the power functions of all links are the same. If each link can have a different power function, we show that the problem is APX-hard. If additionally, the edges may be directed, then we show that no poly-log approximation is possible in polynomial time under standard complexity assumptions. These are the first results on multicast routing in speed scalable networks in the algorithmic literature.

1 Introduction

The amount of energy used by data networks is significant, worldwide more 50 billion kWH are used per year according to a US Department of Energy study [1]. As the number of processors per chip grows, interprocessor communication is widely expected to become the dominant energy component for computer chips. Thus there has been significant recent interest, both within industry

⋆ Supported in part by the NWO Vidi grant 639.022.211.
⋆⋆ Supported in part by NSF awards CCF-0964474 and CCF-1016799.
⋆⋆⋆ Supported in part by NSF awards CCF-0964474 and CCF-1016799 and an IBM Graduate Fellowship.
† Supported in part by NSF grants CCF-0830558, CCF-1115575, CNS-1253218 and an IBM Faculty Award.
‡ Supported in part by NSF grant CCF-0915681.

and academia, to develop methods to manage networks, from on-chip up to wide-area, in a more energy efficient manner. The same US Department of Energy study [1] estimates that at least a 40% reduction in wide-area network energy would be possible if network components were able to dynamically adjust their power in response to the traffic experienced. Simulation results by Benoit et al. [5] suggest using speed scalable link technology, such as those proposed by Kim and Horowitz [11], would save significant energy in an on-chip network for a chip multiprocessor.

Virtual circuit routing is a common means of providing reliable communications in a network. A virtual circuit gives the user a reserved portion of the network with a guaranteed bandwidth in which to route messages. Algorithmic problems associated with virtual circuit routing have been studied for many years (e.g. [4,7]). Here we consider algorithmic problems that combine the traditional virtual circuit routing with the the traditional energy management mechanisms: changing the speed of a component and/or shutting the component off.

We consider the same setting as was proposed in several previous papers. The network is a graph, with a power function associated with each edge.[1] The power used by edge e as

$$\mathrm{en}_e(f_e) = \begin{cases} \sigma_e + f_e^\alpha & \text{if } f_e > 0 \\ 0 & \text{if } f_e = 0 \end{cases}, \tag{1}$$

where f_e is the number of circuits (flow) passing through each edge, σ is the static power, and $\alpha > 0$ is a given constant. An important special case is when all edges have the same fixed cost σ. We call such cost functions *homogeneous* and call the general case *heterogeneous*. We consider both directed and undirected graphs, unless explicitly stated graphs are undirected. The input further consists of a sequence of requests for the establishment of a virtual circuit. In response to the arrival of a source-sink request (s_i, t_i) a routing path (the virtual circuit) P_i connecting s_i to t_i must be established. The objective is to minimize the aggregate power used over all the edges.

Andrews et al. [2] gave a polynomial-time poly-log-approximation algorithm if the graph is undirected, and edges e have homogeneous cost functions. It was previously noted [3] that a polynomial-time algorithm with a better than poly-log approximation ratio would violate standard complexity theoretic assumptions. Gupta et al. [8] considered online algorithms and gave an α^α-competitive online algorithm under the same assumptions, plus the additional assumption that the static power σ_e for each edge was zero.

In this paper we consider applications in which there is a common source vertex, as would be the case if a multicast communication pattern was implemented using unicast. We consider both the online and offline cases and give several positive and negative results:

– For undirected networks and homogeneous power functions of the form given in (1), in Section 3 we give a poly-log competitive online algorithm.

[1] Power functions are associated with edges and not vertices primarily because the resulting algorithmic problems are more tractable.

(These are the same assumptions under which an offline poly-log approximation algorithm was given by [2], although the result of [2] holds more generally without the single source assumption.) This result shows that poly-log competitiveness can be achieved by an online algorithm for multicast communication.
- For undirected networks and homogeneous power functions of the form given in (1), in Section 4, we give an $O(\alpha)$ offline approximation algorithm.
- In Section 5.1, we show that if the graph is directed instead of undirected, and we allow heterogeneous power functions, then even for the s–t routing case (i.e. when all the sinks are also located at the same node), there is no polynomial-time poly-log-approximation algorithm under a standard complexity theoretic assumption.
- In Section 5.2, we show that for the heterogeneous case in an undirected graph, the offline problem is APX-hard even in the s–t case.

In Section 6 we discuss some of the natural open questions arising from these results.

2 Notation and Background

In the Energy-Aware Routing Problem (EERP), we are given an undirected graph $G = (V, E)$ with $|V| = n$ and a distinguished source vertex s. We are given k sinks $R_k = \{t_1, t_2, \ldots, t_k\}$ corresponding to k different routing requests, One unit of flow needs to be routed from the source s to each sink t_i, on a single s-t_i path P_i. Given a solution $P_1, P_2, \ldots, P_k$, the flow on edge e, f_e, is the number of paths that use e, that is $f_e = \sum_{i:e\in P_i} 1$, and the energy used by edge e is defined by equation (1) above. For our positive results, we consider the homogeneous case where all the edges have a common σ. Furthermore, we are interested in the case where $1 \ll \sigma$ and $\alpha > 1$. (If σ is small, for the purposes of approximation, it can be treated as 0, and if $\alpha < 1$, this problem exhibits economies of scale and has been well studied.) The objective is to find a feasible routing which minimizes the total power, which is obtained by summing the power usage over all edges. In other words, the total power used is

$$\sum_{e\in E} \mathrm{en}_e = \sum_{e\in E} \left(\sigma \cdot \mathbf{1}_{(e\in \cup_i P_i)} + \left(\sum_{i=1}^{k} \mathbf{1}_{(e\in P_i)} \right)^{\alpha} \right) \quad (2)$$

We will usually refer to the power used as the *cost incurred.*

We consider both the offline and online variants: in the online variant, the sinks are given online and we must choose a path for sink t_i before learning about sink t_{i+1}.

The fixed σ term is called the *buying* or *opening* cost: the net buying cost is the first term in (2). The second term is sometimes called the *renting* cost. In designing our algorithms, we will sometimes want to balance the renting and buying costs, we therefore define $q = \sigma^{1/\alpha}$, the minimum number of paths that must use an edge for the renting cost to be at least the buying cost.

We will compare our results to an optimal solution and use $\mathsf{opt}(R_k)$ to denote the cost incurred by the set of optimal paths $\{P_1^*, P_2^*, \ldots, P_k^*\}$.

Steiner Trees: We will use Steiner trees, both in our algorithm design and for showing lower bounds. Given $S \subseteq V$, a *min-cost Steiner tree on S* is a tree T in G with the fewest edges that connects all the nodes in S. We denote the number of edges in the min-cost Steiner tree by $\mathsf{StTree}(S)$. Since the union of the paths P_i in any solution $\{P_i\}_{i=1}^k$ to EERP also connects $\{s\} \cup R_k$, we get that the optimal buying cost is at least $\sigma \cdot \mathsf{StTree}(\{s\} \cup R_k)$, and hence

$$\mathsf{opt}(R_k) \geq \sigma \cdot \mathsf{StTree}(\{s\} \cup R_k). \tag{3}$$

We will sometimes use opt as shorthand for $\mathsf{opt}(R_k)$.

Probabilistic Bounds: Our analysis will use the following result on the expectations of the sums of powers of random variables.

Theorem 1 ([10,13]). *Let $X_1, X_2, \ldots, X_N$ be independent non-negative random variables. Let $\alpha > 1$ and $K_\alpha = \Theta(\alpha / \log \alpha)$. Then it is the case that* $\left(\mathbb{E}[(\sum_i X_i)^\alpha]\right)^{1/\alpha} \leq K_\alpha \max\left(\sum_i \mathbb{E}[X_i], \left(\sum_i \mathbb{E}[X_i^\alpha]\right)^{1/\alpha}\right)$.

Corollary 1. *Let $p \geq 0$, and let $X_1, X_2, \ldots, X_n$ be i.i.d. random variables taking value D with probability* $\max\{1, p\}$. *Then* $\mathbb{E}[(\sum_i X_i)^\alpha] \leq (K_\alpha)^\alpha \cdot \max\{1, pN\, D^\alpha + (pND)^\alpha\}$, *where $K_\alpha = \Theta(\alpha / \log \alpha)$.*

Proof. For the case when $p \geq 1$, $X_i = D$ with probability 1, and hence we can conclude that $\mathbb{E}[(\sum_i X_i)^\alpha] = (ND)^\alpha$. For the case when $p \in [0,1]$, $\mathbb{E}[X_i] = pD$, and $\mathbb{E}[X_i^\alpha] = pD^\alpha$. From this we can conclude that the upper bound in Theorem 1 is $K_\alpha \max(pND, (pN)^{1/\alpha} D)$. Taking α^{th} powers and replacing the max by a sum, we get $\mathbb{E}[(\sum_i X_i)^\alpha] \leq (K_\alpha\, D)^\alpha((pN)^\alpha + pN)$. □

3 Online Algorithm for Homogeneous Setting

We now give an online algorithm, in which the sinks arrive one-by-one, and we have to choose the path P_i connecting t_i to the source s before knowing the identity of the next sink t_{i+1}.

3.1 The Algorithm

We assume (without loss of generality, by adding zero cost edges) that each sink appears at a distinct vertex; so the number of sinks $k \leq n$ the number of vertices in G. We let $R_i = \{t_1, t_2, \ldots, t_i\}$ denote the set of demands that have already arrived and let $R_i' = \{s\} \cup R_i$. The algorithm maintains a tree T_i that initially contains only the source s. When a request t_i arrives, a path P_i will be chosen and the tree will be updated, however, the routing paths may use edges outside this tree. We will also maintain a subset of the vertices which we call *leaders*. The set of leaders in step i is denoted by L_i, and is initialized to $L_0 = \{s\}$. We will maintain various counters, all initialized to 0. A counter Λ_j will denote the number of nodes assigned to a particular leader j. A counter ρ_e will count the number of times an edge has been used in non-tree paths from the source to a leader and a counter λ_e will count the number of times an tree-edge is used to route flow.

We will denote trees and paths by their sets of edges.

When a sink t_i arrives, we do the following:

1. Let $\tilde{P}_i$ be the shortest path from t_i to any node in R'_{i-1} and let $T_i = T_{i-1} \cup \tilde{P}_i$.
2. With probability $\min\{1, \frac{c \log n}{q}\}$, let t_i be a leader (i.e., set $L_i = L_{i-1} \cup \{t_i\}$), else let $L_i = L_{i-1}$.
   If t_i is a leader, find a path Q_i from t_i to the root that minimizes
   $$\sum_{e \in Q_i} \left((\rho_e + 1)^\alpha - (\rho_e)^\alpha\right). \tag{4}$$
   For each edge e on the path Q_i, set $\rho_e \leftarrow \rho_e + 1$.
3. Choose the leader $j \in L_i$ that minimizes the expression:
   $$\begin{aligned} &(3/2)^{(\Lambda_j+1)/(q \log n)} - (3/2)^{\Lambda_j/(q \log n)} \\ &+ \sum_{e \in T_i[t_i, j]} \left((3/2)^{(\lambda_e+1)/(q \log n)} - (3/2)^{\lambda_e/(q \log n)}\right) \end{aligned} \tag{5}$$
   where $T_i[a, b]$ is the unique path between nodes a and b in the tree T_i. If this minimizer is the leader $j^* \in L_i$, set $\Lambda_{j^*} \leftarrow \Lambda_{j^*} + 1$; for every edge e in $T_i[t_i, j^*]$, set $\lambda_e \leftarrow \lambda_e + 1$.
4. Set the s-t_i path P_i to be Q_{j^*} (which is a s-j^* path) concatenated with $T_i[j^*, t_i]$. We call the latter part of the path P_i to be the "tree" portion, and the former to be the "non-tree" portion.

Above, $c > 1$ is some constant. The choice of expression (5) is from the online path selection algorithm to minimize the congestion (maximum load) [4]; similarly the choice of (4) is from the online algorithm to minimize the sum of α^{th} powers of edge loads [8].

3.2 Analysis

The algorithm maintains the tree T_i, the source to leader paths Q_{j^*} and the routing paths P_i. Recall that the flow on an edge e is defined from the P_i. Observe that it is an easy consequence of the algorithm that after step k' there is non-zero flow on all edges in $T_{k'} \cup \cup_{j^* \in L_{k'}} Q_{j^*}$, and hence we have "bought" all of these edges. In the following analysis, we will use k as an index into the number of sinks, rather than (necessarily) the total number of sinks. We first bound the cost associated with the tree-edges. We divide the analysis into buying cost (the number of tree edges) and renting cost (a function of the load on the tree edges).

Claim 1. *The total buying cost for edges in T_k is at most $O(\log k)\mathsf{opt}(R_k)$.*

Proof. The edges in T_k are all bought in Step 1. This step implements a greedy Steiner tree algorithm, and hence the number of edges bought by the greedy Steiner tree algorithm is at most $O(\log k)$ times the optimal Steiner tree on R'_k [9]. As mentioned in Section 2 the optimal Steiner tree gives a lower bound on $\mathsf{opt}(R_k)$, and we get that the total cost of edges bought in Step 1 is at most $O(\log k)\mathsf{opt}$. □

Now we show that in step 3, no edge is used too many times.

Lemma 1. *With probability at least $1 - n^{-2\alpha}$ over the choice of the leaders:*

1. *There exists an assignment of sinks to leaders so that (a) each edge of T_k is used in the tree portion of at most $O(q \log n)$ sinks, and (b) each leader is assigned at most $O(q)$ sinks.*
2. *Thus our algorithm obtains a path assignment in Step 3 where each edge of T_k is used $O(q \log^2 n)$ times and each leader is assigned $O(q \log n)$ sinks.*

Proof. Each sink becomes a leader with probability $\min\{1, \frac{c \log n}{q}\}$. So if $q \leq c \log n$ then no edges are ever used and each leader is only assigned one sink (namely itself). For the rest of the argument we assume that $q \geq c \log n$.

For the sake of the analysis, we choose the leaders in a different way: for each sink, we flip $c \log n$ independent coins of bias $1/q$, if the i^{th} coin is the first of these to turn up heads, we designate the sink as a leader of color i. If all coins turn up tails, the sink is not considered a leader. Since the probability that a sink is a leader (of any color) according to this process is $1 - (1 - 1/q)^{c \log n} \leq \frac{c \log n}{q}$, proving the statements with this new way of choosing leaders also proves the original statement (via a standard coupling argument).

Consider the tree T_k and partition it into connected edge-disjoint groups, so that each group (apart from possibly one) contains between $2q$ and $4q$ sinks. This partition can be achieved by rooting the tree T_k, making it binary by adding dummy edges, and repeatedly choosing the deepest subtree containing at least $2q$ sinks. Consider a particular group S of sinks, and order the sinks in S according to their arrival times. The probability that there is at least one leader of color 1 among the first q sinks in S is at least $1 - (1 - 1/q)^q \geq 1/2$. For such groups, at least half their sinks can route to this leader of color 1. Since the groups were chosen to included edge-disjoint parts of the tree, this routing incurs a maximum load of $4q$ on any edge of the tree (and on any color-1 leader). Also, this reduces the number of remaining sinks to $3k/4$ in expectation. Now we can recurse on the remaining sinks: form edge-disjoint groups on them, and assign $3/4$ fraction of these sinks (in expectation) to leaders of color 2, and continue. After repeating this process $c \log n$ times, the expected number of unassigned sinks is at most $(3/4)^{c \log n} \cdot k \leq \frac{k}{n^{3\alpha}} \leq \frac{n}{n^{3\alpha}}$, by setting $c \geq 9\alpha$. Thus (by Markov's inequality) there is no unassigned sink with probability at least $1 - n^{-2\alpha}$.

Altogether, this assignment routes at most $4q$ sinks to each leader, and uses each edge of T_k at most $4c \log n \cdot q$ times. This proves the first part of the lemma.

If each edge of T_k is given capacity $O(q \log n)$ and each leader is given capacity $O(q)$, the above path assignment corresponds to a solution having congestion (i.e. load/capacity) one for the following routing problem: each sink t_i has to use edges of T_k to route unit flow from any node of L_i (i.e. leader among $\{t_1, \dots, t_i\}$). The result of Aspnes et al. [4] implies that path assignment according to (5) gives a solution with congestion $O(\log n)$ times the optimal. This proves the second part of the lemma. □

We now combine the previous two arguments.

Lemma 2. *The expected total cost (from both buying and renting) incurred by the algorithm on the tree portions of the paths $\{P_i\}_{i=1}^k$ is $O(\log^{2\alpha} n \log k)\mathsf{opt}(R_k)$.*

Proof. By Claim 1, the buying cost for the tree edges is $O(\log k)\mathsf{opt}(R_k)$, and hence the number of edges bought $|T_k| \le O(\log k)\mathsf{opt}(R_k)/\sigma$.

By the second part of Lemma 1, with probability at least $1 - n^{-2\alpha}$, each edge of T_k carries at most $O(q \log^2 n)$ flow. Thus the expected renting cost incurred over T_k is at most $|T_k| \cdot O(q \log^2 n)^\alpha + \frac{1}{n^{2\alpha}} |T_k| \cdot k^\alpha \le O(\log n)^{2\alpha} |T_k| \cdot \sigma = O(\log k \cdot \log^{2\alpha} n)\mathsf{opt}(R_k)$. □

We now bound the cost of using the edges in the non-tree portion $\cup Q_{j*}$.

Lemma 3. *Consider the following random experiment: choose a random subset S of sinks, with each sink t_i chosen independently with probability $\min\{1, \frac{c \log n}{q}\}$; thereafter for each $t_i \in S$, send $\Theta(q \log n)$ flow from s to t_i on its optimal path P_i^*. The expected cost (both buying and renting) incurred by this routing is $O(\log^{2\alpha} n)\mathsf{opt}(R_k)$.*

Proof. Since we buy a subset of the edges bought by the optimal solution, the buying cost is bounded by $\mathsf{opt}(R_k)$. For the expected renting cost, consider an edge e, and all the sinks whose optimal paths P_i^* use e: if there are N of them, the optimal's renting cost for e is N^α. Since each sink chooses independently, we can use Corollary 1 with $p = \frac{c \log n}{q}$ and $D = c' q \log n$ to bound the expected renting cost for e. Ignoring terms of the form $O(\alpha)^\alpha$ and using $q^\alpha = \sigma$, we get

$$pND^\alpha + (pND)^\alpha \approx (\log^{\alpha+1} n)\, \sigma N/q + (\log^{2\alpha} n)\, N^\alpha \le (\log^{2\alpha} n)(2N^\alpha + \sigma),$$

which is the claimed polylogarithmic factor times the optimal's cost incurred on this edge. (For the last inequality, observe that if $q \le N$ then $\sigma N/q = q^{\alpha-1} N \le N^\alpha$, and if $q > N$ then $\sigma N/q \le \sigma$.) Now summing over all edges, and using linearity of expectations completes the proof. □

Lemma 4. *The expected cost incurred by our algorithm for routing on the non-tree edges $\cup_{j^*} Q_{j*}$ is at most $O(\log^{2\alpha} n)\mathsf{opt}(R_k)$.*

Proof. Consider a random instance on the original graph where each sink is activated independently (as leader) with probability $\frac{c \log n}{q}$ and requires $\Theta(q \log n)$ unsplittable flow from s, with the objective of minimizing $\sum_{e : f_e > 0} (\sigma + f_e^\alpha)$ where

f_e denotes the flow on edge e. Since the routing is unsplittable, each positive f_e has $f_e \geq \Omega(q \log n) \geq q$; so $\sigma + f_e^\alpha \leq 2 \cdot f_e^\alpha$. Thus (up to a factor of 2) the objective is simply $\sum_e f_e^\alpha$. By Lemma 3, the expected optimal value of this random instance is $O(\log^{2\alpha} n)\mathsf{opt}(R_k)$. The path selection $\{Q_j\}$ in Step 2 corresponds to an α^α-competitive online algorithm for this random instance [8]. Thus, if we send $O(q \log n)$ flow along each of these paths, the expected cost incurred is $\alpha^\alpha \cdot O(\log^{2\alpha} n)\mathsf{opt}(R_k) = O(\log^{2\alpha} n)\mathsf{opt}(R_k)$.

Now, by the second part of Lemma 1, with probability at least $1 - n^{-2\alpha}$, the number of sinks assigned to each leader is $O(q \log n)$, in which case reserving capacity $O(q \log n)$ on each leader's path from s (as in above instance) suffices. With the remaining $n^{-2\alpha}$ probability, the worst case cost is $nk(\sigma + k^\alpha)$. Thus the expected cost of our algorithm on the non-tree portion is $O(\log^{2\alpha} n)\mathsf{opt}(R_k)$. $\square$

Theorem 2. *There is an $O_\alpha\left(\log^{O(\alpha)} n\right)$-competitive randomized online algorithm for single-source* EERP *with homogeneous power functions of form $\sigma + f_e^\alpha$.*

4 Offline Algorithm for Homogeneous Setting

In this section we give an $O(\alpha)$-approximation algorithm for the *offline* EERP problem. with a homogeneous energy function. The algorithm has two phases (similar to the online setting), aggregation and batched routing. We assume that $\sigma \geq \alpha^\alpha$; otherwise aggregation is not necessary and the algorithm proceeds directly to the network flow instance[2].

Set $p := \sigma^{1/\alpha}/\alpha \geq 1$, which may not be integral. We first describe an algorithm to compute a splittable routing for each sink: then we show that this can be easily converted to an unsplittable routing.

Aggregation: Let T denote an approximately minimum Steiner tree, which we can compute in polynomial time. Using an Euler tour of T we can fractionally partition the k demands to obtain r groups $\{V_j \subseteq \{t_1, \ldots, t_k\}\}_{j=1}^r$ where V_j induces a subtree T_j on T, so that:

- For each group $j \in [r]$, there is positive (fractional) demand only on sinks V_j, which totals to exactly p.
- For each sink t_i, the total demand over all groups is exactly one.
- Each edge of T appears in at most two subtrees $\{T_j\}_{j=1}^r$.

If k is not an integral multiple of p, by adding dummy demand, we can ensure that each group contains exactly p demand (this only affects the approximation ratio by a constant factor). In doing so, we may need to add one fractional demand, for that sink the second condition is modified so that the total demand is equal to the fractional amount.

[2] In this case the algorithm is even simpler: all capacities are one and the copies of an edge are: $\lceil\alpha\rceil$ edges of cost σ each; and for each integer $h \geq \lceil\alpha\rceil + 1$, an edge of cost $h^\alpha - (h-1)^\alpha$. That this is an $O(\alpha)$ approximation, follows easily from the arguments for "Batched Routing" below.

Batched routing: We now define a minimum-cost network flow instance G' corresponding to the above grouping $\{V_j\}_{j=1}^r$ of sinks. We create r new sinks $\{t'_j\}_{j=1}^r$ where each t'_j requires flow p and is connected to all sinks in V_j with zero cost and infinite capacity edges. Then we replace each edge e in G by the following parallel edges:

– There are $\lceil \alpha \rceil$ identical edges with capacity p and cost (per unit flow) $\frac{\sigma}{p}$.
– For each integer $h \geq 0$, there is an edge of capacity p and cost $\left(1 + \frac{h}{\alpha}\right)^{\alpha-1} \cdot \frac{\sigma}{p}$.

We use $g(x) = \sigma + x^\alpha$ to denote the homogeneous power function applied to the flow on one edge. The transformation above corresponds to a natural discretization of the power function into linear pieces, and was also used in Andrews et al. [2]. Let $c_e(x)$ denote the minimum cost way to send x units of flow through the above set of parallel edges corresponding to an edge e. This minimum cost routing uses the edges in order of increasing cost.

Before analyzing our algorithm, we prove two technical claims about the behavior of our cost functions. The choice of our discretization parameter p implies:

Claim 2. *For all $x \geq 0$, $g(x + p) \leq 9 \cdot g(x)$.*

Moreover, by the definition of the parallel edges,

Claim 3. *For each $x \geq 0$, $c(x) \leq (\alpha + 1) \cdot g(x)$; and for each $x \geq p$, $g(x) = O(c(x))$.*

We now return to the description of our algorithm. The algorithm computes a minimum cost flow in this network G' with demands of p units to each of $t'_1, \ldots, t'_r$. Since all capacities and demands are multiples of p, we can obtain in polynomial time (by integrality of single commodity flow) an optimal solution given by paths $\{Q_j\}_{j=1}^r$ where each Q_j is a path from s to some $t^*_j \in V_j$ carrying p flow. For each edge $e \in G$ (the original graph), let f_e denote the total flow sent through "copies" of e in this solution; note that since f_e is a multiple of p, either $f_e = 0$ or $f_e \geq p$. Let $E' \subseteq E$ denote the edges e with $f_e > 0$, so the cost of this solution is $\sum_{e \in E'} c(f_e)$.

Lemma 5. *The cost of the flow $\sum_{e \in E'} c(f_e) \leq (\alpha + 1) \cdot \mathsf{opt}$. Moreover, the total energy cost, $\sum_{e \in E'} g(f_e) = O(\alpha) \cdot \mathsf{opt}$.*

Proof. We will show the existence of a feasible solution of cost $(\alpha + 1) \cdot \mathsf{opt}$ to the above network flow instance: since we obtain an optimal solution, our cost $\sum_{e \in E'} c(f_e)$ is no worse. Consider the optimal paths $\{P^*_i\}_{i=1}^k$ in EERP carrying unit flow to each sink $\{t_i\}_{i=1}^k$. For each group $j \in [r]$ and sink $t \in V_j$ we send $\mathsf{demand}_j(t)$ units of flow to t'_j by extending path P^*_t. The property of the aggregation step ensures that each $\{t'_j\}_{j=1}^r$ receives exactly p flow, and the flow f^*_e through any edge in G is exactly the number of paths $\{P^*_i\}_{i=1}^k$ using it. Thus the cost of this solution is $\sum_e c(f^*_e) \ \leq_{Claim\ 3} \ (\alpha + 1) \sum_e g(f^*_e) = (\alpha + 1) \cdot \mathsf{opt}$.

Next, observe that our optimal solution $\{f_e : e \in E'\}$ has any positive flow at least p. Hence using Claim 3, $\sum_{e \in E'} g(f_e) \leq O(1) \sum_{e \in E'} c(f_e) = O(\alpha) \cdot \mathsf{opt}$. □

Obtaining a solution. We now combine the solutions from the above two phases to obtain a *splittable* EERP routing in G. The flow $\{f_e : e \in E'\}$ sends p units of flow to $t_j^* \in V_j$ for each group $j \in [r]$. Then for each $j \in [r]$, using the edges on subtree T_j, these p units can be redistributed from t_j^* so that each sink $t \in V_j$ gets exactly $\mathsf{demand}_j(t)$ flow, and the flow on each edge of T_j is at most p. This flow is a feasible splittable EERP solution, since the total demand of each sink (over all groups) is one. Since each edge in T appears in at most two subtrees $\{T_j\}_{j=1}^r$, the final flow on any edge e is at most $f_e + 2p$. So the cost of this combined routing is at most:

$$\begin{aligned}\sum_{e\in E'} g(f_e + 2p) + \sum_{e\in T\setminus E'} (\sigma + (2p)^\alpha) &\leq 9^2 \cdot \sum_{e\in E'} g(f_e) + 9^2 \cdot \sum_{e\in T\setminus E'} \sigma \\ &\leq 9^2 \sum_{e\in E'} g(f_e) \; + \; 2 \cdot 9^2 \cdot \mathsf{opt}\end{aligned}$$

The first inequality is by Claim 2 and the definition of p. The second inequality is by the fact that T can be chosen to be a 2-approximate Steiner tree. Finally, using Lemma 5, $\sum_{e\in E'} g(f_e) = O(\alpha) \cdot \mathsf{opt}$, and the total cost of this splittable routing is $O(\alpha) \cdot \mathsf{opt}$.

We now obtain an unsplittable routing. If $\{\ell_e\}_{e\in E}$ denotes the flow in this solution, define a network with each edge $e \in G$ having capacity $\lceil \ell_e \rceil$. The source is s and there is a unit demand at each $\{t_i\}_{i=1}^k$. This instance is feasible as shown by the fractional solution $\{\ell_e\}_{e\in E}$. Again, using integrality of flow, we can find an integer valued flow within these capacities, yielding the unsplittable EERP solution. Using Claim 2, we see that the cost is at most $\sum_e g(\ell_e + 1) \leq \sum_e g(\ell_e + p) \; \leq \; 9 \sum_e g(\ell_e) = O(\alpha) \cdot \mathsf{opt}$.

Theorem 3. *There is an $O(\alpha)$-approximation algorithm for EERP with a single source and homogeneous power functions of the form $\sigma + f_e^\alpha$.*

5 Hardness of Approximation Results

5.1 Hardness of *s*-*t* Directed Routing with Heterogeneous Functions

We now consider the heterogeneous case in which all sinks are located at the same node. Since all sinks are located at the same node, we will speak of a demand r (which is equivalent to the number of sinks k in our original formulation). We therefore want to compute an s-t flow of r units having the minimum total power (summed over all edges). We will consider the case when the exponent α being a small constant that is larger than one, i.e. $\alpha = 1 + \epsilon$ where $\epsilon > 0$ is any constant. The main result of this section is the following inapproximability:

Theorem 4. *The s-t routing problem on directed graphs with heterogeneous power functions and constant exponent $\alpha > 1$ does not admit an approximation ratio better than $2^{\log^{1-\delta} n}$ for any $\delta > 0$, unless $NP \subseteq DTIME(\text{polylog(n)})$.*

Proof. We reduce from the label cover problem. The input is a bipartite graph $(A \cup B, F)$ where each vertex in A and B has degree d, a label set L and a relation $\pi_{a,b} \subseteq L \times L$ for each $(a, b) \in F$. We let $|A| = |B| = n$ and $|F| = m = n \cdot d$. The goal is compute a labeling $\phi : A \cup B \to L$ that maximizes the number of consistent edges, where edge $(a, b) \in F$ is consistent if $(\phi(a), \phi(b)) \in \pi_{a,b}$. For any $\delta > 0$ it is known [12] that unless $NP \subseteq DTIME(\text{polylog(n)})$, there is no polynomial time algorithm to distinguish between:

- Yes instances: with optimal value $|F|$, i.e. there is a labeling with all edges consistent.
- No instances: with optimal value at most $|F|/2^{\log^{1-\delta} n}$, i.e. no labeling makes more than $|F|/2^{\log^{1-\delta} n}$ edges consistent.

The reduction here is similar to one for the related *s-t capacitated network design problem* [6], where each edge has a fixed cost and capacity (instead of a power function), and the goal is to choose a minimum-cost set of edges that support f units of flow from s to t. It is straightforward to reduce s-t capacitated network design to our problem when the exponent $\alpha \approx n$. Below we show that the same hardness persists even for any constant exponent $\alpha > 1$.

The graph $G = (V, E)$ for the heterogeneous power s-t directed routing problem is as follows. The vertex set $V = \{s, t\} \cup A \cup B \cup \{a(u) : a \in A, u \in L\} \cup \{b(u) : b \in B, u \in L\}$. The edge set E contains:

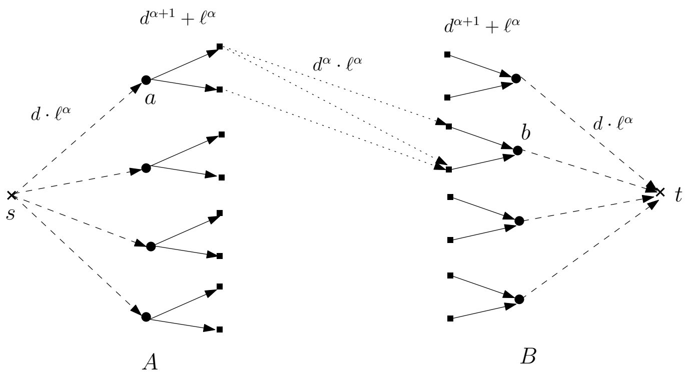
- For each $a \in A$, there is an edge (s, a) with function $d \cdot f^\alpha$. (To ensure functions of the form $\sigma_e + f^\alpha$, we can subdivide this edge into d smaller edges each having function f^α.)
- Similarly, for each $b \in B$, there is an edge (b, t) with function $d \cdot f^\alpha$.
- For each $a \in A$ and $u \in L$, there is an edge $(a, a(u))$ with function $d^{\alpha+1} + f^\alpha$.
- Similarly, for each $b \in B$ and $u \in L$, there is an edge $(b(u), b)$ with function $d^{\alpha+1} + f^\alpha$.
- For each $(a, b) \in F$ and $(u, v) \in \pi_{a,b}$ there is an edge $(a(u), b(v))$ with function $d^\alpha \cdot f^\alpha$. (Again each such edge can be subdivided into d^α edges with function f^α.)

We denote the last set of edges as E'. The flow demand is set to $m = |F| = d \cdot n$ units.

Yes instances: Observe that if the label cover instance has a labeling ϕ that is consistent for all edges, there is a routing in G of total cost at most $7m \cdot d^\alpha$.

No instances: Suppose that there is a routing in G of total power $\rho \cdot m \cdot d^\alpha$. Then we show that one can recover a labeling for the label cover instance that satisfies at least $|F|/\rho^{2+\frac{3}{\alpha-1}}$ edges.

Let f_e denote the flow on edge e in the given routing (having power $\rho \cdot m \cdot d^\alpha$). For each $a \in A$, define $N_a := \{u \in L : f_{(a,a(u))} > 0\}$. Similarly for $b \in B$, $N_b := \{u \in L : f_{(b(u),b)} > 0\}$. We consider an arbitrary flow decomposition of $\{f_e\}_{e \in E}$ into s-t flow-paths (of total value m), and modify it as follows (below $\beta := (4\rho)^{\frac{1}{\alpha-1}}$).



$d^{\alpha+1} + \ell^{\alpha}$
$d^{\alpha+1} + \ell^{\alpha}$
$d^{\alpha} \cdot \ell^{\alpha}$
$d \cdot \ell^{\alpha}$
$d \cdot \ell^{\alpha}$
a
b
s
t
A
B

Fig. 1. The $s - t$ directed network with power functions

1. For each $a \in A$, if $f_{(s,a)} > \beta \cdot d$ or $|N_a| > 4\beta\rho$ then delete all flow-paths through (s, a).
2. For each $b \in B$, if $f_{(b,t)} > \beta \cdot d$ or $|N_b| > 4\beta\rho$ then delete all flow-paths through (b, t).
3. For each $e \in E'$, if $f_e > \beta$ then delete all flow-paths through e.

Claim 4. *The total flow remaining after this pruning is at least* $\frac{m}{4}$.

Proof. We bound the flow lost in each step separately. Consider first the edges $E_a = \{(s, a) : f_{(s,a)} > \beta \cdot d\}$. We have:

$$\frac{\sum_{e\in E_a} f_e}{\sum_{e\in E_a} f_e^{\alpha}} \leq \max_{e\in E_a} f_e^{1-\alpha} \leq \frac{1}{(\beta d)^{\alpha-1}} \implies \sum_{e\in E_a} f_e \leq \frac{\sum_{e\in E_a} f_e^{\alpha}}{(\beta d)^{\alpha-1}}$$

An identical analysis implies that $\sum_{e\in E_b} f_e \leq (\beta d)^{1-\alpha} \cdot \sum_{e\in E_b} f_e^{\alpha}$ where $E_b = \{(b, t) : f_{(b,t)} > \beta \cdot d\}$.

Recall that the total cost due to edges $E_a \cup E_b$ is $d \cdot \sum_{e\in E_a} f_e^{\alpha} + d \cdot \sum_{e\in E_b} f_e^{\alpha} \leq \rho m d^{\alpha}$. This implies:

$$\sum_{e\in E_a} f_e + \sum_{e\in E_b} f_e \leq \frac{\sum_{e\in E_a} f_e^{\alpha} + \sum_{e\in E_b} f_e^{\alpha}}{(\beta d)^{\alpha-1}} \leq \frac{\rho\, m\, d^{\alpha-1}}{4\rho\, d^{\alpha-1}} = \frac{m}{4}.$$

Let $V_a = \{a \in A : |N_a| > 4\rho\beta,\ f_{(s,a)} \leq \beta d\}$ and $V_b = \{b \in B : |N_b| > 4\rho\beta,\ f_{(b,t)} \leq \beta d\}$. The total cost of edges $\{(a, a(u)) : a \in A, u \in L\} \cup \{(b(u), b) : b \in B, u \in L\}$ is at least $\sum_{a\in A} |N_a| \cdot d^{\alpha+1} + \sum_{b\in B} |N_b| \cdot d^{\alpha+1} \geq (|V_a| + |V_b|) \cdot 4\rho\beta\, d^{\alpha+1}$. Since the total routing cost is at most $\rho m d^{\alpha}$, we have $|V_a| + |V_b| \leq \frac{\rho m d^{\alpha}}{4\rho\beta\, d^{\alpha+1}} = \frac{n}{4\beta}$. Thus $\sum_{w\in V_a} f_{(s,w)} + \sum_{w\in V_b} f_{(w,t)} \leq (|V_a| + |V_b|) \cdot \beta d \leq \frac{nd}{4} = \frac{m}{4}$.

Note that the flow lost in Step 1 above is at most $\sum_{e\in E_a} f_e + \sum_{w\in V_a} f_{(s,w)}$, and that in Step 2 is at most $\sum_{e\in E_b} f_e + + \sum_{w\in V_b} f_{(w,t)}$. So the total loss in flow is at most $\frac{m}{2}$. Finally consider Step 3. Let $E'' = \{e \in E' : f_e > \beta\}$. As in the calculation for E_a and E_b, using the fact that $\sum_{e\in E''} f_e^{\alpha} \leq \rho\, m$ (since cost of edge $e \in E'$ is $d^{\alpha} \cdot f_e^{\alpha}$), we have $\sum_{e\in E''} f_e \leq \frac{\rho\, m}{(\beta d)^{\alpha-1}} = \frac{m}{4}$. □

The flow after the above pruning has magnitude at least $\frac{m}{4}$ and edges in E' carry at most β flow each. If we choose one label for each vertex $c \in A \cup B$ randomly from N_c, the expected number of consistent edges in F is at least $\frac{m}{4\beta} \cdot \frac{1}{(4\beta\rho)^2} = \frac{m}{64\rho^2\beta^3} = \frac{|F|}{64\rho^{2+\frac{3}{\alpha-1}}}$.

Finally, the hardness of label cover implies that the s-t routing problem with any exponent $\alpha = 1 + \epsilon$ (for constant $\epsilon > 0$) is hard to approximate better than ratio $2^{\log^{1-\delta} n}$ for any $\delta > 0$. □

5.2 APX-Hardness of Undirected $s - t$ Routing with Heterogeneous Functions

We now consider the same case as the previous section, but in undirected graphs. Undirected graphs tend to be easier to route in than directed graphs, but we are still able to prove an inapproximability result.

Theorem 5. *The $s-t$ routing problem on undirected graphs with heterogeneous power functions and constant exponent $\alpha > 1$ is APX-Hard.*

Proof. The proof is a reduction from the problem of $\mathsf{3SAT}(2d)$, i.e., 3SAT where each variable appears in exactly $2d$ clauses (d as the positive form, and d as the negative form), for which the following hardness is known [14].

Theorem 6. *There exist constants d and ϵ for which it is NP-hard to distinguish between $\mathsf{3SAT}(2d)$ instances which are fully satisfiable and those which are at most $(1 - \epsilon)$ satisfiable.*

From $\mathsf{3SAT}(2d)$ to Independent Set. We first reduce the problem of $\mathsf{3SAT}(2d)$, to that of finding large independent sets in bounded degree graphs. Indeed, given an instance of $\mathsf{3SAT}(2d)$ with $m = 2dn/3$ clauses on n variables, we construct the following graph: For each clause, we create a triangle with nodes corresponding to the literals. These triangles are disjoint across clauses. To tie up the instance, we place edges between x and $\overline{x}$ for all the occurrences of literals x and $\overline{x}$. Notice that there are $N := 3m$ variables, and $M := 3m + d^2n = N(1 + d/2)$ edges in this graph. Furthermore, each node has degree $d + 2$ (two edges in the triangle, and d edges to the opposite literal).

We now relate independent sets on this graph with satisfying assignments in the $\mathsf{3SAT}(2d)$ instance. In the yes case, suppose there is a fully satisfying assignment for the SAT instance. Then, we can pick one satisfying literal from each clause and it forms an independent set (there are no triangle edges picked, and because the assignment is consistent, there are no literal edges as well). The size of this independent set is $N/3 = m$ nodes.

In the opposite direction, suppose we have an independent set of size $\frac{N}{3}(1 - \epsilon) = m(1 - \epsilon)$ nodes in the graph. Then, clearly, it has to pick at most one node from each of the clause triangles. Furthermore, these nodes must correspond to consistent literals (else we would include a literal edge). Therefore, the independent set naturally recovers an assignment which satisfies at least $m(1 - \epsilon)$ clauses. We can therefore conclude this step with the following theorem:

Theorem 7. *There are constants d and ϵ for which it is NP-hard to distinguish between $d+2$-regular graphs on N nodes which have an independent set of size at least $N/3$ and those where all independent sets are at most $N/3(1-\epsilon)$.*

From Independent Set to Power-Routing. We now reduce from independent set instances to the routing instances. Indeed, given a $d+2$-regular graph on N nodes $G=(V,E)$, we create the following routing instance $H=(W,F)$.

Routing Instance H. For each edge $e \in E$, there is an edge-vertex $w_e \in V'$, and for each node $v \in V$, we have a node-vertex $w_v \in V'$. We connect each edge-vertex to the corresponding node-vertices (according to G). That is, if $e=(u,v)$ in G, then we connect w_e with w_u and w_v. These edges have a buy cost of 1 and no load cost. These are called the *intermediate edges* in our graph H. Likewise, we connect each node-vertex w_v vertex to a sink t, with edges of buy cost $B := \frac{1}{2(d+1)}$. Finally, we connect each edge-vertex w_e to a source s with cost l^α, where l is the load. For the demand, we require s to route M units of flow to t. In the remainder of the proof, we will use C to denote $(2N/3)$.

Lemma 6. *If there is a vertex cover of size $C := (2N/3)$ in G, there is a routing solution in H of total cost at most $BC+2M$.*

This is easy to see, as the source can send a unit flow along to each edge-vertex, which will then send the flow to the node which covers it (in the vertex cover in G), and finally this flow gets routed to the sink. For the soundness direction, we show the following lemma in the full version of the paper.

Lemma 7. *For large enough constant α, if there is routing in H of cost at most $(1+\epsilon')(2M+BC)$, then we can recover an "almost vertex cover" of size $C(1+\epsilon')+8\epsilon' M(d+1)$ in G. Here, an "almost vertex cover" is a collection of nodes incident on at least a $(1-10\epsilon'(d+2)^2)$ fraction of the edges.*

We now complete our proof. In particular, we will be interested in the case when $C := 2N/3$ (recall that it was hard to distinguish between independent sets of size $N/3$ and those of size $(1-\epsilon)N/3$ in G). Indeed, suppose there is an independent set of size $N/3$. Then there is a vertex cover of size $2N/3$, and by Lemma 6, there is a routing in H of cost at most $BC+2M$.

Now, in the other direction, suppose there is a routing of cost at most $(1+\epsilon')(BC+2M)$. Then by the above Lemma 7, we can recover a set of $C(1+\epsilon')+8\epsilon' M(d+1)$ nodes which are incident on $(1-10\epsilon'(d+2)^2)$ fraction of the edges. But we can extend this to a complete vertex cover by adding at most $10\epsilon'(d+2)^2 M$ nodes by picking one node from each of the uncovered edges. This implies there is an independent set of size at least $N/3 - O(\epsilon' d^2)N = N/3(1-\epsilon)$ nodes, for small enough constant ϵ' (recall that d is a constant determined in Thm 7). But by Theorem 7, this is impossible if $P \neq NP$, thus proving Theorem 5. $\square$

6 Open Problems

Perhaps the most natural open question is whether there exists a poly-log competitive online algorithm for the case of multiple sources and multiple sinks.

Is the following simple algorithm good? Consider request (s_i, t_i). With probability $1/2^k$ "pretend" that the demand D is 2^k, and open all edges used in the cheapest way to route demand D from s_i to t_i assuming previous routes. Then route one unit of flow between s_i and t_i in the cheapest possible way along open edges.

References

1. Proceedings of the vision and roadmap workshop on routing telecom and data centers toward efficient energy use (October 2008)
2. Andrews, M., Antonakopoulos, S., Zhang, L.: Minimum-cost network design with (dis)economies of scale. In: FOCS, pp. 585–592 (2010)
3. Andrews, M., Fernández, A., Zhang, L., Zhao, W.: Routing for energy minimization in the speed scaling model. In: INFOCOM, pp. 2435–2443 (2010)
4. Aspnes, J., Azar, Y., Fiat, A., Plotkin, S., Waarts, O.: On-line routing of virtual circuits with applications to load balancing and machine scheduling. J. ACM 44(3), 486–504 (1997)
5. Benoit, A., Melhem, R., Renaud-Goud, P., Robert, Y.: Power-aware manhattan routing on chip multiprocessors. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS) (May 2012)
6. Chakrabarty, D., Chekuri, C., Khanna, S., Korula, N.: Approximability of Capacitated Network Design. In: Günlük, O., Woeginger, G.J. (eds.) IPCO 2011. LNCS, vol. 6655, pp. 78–91. Springer, Heidelberg (2011)
7. Gafni, E.M., Bertsekas, D.P.: Path assignment for virtual circuit routing. In: Proceedings of the Symposium on Communications Architectures & Protocols, SIGCOMM 1983, pp. 21–25. ACM, New York (1983)
8. Gupta, A., Krishnaswamy, R., Pruhs, K.: Online primal-dual for non-linear optimization with applications to speed scaling. CoRR, abs/1109.5931 (2011)
9. Imase, M., Waxman, B.M.: Dynamic Steiner tree problem. SIAM J. Discrete Math. 4(3), 369–384 (1991)
10. Johnson, W.B., Schechtman, G., Zinn, J.: Best constants in moment inequalities for linear combinations of independent and exchangeable random variables. Ann. Probab. (1), 234–253 (1985)
11. Kim, J., Horowitz, M.A.: Adaptive supply serial links with sub-1-v operation and per-pin clock recovery. IEEE Journal of Solid-State Circuits 37(11), 1403–1413 (2002)
12. Raz, R.: A parallel repetition theorem. SIAM J. Comput. 27(3), 763–803 (1998)
13. Rosenthal, H.P.: On the subspaces of L^p $(p > 2)$ spanned by sequences of independent random variables. Israel J. Math. 8, 273–303 (1970)
14. Trevisan, L.: Non-approximability results for optimization problems on bounded degree instances. In: ACM Symposium on Theory of Computing, pp. 453–461 (2001)

Reoptimization of the Minimum Total Flow-Time Scheduling Problem

Guy Baram and Tami Tamir

School of Computer Science, The Interdisciplinary Center, Herzliya, Israel
guy.baram@gmail.com, tami@idc.ac.il

Abstract. We consider *reoptimization* problems arising in production planning. Due to unexpected changes in the environment (out-of-order or new machines, modified jobs' processing requirements, etc.), the production schedule needs to be modified. That is, jobs might be migrated from their current machine to a different one. Migrations are associated with a cost – due to relocation overhead and machine set-up times. The goal is to find a good modified schedule, which is as close as possible to the initial one. We consider the objective of minimizing the total flow time, denoted in standard scheduling notation by $P||\sum C_j$.

We study two different problems: (i) achieving an optimal solution using the minimal possible transition cost, and (ii) achieving the best possible schedule using a given limited budget for the transition. We present optimal algorithms for the first problem and for several classes of instances for the second problem.

1 Introduction

This work studies a reoptimization variant of the classical scheduling problem of minimizing the total flow time (denoted in standard scheduling notation by $P||\sum C_j$ [12]). This problem can be solved efficiently by the simple greedy SPT rule [23,9] that assigns the jobs in nondecreasing order by their length. This algorithm, as many other algorithms for combinatorial optimization problems, solves the problem from scratch, for a single arbitrary instance without having any constraints or preferences regarding the required solution - as long as it achieves the optimal objective value. However, many of the real-life scenarios motivating these problems involve systems that change dynamically over time. Thus, throughout the continuous operation of such a system, it is required to compute solutions for new problem instances, derived from previous instances.

Moreover, since there is some cost associated with the transition from one solution to another, a natural goal is to have the solution for the new instance *close* to the original one (under certain distance measure). Thus, solving a *reoptimization* problem combines the challenge of computing an optimal (or close to the optimal) solution for the new instance, with the challenge of efficiently converting the initial solution to the new one. Each of these challenges, even when considered alone, gives rise to many theoretical and practical questions.

Obviously, combining the two challenges is an important goal, which naturally shows up in many applications.

Reoptimization variants of scheduling problems arise naturally in production planning – due to unexpected changes in the environment (out-of-order or new machines, modified jobs' processing requirements, etc.). Migrating tasks among the machines is costly, due to relocation overhead and machine set-up times. This work studies the problem of finding a good modified schedule, which is as close as possible to the initial one. To the best of our knowledge, no previous work combines these two objectives in a scheduling setting.

Applications: As mentioned above, the scenario we consider often arises in manufacturing systems. In fact, our work is relevant to any dynamic scheduling environment. We describe below two less intuitive applications in cloud computing and semiconductor wafers production line.

Consider an RPC (Remote Procedure Call) service. In this environment, a cloud of servers can provide service to a limited number of simultaneous users. If the number of requests is high, another virtual server could be temporarily rented, where the cost for using it is per user. The options are to put the RPC in a queue, thus causing latency in the service, or renting more virtual servers, enabling faster service and paying the additional servers' cost. In this application, the transition cost is not due to the migration itself, but due to the activation cost of the additional resources.

Some of our results will be extended to consider modifications that occur after the processing has begun, that is, at time $t > 0$. For this extension (see Section 2.1.1) we distinguish between environments in which the currently processed jobs can migrate and be restarted on a different machine, and applications in with restarts are not allowed, and a currently processed job must complete its partial processing. The following application describes a system in which restarts are not allowed: In a semiconductor wafers production line, some of the coating methods involve purely physical processes such as high temperature vacuum evaporation (physical vapor deposition - PVD). During the process, a vacuum is created to enable the coating. Once the elements are in a vacuum environment, the process can not be stopped as if the machine halts, it will be severely damaged [16]. Assume that at time $t > 0$ machines are added. Transferring jobs is costly - to capture the transition overhead and the changes required in programming the machines workplan. Also, the elements that are currently produced, that are already in vacuum state, must complete their production.

1.1 Problem Statement and Notation

An instance of our problem consists of a set J_0 of n_0 jobs and a set of m_0 identical machines. Denote by p_j the processing time of job j. A schedule S_0 of the initial instance is given. That is, for every job in J_0, it is specified on which machine it is assigned and on which time interval it is going to be processed. At any time, a machine can process at most one job and a job can be processed by at most one machine.

At time $t \geq 0$, a change in the system occurs. Possible changes include addition or removal of machines and/or jobs, as well as modification of processing time of jobs in J_0. Let J denote the modified set of jobs, and let $n = |J|$. Let M denote the modified set of machines, and let $m = |M|$. Our goal is to suggest a new schedule, S, for the modified instance, with good objective value and small transition cost form S_0. Assignment of a job to a different machine in S_0 and S is denoted *migration* and is associated with a cost. Formally, we are given a price list $\theta_{ii'j}$, such that it costs $\theta_{ii'j}$ to migrate job j from machine i to machine i'. We consider two problems:

1. Rescheduling to an optimal schedule using the minimal possible transition cost.
2. Given a budget B, find the best possible modified schedule that be be achieved without exceeding the budget B.

Some of our results assume identical transition costs, that is, for all j and $i \neq i'$, $\theta_{ii'j} = 1$.[1] For a given schedule, let C_j be the completion time of job j, that is, the time when the process of j completes.

Example: Assume that six jobs of lengths $1, \ldots, 6$ are scheduled on a single machine in an optimal SPT order. Assume that a second machine is added, and that all migrations have unit transition cost. Figure 1(a) presents an optimal modified schedule, for which the total flow-time is $\sum C_j = 34$. The budget required to reach this schedule (or any other schedule with $\sum C_j = 34$) is 3. For a given budget, $B = 2$, it is possible to move, for example, to the modified schedules given in Figures 1(b) and (c), having total flow-time 36 and 35, respectively. The schedule (c) is optimal for this budget. Note that the natural greedy approach of migrating the long jobs if the budget is low (as in schedule (b)) is sub-optimal. Two other natural approaches of prefix-SPT, or suffix-SPT (use the budget to maximize the prefix of the schedule or the suffix of the schedule that is identical to an SPT schedule) are also sub-optimal.[2]

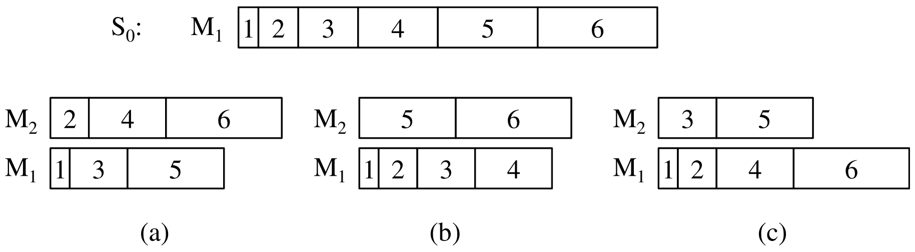


Fig. 1. An initial assignment (top), an optimal reassignment achieved with transition cost 3 (a), a possible (b) and an optimal (c) reassignments achieved with budget 2

[1] Note that the constant 1 can be replaced by any other constant.

[2] In the example, schedule (c) is suffix-SPT and optimal, however, suffix-SPT fail on other instances.

1.2 Related Work

The 'single-shot' minimum total flow-time, $P||\sum C_j$, can be solved in polynomial time by using the shortest processing time (SPT) rule [23,9]. The problem is solvable also on unrelated machines, $R||\sum C_j$, [7,14] by a reduction to a min-weight complete matching problem.

The work on reoptimization problems started with the analysis of *dynamic graph* problems (see e.g. [10,24]). Reoptimization algorithms were developed also for some classic problems on graphs, such as shortest-path [18,17] and minimum spanning tree [1]. A different line of research deals with the computation of a good solution for an NP-hard problem, given an optimal solution for a close instance. Among the problems studied in this setting are TSP, [4,6], Steiner Tree on weighted graphs [11] and Knapsack [2]. A survey of other research in this direction is given in [3]. In all of the above works, the goal is to compute an optimal (or approximate) solution for the modified instance. The resulting solution may be significantly different from the original one, since there is no cost associated with the transition among solutions.

The paper [21] suggests the framework we adopt for this work, in which the solution for the modified instance is evaluated also with respect to its difference from the initial solution. This framework is in use also in [20], to analyze algorithms for data placement in storage area network. Considering both the quality of the solution and the transition cost from an initial solution can also be seen as a special case of *multiobjective* optimization problems. In these problems, there are several weight functions associated with the input elements. The quality of a solution is measured with respect to a combination of these weights (see e.g., [19,13]).

1.3 Our Results

In Section 2 we explore the problem of moving to a modified optimal schedule using the minimal required budget. We present optimal algorithms that return both an optimal schedule and the minimum budget B required to reach an optimal schedule. We first describe an optimal algorithm for arbitrary migration costs and arbitrary changes in the instance. Its running time is dominated by the time required to find a minimum weight complete matching in a complete bipartite graph with $O(nm)$ vertices. We then present a more efficient algorithm for instances with uniform migration costs. The time complexity of this algorithm is varies between $O(n)$ (if the initial schedule is an SPT schedule) and $O(n \log n)$ (for arbitrary initial schedule). The first algorithm is described assuming the modification takes place at time $t = 0$. In Section 2.1.1 we describe how and under which conditions it can be extended to handle modifications at time $t > 0$. The second algorithm is valid for changes at any time $t \geq 0$.

In section 3 we consider the problem of rescheduling with a limited budget. The goal is to utilize the budget in the best possible way, that is, the modified schedule should have a low total flow-time - the minimal possible among all schedules that can be achieved using the given budget. Our results for this model assume unit migration costs, thus, the budget B gives the maximal number of allowed migrations. We present optimal algorithms for two cases: when the budget is a constant and when migrations are allowed only to new machines.

We conclude, in Section 4, with a discussion and some directions for future work. We note that our results can be applied also on a sequence of modifications. That is, the environment might change more than once, and the algorithms are performed after each modification. Due to space constraints, some of the proofs are omitted. All proofs are available in the full version [5].

2 Optimal Modified Schedule Using Minimum Budget

In this section we consider the problem of moving to a modified optimal schedule with respect to the minimal total flow objective using the minimal required budget.

2.1 Arbitrary Costs and Modifications

Let S_0 be a given initial schedule. We do not assume that S_0 is optimal nor that it has a specific structure or properties. Assume that at time $t = 0$, the environment is modified. Possible modifications include addition or removal of machines and/or jobs, and changes in jobs' processing times. The price list $\theta_{ii'j}$ specifies for every job j assigned to machine i, how much it costs to migrate j to machine i'. The goal is to find a new schedule, S, that is optimal with respect to the total flow-time, and has the minimal transition cost from S_0 among all optimal schedules.

We reduce the problem into a minimum weight complete matching problem in a bipartite graph. This approach was used by Horn [14], and Bruno, Coffman and Sethi [7] for solving the problem of minimum flow time on unrelated machines ($R||\sum C_j$). While the processing time of the jobs do not change due to migrations, it is possible to adopt this technique for our problem by setting the weights in the corresponding bipartite graph in a way that reflects the migration overhead.

Recall that n and m represent the number of jobs and machines in the modified instance. Let $G = (V, E)$, where $V = J \cup U$. The set J represents the set of n jobs (a single node per job). The set U consists of mn nodes, q_{ik}, for $i = 1, \ldots, m$ and $k = 1, \ldots, n$, where node q_{ik} represents the k^{th} from last position on machine i. The edge set E includes an edge (v_j, q_{ik}) for every node in J and every node in U (a complete bipartite graph). The following is an optimal algorithm for our problem. Note that edge weights (determined in Steps 1-2) consist of two components: first, a dominant component corresponding to the contribution of a job assigned in a specific position to the total flow-time, and second, a minor component corresponding to the associated transition cost. Both components are combined to form a single weight. Figure 2 illustrates the edges corresponding to a single job.

Algorithm 1. An optimal algorithms for rescheduling using minimum budget

```
1. Let $\theta_{ii'j}$ be a price list, i.e., it costs $\theta_{ii'j}$ to migrate job $j$ from machine $i$ to machine $i'$. In particular, for all $i, j$, $\theta_{iij} = 0$.
   Let $\Delta = \max_{j,i,i'} \theta_{ii'j}$ and let $Z$ be a constant lager than $n\Delta$.
2. Let $G$ be the complete bipartite graph corresponding to the problem. Set the edge weights as follows:
   – For every job that is assigned to $i$ , the weight of $(v_j, q_{ik})$ is $Zkp_j$.
   – For every $i' \neq i$, the weight of $(v_j, q_{i'k})$ is $Zkp_j + \theta_{ii'j}$.
3. Find a min-cost complete matching in $G$. Let $H$ denote the set of edges in this matching.
4. Return the schedule corresponding to $H$. That is, for every $(v_j, q_{i',k}) \in H$, assign $j$ in the $k^{th}$ from last position on machine $M_{i'}$. The minimum transition cost is $\sum_{(v_j,q_{i',k})\in H} \theta_{ii'j}$, where $i$ is the machine on which $j$ is assigned in $S_0$.
```

In the following claims we show that H induces an optimal schedule with the minimal possible transition cost from S_0. First, we show that H corresponds to a schedule with minimum total flow-time, then we show that among all schedules achieving minimum total flow-time, the schedule induced by H has minimum transition cost from S_0.

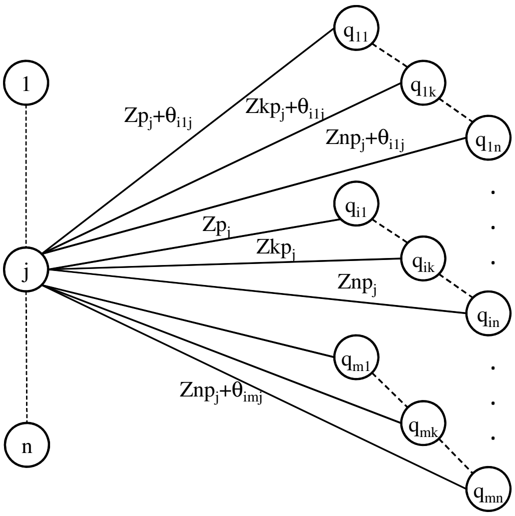


Fig. 2. The bipartite graph for Algorithm 1. The job j is assigned to machine i in S_0.

Claim. The set of edges H corresponds to a feasible schedule with minimum total flow-time.

Claim. Among all schedules achieving minimum total flow, the schedule induced by H has the minimal transition cost.

Proof. Let H^* be any perfect matching in G, corresponding to a schedule, $S(H^*)$, achieving minimum total flow-time. We show that the transition cost to $S(H^*)$ is not lower than the transition cost to $S(H)$. We know that H is a min-cost matching in G, therefore, $\sum_{e \in H} w(e) \leq \sum_{e \in H^*} w(e)$. Also, since both achieve minimum total flow-time and the weights w' reflect the total flow-time without the transition costs, $\sum_{e \in H} w'(e) = \sum_{e \in H^*} w'(e)$. The definition of w implies that for every matching H', it holds that $\sum_{e \in H'} w(e) = Z \sum_{e \in H'} w'(e) + \sum_{e=(v_j,q_{ik}) \in H'} \theta_{ii'j}$, where the second term is exactly the transition cost from the initial schedule to the schedule induced by H'. We conclude that the transition cost to $S(H^*)$ is not lower than the transition cost to $S(H)$.

2.1.1 Extension: When the Modification Occurs at Time $t > 0$

The change in the system might occur after the processing has begun, that is, at time $t > 0$. Let J_t be the set of jobs processed at time t. In some systems the processing of $j \in J_t$ must complete on its current machine. In others, j can be migrated to another machine. If reassigned, the corresponding transition cost is applied and the job must restart. We assume that preemptions are not allowed[3]. For every machine i, let γ_i denote the time required to complete the job from J_t processed at time t on machine i.

When restarts are not allowed, the only modification we consider is machines addition. Note that if machines can be removed, and restarts are not allowed then the problem is not well-defined for the jobs that are currently processed. The problem can be viewed as a scheduling problem in which machine i is available starting at time γ_i. Algorithm 1 can be generalized by setting the weights in the bipartite graph (described in Section 2.1) in the following way:

- For every job that is assigned to M_i , the weight of (v_j, q_{ik}) is $Z(kp_j + \gamma_i)$.
- For every $i' \neq i$, the weight of $(v_j, q_{i'k})$ is $Z(kp_j + \gamma_{i'}) + \theta_{ii'j}$.

When restarts are allowed, for every job $j \in J_t$ an additional possibility is to migrate j to a different machine and restart its processing. For this case our extension assumes that the initial schedule was optimal, that is, in SPT order. We set the weights in the bipartite graph as follows:

- For every job $j \in J_t$ that is currently processed on M_i the weight of (v_j, q_{ik}) is $Zk\gamma_i$.
- For every job $j \notin J_t$ that is assigned to M_i, the weight of (v_j, q_{ik}) is Zkp_j.
- For every $i' \neq i$, the weight of $(v_j, q_{i'k})$ is $Zkp_j + \theta_{ii'j}$.

Note that the above weights correspond to the contribution of jobs to the total flow-time, assuming the following property: if a currently processed job j remains on M_i then in the optimal modified assignment it is processed first on M_i. The proof of this property and the extensions' proofs are given in the full version [5].

[3] Enabling preemptions affects all the jobs of the instance, thus causing the problem to be intractable [22].

2.2 An Efficient Algorithm for Identical Migration Costs

In this section we consider systems with identical migration costs, that is, for all j, i, i', it holds that $\theta_{j,i,i'} = 1$. We present an efficient algorithm for finding an optimal modified schedule using the minimal possible budget. The algorithm can be applied for addition or removal of machines and/or jobs, as well as changes in jobs' processing times.

The algorithm is based on some properties of the SPT algorithm [23,9] for $P||\sum C_j$. For completeness, we describe a specific form of SPT algorithm: Given an instance of n jobs and m parallel machines, add dummy jobs of length 0 such that the total number of jobs is a multiple of m. Specifically, if n is not a multiple of m, then add to the instance $m - (n \bmod m)$ jobs of length 0. The dummy jobs can be scheduled on arbitrary machines and (when rescheduling) their migration cost is 0. Given that n is a multiple of m, the SPT algorithm can be described as follows: First, sort the jobs in non-decreasing order of processing time (break ties arbitrarily). Next, partition the jobs into n/m *rounds* of m jobs each. The k-th round consists of the jobs indexed $(k-1)m+1, \dots, km$ in the sorted list. Schedule on each machine one job from the first round, followed by one job from the second round, etc.

We use the following known property of SPT schedules: *the internal assignment of jobs from a particular round to the machines does not affect the total flow-time.* That is, any schedule in which the m jobs of round k are assigned on the k-th slots of the m machines is optimal.

Let L be the set of job lengths in the modified instance. The set L includes at most n distinct values. By the above property of SPT schedules, an optimal schedule can be characterized by the numbers $n_{\ell,k}$, for all $\ell \in L$ and $1 \le k \le \frac{n}{m}$, where $n_{\ell,k}$ is the number of jobs of length ℓ in round k, in any optimal schedule. Moreover, the problem of finding an optimal schedule using minimum transition cost reduces to the problem of finding a schedule obeying the optimal $n_{\ell,k}$ values with a minimal number of migrations from the initial schedule. The following is an overview of our optimal algorithm:

Algorithm 2. An efficient optimal algorithm for rescheduling with identical migration costs.

1. For every length $\ell \in L$ and round $1 \le k \le \frac{n}{m}$, calculate $n_{\ell,k}$, the number of jobs of length ℓ in round k, in any optimal modified schedule.
2. Partition L into two sets of job lengths: Let $L_1 \subseteq L$ be the set of lengths such that $\ell \in L_1$ if and only if $n_{\ell,k} > 0$ for a single round k. Let $L_2 = L \setminus L_1$ be the set of lengths such that $\ell \in L_2$ if and only if $n_{\ell,k} > 0$ for more than a single round.
3. For every round $1 \le k \le \frac{n}{m}$, schedule a maximal number of non-migrating jobs in round k. First, assign jobs having lengths in L_1, then in L_2. When assigning jobs from L_2, give higher priority to short jobs.
4. Schedule migrating jobs.

The idea is to assign first a maximal number of non-migrating jobs, and then assign the migrating jobs. When assigning the non-migrating jobs, we first assign the more restricted jobs – having lengths in L_1, and then the more flexible jobs whose lengths are in L_2.

Denote by S the schedule built by the algorithm. Steps (3-4) are implemented as follows: Denote by $S_{i,k}$ the slot in the k^{th} round on machine i. Initially, for all $1 \leq i \leq m, 1 \leq k \leq \frac{n}{m}$ it holds that $S_{i,k}$ is available (=EMPTY). During steps (3-4) some slots are assigned to non-migrating jobs. Whenever a job j of length ℓ is assigned to the k-th slot on machine i, the corresponding variable $S_{i,k}$ is set to j, and the corresponding counter of $n_{\ell,k}$ is reduced by one. Specifically, steps (3-4) are implemented as follows:

Step 3: Step 3 consists of $\frac{n}{m}$ iterations. In iteration k, the algorithm assigns non-migrating jobs into slots of round k. Consider a slot $S_{i,k}$. Let $ForFree(i,k)$ denote the set of jobs that can be assigned to $S_{i,k}$ with no migration. Formally, $j \in ForFree(i,k)$ if and only if (i) $n_{p_j,k} > 0$, (ii) j is assigned to M_i in S_0, and (iii) j was not assigned to M_i in earlier rounds.

In step 3, if possible, the algorithm assigns to $S_{i,k}$ a job from $ForFree(i,k)$ giving priority to lengths in L_1, and then to shorter lengths in L_2. Formally,

```
For $k = 1$ to $\frac{n}{m}$
  For $i = 1$ to $m$
    Calculate $ForFree(i,k)$.
    If $ForFree(i,k) \neq \emptyset$
      If there exists $j \in ForFree(i,k)$ such that $p_j \in L_1$. Set $S_{i,k} = j$ , $n_{p_j,k} --$.
      Else, let $j$ be the shortest job in $ForFree(i,k)$ such that $p_j \in L_2$.
        Set $S_{i,k} = j$ , $n_{p_j,k} --$.
```

Step 4: Step 4 consists of $\frac{n}{m}$ iterations. In iteration k, the algorithm assigns, with migrations, jobs to slots $S_{i,k}$ for which $ForFree(i,k) = \emptyset$. Formally,

```
While there exist $\ell, k$ such that $n_{\ell,k} > 0$,
  Assign any unassigned job $j$ of length $\ell$ to any machine $i$ s.t. $S_{i,k} = EMPTY$.
    Set $S_{i,k} = j$ , $n_{\ell,k} --$.
```

The number of migrations is the number of non-dummy jobs assigned in step 4. This number is the minimum budget required to reach an optimal schedule. We prove the optimality of the algorithm by combining two lemmas.

Lemma 1. *The algorithm produces an optimal schedule with respect to the total flow-time.*

Proof. The schedule S satisfies the $n_{\ell,k}$ values calculated by SPT algorithm, therefore it must be optimal. Since these values were calculated according to the amounts of jobs in the modified instance, all jobs are assigned, that is, in Step 4, while there exist ℓ, k such that $n_{\ell,k} > 0$, it is guaranteed that there is an available empty slot for a job of length ℓ in round k.

Lemma 2. *Every schedule minimizing the total flow-time requires at least the same number of migrations as the number of migrations applied by the algorithm.*

Proof. We prove the following greedy choice property: *for every round k there exists an optimal solution minimizing the total number of migrations, in which the non-migrating jobs assigned to round k are identical to those selected by the algorithm.* The following simple observation will be used to analyze the assignment of jobs having lengths in L_2.

Observation 1. *For every round k, there are at most two lengths $\ell_1, \ell_2 \in L_2$ such that $n_{\ell_1,k} > 0$ and $n_{\ell_2,k} > 0$.*

Proof. By definition, jobs of lengths in L_2 span across more than one round in any optimal schedule. Another known property of SPT schedules is that all job lengths in round k are not shorter than job lengths in round $k-1$ and not longer than job lengths in round $k+1$. Therefore, it is not possible to have three different lengths, all spanning over round k and an additional round. In order to preserve the above SPT property, jobs of the middle length, must all be assigned to round k.

We prove the greedy choice property for round k: Assume that an optimal schedule agrees with the algorithm in rounds earlier than k, and consider the assignment to round k. For every machine i, if $ForFree(i,k) = \emptyset$ then this is valid also for the optimal assignment, and a migration from another machine to $S_{i,k}$ is inevitable. If $ForFree(i,k)$ includes at least one job then we use exchange argument to show that any selection of job to $S_{i,k}$ that is different from the algorithm's choice can be changed to the algorithm's choice without hurting the total number of non-migrating jobs. Let $j \in ForFree(i,k)$ be the job assigned by the algorithm to $S_{i,k}$. Let $j' \neq j$ be the job assigned in the optimal schedule to $S_{i,k}$. If $j' \notin ForFree(i,k)$, then by switching j and j', we can only reduce the number of non-migrating jobs. If $j' \in ForFree(i,k)$, we distinguish between two cases:

1. $p_j \in L_1$. In this case, j must be assigned to round k, and assigning it to $S_{i,k}$ is the only way to assign it for free. By switching the assignment of j' and j in the optimal assignment, we avoid the migration of j, and cause a migration to j', thus, the total number of migrations does not increase.
2. $p_j \in L_2$. Since the algorithm gives priority to jobs whose lengths are in L_1, it must be that all job lengths in $ForFree(i,k)$ are in L_2 and in particular, $p_{j'} \in L_2$. By Observation 1, p_j, p'_j are the only lengths of jobs in $ForFree(i,k)$. Among lengths in L_2, the algorithm gives priority to shorter jobs, therefore, $p_j < p_{j'}$. Moreover, k is the last round in which jobs of length p_j will be assigned, as otherwise, the SPT order is not preserved (given that jobs of length $p_{j'}$ are assigned on both k and $k+1$). Therefore, assigning j to $S_{i,k}$ is the only way to assign it for free. By switching the assignment of j' and j in the optimal assignment, we avoid the migration of j, and cause a migration to j', thus, the total number of migrations does not increase.

We conclude that any optimal assignment can be modified such that it agrees with the algorithm's choice, without hurting the number of migrations. Thus, the algorithm produces an optimal assignment.

Thus, our algorithm produces an optimal schedule using the minimal number of migrations.

Time Complexity Analysis: Algorithm 2 consists of 4 steps. In order to calculate the $n_{\ell,k}$ values in step 1 the jobs should be sorted by processing times. If the initial schedule S_0 is arbitrary, or if the modification includes jobs addition or jobs' length modification, then the sorting takes in $O(n \log n)$ time. If the initial schedule is optimal, that is, in SPT order, and the modification does not include jobs' length modification, then the algorithm only needs to sort the jobs of each round in S_0 separately, and concatenate the resulting lists. As there are m_0 jobs in each round we get an $O(n \log m_0)$ time algorithm. If in the initial SPT schedule the jobs are assigned sequentially on the machines, or if m_0 is a constant, then Step 1 takes $O(n)$ time.

The partition of job lengths into L_1, L_2 in Step 2 is clearly linear. Step 3 iterates on the rounds and in each round assigns jobs using the already sorted list. The $ForFree$ structure can be implemented using a list of pointers. Since $ForFree$ jobs are assigned in a non-decreasing order and by observation 1, we conclude that this step takes $O(m\frac{n}{m}) = O(n)$. In step 4, the algorithm assigns the remaining jobs in time $O(n)$.

We conclude that the time complexity of the algorithm varies between $O(n)$ and $O(n \log n)$, depending on the initial schedule and the allowed modification in the instance.

3 Rescheduling with a Limited Budget - Unit Migration Costs

In this section we consider the rescheduling problem assuming a limited budget. Naturally, the goal is to utilize the budget in the best possible way, that is, the modified schedule should have a low total flow-time – the minimal possible among all schedules that can be achieved using the given budget. We assume unit migration costs, that is, $\theta_{ii'j} = 1$, independent of the job j and the involved machines. Thus, the budget B gives the maximal number of allowed migrations. We also assume that $n > B$, as otherwise an optimal schedule can be found by ignoring the migration costs.

The problem can be described as the following weighted matching problem: Similar to the technique used in Section 2.1, let $G = (V, E)$, be a complete bipartite graph with n nodes on one side and mn nodes in the other side. The node q_{ik}, for $i = 1, \ldots, m$ and $k = 1, \ldots, n$, corresponds to the k^{th} from last position on machine i. The edge (j, q_{ik}) has weight kp_j, reflecting the contribution of j to the total flow-time if it is assigned on the k^{th} from last position on machine i. We color the edges of G as follows: If an edge (j, q_{ik}) corresponds to a migration of j, that is, i is not the machine j is assigned to in S_0, then the edge is red, otherwise the edge is blue.

It is easy to verify that a min-weight perfect matching with at most B red edges corresponds to an optimal reschedule. For an arbitrary bipartite graph with arbitrary weights, the complexity of the above restricted matching problem is unknown. Some special cases for which efficient algorithms exist include bipartite graphs with unit-weights [15], or with equal sizes ($K_{n,n}$) [25]. The more general problem of determining whether a complete weighted bipartite graph has a complete matching with a specific weight w in known to be NP-hard [8]. We present optimal polynomial time algorithms for several classes of instances of our problem.

3.1 The Budget B Is a Constant

Assume that the modification occurs at time $t = 0$, and the budget B is a constant. Clearly, every job j may either migrate or not, and as the budget is a constant, there are at most n^B possible ways to select the subset of jobs that are allowed to migrate. The following algorithm considers each selection separately.

Algorithm 3. An optimal algorithm for rescheduling when the budget B is a constant

```
For every possible selection of $B$ jobs $J' \subset J$:
    1. Let $G = (V, E)$, be a bipartite graph with $n$ nodes on one side and $mn$ nodes in
       the other side. The node $q_{ik}$, for $i = 1, \ldots, m$ and $k = 1, \ldots, n$, corresponds to the
       $k^{th}$ from last position on machine $i$. For every job $j \in J'$, there is an edge $(j, q_{ik})$
       for every $i = 1, \ldots, m$ and $k = 1, \ldots, n$. For every job $j \notin J'$, there is an edge
       $(j, q_{ik})$ for every $k = 1, \ldots, n$, but only for the machine $i$ on which $j$ is assigned to
       in $S_0$. The weight of $(j, q_{ik})$ is $kp_j$.
    2. Find a min-cost complete matching in $G$.
Return the schedule induced by the minimal min-cost matching.
```

Theorem 2. *Algorithm 3 returns a modified schedule whose total flow-time is minimal among all schedules achieved with budget at most B.*

3.2 Migrations Are Allowed Only to New Machines

Another case for which it is possible to solve the problem optimally is when the system's modification consists of machines addition and the only allowed migrations are to the new machines. This scenario arises in practice when the system is upgraded with new machines that are ready to receive tasks, while the old machines are not capable to accept new tasks. We present an optimal algorithm for this problem based on a reduction to a min-cost max-flow problem. An illustration of the flow network is given in Figure 3.

An Overview of the Flow Network: The set of nodes r_{ik} for $1 \leq i \leq m_0,\ 1 \leq k \leq n$ correspond to positions on the initial machines. The set of nodes $q_{i'k}$ for $1 \leq i' \leq m',\ 1 \leq k \leq B$ correspond to positions on the added machines. All the q-nodes are connected to node d. The capacity of the edge (d, t) is the budget B. This limited capacity guarantees that the total number of slots occupied on the new machines will not exceed B. The set of nodes $1 \leq j \leq n$ correspond to the jobs. Every job j that is assigned to machine i in S_0 is connected to the nodes corresponding to positions on machine i and to all the q-nodes. The capacities of all edges except for (d, t) are 1. The cost of an edge connecting job j to a node corresponding to a k^{th} from last position (on any machine) is kp_j. All other edges have cost 0.

Theorem 3. *A minimum-cost maximum-flow (of value n) in G corresponds to an optimal schedule without exceeding the budget B.*

Proof. (Sketch) First, note that every valid schedule corresponds to a maximum-flow in G. On the other hand, not every maximum-flow in G corresponds to a schedule, since a job might be assigned to the k^{th} from last position in some machine, while less than k jobs are assigned to that machine. However, such a maximum-flow is clearly not of minimal cost - a better matching can be obtained by shifting the $k' < k$ jobs assigned to that machine into the k' last slots. Therefore, a schedule of minimum total flow-time corresponds to a minimum-cost maximum-flow in G.

As the capacity of (d, t) is B, while all other edges' capacity is 1, at most B q-nodes have incoming flow. These nodes correspond to migrating jobs. Thus, a minimum-cost maximum-flow in G corresponds to an optimal schedule without exceeding the budget B.

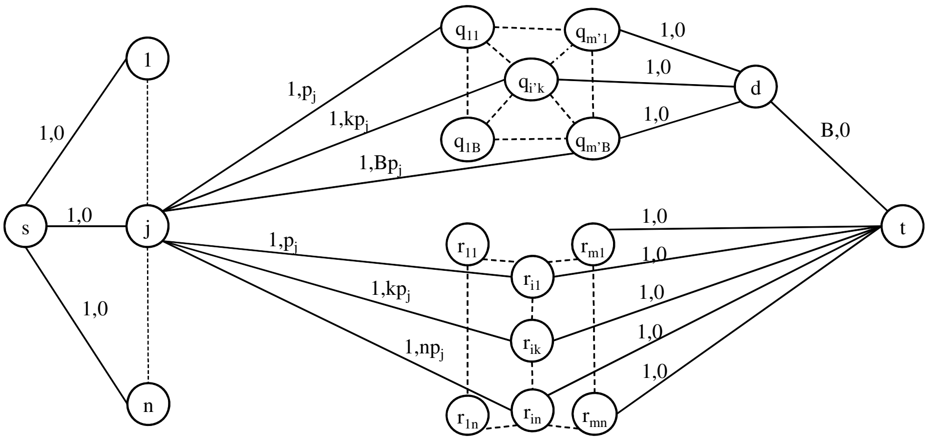


Fig. 3. The flow network built for the rescheduling with limited budget problem. Each edge is labeled by its capacity and the cost of one flow unit.

This algorithm can be extended for the case in which the systems' modification occurs at time $t > 0$ - similar to the extensions described in Section 2.1.1. If restarts are allowed, then our extension assumes that every currently processed job is the shortest job on its machine (which is true if the initial schedule is optimal, or if the schedule is a result of our algorithm - even on a sequence of modifications). If restarts are not allowed then our extension is valid for any initial schedule.

4 Conclusions and Future Work

We studied reoptimization problems arising in production planning, in which the goal is to combine the objective of finding a schedule with low total flow-time, with the goal of efficiently converting a given initial schedule to the modified one. We presented the first positive results in this framework. We presented algorithms for finding an optimal schedule achieved using the minimal possible transition cost, and algorithms for optimal utilization of a limited number of migrations.

Several interesting important problems remain open:

1. Identify the complexity status of the second problem for arbitrary transition costs and arbitrary modifications. As explained in Section 3, even with unit transition costs this is a special case of a more general open problem (min-weight matching with limited number of red edges).
2. Identify the range of budget B for which it is guaranteed that an optimal reschedule can be achieved using no internal migrations. It is easy to see that this range is included in $m' < B \leq m' \frac{n}{m_0+m'}$.
3. Another open research direction is to consider different objective functions. In particular, minimizing the makespan of the schedule, given by the last completion time of some job. Since the problem is NP-hard, the reoptimization problem is clearly also NP-hard. The goal is to develop an algorithm for the reoptimization problem whose approximation-ratio is similar to the best approximation-ratio known for the original problem.

References

1. Amato, G., Cattaneo, G., Italiano, G.F.: Experimental analysis of dynamic minimum spanning tree algorithms. In: Proc. of 8th SODA (1997)
2. Archetti, C., Bertazzi, L., Speranza, M.G.: Reoptimizing the 0-1 knapsack problem. Discrete Applied Mathematics 158(17) (2010)
3. Ausiello, G., Bonifaci, V., Escoffier, B.: Complexity and approximation in reoptimization. In: Cooper, B., Sorbi, A. (eds.) Computability in Context: Computation and Logic in the Real World. Imperial College Press/World Scientific (2011)
4. Ausiello, G., Escoffier, B., Monnot, J., Paschos, V.T.: Reoptimization of minimum and maximum traveling salesmans tours. J. of Discrete Algorithms 7(4), 453–463 (2009)

5. Baram, G., Tamir, T.: Reoptimization of the minimum total flow-time scheduling problem (full version), http://www.faculty.idc.ac.il/tami/Papers/BTfull.pdf
6. Böckenhauer, H.J., Forlizzi, L., Hromkovič, J., Kneis, J., Kupke, J., Proietti, G., Widmayer, P.: On the approximability of TSP on local modifications of optimally solved instances. Algorithmic Operations Research 2(2) (2007)
7. Bruno, J.L., Coffman, E.G., Sethi, R.: Scheduling independent tasks to reduce mean finishing time. Communications of the ACM 17, 382–387 (1974)
8. Chandrasekaran, R., Kaboadi, S.N., Murty, K.G.: Some NP-complete problems in linear programming. Operations Research Letters 1, 101–104 (1982)
9. Conway, R.W., Maxwell, W.L., Miller, L.W.: Theory of Scheduling. AddisonWesley (1967)
10. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. In: Atallah, M.J. (ed.) CRC Handbook of Algorithms and Theory of Computation, ch. 8 (1999)
11. Escoffier, B., Milanič, M., Paschos, V.T: Simple and fast reoptimizations for the Steiner tree problem. DIMACS Technical Report 2007-01
12. Graham, R.L., Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G.: Optimization and approximation in deterministic sequencing and scheduling: A survey. Annals of Discrete Math. 5, 287–326 (1979)
13. Grandoni, F., Zenklusen, R.: Optimization with more than one budget. In: Proc. of ESA (2010)
14. Horn, W.: Minimizing average flow-time with parallel machines. Operations Research 21, 846–847 (1973)
15. Karzanov, A.V.: Maximum matching of given weight in complete and complete bipartite graphs. Kibernetika 1, 7–11 (1987); English translation in CYBNAW 23, 8–13
16. Mattox, D.: Handbook of Physical Vapor Deposition (PVD) Processing, 2nd edn. Elsevier (2010)
17. Nardelli, E., Proietti, G., Widmayer, P.: Swapping a failing edge of a single source shortest paths tree is good and fast. Algorithmica 35 (2003)
18. Pallottino, S., Scutella, M.G.: A new algorithm for reoptimizing shortest paths when the arc costs change. Operations Research Letters 31 (2003)
19. Ravi, R., Goemans, M.X.: The Constrained Minimum Spanning Tree Problem. In: Karlsson, R., Lingas, A. (eds.) SWAT 1996. LNCS, vol. 1097, pp. 66–75. Springer, Heidelberg (1996)
20. Shachnai, H., Tamir, G., Tamir, T.: Minimal Cost Reconfiguration of Data Placement in Storage Area Network. In: Bampis, E., Jansen, K. (eds.) WAOA 2009. LNCS, vol. 5893, pp. 229–241. Springer, Heidelberg (2010)
21. Shachnai, H., Tamir, G., Tamir, T.: A Theory and Algorithms for Combinatorial Reoptimization. In: Fernández-Baca, D. (ed.) LATIN 2012. LNCS, vol. 7256, pp. 618–630. Springer, Heidelberg (2012)
22. Sitters, R.A.: Two NP-Hardness Results for Preemptive Minsum Scheduling of Unrelated Parallel Machines. In: Aardal, K., Gerards, B. (eds.) IPCO 2001. LNCS, vol. 2081, pp. 396–405. Springer, Heidelberg (2001)
23. Smith, W.E.: Various optimizers for single-stage production. Naval Research Logistics Quarterly 3, 59–66 (1956)
24. Thorup, M., Karger, D.R.: Dynamic Graph Algorithms with Applications. In: Halldórsson, M.M. (ed.) SWAT 2000. LNCS, vol. 1851, pp. 1–9. Springer, Heidelberg (2000)
25. Yi, T., Murty, K.G., Spera, C.: Matchings in colored bipartite networks. Discrete Applied Mathematics 121, 261–277 (2002)

Energy Efficient Caching for Phase-Change Memory

Neal Barcelo, Miao Zhou, Daniel Cole, Michael Nugent, and Kirk Pruhs

Department of Computer Science,
University of Pittsburgh, Pittsburgh, Pennsylvania 15260,
{ncb30,miaozhou,dcc20,mnugent,kirk}@cs.pitt.edu

Abstract. Phase-Change Memory (PCM) has the potential to replace DRAM as the primary memory technology due to its non-volatility, scalability, and high energy efficiency. However, the adoption of PCM will require technological solutions to surmount some deficiencies of PCM, such as writes requiring significantly more energy and time than reads. One way to limit the number of writes is by adopting a last-level cache replacement policy that is aware of the asymmetric nature of PCM read/write costs. We first develop a cache replacement algorithm, Asymmetric Landlord (AL), and show that it is theoretically optimal in that it gives the best possible guarantee on relative error. We also propose an algorithm Variable Aging (VA), which is a variation of AL. We have carried out a simulation analysis comparing the algorithms LRU, N-Chance, AL, and VA. For benchmarks that are a mixture of reads and writes, VA is comparable or better than N-Chance, even for the best choice of N, and uses at least 11% less energy than LRU. For read dominated benchmarks, we find that AL and VA are comparable to LRU, while N-Chance (using the N that was best for benchmarks that were a mixture of reads and writes) uses at least 20% more energy.

Keywords: cache replacement, algorithms, phase change memory, energy, power.

1 Introduction

Dynamic Random Access Memory (DRAM) has been the memory of choice for most computer systems for decades, but it now faces two critical problems. First, DRAM is projected to encounter severe scalability problems in coming years because DRAM relies on charge placement and control logic, which are inherently unscalable [1]. Second, increasingly large DRAM components have become a major consumer of energy in computer systems. Currently main memory consumes up to 40% of the energy in computer systems, comparable to, or slightly higher than, the energy consumption of processors [2].

In reaction to these problems, *Phase-Change Memory* (PCM) has been proposed as a replacement technology for DRAM (or maybe an add-on technology to DRAM). Desirable properties of PCM include: non-volatility, good scalability, and high energy efficiency (due to its low read power and very low idle power). Undesirable properties of PCM include: non-durability to writes (PCM memory can fail after 10^7 or 10^8 writes, which can happen in days or even hours without wear-leveling techniques [3–6]), PCM is slower than DRAM, and with current technology, PCM writes have up to 20 times higher latency, and 10 times higher energy consumption than PCM reads [1,7].

To mitigate these shortcomings, researchers have proposed to organize PCM main memory with a small cache [1,8,9]. The cache could be an on-chip cache or an off-chip DRAM cache. Logically, it is the *last-level cache* (LLC) of the PCM main memory. The LLC improves performance by caching highly accessed data, and further, extends PCM lifetime and reduces energy consumption by filtering a large portion of destructive and energy costly PCM writes.

The LLC replacement policy will be crucial for improving the energy efficiency of PCM as a main memory technology. An ideal replacement strategy must take into account the read/write asymmetry of PCM: PCM reads are relatively fast and energy efficient, while PCM writes are very slow and energy hungry. PCM bits are stored as the physical state of chalcogenide material, and writes are costly because changing this state is a process of heating followed by controlled cooling. However, since PCM can be written to one bit at a time (unlike DRAM), how much time and energy is needed to write a page is directly related to how many bits of that page have changed. So intuitively a good replacement strategy should be more hesitant (than a replacement strategy for DRAM) to evict dirty cache pages since this involves writing to PCM, and just how hesitant may depend on the number of bits that have changed in that page. The most common DRAM replacement strategy, Least Recently Used (LRU), does not differentiate between evicting clean and dirty pages.

To the best of of our knowledge, there is only one paper in the literature specifically addressing the issue of developing a good cache replacement strategy for PCM. Ferreira et al. [6] proposed a policy N-Chance, where N is a tunable integer parameter, that is a variation of LRU. N-Chance evicts the least recently accessed clean page from cache, unless all of the N least recently accessed pages are dirty, in which case it evicts the least recently accessed page. Note that when $N = 1$, N-Chance is exactly LRU, so there is always a choice of the parameter N where N-Chance is at least as good as LRU. In a simulation study, [6] showed that N-Chance can be significantly better than LRU on common benchmarks, given an appropriate choice for the parameter N. However, [6] also showed that the best choice of the parameter N can vary significantly based on the application, and on the ratio of the write to read cost in PCM.

The goal of the research reported here is to theoretically and experimentally investigate other replacement strategies that account for the read/write cost asymmetry of PCM, and in particular, to develop algorithms that do not require an application specific tunable parameter and that take into account how dirty a page is. We primarily focus on the energy costs of accessing PCM. We adopt the most natural simple model, which assumes that a PCM read costs 1 unit of energy, and a PCM write costs some constant $c > 1$ units of energy.

Most naturally c might be the cost to change half of the bits in the page, which would be approximately the number of bits that would have to be written if there was no correlation between a page's original value and the page's new value. Actually our theoretical results generalize to a setting where the write cost is concave in the number of times that page has been written to while being in cache. Discussion of why this might be a reasonable assumption for common applications can be found in [10]. For example, if a random fixed percentage of the bits are changed with each write then the cost to write back to PCM is concave in the number of writes.

One of our research goals was to design provably scalable algorithms. A scalable algorithm guarantees a bounded error relative to the minimum possible achievable cost with slightly less resources, which in our case means a slightly smaller cache. A scalable algorithm guarantees that for every access pattern, if the PCM energy costs were high for the scalable algorithm, then the energy costs would be high for every algorithm equipped with a slightly smaller cache. Thus intuitively scalability means that, for every access pattern, either the cache size was at a phase change point for the access pattern where the costs change dramatically for that cache size, or the high energy costs were unavoidable for that access pattern (and thus not the algorithm's fault).

It is not too difficult to see that even for very simple memory access patterns, N-Chance can have very high PCM energy costs, even if low energy costs are achievable with a smaller cache. Intuitively if N is too large, N-Chance can have an unnecessarily high cost for cycles of read requests to clean pages, because N-Chance can indefinitely freeze up to N dirty pages in cache, which effectively reduces the cache size available for the clean pages. And if N is too small, N-Chance can have unnecessarily high cost for a sequence that alternates between reads to one working set of pages, and writes to another working set of pages, because N-Chance won't keep enough of the dirty pages in cache.

In section 3 we develop and analyze a scalable cache replacement algorithm Asymmetric Landlord (AL) that guarantees a bounded error. We show that for all access patterns, the PCM energy cost for AL is at most $\frac{1}{\epsilon}$ times the minimum possible cost for that access pattern for a cache of size $(1-\epsilon)$ times the current cache. Roughly speaking, AL associates a Time-To-Live (TTL) value with each page in cache, and always evicts the least recently used page among the pages with the lowest TTL value (and reduces the TTL of the other pages by the TTL of the evicted page). When clean pages are brought into cache, they are initially given a TTL of 1. When dirty pages are brought into cache they are given a TTL of $c_1 + 1$, where c_1 is the average energy cost to write back a page that has been written to once. When write costs are small, AL is intuitively similar to LRU (but it is not exactly LRU). Further, AL does not have any application specific tunable parameters.

In an effort to more intuitively describe AL (there are a few more cases than described above), we discovered another intuitive algorithm, that we call Variable Aging (VA). VA is another generalization of LRU. In LRU, each page in cache can be viewed as having an age equal to the number of accesses since it was last accessed (so pages are reborn when they are accessed), where upon eviction, LRU evicts the oldest page in cache. In VA, the rate at which pages age is inversely proportional to the energy cost that the page will incur when evicted. So clean pages age at a rate of 1, and dirty pages age at a rate of $1/c$, where c is the average cost of writing a page to memory. Again VA always evicts the oldest page. So both AL and VA have ages, but in AL the age is a time until death, and in VA the age is a time since rebirth. It is not difficult to see that there are access patterns where VA can have high energy costs, even if low energy costs were achievable with a small cache.

LRU, N-Chance, AL, and VA, all assume that pages must be brought into cache to be accessed. In principle, one might also read-through or write-through cache by accessing the page directly from PCM, avoiding having to move the page into cache. This would be the right choice for example if the cache was all dirty, and there was a single read to a page that was not in cache; a read-through would cost a PCM read, while an eviction would cost a PCM read and a PCM write. In section 3 we develop a replacement algorithm that incorporates read-throughs and write-throughs (and also the possibility of variable sized pages). We show that for all access patterns, the PCM energy cost for this algorithm is at most $\frac{2}{\epsilon}$ times the minimum possible cost (again assuming the possibilities of read-throughs and write-throughs) for that access pattern for a cache of size $(1-\epsilon)$ times the current cache.

We have carried out a simulation analysis comparing the algorithms that do not use read-through and write-through, namely LRU, N-Chance, AL, and VA. In section 4, we give the experimental set-up. We evaluate the energy efficiency and performance of the proposed policies with SPEC CPU 2006 benchmarks using a trace-driven cycle-accurate simulator. In section 5, we report on our experimental results. For benchmarks that are a mixture of reads and writes, VA is comparable or better than N-Chance, even for the best choice of N, and uses at least 11% less energy than LRU. For read dominated benchmarks, we find that AL and VA are comparable to LRU, while N-Chance (using the N that was best for benchmarks that were a mixture of reads and writes) uses at least 20% more energy.

2 Related Work

The design and analysis of AL is heavily indebted to the design and analysis of the Landlord algorithm for browser caching [11]. In browser caching, there is a cost and a size associated with each page. There are two main differences between browser caching, and caching for PCM. The first is that the costs in browser caching can take on any value, not just 1 or c. This would seem to make browser caching harder than PCM caching. The second difference is that the page cost in PCM is time dependent, as it can rise from 1 to c when a previously clean page is written to. This would seem to make PCM caching harder than browser caching. And in fact, while the problem of optimal browser caching with equal sized files can be computed by a min-cost flow computation [12], we do not know how to compute the optimal PCM caching cost with a polynomial time algorithm. The main conceptual difficulty in adapting the Landlord results in [11] to our purposes was that in PCM caching a cost can be incurred without a change in the contents of cache (when a page is dirtied).

For online randomized caching with uniform costs and page sizes, [13] presented the $\log 2k$-competitive marking algorithm and [14] gave a $\log k$-competitive algorithm, which is the best possible competitive ratio. [15] gave a $O(\log^2 k)$-competitive algorithm when pages have arbitrary sizes and weights, and recently [16] used similar ideas to develop a $O(\log k)$-competitive algorithm for this problem.

One might also wish to limit the number of writes to reduce wear in a memory technology where durability is an issue, such as flash or PCM. [17] develops an algorithm similar to N-Chance for flash memory, with part of the motivation being the asymmetric cost of reads and writes in flash memory.

3 Theoretical Development of the Asymmetric Landlord Algorithm

3.1 Problem Model

Consider the following model for the problem of *asymmetric weighted page caching* to minimize total energy consumption. There is a fully associative cache of size k, and we are given a sequence of requests $I = r_1, r_2, \dots r_m$ where $r_i = (f_i, s)$ denotes a request for page f_i with $s \in \{r, w\}$ denoting a read or write. There is a read cost, normalized to 1, representing the cost in terms of energy of reading a page from slow memory (PCM). There is a concave function $\mathcal{C}$ mapping the number of writes to a page to the cost in terms of energy of writing it to slow memory. An algorithm must satisfy each request by bringing the requested page into cache and when necessary evicting other pages from cache to maintain that the number of pages in cache is no more than k. All requests are received on-line, meaning an algorithm must make a decision for request r_i without knowledge of future requests r_j, where $j > i$. We assume that there is no cost for reading or writing to a page in cache, however, any page that is written to s times while in cache must pay the write cost $\mathcal{C}(s)$ when evicted. For convenience, we define the incremental cost of the i^{th} write to be $c_i = \mathcal{C}(i) - \mathcal{C}(i-1)$. We say a page has "dirtiness i" if it has been written to i times without being evicted, and a page is "dirty" if has dirtiness is at least 1. For an on-line algorithm A, and a request sequence I, let $\mathcal{A}_k(I)$ denote the cost of algorithm A with cache size k, where the cost is the total energy consumed by A in satisfying all requests in I. We say that an algorithm A is (k/h)-cache c-competitive if $\max_I \frac{\mathcal{A}_k(I)}{\mathcal{O}_h(I)} \leq c$, where $\mathcal{O}_h(I)$ is the optimal cost using a cache of size $h \leq k$.

The algorithm Asymmetric Landlord (AL) was informally described in the introduction; a formal description may be found in the adjacent figure. The one situation not discussed in the informal description was how to handle the case that a page in cache with dirtiness i is written to. In this case AL increments the age of this page by c_{i+1}. In Theorem 1 we show that Asymmetric Landlord is (k/h)-cache $(k/(k-h+1))$-competitive. Even for symmetric read and write costs, this is the best achievable ratio [18]. AL and the proof of Theorem 1 are derived from the Landlord algorithm and corresponding analysis given in [11].

Algorithm 1. Asymmetric Landlord

```
1: When there is a request $(g, s)$
2: **if** $g$ is not in the cache **then**
3:     $\Delta = \min_{f \in \text{cache}} \text{TTL}[f]$
4:     For each page $f$ in cache, decrease $\text{TTL}[f]$ by $\Delta$
5:     Evict the oldest page $f$ such that $\text{TTL}[f] = 0$
6:     Bring $g$ into the cache
7:     **if** $s = r$ **then** Set $\text{TTL}[g] = 1$
8:     **else** Set $\text{TTL}[g] = c_1 + 1$
9: **else Case 1:** $g$ has dirtiness $i$ and $s = w$ **then** Set $\text{TTL}[g] = \max(\text{TTL}[g] + c_{i+1}, c_1 + 1)$
10:     **Case 2:** $g$ is clean and $s = r$ **then** Set $\text{TTL}[g] = 1$
11:     **Case 3:** $g$ is dirty and $s = r$ **then** Do nothing
```

Theorem 1. *Asymmetric Landlord (AL) is (k/h)-cache $\frac{k}{k-h+1}$-competitive for asymmetric weighted page caching.*

Proof. We use a potential function argument very similar to the one found in [11]. That is we find a function Φ that maps the state of AL and the state of the optimal solution to a nonnegative integer, where Φ is zero if the cache is empty in both states. Further we need that for every access:

$$(k-h+1) \cdot \Delta\mathcal{A}_k + \Delta\Phi \leq k \cdot \Delta\mathcal{O}_h \tag{1}$$

where Δ denotes the change due to that request. The claim then follows by summing up these inequalities over all requests, and noting that the various terms telescope.

To define the potential function Φ that we use, let AL be the set of pages in the cache of Asymmetric Landlord, OPT be the set of pages in the Optimal's cache, and $\text{OPT}_l \subseteq \text{OPT}$ be the set of pages in Optimal's cache that have dirtiness l or higher. By convention, we assume that for any page f not in the cache $\text{TTL}[f] = 0$. Define the potential function

$$\Phi = (h-1) \cdot \sum_{f \in \text{AL}} \text{TTL}[f] + k\Big(\sum_{f \in \text{OPT}} \max\Big(1 + \sum_{l : f \in \text{OPT}_l} c_l - \text{TTL}[f], 0\Big)\Big) \tag{2}$$

We define $\Phi_1 = (h-1) \cdot \sum_{f \in \text{AL}} \text{TTL}[f]$ and $\Phi_2 = k \cdot (\sum_{f \in \text{OPT}} \max(1 + \sum_{l : f \in \text{OPT}_l} c_l - \text{TTL}[f], 0))$, so $\Phi = \Phi_1 + \Phi_2$. Clearly Φ is initially 0 when the cache is empty, and $\Phi \geq 0$ since the max term is at least 0. For simplicity, we split up write requests into a read followed by a write (any requested page not in the cache is always read first, so writes are always performed on pages in the cache). When there is a write request to a page having dirtiness l for either algorithm, we assume that the cost of c_{l+1} is paid immediately as opposed to when the page is evicted. In order to prove inequality (1), we show that:

- if Optimal brings a (clean) page into cache, Φ increases by at most k.
- if Asymmetric Landlord brings a page into cache, Φ decreases by at least $k - h + 1$ (regardless of whether or not it is a write).
- if there is a write to a page having dirtiness l_1 in Optimal and l_2 in Landlord, Φ increases by at most $kc_{l_1+1} - (k - h + 1)c_{l_2+1}$.
- at all other times Φ does not increase.

The total effect of each request on Φ can be broken down into steps, and we analyze the effect of each step on Φ. Note that each step assumes all previous steps have completed.

- *Optimal evicts a page f*: Since this just removes a term from Φ_2 and each term is nonnegative, Φ cannot increase.
- *Optimal retrieves a page g*: Optimal pays the read cost 1 (if it is a write request, the write is performed in a future step). Since $\text{TTL}[g] \geq 0$, Φ can increase by at most k.
- *Asymmetric Landlord decreases* $\text{TTL}[f]$ *for all $f \in AL$*: All TTL's are decreased by the same amount, call it Δ. Φ_1 decreases by $\Delta(h-1)k$. Φ_2 increases by at most Δk for each page in both OPT and AL. Therefore, the net decrease in Φ is at least Δ times $(h-1)k - k \cdot \text{size}(\text{OPT} \cap \text{AL})$ where $\text{size}(\text{OPT} \cap \text{AL})$ denotes the number of pages that appear in both OPT and AL. We know the requested page g is in OPT but not AL, so $\text{size}(\text{OPT} \cap \text{AL}) \leq h-1$. Therefore, the decrease in the potential function is at least Δ times $(h-1)k - k(h-1) = 0$ and thus Φ does not increase.
- *Asymmetric Landlord evicts a page f*: Asymmetric Landlord only evicts f when $\text{TTL}[f] = 0$. Thus Φ is unchanged.
- *Asymmetric Landlord retrieves the requested page g and sets* $\text{TTL}[g]$ *to* 1: In this step, Asymmetric Landlord pays the read cost of 1. Since g was not previously in the cache (and so $\text{TTL}[g]$ was zero), and because we know $g \in \text{OPT}$, Φ decreases by $-(h-1) + k = k - h + 1$.
- *Optimal writes to page g with dirtiness l_1 and Asymmetric Landlord writes to page g with dirtiness l_2 and adds c_{l_2+1} to* $\text{TTL}[g]$: Here, Optimal pays the write cost c_{l_1+1} and Asymmetric Landlord pays the write cost c_{l_2+1}. First note that an increase of $\text{TTL}[g]$ by c_{l_2+1} increses Φ_1 by $(h-1)c_{l_2+1}$. There are two cases: either g is dirtier in Asymmetric Landlord's cache, or it is not.

  If g is dirtier in Asymmetric Landlord's cache, then $l_2 > l_1$. When g is added to OPT_{l_1+1}, Φ_2 increases by some amount δ_1, which is at most c_{l_1+1}. Adding c_{l_2+1} to $\text{TTL}[g]$ then decreases Φ_2 by $\delta_2 = \min(c_{l_2+1}, \delta_1)$. Since $\mathcal{C}$ is concave, $c_{l_2+1} \leq c_{l_1+1}$, so $\delta_1 - \delta_2$ has a maximum value of $c_{l_1+1} - c_{l_2+1}$. Since the total increase toΦ_2 is $k(\delta_1 - \delta_2) \leq k(c_{l_1+1} - c_{l_2+1})$, the total increase in Φ is at most $k(c_{l_1+1} - c_{l_2+1}) + (h-1)c_{l_2+1} = kc_{l_1+1} - (k-h+1)c_{l_2+1}$.

If g is not dirtier in Asymmetric Landlord's cache, then $l_2 \leq l_1$. Since before the increase to TTL$[g]$ it was the case that TTL$[g] \leq 1 + \sum_{l \leq l_2} c_l \leq 1 + \sum_{l \leq l_1} c_l$ the addition of g to OPT$_{l_1+1}$ increases Φ_2 by c_{l_1+1} and the increase to TTL$[g]$ decreases Φ_2 by c_{l_2+1}, so the total increase to Φ_2 is $k(c_{l_1+1} - c_{l_2+1})$, and thus the total increase of Φ is $kc_{l_1+1} - (k-h+1)c_{l_2+1}$.

– *Asymmetric Landlord increases TTL*$[g]$ *to a maximum of* 1 *if the request is a read or* $c_1 + 1$ *if the request is a write*: Again, we know $g \in$ OPT, and if the request is a write we further know $g \in$ OPT$_1$. If TTL$[g]$ changes, it can only increase. In this case, since $h - 1 < k$, Φ decreases (though the amount of decrease depends on whether or not the request is a write).

3.2 Asymmetric Landlord with Read-throughs and Write-throughs

In this section, we consider the situation where a page need not be brought into cache to be accessed. We also assume that pages may be of different sizes, and that there is only one level of dirtiness (i.e., $c_i = 0$ for $i > 1$ and $c_1 = c$). Note that in the case of a write-through, the cost is just the cost to write the page (i.e., the cost is c, not $c + 1$). For a page f, we denote its size by $size(f)$. The algorithm AL2 that we give is similar to Asymmetric Landlord. Intuitively, the difference is when there is a cache miss, AL2 gives the accessed page the TTL that it would have gotten had it been moved into cache, and then performs the steps of AL; if AL would then have evicted this new page, then AL2 will access the page directly from PCM memory. To accomplish this, AL2 keeps both TTL$_r$ and a TTL$_w$ credits for each page which are affected differently by reads and writes. The time-to-live of a page can be thought of as the sum of these two values. The pseudo-code for AL2 can be found in the adjacent figure. Note that in line 10 when decreasing the sum we must make two assignment statements to ensure neither TTL goes negative. We then claim that AL2 is k/h-cache $2(k+1)/(k-h+1)$-competitive.

Theorem 2. *Asymmetric Landlord 2 is* $2(k+1)/(k-h+1)$ *competitive for asymmetric weighted page caching with read-throughs and write-throughs.*

Proof. To prove theorem 2, we start with a definition of fake caches that we will need to fully define the potential function.

Fake Caches: The concept of a fake cache is used to maintain a consistent relationship between OPT's cache and AL2's cache even if one or both of the algorithms satisfy a request for file f with a read-through or write-through. Both OPT and AL2 have a fake cache that is empty between requests, and during a request (sometimes) holds the currently requested file. Whether a fake cache holds the currently requested file simply depends on the actions of AL2 and OPT. Fake caches are not part of either algorithm, rather, they can be thought of as part of the potential function in that they are used for the accounting of costs only. We now describe each fake cache:

Algorithm 2. Asymmetric Landlord 2

```
1: When there is a request $(g, s)$
2: if $g$ is not in the cache then
3:     $\mathrm{TTL}_r[g] = 1$
4:     if $s = w$ then $\mathrm{TTL}_w[g] = c$
5:     until $|C| + size(g) \le k$ or $\mathrm{TTL}_r[g] + \mathrm{TTL}_w[g] = 0$
6:         $\Lambda = \min_{f \in C \cup g}(TTL_r[f] + TTL_w[f])/size[f]$
7:         for each $f \in C \cup g$: Decease $\mathrm{TTL}_r[f]$ + $\mathrm{TTL}_w[f]$ by $\Delta\cdot$ size$[f]$
8:         Evict all $f$ such that $\mathrm{TTL}_r[f] + \mathrm{TTL}_w[f] = 0$
9:     if $\mathrm{TTL}_r[g] + \mathrm{TTL}_w[g] > 0$ then Bring $g$ into the cache
10:    else Perform requested operation directly in memory
11: else Case 1: $g$ is clean and $s = w$ then Set $\mathrm{TTL}_w[g] = c$
12:     Case 2: $g$ is dirty and $s = w$ then Set $\mathrm{TTL}_w[g] = c$
13:     Case 3: $g$ is clean and $s = r$ then Set $\mathrm{TTL}_r[g] = 1$
14:     Case 4: $g$ is dirty and $s = r$ then Do nothing
```

OPT's Fake Cache: If OPT fulfils a request for file f with a read-through or write-through, then we say that OPT pays the read or write cost for f and brings f into a "fake" cache. At some point before the next request, we say that OPT then evicts f from the fake cache. When, exactly, the eviction occurs is dependent on AL2's actions for the current request, but it is always prior to the next request.

AL2's Fake Cache: On an AL2 cache miss, we say that AL2 sets $\mathrm{TTL}_r[f]$ to 1, puts f in AL2's "fake" cache and pays cost 1. If the request is a write then when AL2 sets $\mathrm{TTL}_w[f]$ to c, AL2 writes to clean file f in the fake cache and pays c. The decrease step will then determine if f is brought into the actual cache or evicted from the fake cache (i.e. a read or write through is performed). In either case, the fake cache will be empty prior to the next request.

We can now define the potential function to prove our result.

$$\begin{aligned}\Phi =& \frac{h}{k-h+1}\sum_{f\in AL2}(TTL_r[f] + TTL_w[f]) \\ &+ \frac{k+1}{k-h+1}\Big(\sum_{f\in OPT}(1 - TTL_r[f]) + \sum_{f\in OPT_d}(c - TTL_w[f])\Big)\end{aligned}$$

where OPT and OPT_d are the set of all pages and dirty pages respectively in OPT's cache and AL2 is the set of pages in AL2's cache. OPT and OPT_d contain the file in OPT's fake cache, and likewise AL2 contain the file in AL2's fake cache.

To prove theorem 2, we will show that for every request,

$$\Delta\mathcal{A}_k + \Delta\Phi \le 2\frac{k+1}{k-h+1}\Delta\mathcal{O}_h \tag{3}$$

To do this we start with what we call Actions, which are basic operations that either AL2 or OPT perform to fulfill a request. We will first show that equation 3 holds independent of when the Action happens or as long as some precondition is met. Then we consider what OPT and AL2 do when requests arrive. We show that when a request arrives, AL2 and OPT perform a sequence of Actions plus some additional steps, and that equation 3 holds at every point in this sequence.

Actions:

1. *OPT evicts from real or fake cache*: As $1 \geq \mathrm{TTL}_r[f]$ and $c \geq \mathrm{TTL}_w[f]$, Φ cannot increase.
2. *Landlord evicts*: Landlord only evicts when $\mathrm{TTL}_r[f] + \mathrm{TTL}_w[f] = 0$, thus Φ cannot increase.
3. *OPT reads from memory, writes to memory, or writes to a clean page in real or fake cache*:
   (a) OPT reads: OPT pays 1, and Φ can increase by at most $\frac{k+1}{k-h+1}$, giving, $\Delta\Phi \leq \frac{k+1}{k-h+1} \leq 2\frac{k+1}{k-h+1}\Delta\mathrm{OPT}_h$
   (b) OPT writes: Follow the read case with cost of c instead of cost of 1.
4. *AL2 reads g from memory, writes g to memory, or writes to a clean page g in real or fake cache* (precondition: $g \in OPT$ if reading, $g \in OPT_d$ if writing):
   (a) AL2 reads g and sets $\mathrm{TTL}_r[g]$ to 1: AL2 pays a cost of 1 while Φ increases by $\frac{h}{k-h+1}$ and decreases by $\frac{k+1}{k-h+1}$ as we know $g \in OPT$. This gives,$1 + \frac{h}{k-h+1} - \frac{k+1}{k-h+1} = (1 - \frac{k-h+1}{k-h+1}) = 0$
   (b) AL2 writes to clean page g and sets $\mathrm{TTL}_w[g]$ to c: Follow the read case but with cost of c instead of cost of 1 and OPT_d instead of OPT.
5. *Landlord performs a decrease* (precondition: OPT's fake cache is empty): If $\mathrm{size}(X)$ denotes the sum of the sizes of all files in X, then because the decrease of $\mathrm{TTL}_r[f]+\mathrm{TTL}_w[f]$ is $\Lambda\,\mathrm{size}(f)$, Φ increases by $\Lambda\frac{k+1}{k-h+1}\mathrm{size}((OPT \cup OPT_d) \cap AL2)$ and decreases by $\Lambda\frac{h}{k-h+1}\mathrm{size}(AL2)$. Thus we get, $\Delta\Phi = \Lambda(\frac{k+1}{k-h+1}\mathrm{size}((OPT \cup OPT_d) \cap AL2) - \frac{h}{k-h+1}\mathrm{size}(AL2))$. However, because OPT's fake cache is empty, any file in OPT_d is also in OPT, thus $\mathrm{size}((OPT \cup OPT_d) \cap AL2) = \mathrm{size}(OPT \cap AL2) \leq \mathrm{size}(OPT) \leq h$. It must be that $\mathrm{size}(AL2) \geq k+1$, otherwise everything could fit in AL2's cache and AL2 wouldn't perform a decrease. Thus we have that $\Delta\Phi \leq \Lambda(h\frac{k+1}{k-h+1} - (k+1)\frac{h}{k-h+1}) = 0$.

We break Requests into two main cases, when AL2 has a cache hit and when AL2 has a cache miss. For the cache miss case, we have three sub-cases, if OPT does a read-through, if OPT does a write-through, or if OPT does neither (OPT will never do both for a single request). We break each case into a number of steps where each step is either an Action or a more complex step which we explain in detail. If a step is an Action, then the number at the end of a step represents the Action of the current step, thus implying 3 holds at that step. For steps consisting of an Action with a precondition, we specify why the precondition is met at that step. For the rest of the steps we describe explicitly why equation 3 holds. Lastly, note that the case of a write request to clean file in AL2's cache will be accounted for as an AL2 cache miss. This is mainly because AL2 pays a cost in this case, while in the normal AL2 cache hit case, AL2 pays no cost.

Requests:

1. *AL2 Cache Hit*:
   (a) (if necessary) OPT evicts file $h \neq g$ from cache. (1)
   (b) (if $g \notin OPT$) OPT reads from memory, writes to memory, or writes to a clean page in real or fake cache. (3)
   (c) Landlord resets either $\mathrm{TTL}_r[g]$ or $\mathrm{TTL}_w[g]$: Call the amount increased in either case $\lambda \geq 0$. In both cases Φ increases by $h/(k-h+1) \cdot \lambda$. In the first case, it must be that $g \in OPT$, and in the second $g \in OPT_d$, thus in both cases, Φ decreases by $(k+1)/(k-h+1) \cdot \lambda$. Because $k+1 > h$, Φ cannot increase.
   (d) (if necessary) OPT evicts from fake cache. (1)
2. *AL2 Cache Miss*:
   (a) OPT doesn't perform a read-through or write-through:
      i. (if necessary) OPT evicts file $h \neq g$ from cache. (1)
      ii. (if necessary) OPT reads from memory and/or writes to a clean file in cache. (3)
      iii. AL2 reads/writes into fake or real cache: By the definition of the case, it must be that $g \in OPT$ and if the request is a write, $g \in OPT_d$ also, thus we can apply 4.
      iv. (if necessary) AL2 performs a decrease step: By the definition of the case, OPT's fake cache is empty, thus we can apply 5.
      v. (if necessary) AL2 evicts. (2)
   (b) OPT does a read-through:
      i. OPT reads into fake cache. (3)
      ii. AL2 reads into fake cache: By the definition of the case and the previous step, it must be that $g \in OPT$, thus we can apply 4.
      iii. OPT evicts from fake cache. (1)
      iv. (if necessary) AL2 decreases: By the previous step, OPT's fake cache is empty, thus we can apply 5.
      v. (if necessary) AL2 evicts. (2)
   (c) OPT does a write-through:
      i. We combine the case when OPT writes into fake cache and when AL2 reads/writes into fake cache. Here $\Delta\mathrm{OPT}_h = c$ and $\Delta\mathrm{AL2}_k = c+1$. When OPT writes, Φ increases by $\frac{k+1}{k-h+1}c$, when $AL2$ does the read, Φ increases by $\frac{h}{k-h+1}$. When $AL2$ does the write, Φ decreases by $\frac{k+1}{k-h+1}c$ and increases by $\frac{h}{k-h+1}c$. After cancelling we have that $\Delta\mathrm{AL2}_k + \Delta\Phi$ is bounded by $c+1+\frac{h}{k-h+1}+\frac{h}{k-h+1}c \leq 2\frac{k+1}{k-h+1}\Delta\mathrm{OPT}_h$.
      ii. OPT evicts from fake cache. (1)
      iii. (if necessary) LL decreases: By the previous step, OPT's fake cache is empty, thus we can apply 5.
      iv. (if necessary) LL evicts. (2)

We have shown that for all requests, equation 3 holds. The desired result then follows by summing over all requests.

4 Experimental Methodology

4.1 The Variable Aging Algorithm

Here we formally describe the Variable Aging (VA) algorithm. Let c be the average cost of writing a page to memory, and 1 be the cost of reading a page from memory. Each page in cache has an age associated with it. When a page f is requested, each page in cache has its age increased by 1 if it is clean and $1/c$ if it is dirty. If f is not in cache, if necessary (i.e., if the cache is full) evict the page with the highest age, and bring f into the cache. Reset the age of f to 0.

4.2 Methodology and Experimental Setup

The main memory architecture we consider consists of three levels of caches and the PCM main memory. The L1 instruction and data cache is a 4-way 64KB cache and the L2 cache is a unified 2MB 8-way associative cache. The 8MB L3 cache (i.e., the last level cache or LLC) works as a traditional write-allocate cache with a write-back policy (to PCM). That is, when a modified cache line is evicted from the LLC, it must be written to the PCM. A primary benefit the LLC provides to PCM main memory is that by coalescing a sequence of writes to the same line in the cache, the LLC partially mitigates the negative impacts of PCM writes. We assume the PCM write energy cost is 10x of PCM read [1,19].

We use Simics [20] to model the processor, L1 and L2 caches, and generate memory traces, which are input to an in-house cycle-accurate simulator that models the LLC and PCM. The memory trace contains, for each memory request by the CPU, the time stamp (assuming zero memory latency, that is, counting only CPU cycles to execute the task and L1/L2 cache latency), the type of request (read vs. write) and the physical address of the memory reference.

We use the SPEC CPU2006 [21] benchmarks for evaluation. Each benchmark was run in Simics for 2 billion instructions.

5 Evaluation Results

To understand the effectiveness of our schemes, we compare Asymmetric Landlord (AL) and Variable Aging (VA) to N-Chance [6].

Figures 1 (a) and (b) show the number of PCM reads and writes of 4-, 8-, 12-, 16-Chance, AL and VA normalized to LRU. The bar labeled “average” is the average result of all 15 benchmarks. In general, picking a large parameter N for N-Chance reduces the number of PCM writes at the cost of incurring more PCM reads. However, the impact of the parameter N on the number of PCM reads and writes depends on benchmarks.

To better understand this impact, we categorize the benchmarks into three groups. For *gcc* and *mcf*, N-Chance reduces the number of PCM reads as well as writes. This is because the dirty pages of these benchmarks are frequently accessed, and keeping the frequently accessed dirty pages is good for reducing

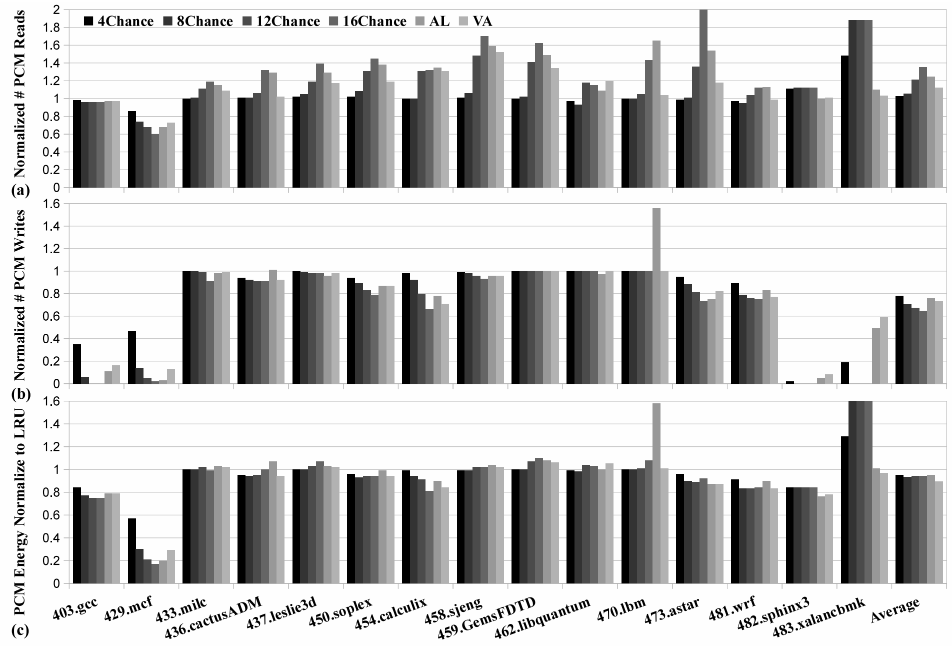


Fig. 1. Impact on number of (a) PCM reads, and (b) PCM writes (c) Energy

the LLC miss rate and writeback rate. As a result, 16-Chance is always the best choice. On the contrary, for *GemsFDTD*, *lbm*, and *xalancbmk*, N-Chance slightly reduces the number of PCM writes, while increasing the number of PCM reads substantially. Notice that even though 8-Chance eliminates all the PCM writes for *xalancbmk*, the benefit is limited due to the fact that memory accesses of *xalancbmk* are dominated by read references (i.e., $\geq$ 98%). In this case, we should pick $N = 1$. Lastly, for other benchmarks such as *milc*, *cactusADM*, and *leslie3d*, N-Chance reduces the number of PCM writes with comparable increase in the number of PCM reads. In this case, it is difficult to identify the value of N which leads to the best energy efficiency. In summary, there exists no universal value of the parameter N for N-Chance policy that minimizes the energy consumptions for applications.

On average, 4-, 8-, 12-, and 16-Chance increase the number of PCM reads by 3%, 5%, 21%, and 35%, and reduce the number of PCM writes by 22%, 30%, 33%, and 35% over LRU, respectively. As one might imagine, N-Chance will tend to rigidly reserve part of the LLC space to preferably store dirty pages, effectively reducing the LLC cache size therefore hurting the cache hit rate. For instance, for *xalancbmk*, N-Chance saves limited number of PCM writes (i.e., the total number of dirty pages is limited), but increases the number of PCM reads by up to 88%, compared to LRU. Compared to N-Chance, AL and VA do a better job in balancing the trade-off between PCM write reduction and PCM read increase, and adapting to various applications. Instead of rigidly reserving partial LLC capacity for dirty pages, AL and VA make replacement decisions based on liveliness and age information: AL assigns different TTL values to clean and dirty pages; VA ages clean and dirty pages at different rates. These mechanisms ensure that AL and VA do not suffer from the instances mentioned above. On average, AL and VA only increase the number of PCM reads by 11% and 3% for *xalancbmk* over LRU, respectively.

Figure 1 (c) shows the PCM energy consumption of 4-, 8-, 12-, 16-Chance, AL and VA normalized to LRU. Overall, N-Chance and our proposed policies save energy. For example, for *mcf*, 16-Chance outperforms LRU by 83% due to the 40% reduction in the number of PCM reads and the 98% reduction in the number of PCM writes. AL and VA also save 80% and 91% of the PCM energy consumption for *mcf*. As we pointed out, N-Chance fails to pick a universal parameter that is suitable for all benchmarks. 16-Chance is the best choice for *mcf*, *milc*, and *calculix*; 12-Chance is favorable for *astar*; 8-Chance ensures the best energy efficiency for *cactusADM*, and *libquantum*; while 1-Chance (i.e., LRU) is suitable for *xalancbmk*. The benchmark *xalancbmk* represents a pathological case for N-Chance replacement policy such that any parameter N except 1 will result in large increase in energy consumption. On the contrary, AL and VA can easily adjust to the access patterns of the applications to avoid the pathological cases. For instance, for *xalancbmk*, AL results in the same energy consumption as LRU, and VA even outperforms LRU by 3%.

AL achieves comparable energy savings against the best N-Chance for most of the benchmarks except *cactusADM* and *lbm*. VA delivers comparable energy efficiency to, or even outperforms the best N-Chance. On average, 4-, 8-, 12-, 16-Chance, AL and VA reduce the PCM energy by 5%, 7%, 6%, 6%, 5%, and 10% over LRU, respectively.

Acknowledgment. Supported in part by an IBM Faculty Award, and NSF grants CCF-0830558, CNS-1012070, CNS-1253218, and CCF-1115575. We thank Neal Young and Rami Melhem for invaluable discussions.

References

1. Lee, B.C., Ipek, E., Mutlu, O., Burger, D.: Architecting phase change memory as a scalable dram alternative. In: 36th International Symposium on Computer Architecture, ISCA, pp. 2–13 (2009)
2. Li, D., Vetter, J.S., Marin, G., McCurdy, C., Cira, C., Liu, Z., Yu, W.: Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In: 26th IEEE International Parallel and Distributed Processing Symposium, IPDPS, pp. 945–956 (2012)
3. Zhou, P., Zhao, B., Yang, J., Zhang, Y.: A durable and energy efficient main memory using phase change memory technology. In: 36th International Symposium on Computer Architecture, ISCA, pp. 14–23 (2009)
4. Qureshi, M.K., Karidis, J.P., Franceschini, M., Srinivasan, V., Lastras, L., Abali, B.: Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In: 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, pp. 14–23 (2009)

5. Cho, S., Lee, H.: Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. In: 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, pp. 347–357 (2009)
6. Ferreira, A.P., Zhou, M., Bock, S., Childers, B.R., Melhem, R.G., Mossé, D.: Increasing pcm main memory lifetime. In: Design, Automation and Test in Europe, DATE, pp. 914–919 (2010)
7. Chen, S., Gibbons, P.B., Nath, S.: Rethinking database algorithms for phase change memory. In: Fifth Biennial Conference on Innovative Data Systems Research, CIDR, pp. 21–31 (2011)
8. Ferreira, A.P., Childers, B.R., Melhem, R.G., Mossé, D., Yousif, M.: Using pcm in next-generation embedded space applications. In: IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, pp. 153–162 (2010)
9. Qureshi, M.K., Srinivasan, V., Rivers, J.A.: Scalable high performance main memory system using phase-change memory technology. In: 36th International Symposium on Computer Architecture, ISCA, pp. 24–33 (2009)
10. Hay, A., Strauss, K., Sherwood, T., Loh, G.H., Burger, D.: Preventing pcm banks from seizing too much power. In: 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, pp. 186–195 (2011)
11. Young, N.E.: On-line file caching. In: Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, pp. 82–86 (1998)
12. Chrobak, M., Woeginger, G.J., Makino, K., Xu, H.: Caching Is Hard – Even in the Fault Model. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part I. LNCS, vol. 6346, pp. 195–206. Springer, Heidelberg (2010)
13. Fiat, A., Karp, R.M., Luby, M., McGeoch, L.A., Sleator, D.D., Young, N.E.: Competitive paging algorithms. J. Algorithms 12(4), 685–699 (1991)
14. McGeoch, L.A., Sleator, D.D.: A strongly competitive randomized paging algorithm. Algorithmica 6(6), 816–825 (1991)
15. Bansal, N., Buchbinder, N., Naor, J.: Randomized competitive algorithms for generalized caching. In: 40th Annual ACM Symposium on Theory of Computing, STOC, pp. 235–244 (2008)
16. Adamaszek, A., Czumaj, A., Englert, M., Räcke, H.: An $o(\log k)$-competitive algorithm for generalized caching. In: Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, pp. 1681–1689 (2012)
17. Park, S.Y., Jung, D., Kang, J.U., Kim, J., Lee, J.: Cflru: a replacement algorithm for flash memory. In: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES, pp. 234–241 (2006)
18. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update rules. In: 16th Annual ACM Symposium on Theory of Computing, STOC, pp. 488–492 (1984)
19. Joo, Y., Niu, D., Dong, X., Sun, G., Chang, N., Xie, Y.: Energy- and endurance-aware design of phase change memory caches. In: Design, Automation and Test in Europe, DATE, pp. 136–141 (2010)
20. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer 35, 50–58 (2002)
21. Henning, J.L.: Spec cpu 2006 benchmark descriptions. SIGARCH Comput. Archit. News 34, 1–17 (2006)

Shortest-Elapsed-Time-First on a Multiprocessor

Neal Barcelo[1], Sungjin Im[2], Benjamin Moseley[2], and Kirk Pruhs[1]

[1] Department of Computer Science, University of Pittsburgh
{ncb30,kirk}@cs.pitt.edu
[2] Computer Science Department, University of Illinois
{im3,bmosele2}@illinois.edu

"I would like to call it a corollary of Moore's Law that the number of cores will double every 18 months." — Anant Agarwal, founder and chief technology officer of MIT startup Tilera

Abstract. We show that SETF, the idealized version of the uniprocessor scheduling algorithm used by Unix, is scalable for the objective of fractional flow on a homogeneous multiprocessor. We also give a potential function analysis for the objective of weighted fractional flow on a uniprocessor.

1 Introduction

At the hardware level, Moore's law continues unabated, with the number of transistors per chip doubling about every 1.5 to 2 years. However, we are in the midst of a revolutionary change in the effect of Moore's law on the software layers of the information technology stack. Instead of an exponential increase in processor speed over time, these layers are now expected to see an exponential increase in the number of processors over time. MIT startup Tilera now produces chips with up to 100 processors, and the expectation is that chips with 1000 processors will be available within the decade.

The natural research question motivating our research is whether the standard priority scheduling algorithms used for uniprocessors will be appropriate in the multiprocessor setting. In particular, we consider scheduling algorithm Shortest Elapsed Time First (SETF), as it is the idealized version of Unix's uniprocessor scheduling algorithm. (Of course the implementation in Unix has many practical kludges/modifications, such as maintaining equivalence queues of jobs that have been processed about the same amount so as to logarithmically bound the number of preemptions per job). For a uniprocessor using SETF for scheduling, all jobs that have been processed the least share the processing equally; it is useful to think of SETF giving higher priority to jobs that have been processed less. The natural generalization of SETF to a homogeneous multiprocessor setting assigns jobs to processors in priority order (recall jobs that have been processed less have higher priority); the x jobs of the next priority are either assigned to x processors, or evenly share the remaining unassigned processors if there are less than x previously unassigned processors.

Two natural quality of service measures for individual jobs are integer flow and fractional flow. The integer flow of a job is the total time a job has to wait to be completed. The fractional flow of a job is the integral over times between when a job arrives and when it is completed of the fraction of the job that is uncompleted. Integer flow is a more appropriate objective if no benefit is gained from a job being partially completed, and fractional flow is a more appropriate objective if some benefit is gained from partially completing a job. The corresponding two natural scheduling objectives are the integer flow of the schedule, which is the sum of the integer flow of the jobs, and fractional flow of the schedule, which is the sum of the fractional flow of the jobs.

On a uniprocessor, SETF is known to be scalable, $(1+\epsilon)$-speed $O(1)$-competitive, for the standard objective of integer flow [1], and it is known that speed augmentation is required to achieve bounded competitiveness in a general operating system setting requiring a nonclairvoyant scheduler, that is one that does not know the size of the jobs [2]. To the best of our knowledge, there are no results in the literature explicitly analyzing the fractional flow for nonclairvoyant algorithms on a uniprocessor (although it is possible that such results might be derivable from results on integer flow).

The main result of this paper is to show in Section 2 that for a homogeneous multiprocessor, SETF is universally[1] scalable for the objective of fractional flow.

The analysis in [1] shows that SETF is locally competitive for integer flow on a uniprocessor, that is, at all points in time the increase for the quality of service objective for SETF is not too much greater than the increase for an arbitrary schedule. But it is straightforward to see that no online scheduling algorithm can be locally competitive for either fractional or integer flow on a homogeneous multiprocessor. Thus the next logical approach is to try to use an amortized local competitiveness argument using the so called “standard potential function” method for these sorts of scheduling problems (for more background, see [3]). However, this standard approach is not immediately applicable in this setting as this approach requires a reasonably simple algebraic expression for the online algorithm’s future cost given no more job arrivals, and after some thought, one can see that a simple algebraic expression does not exist for SETF’s future costs on a multiprocessor. For fractional flow, we are able to surmount this difficultly by using a potential function based on an algebraic expression for SETF’s future costs on a uniprocessor. The primary differences between our potential function and the “standard potential function” are that it takes the difference of future costs between the work remaining in the optimal schedule and the online algorithm instead of the future cost of the difference in remaining work, and additionally our potential function discounts the optimal’s future costs. These modifications are necessary to get the running condition to hold; however, these modifications cause the potential function to jump when jobs arrive. Fortunately, we are able to complete the analysis by showing that the aggregate increase in these jumps

[1] An algorithm is said to be universally scalable if it is $(1+\epsilon)$-speed $O(f(\epsilon))$ competitive for any fixed constant $\epsilon > 0$ and the algorithm is not parameterized by ϵ. Here f is a function of only ϵ.

can be bounded by total processing times of all the jobs. Unfortunately we are unable to make this approach work for integer flow for SETF. Although one can resort to a technique to convert an algorithm that is fractionally scalable to an algorithm that is integrally scalable (see [4] for details). This technique combined with our analysis shows that a variation of SETF is scalable for integer flow.

There are two closely related results in the literature. It was known that if newly arriving jobs were randomly assigned to a processor, and if each processor ran SETF, that the resulting algorithm is universally scalable in expectation for integer flow [5]. Roughly this analysis combines the fact that SETF is universally scalable on a uniprocessor, with the fact that randomly assigning jobs roughly balances the processor loads (although the fact that there will be some unevenness in the loads in part explains why the competitive ratio that is proved is quite large, something like $O(\frac{1}{\epsilon^7})$). It is also known that the algorithm Late Arrival Processor Sharing (LAPS) is existentially[2] scalable for integer flow on a homogeneous multiprocessor [6].

There are often situations where one would like the operating system to view some jobs as being more important than other jobs. One way to formalize this is to assume that jobs have weights specifying their importance, and then consider the objective of minimizing the weighted fractional or integral flow of the jobs. WSETF is a natural generalization of SETF, where jobs are prioritized by the ratio of their weight to the time that jobs have been processed. It was shown in [7] that WSETF is scalable for a uniprocessor using a local competitiveness argument. In Section 3, we show that WSETF is scalable using an amortized local competitiveness argument using a potential function. As in our analysis of SETF, the starting point for the design of the potential function was an algebraic expression for the future cost of WSETF. However, we again had to make modifications to the "standard potential function" in order for the running condition to hold. We believe that our analysis is at least modestly interesting for a couple reasons. When one is analyzing algorithms in non-work-conserving scheduling settings, there is usually no hope of using a local competitiveness argument. In this context, a scheduling environment is said to be non-work-conserving if at any given time two reasonable scheduling algorithms could have completed a different amount of total work thus far. The lack of a potential function analysis of SETF and WSETF meant that these algorithms could not be used to design algorithms in non-work-conserving scheduling settings. For example, the analysis of nonclairvoyant speed scaling algorithms for a speed scalable processor in [8] considered the Late Arrival Processor Sharing Algorithm (LAPS) instead of SETF because a potential function analysis was known for LAPS [9]. It is our hope that our potential functions for SETF and WSETF will be useful in other non-work-conserving scheduling settings. Although in fairness we need to mention that we were unable to adapt our potential function analysis of WSETF for a uniprocessor to the multiprocessor setting because we do not

[2] An algorithm is said to be existentially scalable if it is $(1+\epsilon)$-speed $O(f(\epsilon))$ competitive for any fixed constant $\epsilon > 0$ and the algorithm is parameterized by ϵ. Here f is a function of only ϵ.

know how to bound the aggregate increases in the potential function when jobs arrive. But it is our hope that that one further idea would be enough to surmount this issue, and allow the application of this potential function (or some variation thereof) to non-work-conserving scheduling settings. Note that an existentially scalable algorithm, Weighted Late Arrival Processor Sharing, is known for the objective of integer flow on a homogeneous multiprocessor [10].

There is currently a debate within the architectural community as to whether a homogeneous multiprocessor or a heterogeneous multiprocessor is a better design [11]. There are advantages to each option. [12] points out that some standard priority scheduling algorithms, such as Highest Density First and WSETF, are not scalable for a heterogeneous multiprocessor, and that it is not clear whether other standard priority algorithms, such as Shortest Remaining Processing Time, Shortest Job First, and SETF, are scalable. So while this paper certainly does not settle the issue, taken together with [12], the results in this paper indicate that one advantage of homogeneous multiprocessors over heterogeneous multiprocessors is that they seem to be easier to schedule, and that in fact the standard uniprocessor scheduling algorithms should perform similarly well on a homogeneous multiprocessor as on a uniprocessor.

1.1 Basic Definitions

The input consists of n jobs. We let r_i denote the release time of job i, p_i denote the size of job i, and in some instances, w_i denote the weight of job i. An online scheduler does not learn about job i until time r_i. At time r_i, a nonclairvoyant scheduler learns the weight w_i but not the size p_i. For each time t, the online algorithm must choose some job i to run such that $r_i \geq t$. We assume that the processor has unit speed, so a job of size p_i, takes p_i units of time to complete. If C_i is the completion time for job i, then $\int_{t=r_i}^{C_i} w_i \, dt$ is the weighted integer flow for job i. The integer flow of a schedule is the sum over the jobs of the integer flow of each job. The weighted fractional flow of job i, is $\int_{t=r_i}^{\infty} w_i \cdot \frac{p_i(t)}{p_i} \, dt$, where $p_i(t)$ represents the remaining processing time of job i. The fractional flow of a schedule is the sum over the jobs of the fractional flow of each job. If the schedule is not obvious from context, we superscript a variable with the name of the schedule that is referred to.

An algorithm A is s-speed c-competitive if

$$\max_I \frac{A_s(I)}{\text{OPT}_1(I)} \leq c,$$

where $A_s(I)$ denotes the cost of algorithm A on input I with a speed s processor, $\text{OPT}_1(I)$ denotes the cost of the optimal schedule with a speed 1 processor, and the maximum is taken over all possible inputs. A class $\{A_{(1+\epsilon)}\}$ of algorithms is existentially scalable if for all $\epsilon > 0$, $A_{(1+\epsilon)}$ is $(1+\epsilon)$-speed $O(f(\epsilon))$-competitive for some function f that only depends on ϵ. An algorithm A is universally scalable if for all $\epsilon > 0$, A is $(1+\epsilon)$-speed $O(f(\epsilon))$-competitive for some function f.

To show that an algorithm A is $(c+d)$-competitive using a locally amortized competitiveness argument, one finds a potential function Φ such that the following conditions hold [3]:

Boundary condition: Φ is initially 0 and finally non-negative.
Completion condition: Φ does not increase due to completion of jobs by A or OPT.
Arrival condition: Φ does not increase by more than $d \cdot \mathrm{OPT}$ due to arrival of jobs.
Running condition: At all times t when no job arrives or is completed, we have,

$$\frac{d}{dt}A + \frac{d}{dt}\Phi(t) \leq c\frac{d}{dt}\mathrm{OPT}$$

Here $\frac{d}{dt}A$ denotes the increase in the objective in A's schedule, while $\frac{d}{dt}\mathrm{OPT}$ denotes the increase in the objective in OPT's schedule. $(c+d)$-competitiveness follows by integrating these conditions over time.

2 SETF on a Homogeneous Multiprocessor

As our first result, we show in Theorem 1 that SETF is universally scalable on a homogeneous multiprocessor for the objective of fractional flow using an amortized local competitiveness argument.

Theorem 1. SETF *is* $(1+\epsilon)$*-speed* $(1+\frac{5}{\epsilon})$*-competitive on a homogeneous multiprocessor for the objective of fractional flow.*

Proof. We use A to denote SETF. Let m denote the number of homogeneous mutliprocessors. We let $q_j^A(t)$ denote the amount of job j that has been processed up to time t. Note that $q_j^A(t)+p_j^A(t) = p_j$. Let, $(x)^+$ return x when x is positive, and 0 otherwise. Then, define $p_{i,j}^A(t) := (\min(p_i, p_j) - q_j^A(t))^+$. This represents the amount of time job i must wait on job j assuming no more jobs arrive. Note that it is possible that $i = j$. Similarly for OPT, $p_{i,j}^O(t) := (\min(p_i, p_j) - q_j^O(t))^+$. We let $Q_A(t)$ and $Q_O(t)$ denote the algorithm A's queue and OPT's queue, at time t respectively. Finally, let $Z_i^A(t) := \sum_{j \in Q_A(t)} p_{i,j}^A(t)$. Similarly, $Z_i^O(t) := \sum_{j \in Q_O(t)} p_{i,j}^O(t)$. We use an amortized local competitiveness argument. We define the potential function $\Phi(t)$ as follows.

$$\begin{aligned}\Phi(t) &= \frac{1}{m\epsilon}\sum_{i \in Q_A(t)} \frac{p_i^A(t)}{p_i}\left(Z_i^A(t) + mp_i^A(t) - Z_i^O(t)\right)\\ &= \frac{1}{m\epsilon}\sum_{i \in Q_A(t)} \frac{p_i^A(t)}{p_i}\Big(\sum_{j \in Q_A(t)} (\min(p_i,p_j) - q_j^A(t))^+ + mp_i^A(t)\\ &\qquad - \sum_{j \in Q_O(t)} (\min(p_i,p_j) - q_j^O(t))^+\Big)\end{aligned}$$

Boundary Condition: The boundary condition is trivially satisfied, as there are no jobs contributing to Φ at $t = 0$ or when all jobs have been finished.

Job Completion: Fix some job $i \in Q_A(t)$. Consider first when A completes job i. Note that at this time, $p_i^A(t) = 0$ and therefore there is no change in Φ from removing this term from the sum. Next, consider when A completes some job $j \neq i$. Since, $q_j^A(t) = p_j$, $p_{i,j}^A(t) = 0$, so there is no change in Φ from removing this term. Similarly, the completion of a job by OPT does not change Φ.

Job Arrival: We first show the following lemma.

Lemma 1. *Consider any job $i \in Q_A(t)$ and time t. Then it is the case that $Z_i^A(t) - Z_i^O(t) \leq mp_i$.*

Proof. Fix time t. Let $J(t)$ denote the set of all jobs in A's queue that have been processed less than job i's total processing time. More formally, we have $J(t) = \{j \in Q_A(t) \mid q_j^A(t) < p_i\}$. If $|J(t)| \leq m$, then there are at most m terms contributing to $Z_i^A(t)$ each of which have value at most p_i and so the desired result holds. So suppose $|J(t)| > m$. Consider the earliest time $t' \leq t$ such that at any time $\tau \in [t', t]$, $|J(\tau)| > m$. By definition of t', at time $t' - \delta$, there are at most m jobs that have elapsed processing times at most p_i. Now consider all jobs, denoted by S, which arrive during $[t', t]$. Note that for any time $\tau \in [t', t]$, for any job j that is run, $q_j^A(\tau) < p_i$ since $|J(\tau)| > m$. Therefore, $J(t) \subseteq J(t' - \delta) \cup S$. Consider $J(t)$'s contribution to $Z_i^A(t) - Z_i^O(t)$ at time t. Let $t'' = t' - \delta$.

$$\begin{aligned}
&\sum_{j \in J(t)} (\min(p_i, p_j) - q_j^A(t))^+ - (\min(p_i, p_j) - q_j^O(t))^+ \\
&\leq \sum_{j \in J(t)} (\min(p_i, p_j) - q_j^A(t)) - (\min(p_i, p_j) - q_j^O(t)) \qquad (1)\\
&= \sum_{j \in J(t)} (q_j^O(t) - q_j^A(t)) \\
&\leq \sum_{j \in J(t'')} (q_j^O(t'') - q_j^A(t'')) + \sum_{j \in J(t'')} (q_j^O(t) - q_j^O(t'')) - (q_j^A(t) - q_j^A(t'')) \\
&\quad + \sum_{j \in S} (q_j^O(t) - q_j^A(t)) \qquad (2)\\
&\leq mp_i \qquad (3)
\end{aligned}$$

Inequality (1) holds as based on the definition of $J(t)$ the first term in the sum will always be positive. (2) holds by noting that $J(t) = J(t') \cup S$ and rearranging terms while letting $\delta \to 0$. Finally, (3) is true because the first sum is less than mp_i as there are at most m terms of value p_i. Further, $\sum_{j \in J(t'')} (q_j^A(t) - q_j^A(t'')) + \sum_{j \in S} q_j^A(t)$ represents the total work that SETF did during this interval and $\sum_{j \in J(t'')} (q_j^O(t) - q_j^O(t'')) + \sum_{j \in S} q_j^O(t)$ cannot be more than the work that OPT did during this interval, therefore their difference is non-positive.

Given this lemma, note that when job i arrives, Φ increases by at most $\frac{2}{\epsilon}p_i$ and so summing over all arrivals, the increase is at most $\frac{4}{\epsilon}\mathrm{OPT}$ since $p_i/2$ is a lower bound for job i's fractional flow time in any schedule.

Running Condition: First note that $\frac{d}{dt}A = \sum_{i \in Q_A(t)} \frac{p_i^A(t)}{p_i}$. Also, $\frac{d}{dt}\mathrm{OPT} = \sum_{i \in Q_O(t)} \frac{p_i^O(t)}{p_i}$. We now bound the change in Φ at some time t when no jobs arrive or complete. We have that,

$$\frac{d}{dt}\Phi(t) = \frac{1}{m\epsilon} \sum_{i \in Q_A(t)} \Big(\frac{d\frac{p_i^A(t)}{p_i}}{dt} \cdot (Z_i^A(t) + mp_i^A(t) - Z_i^O(t)) + \frac{p_i^A(t)}{p_i} \cdot \frac{d(Z_i^A(t) + mp_i^A(t) - Z_i^O(t))}{dt} \Big)$$

First consider the change of $\frac{p_i^A(t)}{p_i}$. This occurs only when job i is being processed by SETF. Since SETF runs at speed $(1+\epsilon)$, $\frac{p_i^A(t)}{p_i}$ is decreasing at a rate of $(1+\epsilon)\frac{1}{p_i}$. To bound the overall rate of increase in Φ this can have, we ignore the positive terms $Z_i^A(t)$ and $mp_i^A(t)$ and consider only $-Z_i^O(t)$. Then, the rate of increase in Φ due to change in $\frac{p_i^A(t)}{p_i}$ is bounded by

$$\frac{1}{m\epsilon}(1+\epsilon)\frac{1}{p_i} \sum_{j \in Q_O(t)} (\min(p_i, p_j) - q_j^O(t))^+$$

To bound this sum, there are two cases to consider. First, consider all jobs j such that $p_j < p_i$. Then, we have that

$$\frac{1}{p_i}(\min(p_i, p_j) - q_j^O(t))^+ = \frac{1}{p_i} p_j^O(t) \leq \frac{p_j^O(t)}{p_j}$$

For all jobs j such that $p_j \geq p_i$, we have that

$$\frac{1}{p_i}(\min(p_i, p_j) - q_j^O(t))^+ = \left(\frac{p_i - q_j^O(t)}{p_i} \right)^+ = \left(1 - \frac{q_j^O(t)}{p_i} \right)^+ \leq \frac{p_j^O(t)}{p_j}$$

So, in total we have that

$$\begin{aligned} &\frac{1}{m\epsilon}(1+\epsilon)\frac{1}{p_i} \sum_{j \in Q_O(t)} (\min(p_i, p_j) - q_j^O(t))^+ \\ &\leq \frac{1}{m\epsilon}(1+\epsilon) \sum_{j \in Q_O(t)} \frac{p_j^O(t)}{p_j} \\ &= \frac{1+\epsilon}{m\epsilon} \frac{d}{dt}\mathrm{OPT} \end{aligned}$$

Since there are at most m such jobs as i running, the total rate of increase in Φ due to change in $\frac{p_i^A(t)}{p_i}$ is bounded by $(1 + \frac{1}{\epsilon})\frac{d}{dt}\mathrm{OPT}$.

We now turn our attention to the change of $(Z_i^A(t)+mp_i^A(t)-Z_i^O(t))$ for any job $i \in Q_A(t)$. Note that $p_i^A(t) > 0$, i.e. $q_i^A(t) < p_i$. If SETF is working on job i, then $mp_i^A(t)$ decreases at a rate of $m(1+\epsilon)$. Otherwise, if SETF does not work on i at time t, then there must exist m jobs j such that $q_j^A(t) < p_i$ that SETF is working on. In either case, $Z_i^A(t)+mp_i^A(t)$ decreases at a rate of $m(1+\epsilon)$. On the other hand, $Z_i^O(t)$ can increase at a rate of at most m. Therefore, the rate of change of Φ due to change in $(Z_i^A(t)+mp_i^A(t)-Z_i^O(t))$ is bounded by,

$$\frac{1}{m\epsilon}\sum_{i\in Q_A(t)}\frac{p_i^A(t)}{p_i}(-m(1+\epsilon)+m) = -\sum_{i\in Q_A(t)}\frac{p_i^A(t)}{p_i} = -\frac{d}{dt}A$$

So, in total, we have that

$$\frac{d}{dt}A+\frac{d}{dt}\Phi(t) \le \frac{d}{dt}A+\left(1+\frac{1}{\epsilon}\right)\frac{d}{dt}\mathrm{OPT}-\frac{d}{dt}A = \left(1+\frac{1}{\epsilon}\right)\frac{d}{dt}\mathrm{OPT}$$

We note that one can achieve an existentially scalable nonclairvoyant algorithm for integer flow by maintaining the invariant that each job is either done or has processed $(1+\epsilon)$ times as much as SETF would have processed it on $(1+\epsilon)$ slower processors.

3 WSETF on a Uniprocessor

We now show that WSETF is scalable on a single processor for the objective of weighted fractional flow. Recall the WSETF shares the processor equally among all jobs that have maximal ratio between weight and the amount that the job has been processed.

Theorem 2. WSETF *is* $(1+\epsilon)$*-speed* $(1+\frac{3}{\epsilon})$*-competitive on a uniprocessor for the objective of weighted fractional flow.*

Proof. We use A to denote the algorithm WSETF. Let $q_j^A(t)$ denote the amount of job j that has been processed up to time t. Let $p_{i,j}^A(t) := (\min(\frac{w_j}{w_i}p_i, p_j) - q_j^A(t))^+$ and $p_{i,j}^O(t) := (\min(\frac{w_j}{w_i}p_i, p_j) - q_j^O(t))^+$. We again use an amortized local competitiveness argument. Consider the following potential function $\Phi(t)$.

$$\begin{aligned}\Phi(t) &= \frac{1}{\epsilon}\sum_{i\in Q_A(t)}\frac{w_i\cdot p_i^A(t)}{p_i}(Z_i^A(t)+p_i^A(t)-Z_i^O(t))\\ &= \frac{1}{\epsilon}\sum_{i\in Q_A(t)}\frac{w_i\cdot p_i^A(t)}{p_i}\Big(\sum_{j\in Q_A(t)}(\min(\frac{w_j}{w_i}p_i,p_j)-q_j^A(t))^+ + p_i^A(t)\\ &\qquad -\sum_{j\in Q_O(t)}(\min(\frac{w_j}{w_i}p_i,p_j)-q_j^O(t))^+\Big)\end{aligned}$$

It is worth noting that $\sum_{i\in Q_A(t)}\frac{w_i\cdot p_i^A(t)}{p_i}(Z_i^A(t)+p_i^A(t))$ represents the approximate future cost of WSETF assuming no more jobs arrive. We now verify that all four conditions hold.

Boundary Condition: The boundary condition is trivially satisfied, as there are no jobs contributing to Φ at $t = 0$ or when all jobs have been finished.

Job Completion: Consider first when A completes job i. Note that at this time, $p_i^A(t) = 0$ and therefore there is no change in Φ from removing this term from the sum. Next, consider when A completes job j. Since, $q_j^A(t) = p_j$, $p_{i,j}^A(t) = 0$, so there is no change in Φ from removing this term. Similarly, the completion of job i or job j by OPT does not change Φ.

Job Arrival: We first show the following lemma.

Lemma 2. *Consider any job $i \in A(t)$ and time t. Then it is the case that $Z_i^A(t) - Z_i^O(t) \leq 0$.*

Proof. Fix time t. Consider the earliest time $t' \leq t$ such that at any time $\tau \in [t', t] = I$, WSETF works only on jobs j such that $q_j(\tau) \leq \frac{w_j}{w_i} p_i$. By definition of t', at time $t' - \epsilon$, all unfinished jobs have elapsed processing times at least $\frac{w_j}{w_i} p_i$, which thus contribute zero to $Z_i^A(t)$, so we can ignore those jobs. Now consider all jobs, denoted by S, which arrive during $[t', t]$. Consider S's contribution to $Z_i^A(t) - Z_i^O(t)$ at time t,

$$\begin{aligned}
&\sum_{j \in S} (\min(\frac{w_j}{w_i} p_i, p_j) - q_j^A(t))^+ - (\min(\frac{w_j}{w_i} p_i, p_j) - q_j^O(t))^+ \\
&\leq \sum_{j \in S} (\min(\frac{w_j}{w_i} p_i, p_j) - q_j^A(t)) - (\min(\frac{w_j}{w_i} p_i, p_j) - q_j^O(t)) \qquad (4)\\
&= \sum_{j \in S} q_j^O(t) - q_j^A(t) \qquad (5)
\end{aligned}$$

Note that (4) holds as based on the definition of S and I, for any job $j \in S$, $q_j^A(t) \leq \frac{w_j}{w_i} p_i$. Now consider the term in (5), $\sum_{j \in S} q_j^O(t) - q_j^A(t)$. First note that $\sum_{j \in S} q_j^A(t)$ captures the total work that WSETF did during the interval, and further $\sum_{j \in S} q_j^O(t)$ cannot exceed the amount of work that OPT did during the same interval. Therefore, this term is non-positive.

Given this lemma, note that when job i arrives, Φ increases by at most $\frac{1}{\epsilon}(w_i \cdot p_i)$. So, summing over all arrivals, Φ increases by at most $\frac{1}{\epsilon} \sum_i w_i \cdot p_i \leq \frac{2}{\epsilon}\mathrm{OPT}$ as desired.

Running Condition: First note that $\frac{d}{dt} A = \sum_{i \in Q_A(t)} w_i \cdot \frac{p_i^A(t)}{p_i}$. Also, $\frac{d}{dt}\mathrm{OPT} = \sum_{i \in Q_O(t)} w_i \cdot \frac{p_i^O(t)}{p_i}$. We now bound the change in Φ at some time t when no job arrives or is completed. We have that,

$$\begin{aligned}
\frac{d}{dt}\Phi(t) = \frac{1}{\epsilon} \sum_{i \in Q_A(t)} \Big(&\frac{d \frac{w_i p_i^A(t)}{p_i}}{dt} \cdot (Z_i^A(t) + p_i^A(t) - Z_i^O(t)) \\
&+ \frac{w_i p_i^A(t)}{p_i} \cdot \frac{d(Z_i^A(t) + p_i^A(t) - Z_i^O(t))}{dt} \Big) \qquad (6)
\end{aligned}$$

First consider the change of $\frac{w_i p_i^A(t)}{p_i}$. This occurs only when job i is being processed by A. We assume without loss of generality that A works on a single job at each time t. Then, since WSETF runs at speed $(1+\epsilon)$, $\frac{w_i p_i^A(t)}{p_i}$ is decreasing at a rate of $(1+\epsilon)\frac{w_i}{p_i}$. To bound the overall rate of increase in Φ this can have, we ignore the positive terms $Z_i^A(t)$ and $p_i^A(t)$ and consider only $-Z_i^O(t)$. Then, the rate of increase in Φ due to change in $\frac{w_i p_i^A(t)}{p_i}$ is bounded above by

$$\frac{1}{\epsilon}(1+\epsilon)\frac{w_i}{p_i}Z_i^O(t) = \left(1+\frac{1}{\epsilon}\right)\frac{w_i}{p_i}\sum_{j\in Q_O(t)}\left(\min(\frac{w_j}{w_i}p_i, p_j) - q_j^O(t)\right)^+$$

To bound this sum, we again consider two cases. First, consider all jobs j such that $\frac{w_j}{p_j} > \frac{w_i}{p_i}$. Then,

$$\frac{w_i}{p_i}(\min(\frac{w_j}{w_i}p_i, p_j) - q_j^O(t))^+ = \frac{w_i}{p_i}(p_j - q_j^O(t)) \le w_j \cdot \frac{p_j^O(t)}{p_j}$$

Now, for all jobs j such that $\frac{w_j}{p_j} \le \frac{w_i}{p_i}$, we have that

$$\frac{w_i}{p_i}(\min(\frac{w_j}{w_i}p_i, p_j) - q_j^O(t))^+ = \frac{w_i}{p_i}(\frac{w_j}{w_i}p_i - q_j^O(t))^+ = (w_j - \frac{w_i}{p_i}q_j^O(t))^+ \le w_j \cdot \frac{p_j^O(t)}{p_j}$$

Combining these, we have that

$$\left(1+\frac{1}{\epsilon}\right)\frac{w_i}{p_i}\sum_{j\in Q_O(t)}\left(\min(\frac{w_j}{w_i}p_i, p_j) - q_j^O(t)\right)^+ \le \left(1+\frac{1}{\epsilon}\right)\frac{d}{dt}\mathrm{OPT}$$

We now turn our attention to the change of $(Z_i^A(t) + p_i^A(t) - Z_i^O(t))$ for any job $i \in Q_A(t)$. Note that $p_i^A(t) > 0$, i.e. $q_i^A(t) < p_i$. Thus if WSETF does not work on i at time t, then there must exist a job j such that $\frac{q_j^A(t)}{w_j} < \frac{p_i}{w_i}$ that WSETF is working on. In either case, $Z_i^A(t) + p_i^A(t)$ decreases at a rate of $1+\epsilon$. On the other hand, $Z_i^O(t)$ can increase at a rate of at most 1. Therefore, the rate of change in Φ due to change in $(Z_i^A(t) + p_i^A(t) - Z_i^O(t))$ is bounded above by

$$\frac{1}{\epsilon}\sum_{i\in Q_A(t)}\frac{w_i p_i^A(t)}{p_i}(-(1+\epsilon)+1) = -\sum_{i\in Q_A(t)}\frac{w_i p_i^A(t)}{p_i} = -\frac{d}{dt}A$$

So, in total, we have that

$$\frac{d}{dt}A + \frac{d}{dt}\Phi(t) \le \frac{d}{dt}A + \left(1+\frac{1}{\epsilon}\right)\frac{d}{dt}\mathrm{OPT} - \frac{d}{dt}A = \left(1+\frac{1}{\epsilon}\right)\frac{d}{dt}\mathrm{OPT}$$

Our analysis of WSETF does not extend to a homogeneous multiprocessor because we do not know how to bound the jumps in the potential function when jobs arrive, in part because when a job i arrives the increase in the potential involves terms of the form $p_i w_j$. We are able to surmount this difficulty in our analysis of SETF because all jobs have equal weight, and the sum of the processing times is a lower bound to optimal.

Acknowledgment. Supported in part by an IBM Faculty Award, and NSF grants CCF-1016684, CCF-0830558, CCF-1115575, CNS-1012070, CNS-1115575, and CNS-1253218.

References

1. Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance. J. ACM 47(4), 617–643 (2000)
2. Motwani, R., Phillips, S., Torng, E.: Non-clairvoyant scheduling. Theor. Comput. Sci. 130(1), 17–47 (1994)
3. Im, S., Moseley, B., Pruhs, K.: A tutorial on amortized local competitiveness in online scheduling. SIGACT News 42(2), 83–97 (2011)
4. Chadha, J.S., Garg, N., Kumar, A., Muralidhara, V.N.: A competitive algorithm for minimizing weighted flow time on unrelated machines with speed augmentation. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, pp. 679–684 (2009)
5. Chekuri, C., Khanna, S., Goel, A., et al.: Multi-processor scheduling to minimize flow time with resource augmentation. In: Proc. 36th Symp. Theory of Computing (STOC), pp. 363–372. ACM (2004)
6. Edmonds, J., Pruhs, K.: Scalably scheduling processes with arbitrary speedup curves. In: SODA, pp. 685–692 (2009)
7. Bansal, N., Dhamdhere, K.: Minimizing weighted flow time. ACM Trans. Algorithms 3(4) (November 2007)
8. Chan, H.L., Edmonds, J., Lam, T.W., Lee, L.K., Marchetti-Spaccamela, A., Pruhs, K.: Nonclairvoyant speed scaling for flow and energy. Algorithmica 61(3), 507–517 (2011)
9. Chan, H.L., Edmonds, J., Pruhs, K.: Speed scaling of processes with arbitrary speedup curves on a multiprocessor. Theory Comput. Syst. 49(4), 817–833 (2011)
10. Bansal, N., Krishnaswamy, R., Nagarajan, V.: Better Scalable Algorithms for Broadcast Scheduling. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part I. LNCS, vol. 6198, pp. 324–335. Springer, Heidelberg (2010)
11. Merrit, R.: Cpu designers debate multi-core future. EE Times (February 2010)
12. Gupta, A., Im, S., Krishnaswamy, R., Moseley, B., Pruhs, K.: Scheduling heterogeneous processors isn't as easy as you think. In: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, pp. 1242–1253. SIAM (2012)

Efficient Route Compression for Hybrid Route Planning*

Gernot Veit Batz, Robert Geisberger, Dennis Luxen, Peter Sanders, and Roman Zubkov

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
{batz,luxen,sanders}@kit.edu

Abstract. We describe an algorithmic framework for lossless compression of route descriptions. This is useful for hybrid route planning where routes are computed by a server and then transmitted to a client device in a car using some mobile radio communication where bandwidth may be low. Compressed routes are represented by only a few via nodes which are the connection points when the route is decomposed into unique optimal segments. To reconstruct the route efficiently a client device needs basic but fast route planning capability. Contraction hierarchies make this approach fast enough for practice: Compressing takes only a few milliseconds. And previous experiments suggest that a client can decompress each route segment virtually instantaneously. So, as the segments can be decompressed successively while driving, it is not likely that the driver experiences any delay except for the time needed by the mobile communication.

1 Introduction

Today GPS-based car navigation is quite common. Routes can be computed either by a device located in the car or by a *server* system located in a computing center. The latter requires that routes are transmitted to the *client* device in the car using some *mobile radio communication* like the cellular phone network. We denote this server-based mobile setting as *hybrid route planning.*

Hybrid route planning is not only useful to take the current traffic situation or the latest changes of POI data into account. It also makes the benefits of several advanced route planning algorithms available for car drivers. Such algorithms, which compute high quality routes within milliseconds, are often quite sophisticated and adapting them to work well on mobile devices is usually not trivial – if at all possible. Examples are time-dependent route planning [1–7] where routes depend on the departure time, flexible route planning [8] where routes depend on a freely selectable parameter which models a tradeoff between energy consumption and travel time for example, multi-criteria route planning [9] where routes are assessed with respect to multiple costs, customizable route planning [10] where cost functions can be altered rapidly, or the computation of alternative routes [11, 12]. Possible benefits are, for example,

* This work was partially supported by DFG project SA 933/5-1,2.

– that routes are optimized with respect to the time of day which means that congestions can be avoided based on statistical data,
– that inconvenient roads can be avoided and that toll charges, energy consumption, or detours can be reduced even if travel time is the main objective,
– or that the driver can dynamically choose or fine-tune the cost function.

To make hybrid route planning convenient to use, the latency experienced by the driver should be as small as possible. However, the descriptions of the routes that have to be transmitted over the mobile communication can be quite complex and bandwidths can be low (as in the countryside for example). So, good compression rates are desirable. Also, the time needed for compressing and decompressing the routes has to be small.

In this work we present an algorithmic framework for lossless and efficient compression of route descriptions in the context of hybrid route planning. It provides good compression rates and the running times needed for compressing and decompressing are small. As a result, the user should not notice any latency except for the latency of the mobile communication. Our approach requires that

– the client has basic but fast route planning capability,
– client and server use the same road network topology, and
– the cost function used by the client changes rarely and is known to the server.

A route provided by a sophisticated algorithm running on the server is most probably not an optimal route with respect to a simple cost function that can be handled by the mobile client device. We observe, however, that the route can be composed of a few *unique* route *segments* which are optimal with respect to the client. Our compression exploits this in a simple but effective way: We represent the route by only giving the few locations where this unique optimal route segments meet. We call these locations the *via nodes*. The client can reconstruct the route from the few via nodes by simply computing the optimal routes between them. The uniqueness of the segments guarantees that the reconstructed route is exactly the route originally provided by the server.

To provide efficient decompression, the client device must be able to to perform fast and exact[1] computation of optimal routes. There, it is enough to decompress the *first* route segment fast. All other segments can be decompressed successively when driving. *Contraction hierarchies (CH)* [13, 14] is a fast and exact method for route planning which has also been adapted to run efficiently on mobile devices. Using these mobile CH the client should be able to decompress each segment of the route within less than 0.1 s [15]. From the drivers point of view this is as good as instantaneous. Note that Dijkstra's well known algorithm is *not* an alternative. Though it computes optimal routes, it has running times of more than a second even on server systems which is far to slow.

Our Contributions. With CH the client is already able to decompress each segment of the route fast enough. So, we focus on the server-side algorithmic methods of computing the compressed representation of a path P as a sequence Q

[1] We speak of *exact* route planning to indicate that the computed routes are optimal with respect to the underlying cost function and no heuristic is used.

of via nodes. All these algorithmic methods are instances of a generic frame algorithm (Sect. 2). The first instantiation of the frame algorithm is based on Dijkstra's well known algorithm and yields the *minimum* possible number of via nodes. In practice, however, the Dijkstra-based compression is to slow. The second instantiation uses an algorithmic scheme inspired by binary search. It also yields minimal sequences of via nodes. Like the frame algorithm the binary scheme is a generic algorithm itself. More precisely, it requires a subroutine that decides whether a route is the unique optimal route between two nodes. This subroutine is invoked $O(|Q|\log|P|)$ times (Sect. 3).

Realizing the subroutine using the aforementioned CH (Sect. 4) yields a very fast compression technique, fast enough for practice. Interestingly, this approach needs sometimes even less via nodes than the Dijkstra-based approach (Sect. 5). This is because the definition of a unique optimal route is different in the context of CH. Our experiments indicate that via nodes provide good compression rates in practice. Also the running time needed by the compression is quite low if CH are used to realize the subroutine in the generic binary scheme (Sect. 6).

Related Work. We described some of the ideas presented in this work previously in a technical report [16]. To our knowledge, there is no other publication directly covering the efficient representation of routes beyond traditional data compression and error-correction. Tao et al. [17] show how to efficiently compute a representation that includes at least one out of every k consecutive nodes. While this can be seen as a compact representation of a shortest path it is not clear how to conduct a loss-less reconstruction of the represented path. Via nodes have been applied to CH in a different way [11, 12] to provide *reasonable* alternatives to the optimal route. Here, reasonable means that the alternative is not much longer, has not too much in common with the optimal route and is locally optimal.

2 Via Nodes and a Generic Frame Algorithm

We model road networks as directed graphs $G = (V, E)$. As the mobile device has limited main memory and computing power, it uses a very simple cost function: Every edge (u, v) has a constant weight $c(u, v) \in \mathbb{R}_{>0}$ assigned. Routes are modeled as paths in G. A path $\langle u_1, \ldots, u_k\rangle$ is a *shortest* path if it has minimal cost $c(\langle u_1, \ldots, u_k\rangle) := c(u_1, u_2) + \cdots + c(u_{k-1}, u_k)$ among all paths from u_1 to u_k. A shortest path $\langle u, \ldots, v\rangle$ is called a *unique* if it is the only shortest path from u to v. Subpaths of unique shortest paths are unique shortest paths too. By $G^\top$ we denote the *transpose* graph of G where all edges are reversed.

Consider a path $P := \langle u_1, \ldots, u_n\rangle$ which is not necessarily a shortest path. Let $Q := \langle\langle u_{i_1}, \ldots, u_{i_k}\rangle\rangle$ be a subsequence of P s.t. the subpaths $\langle u_1, \ldots, u_{i_1}\rangle$, $\langle u_{i_k}, \ldots, u_n\rangle$, and $\langle u_{i_j}, \ldots, u_{i_{j+1}}\rangle$ of P are unique shortest paths for all $1 \le j < k$. Then, we call Q a *representation of* P *(with via nodes)*. Certainly, P is completely determined by Q. For $|Q| \ll |P|$ we can speak of a compressed representation.

Observation 1. *If all edges in* G *are unique shortest paths, then all paths in* G *can be represented with via nodes.*

Algorithm 1. Generic frame algorithm computing a representation with via nodes for a path P. Requires a subprocedure *uniqueShortestPrefix*(*Path*) : *Path*.

```
function frameAlgorithm($P$ : Path) : Sequence
    $Q := \langle\!\langle\rangle\!\rangle$ : Sequence
    while $P \neq \langle\rangle$ do
        $R :=$ uniqueShortestPrefix($P$)
        append last node of $R$ to the end of $Q$
        remove prefix $R$ from $P$
    remove last node of $Q$
    return $Q$
```

A trivial representation of P is P itself. If not all edges of G are unique shortest paths, then this property can be established easily by a simple transformation of G: For every edge (u, v) in G we run a Dijkstra search starting from u with the constraint that the edge (u, v) must not be relaxed. This yields a shortest path P_{uv} in the graph $(V, E \setminus \{(u, v)\})$. If $c(P_{uv}) \leq c(u, v)$ holds, we introduce a new node x to G and replace (u, v) by the new edges (u, x) and (x, v) with $c(u, x) := c(x, v) := c(u, v)/2$. In the following we assume that all edges of G are unique shortest paths. Thus, every path in G can be represented with via nodes. A representation Q of P with via nodes is called *minimal* if there exists no other representation Q' of P s.t. $|Q'| < |Q|$. If P is a unique shortest path, then the minimal representation is $\langle\!\langle\rangle\!\rangle$.

A prefix $R_i := \langle u_1, \dots, u_i\rangle$ with $i < n$ is called a *real* prefix of P. If R_i is a unique shortest path, then R_i is called a *unique shortest prefix* of P. If R_{i+1} is *not* a unique shortest path, then R_i is called the *maximal* unique shortest prefix of P. A generic frame algorithm (see Algorithm 1) computes a representation with via nodes for a given path. It requires that a procedure *uniqueShortestPrefix* is present, which computes a unique shortest prefix of a given path. Obviously, *uniqueShortestPrefix* is called $O(|Q|)$ times. The frame algorithm can also be used to find *minimal* representations as the following theorem shows.

Theorem 2. *If uniqueShortestPrefix provides the <u>maximal</u> unique shortest prefix, then Algorithm 1 yields a <u>minimal</u> representation with via nodes.*

Proof. Set $\langle s, \dots, t\rangle := P$ and let $Q = \langle\!\langle u_1, \dots, u_k\rangle\!\rangle$ be the result of the frame algorithm. Assume there is another representation $Q' = \langle\!\langle v_1, \dots, v_\ell\rangle\!\rangle$ of P with $\ell < k$. Then $\langle s, \dots, v_i\rangle$ is a prefix of $\langle s, \dots, u_i\rangle$ for $1 \leq i \leq \ell$. For $i = 1$ this is true because $\langle s, \dots, u_1\rangle$ is computed by *uniqueShortestPrefix* and hence the maximal unique shortest prefix of P. For $i > 1$ we apply induction and assume $\langle s, \dots, v_{i-1}\rangle$ is prefix of $\langle s, \dots, u_{i-1}\rangle$. But then $\langle s, \dots, v_i\rangle$ must also be prefix of $\langle s, \dots, u_i\rangle$. Otherwise $\langle u_{i-1}, \dots, u_i\rangle$, which is the maximal unique shortest prefix of $\langle u_{i-1}, \dots, t\rangle$, would be a real prefix of the unique shortest path $\langle u_{i-1}, \dots, v_i\rangle$. But this is not possible. Hence, $\langle s, \dots, v_\ell\rangle$ is a prefix of $\langle s, \dots, u_\ell\rangle$. But this means that $\langle u_\ell, \dots, t\rangle$ is subpath of $\langle v_\ell, \dots, t\rangle$ and hence a unique shortest path.

Algorithm 2. Modification of Dijkstra's algorithm computing a maximal unique shortest prefix path of a given path $P = \langle u_1, \dots, u_n \rangle$.

```
1  function uniqueShortestPrefix($\langle u_1, \dots, u_n \rangle$ : Path) : Path
2      $d[u] := \infty$, $p[u] := \bot$, $unique[u] := true$ for all $u \in V$
3      $k := n$, $d[u_1] := 0$
4      $M := \{(u_1, 0)\}$ : PriorityQueue
5      while $M \neq \emptyset$ do
6          $u := M.deleteMin()$                                    // settle node $u$
7          if $u = u_i \in P$ then
8              if $\neg unique[u_i]$ or $c(\langle u_1, \dots, u_i \rangle) \neq d[u_i]$ then
9                  $k := \min\{k, i-1\}$
10         for $(u, v) \in E$ do
11             if $d[u] + c(u, v) < d[v]$ then                     // relax edge $(u, v)$
12                 if $d[v] = \infty$ then $M.insert(v, d[u] + c(u, v))$
13                 else $M.decreaseKey(v, d[u] + c(u, v))$
14                 $d[v] := d[u] + c(u, v)$
15                 $p[v] := u$
16                 $unique[v] := true$
17             else if $d[u] + c(u, v) = d[v]$ then $unique[v] := false$
18         if $d[u] > c(\langle u_1, \dots, u_k \rangle)$ then break
19     return $\langle u_1, \dots, u_k \rangle$
```

But this contradicts the fact that $\langle u_\ell, \dots, u_{\ell+1} \rangle$ with $u_{\ell+1} \neq t$ is the maximal unique shortest prefix of $\langle u_\ell, \dots, t \rangle$. □

All compression methods described in this work are instantiations of the frame algorithm with different realizations of the procedure *uniqueShortestPrefix*.

3 Dijkstra-Based Compression and a Generic Scheme

The first realization of the procedure *uniqueShortestPrefix* which we describe is actually Dijkstra's well known algorithm plus some additional actions. It returns the maximal unique shortest prefix of a given path as we will see (Algorithm 2).

The original version of Dijkstra's algorithm computes shortest paths from a given start node s to all reachable nodes in a graph. To do so it successively labels all nodes w with labels $d[w]$ and $p[w]$, where $d[w]$ is the tentative cost from s to w and $p[w]$ the predecessor of w on the corresponding tentative path. After termination we can obtain a shortest path P_{su} from s to a node u by successively selecting the next predecessor node starting from u:

$$P_{su} \;=\; \big\langle s = p[\dots p[u] \dots], \dots, p[p[u]], p[u], u \big\rangle$$

When a node is *settled* (i.e., removed from the priority queue, Line 6), its tentative cost equals the cost of a shortest path and changes no more. A detailed explanation of Dijkstra's algorithm can be found in textbooks (e.g., [18]). Our modified Dijkstra starts from u_1 and maintains an index k which is initialized with n and repeatedly decreased until $\langle u_1, \ldots, u_k \rangle$ is a unique shortest prefix.

Lemma 3. *For $u \in V$ let $P_u := \langle u_1, \ldots, p[p[u]], p[u], u \rangle$ be the shortest path found by Algorithm 2. If $unique[w]$ holds for all $w \in P_u$, then P_u is unique.*

Proof. Otherwise, there is a shortest path $P' := \langle u_1, \ldots, w', w \rangle$ with $w' \notin P_u$ for some $w \in P_u$. After $(p[w], w)$ and (w', w) have been *relaxed* (Lines 11 to 17) we have $\neg unique[w]$ because P' and $\langle u_1, \ldots, p[w], w \rangle \subseteq P_u$ are shortest paths, and $d[w] = d[p[w]] + c(p[w], w) = d[w'] + c(w', w)$ can not further decrease. But this also means that $unique[w]$ will not be changed anymore – a contradiction. □

Lemma 4. *Algorithm 2 computes a unique shortest prefix.*

Proof. Let k_0 be the final value of k. Surely, $\langle u_1, \ldots, u_{k_0} \rangle$ is a shortest path. Otherwise, the algorithm would return a real prefix of $\langle u_1, \ldots, u_{k_0} \rangle$ because of the second condition in Line 8. Also, $\langle u_1, \ldots, u_{k_0} \rangle$ is unique according to Lemma 3 as $unique[u_j]$ holds for $1 \leq j \leq k_0$. Otherwise, we would have $\neg unique[u_j]$ for some u_j at the time when u_j is settled, because *unique* does not change for settled nodes as their tentative cost is already minimal. But then, the algorithm would perform $k := j - 1 < k_0$ which can not be the case. □

Theorem 5. *Algorithm 2 computes the <u>maximal</u> unique shortest prefix.*

Proof. Otherwise, $\langle u_1, \ldots, u_{k_0+1} \rangle$ is a unique shortest prefix with k_0 the final value of k. As the algorithm sets k to k_0, one of the conditions in Line 8 must be fulfilled. But as $\langle u_1, \ldots, u_{k_0+1} \rangle$ is a shortest path, it is $\neg unique[u_{k_0+1}]$ which holds when u_{k_0+1} is settled. This means that u_{k_0+1} is reached by two different paths with the same cost which must be minimal as it is not decreased afterwards. But we assumed that the shortest path $\langle u_1, \ldots, u_{k_0+1} \rangle$ is unique. □

So, if we instantiate the procedure *uniqueShortestPrefix* in the frame algorithm (Algorithm 1) by our modified Dijkstra search (Algorithm 2), we get a method to compute the minimal representation of a given path with via nodes.

But we also consider another realization of *uniqueShortestPrefix* which we call the *generic binary scheme* (Algorithm 3). It is heavily inspired by binary search and like the frame algorithm it is also generic. It requires that a procedure *isUniqueShortestPath* is present, which decides whether a given path is a unique shortest path or not. This subprocedure is invoked $O(\log |P|)$ times.

Corollary 6. *Algorithm 3 computes the <u>maximal</u> unique shortest prefix.*

Instantiating the procedure *uniqueShortestPrefix* in the frame algorithm (Algorithm 1) by the generic binary scheme (Algorithm 3) we get a further generic method to compute the minimal representation with via nodes Q of a path P. Obviously, the procedure *isUniqueShortestPath* is called $O(|Q| \log |P|)$ times.

In the following we instantiate the subprocedure *isUniqueShortestPath* in the generic binary scheme using the aforementioned CH. This yields a quite fast

Algorithm 3. A generic binary scheme checking whether $\langle u_1, \ldots, u_n \rangle$ is a unique shortest path. Requires a subprocedure *isUniqueShortestPath*(*Path*) : *bool*.

```
1 function uniqueShortestPrefix($\langle u_1, \ldots, u_n \rangle$ : Path) : Path
2     $(\ell, m, r) := (1, n, n)$
3     while $\ell + 1 < r$ do
4         if isUniqueShortestPath($\langle u_1, \ldots, u_m \rangle$) then $\ell := m$
5         else $r := m$
6         $m := \lfloor (\ell + 1 + r)/2 \rfloor$
7     return $\langle u_1, \ldots, u_\ell \rangle$
```

realization. It should be noted, however, that the structure of a CH is different from the structure of the original road network. As a result, via nodes are no longer nodes where unique shortest paths meet, but nodes where paths meet that are *uniquely representable* with respect to CH.

4 Representing Paths Uniquely with CH

In the CH framework [13, 14] we derive a *hierarchical* structure from the original road network G in a relatively expensive *preprocessing* step. There, all nodes of G are ordered by some notion of *importance* with more important nodes higher up in the hierarchy. Roughly, a node is more important, the more shortest paths run over it. The hierarchy is constructed bottom up by successively *contracting* the least important remaining node. Contracting a node v means, that v is removed from the graph while preserving all shortest paths. To preserve the shortest paths we have to insert an artificial *shortcut* edge (u, w) for every removed path $\langle u, v, w \rangle$ which is a unique shortest path at that time. If a shortcut (u, w) is inserted, we set $c(u, w) := c(u, v) + c(v, w)$ and annotate (u, w) with the *middle* node v such that (u, w) can be *expanded* to $\langle u, v, w \rangle$. When inserting (u, w) it may happen that an edge (u, w) is already present. In this case we *merge* the two edges. This means we check whether $c(u, w) > c(u, v) + c(v, w)$ holds, and if it does we replace the middle node by v and the weight by $c(u, v) + c(v, w)$.

Having contracted all nodes we have a hierarchy of graphs which we store in a condensed way: Every node is materialized exactly once and the original edges and all shortcuts are put together. The resulting graph is the actual contraction hierarchy (also abbreviated CH) H. We have $G \subseteq H$. Fig. 1 shows an example.

Corollary 7. *Let H be a CH derived from G. Then he shortest path distances in H and G are equal (but usually H contains paths not present in G).*

We say an edge (u, v) in H leads *upward* if u is less important than v. Otherwise, we say (u, v) leads *downward*. Let $H_\uparrow \subseteq H$ and $H_\downarrow \subseteq H$ be the subgraphs that only consist of upward and downward edges respectively. Then we have $H = H_\uparrow \cup H_\downarrow$ where $H_\uparrow$ and $H_\downarrow$ are edge disjoint DAGs. Most probably, a path $P := \langle s, \ldots, t \rangle$ in H contains shortcuts. These can be expanded recursively until

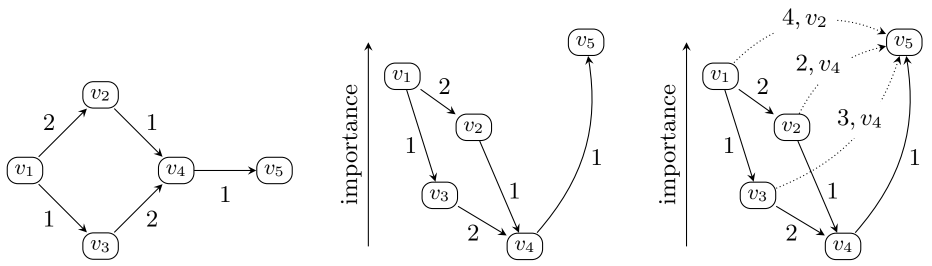


Fig. 1. Example road network (left) whose nodes are ordered by importance according to $v_4 \prec v_3 \prec v_2 \prec v_1 \prec v_5$ (middle). Doing preprocessing we get a CH (right) with three shortcut edges (dotted) annotated with their weight and the respective middle node. When contracting v_3 we insert no shortcut because there is more than one shortest path from v_1 to v_5 in the remaining network which still contains the nodes v_1, v_2, v_5.

no shortcuts are present in the resulting path $P' \subseteq G$. We say that P *represents* P'. A path $\langle s, \ldots, x, \ldots, t\rangle$ in H with $\langle s, \ldots, x\rangle \subseteq H_\uparrow$ and $\langle x, \ldots, t\rangle \subseteq H_\downarrow$ is called an *up-down-path* in H with *top* node x.

Lemma 8 ([14]). *Let H be a CH derived from G and s, t be two nodes. Then, there is an up-down-path in H that represents a shortest path from s to t in G.*

Up-down-paths being also shortest paths are called *shortest* up-down-paths. Shortest paths in G can only be represented by shortest up-down-paths in H. If there is exactly one shortest up-down-path in H representing a shortest path $P \subseteq G$, we say P is *uniquely representable (by a shortest up-down-path) in H.*

A shortest up-down-path from a node s to a node t can be found by performing a *bidirectional* Dijkstra search that runs *upward*. That is two Dijkstra searches that run at the same time, a *forward* and a *backward* search each starting from s and t and running in $H_\uparrow$ and $H_\downarrow^\top$ respectively.[2] This is exactly what Algorithm 4 does. Similar to the Algorithm 2 we maintain tentative cost, predecessor, and uniqueness information, but separately for forward and backward search: d_s and d_t, for example, denote the tentative cost of forward and backward search respectively. Whenever the two searches meet in a node u, we put u into the set C of top node *candidates*, but only if the weight of the corresponding up-down-path with top node u is minimal at that time (Line 10). After the bidirectional search is finished, we check whether the shortest of the up-down-paths that we found is unique (Lines 19 to 26). Algorithm 4 runs very fast because well-constructed CH are *flat* and *sparse*. This means H only contains few shortcuts and the paths in $H_\uparrow$ and $H_\downarrow^\top$ only have few hops. Note that we also apply *stall-on-demand* [13, 14] to further reduce running time.

Theorem 9. *A path $\langle s, \ldots, t\rangle \subseteq G$ is uniquely representable by a shortest up-down-path in H, if and only if Algorithm 4 returns true.*

[2] In reality we run the two searches in an alternating manner instead of simultaneously.

Algorithm 4. Modified bidirectional Dijkstra search checking whether $\langle s, \dots, t\rangle$ is uniquely representable by a shortest up-down-path in the CH $H = H_\uparrow \cup H_\downarrow$.

```
1  function isUniquelyRepresentable($P := \langle s, \dots, t\rangle$ : Path) : bool
2      $d_X[u] := \infty$, $p_X[u] := \bot$, $unique_X[u] := true$ for all $u \in V$, $X \in \{s, t\}$
3      $d_s[s] := d_t[t] := 0$
4      $M_s := \{(s, 0)\}, M_t := \{(t, 0)\}$ : PriorityQueue
5      $X := t$, $C := \emptyset$                          // search direction, candidate set
6      while $M_s \neq \emptyset$ or $M_t \neq \emptyset$ do
7          if $\min M_s \cup M_t > \min\{d_s[x] + d_t[x] \mid x \in C\} \cup \{\infty\}$ then break
8          if $M_{\neg X} \neq \emptyset$ then $X := \neg X$           // with $s = \neg t$ and $t = \neg s$
9          $u := M_X.deleteMin()$
10         if $d_s[u] + d_t[u] \leq \min\{d_s[x] + d_t[x] \mid x \in C\}$ then $C := C \cup \{u\}$
11         foreach edge $(u, v)$ in $H_X$ do                 // $H_s := H_\uparrow$, $H_t := H_\downarrow^\top$
12             if $d_X[u] + c(u, v) < d_X[v]$ then
13                 if $d_X[v] = \infty$ then $M_X.insert(v,\ d_X[u] + c(u, v))$
14                 else $M_X.updateKey(v,\ d_X[u] + c(u, v))$
15                 $d_X[v] := d_X[u] + c(u, v)$
16                 $p_X[v] := u$
17                 $unique_X[v] := true$
18             else if $d_X[u] + c(u, v) = d_X[v]$ then $unique_X[v] := false$
19     if there is exactly one $x \in C$ minimizing $d_s[x] + d_t[x]$ then
20         $x_0 := \operatorname{argmin}_{x \in C} d_s[x] + d_t[x]$
21         $P_s := \langle s, \dots, p_s[p_s[x_0]], p_s[x_0], x_0\rangle \subseteq H_\uparrow$
22         $P_t := \langle x_0, p_t[x_0], p_t[p_t[x_0]], \dots, t\rangle \subseteq H_\downarrow$
23         if concatenated up-down-path $P_s P_t$ not represents $P$ then return false
24         if there is $X \in \{s, t\}$, $w \in P_X$ s.t. $\neg unique_X[w]$ then return false
25         return true
26     return false
```

Proof. First note that both forward and backward search settle the top node of every shortest up-down-path from s to t adding it to C (Line 10). So, in the end C contains the top nodes of all shortest up-down-paths from s to t and $\min\{d_s[x] + d_t[x] \mid x \in C\}$ is the respective cost of these shortest up-down-paths.

Now, assume the algorithm returns *false*. Then, only the following reasons are possible: First, there is no up-down-path from s to t in H at all, or there are multiple shortest up-down-paths with different top nodes (Line 19 with the above statement). Second, P_s or P_t is not unique in $H_\uparrow$ or $H_\downarrow$ respectively (Line 24 with Lemma 3), so there are multiple shortest up-down-paths even if they have the same top node. Third, the concatenated up-down-path $P_s P_t$ does not represent P (Line 23) even it is the only shortest up-down-path from s to t in H.

Assume the algorithm returns *true*. We know that all shortest up-down-paths from s to t have the same top node (Line 19). Also, P_s and P_t are unique in $H_\uparrow$ and $H_\downarrow$ respectively (Line 24 with Lemma 3). So, $P_s P_t$ is the only shortest up-down-path from s to t in H and it represents P (Line 23). □

5 Compression Based on CH

If we instantiate *uniqueShortestPrefix* in the frame algorithm (Algorithm 1) with the binary scheme (Algorithm 3) and *isUniqueShortestPath* in the binary scheme with *isUniquelyRepresentable* (Algorithm 4), we get a very fast *CH-based method* to compute representations with via nodes.

As mentioned before, the resulting representations are no longer in terms of the original road network G but in terms of the CH H which has different properties. Consider a path $P := \langle u_1, \dots, u_n \rangle \subseteq G$ which is not necessarily a shortest path. Let $Q := \langle\!\langle u_{i_1}, \dots, u_{i_k} \rangle\!\rangle$ be a subsequence of P with the property that the subpaths $\langle u_1, \dots, u_{i_1} \rangle$, $\langle u_{i_k}, \dots, u_n \rangle$, and $\langle u_{i_j}, \dots, u_{i_{j+1}} \rangle$ of P with $1 \leq j < k$ are all uniquely representable by up-down-paths in H. Then, we call Q a *CH-based representation of P (with via nodes).* Note that the original road network G is not enough to reconstruct the path from a CH-based representation with via nodes. Instead, we have to compute unique shortest up-down-paths between the via nodes using bidirectional upward searches in the CH. This is due to Observation 10.

Observation 10. *Let H be a CH derived from G. Then, a not unique shortest path $P \subseteq G$ may still be uniquely representable by an up-down-path in H.*

To understand that take a look at Fig. 1. There, the CH contains exactly one shortest up-down-path from v_1 to v_5, namely $\langle v_1, v_5 \rangle$ which represents the shortest path $\langle v_1, v_2, v_4, v_5 \rangle$ in the original road network. However, this shortest path is not unique as the original road network also contains another shortest path from v_1 to v_5, namely $\langle v_1, v_3, v_4, v_5 \rangle$.

As a consequence of Observation 10 less via nodes may be needed by a representation in terms of CH than in terms of the original road network. Again, look at Fig. 1. The minimal representation with via nodes of the path $\langle v_1, v_2, v_4, v_5 \rangle$ in terms of the original road network is $\langle\!\langle v_2 \rangle\!\rangle$. The minimal CH-based representation is $\langle\!\langle \rangle\!\rangle$.

All subpaths of unique shortest paths in G are unique shortest paths themselves. In case of CHs, however, the analogous condition does not hold. Again, Fig. 1 shows an example: The path $\langle v_1, v_2, v_4, v_5 \rangle$ in the original network is uniquely representable by an up-down-path in H but its subpath $\langle v_1, v_2, v_4 \rangle$ is not uniquely representable as there are two up-down-paths from v_1 to v_4.

Observation 11. *Let H be a CH derived from G. Then, a shortest path $P \subseteq G$ may be uniquely representable in H, but one of its subpaths may be not.*

It is because of Observation 11 that the CH-based method does not necessary yield the minimal possible number of via nodes with respect to H. However, we are never worse than the minimal representation in terms of the original road network G. This is due to Lemma 12 as we show in the proof of Theorem 13.

Lemma 12. *Let H be a CH derived from G. Then, every unique shortest path $P \subseteq G$ is uniquely representable by an up-down-path in H.*

Theorem 13. *A CH-based representation computed by our binary CH-based method needs not more via nodes than a minimal representation with respect to the original road network.*

Proof. Let $P = \langle s, \dots, t \rangle \subseteq G$ be the given path. Let $Q_G := \langle\!\langle u_1, \dots, u_k \rangle\!\rangle$ be a minimal representation of P with respect to G. Assume our binary method finds a CH-based representation $Q_H := \langle\!\langle v_1, \dots, v_\ell \rangle\!\rangle$ with respect to H such that $\ell > k$. Then, we know that i, j exist such that the subpath $\langle v_j, \dots, v_{j+1} \rangle \subseteq P$ is also a subpath of $R := \langle u_i, \dots, u_{i+1} \rangle$ with $v_{j+1} \neq u_{i+1}$.[3] But all subpaths $\langle v_j, \dots, w \rangle \subseteq R$ are unique shortest paths in G and thus, by Lemma 12, uniquely representable by a shortest up-down-path in H. So, the binary scheme (Algorithm 3) instantiated with Algorithm 4 does not return $\langle v_j, \dots, v_{j+1} \rangle$ as resulting prefix path but a longer one – a contradiction. □

6 Experiments

Setup. As input we use a German road network provided by PTV AG for scientific use. It has 4.7 M nodes, 10.8 M edges, and 7.2 % time-dependent edge weights reflecting the travel times of midweek (Tuesday till Thursday) traffic collected from historical data – that is a high traffic scenario. For all edges (u, v) also the *driving distance* $dd(u, v)$ is available. The units of time and distance are 0.1 s and 1 m respectively. From this we obtain four different *metrics*, that is edge weights and objective functions defining different kinds of optimal routes. With these metrics we simulate the hybrid route planning scenario, where server-provided routes are not necessary optimal with respect the client's objective function.

In the *time-dependent* metric edge weights are time-dependent travel times. Optimal routes minimize the travel time depending on the departure time [2–4, 6]. In the *free flow* metric we also minimize travel time but there is no time-dependency. As weight of an edge (u, v) we use the minimum travel time $mtt(u, v)$ of the respective time-dependent edge weight. In the *distance* metric we simply use the driving distance $dd(u, v)$ as weight of an edge (u, v). Optimal routes are minimum distance routes. With the *energy* metric we optimize an approximation of energy consumption. As weight of an edge (u, v) we use $dd(u, v) + 4 \cdot mtt(u, v)$. With typical gasoline prices we assume that driving 1 km costs 0.1 €. This implies that travel time is prized with a rate of 14.4 € per hour.

To simulate the server, we compute optimal routes with respect to the metrics *time-dependent* and *distance*. To simulate possible objective functions of the client, we use the metrics *free flow*, *distance*, and *energy*. This leads to five combinations of *server metrics* and *client metrics*. For all three client metrics the road network contains edges which are not unique shortest paths. This means we have to transform the road network a little as described in Sect. 2. For the client metrics *free flow*, *distance*, and *energy* this increases the number of nodes by 2.37 %, 1.48 %, and 2.16 % respectively. The reported average numbers of nodes of the uncompressed paths refers to the non-transformed network.

[3] This can be shown by induction over k.

Table 1. Behavior of the Dijkstra-based and the binary CH-based compression for all five combinations of server and client metrics. Algor.= numbers of combined algorithms as used in this work, max.= maximum, rate= compression rate. All figures except for the maxima are average values.

# route nodes	client metric	method	Algor.	via nodes #	max.	rate[%]	time [ms]
server metric: time-dependent							
996	free flow	Dijkstra-based	1+2	0.071	3	0.006	1 500.78
		binary CH-based	1+3+4	0.068	3	0.006	0.36
	distance	Dijkstra-based	1+2	9.771	26	1.045	481.60
		binary CH-based	1+3+4	9.677	25	1.036	20.98
	energy	Dijkstra-based	1+2	1.103	6	0.125	1 326.17
		binary CH-based	1+3+4	1.094	6	0.124	1.72
server metric: distance							
1 763	free flow	Dijkstra-based	1+2	29.312	76	1.689	162.49
		binary CH-based	1+3+4	29.284	76	1.688	12.56
	energy	Dijkstra-based	1+2	24.902	69	1.434	182.87
		binary CH-based	1+3+4	24.876	69	1.433	15.63

The experimental evaluation was done on different 64 bit machines with Ubuntu Linux 10.04. The running times have been measured on a machine with 8 GiB main memory and a Core i5 Double-Core CPU at 3.33 GHz. There, all programs were compiled using GCC 4.4.3 with optimization level 3. We evaluate the performance of our compression algorithms in terms of running time, number of via nodes, and compression rate. The compression rate is defined as the number of via nodes divided by the number of nodes of the uncompressed path.

Results. To generate "server provided" routes, we randomly select 1 000 pairs of start and destination nodes computing the optimal routes with respect to both server metrics. For the metric *time-dependent* we also select random departure times from $[0, 24\text{h})$. Table 1 shows the resulting performance of the Dijkstra-based and the binary CH-based compression. The average compression rate is never worse than 1.7 % which means that 29 via nodes are needed to represent a path with 1 763 nodes. The maximum number of via nodes is 76. The number of via nodes gets larger if server and client metrics are less correlated. The compression rate achieved by the CH-based method is only sightly better than for the Dijkstra-based method. However, the minimum number of via nodes possible with CH is unknown. And unfortunately our implementation of Algorithm 4 is a bit pessimistic and potentially rejects some uniquely representable subpaths.

The binary CH-based method runs much faster than the Dijkstra-based one. With average compression times below 21 ms it is fast enough for high throughput servers. Previous experiments with mobile CH [15] suggest that a client needs clearly less than 0.1 s to decompress each segment of a compressed route. So, for our German road network compression and decompression should not raise any noticeable latency – remember that it is enough to decompress the *next* segment fast. It is not surprising that the CH-based method runs faster if the number of via nodes is very small as *isUniqueShortestPath* is invoked $O(|Q|\log|P|)$ times. It looks surprising, however, that the compression runs faster with 29.3 than with 9.7 nodes. A possible explanation is that different metrics entail different distributions of via nodes as well as different numbers of shortcuts in the CH. Both can influence the compression time of a single segment.

The running time of the Dijkstra-based method behaves contrary to the CH-based one: It runs faster the more via nodes we need. Whenever Algorithm 2 finds a maximum unique shortest prefix, it can be stopped (Line 18). But Dijkstra's algorithm has roughly quadratic running time on road networks. So, if we stop it more early but invoke it more often, the overall running time decreases.

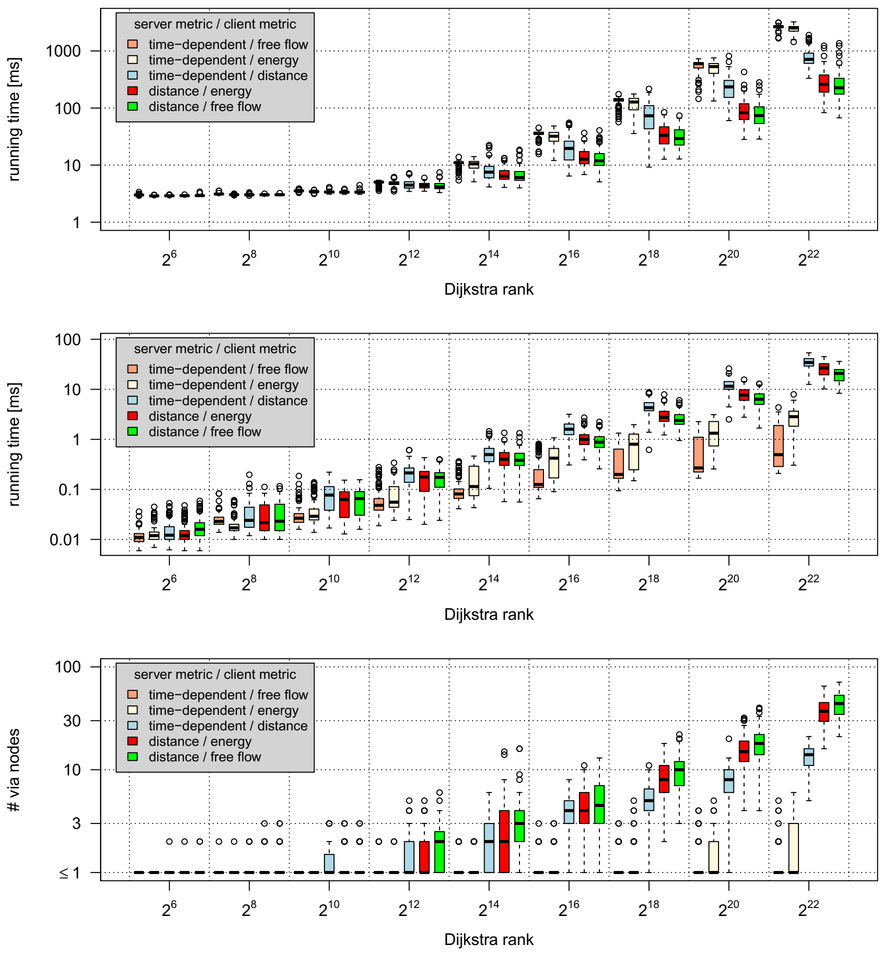


Fig. 2. The compression time of the Dijkstra-based (top) and the CH-based method (middle) plotted over the Dijkstra rank for all five combinations of server and client metrics. The number of via nodes computed by the CH-based method is also plotted (bottom). There are 100 compressed routes per rank and combination.

Fig. 2 shows the compression time of both methods as well as the number of via nodes plotted over Dijkstra rank[4]. For all combinations of server and client metrics the compression time of both methods as well as the number of via nodes increases with the Dijkstra rank.

7 Conclusions and Future Work

We describe an algorithmic framework for convenient hybrid route planning. Routes computed by servers can be transmitted to client devices in cars efficiently, even when the bandwidth is low. To do so routes are represented as sequences of only a few via nodes. These are the connection points when routes are decomposed into unique shortest subpaths with respect to the clients objective function. Utilizing CH we achieve very good performance: On a German road network an average compression takes less than 21 ms and yields less than 30 via nodes. The maximum number of via nodes we observe is 76. Using mobile CH the client can decompress the first subpath of the route in less than 0.1 s as previous experiments suggest [15]. The following subpaths can be decompressed one after another during driving. So, except for the time needed by the mobile communication the driver will most likely not experience any latency. Note that the low number of via nodes also helps to keep the communication time small.

We also describe a Dijkstra-based method that runs much slower than the CH-based one. But applying Arc-Flags [19] or ALT [20], two algorithmic techniques for fast and exact route planning, may bring a substantial speedup there. An interesting question is, whether subpaths that are uniquely representable with CH can be computed "directly", that is without repeated bidirectional Dijkstra searches. This could further speedup the CH-based compression. Finally, it should be noted that we could also use via *edges* instead of via nodes.

References

1. Delling, D., Wagner, D.: Time-Dependent Route Planning. In: Ahuja, R.K., Möhring, R.H., Zaroliagis, C.D. (eds.) Robust and Online Large-Scale Optimization. LNCS, vol. 5868, pp. 207–230. Springer, Heidelberg (2009)
2. Batz, G.V., Delling, D., Sanders, P., Vetter, C.: Time-Dependent Contraction Hierarchies. In: Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX 2009), pp. 97–105. SIAM (April 2009)
3. Batz, G.V., Geisberger, R., Neubauer, S., Sanders, P.: Time-Dependent Contraction Hierarchies and Approximation. In: [21], pp. 166–177
4. Kieritz, T., Luxen, D., Sanders, P., Vetter, C.: Distributed Time-Dependent Contraction Hierarchies. In: [21] pp. 83–93
5. Batz, G.V., Sanders, P.: Time-Dependent Route Planning with Generalized Objective Functions. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 169–180. Springer, Heidelberg (2012)

[4] For $i = 6..22$ we each select 100 random queries such that the time-dependent variant of Dijkstra's algorithm settles 2^i nodes. We call 2^i the *Dijkstra rank*.

6. Delling, D.: Time-Dependent SHARC-Routing. Algorithmica 60(1), 60–94 (2011), Special Issue: European Symposium on Algorithms (2008)
7. Brunel, E., Delling, D., Gemsa, A., Wagner, D.: Space-Efficient SHARC-Routing. In: [21], pp. 47–58
8. Geisberger, R., Kobitzsch, M., Sanders, P.: Route Planning with Flexible Objective Functions. In: Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX 2010), pp. 124–137. SIAM (2010)
9. Delling, D., Wagner, D.: Pareto Paths with SHARC. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 125–136. Springer, Heidelberg (2009)
10. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable Route Planning. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 376–387. Springer, Heidelberg (2011)
11. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Alternative Routes in Road Networks. In: [21], pp. 23–34
12. Luxen, D., Schieferdecker, D.: Candidate Sets for Alternative Routes in Road Networks. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 260–270. Springer, Heidelberg (2012)
13. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
14. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact Routing in Large Road Networks Using Contraction Hierarchies. Transportation Science (accepted for publication, 2012)
15. Sanders, P., Schultes, D., Vetter, C.: Mobile Route Planning. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 732–743. Springer, Heidelberg (2008)
16. Batz, G.V., Geisberger, R., Luxen, D., Sanders, P.: Compressed Transmission of Route Descriptions. Technical report, Karlsruhe Institute of Technology (KIT) (2010), arXiv:1011.4465v1
17. Tao, Y., Sheng, C., Pei, J.: On k-skip shortest paths. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, pp. 421–432. ACM, New York (2011)
18. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox. Springer (2008)
19. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning Graphs to Speed Up Dijkstra's Algorithm. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 189–202. Springer, Heidelberg (2005)
20. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A* Search Meets Graph Theory. In: Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA 2005), pp. 156–165. SIAM (2005)
21. Festa, P. (ed.): SEA 2010. LNCS, vol. 6049. Springer, Heidelberg (2010)

Multipath Spanners via Fault-Tolerant Spanners

Shiri Chechik[1,⋆], Quentin Godfroy[2,⋆⋆], and David Peleg[3,⋆]

[1] Microsoft Research, Silicon Valley Center, USA
shiri.chechik@gmail.com
[2] LaBRI, Université Bordeaux-I, 351 cours de la Libération, Talence, France
quentin@godfroy.eu
[3] Department of Computer Science, The Weizmann Institute, Rehovot, Israel
david.peleg@weizmann.ac.il

Abstract. An s-spanner H of a graph G is a subgraph such that the distance between any two vertices u and v in H is greater by at most a multiplicative factor s than the distance in G. In this paper, we focus on an extension of the concept of spanners to *p-multipath distance*, defined as the smallest length of a collection of p pairwise (vertex or edge) disjoint paths. The notion of multipath spanners was introduced in [15, 16] for edge (respectively, vertex) disjoint paths. This paper significantly improves the stretch-size tradeoff result of the two previous papers, using the related concept of fault-tolerant s-spanners, introduced in [6] for general graphs. More precisely, we show that at the cost of increasing the number of edges by a polynomial factor in p and s, it is possible to obtain an s-multipath spanner, thereby improving on the large stretch obtained in [15, 16].

1 Introduction

Consider a graph $G = (V, E)$. An s-spanner H of a graph G is a spanning subgraph that preserves distances between all pairs of nodes within a factor of s, namely, such that $\delta(u, v, H) \leq s \cdot \delta(u, v, G)$ for every two nodes $u, v \in V$, where $\delta(u, v, G')$ for a graph G' is the distance from u to v in G'.

Graph spanners were introduced in [21, 22] in the context of distributed networks. It is well-known how to efficiently construct an s-spanner of size $O(n^{1+\frac{2}{s+1}})$ [1], for an odd integer s. Spanners have numerous applications, such as synchronizers [22], efficient routing [7, 8, 22, 25, 26], broadcasting [14], near-shortest path algorithms [11–13], covers [2], dominating sets [10], distance oracles [4, 27], or emulators and distance preservers [5]. For recent reviews on spanners see [23, 28].

This paper considers spanners for the *multipath* graph length. A *p-vertex (resp., p-edge) multipath* between two vertices u and v is a subgraph consisting

⋆ Supported in part by the Israel Science Foundation (grant 894/09), the United States-Israel Binational Science Foundation (grant 2008348), the Israel Ministry of Science and Technology (infrastructures grant), and the Citi Foundation.
⋆⋆ Supported by the Institut Universitaire de France.

of the union of p pairwise vertex-disjoint (resp., edge-disjoint) paths between u and v (except for the endpoints). The cost of a p-vertex (resp., p-edge) multipath between u and v is the sum of the weights of its edges.

For a weighted graph G and two vertices u and v, let $\delta_V^p(u,v,G)$ (resp., $\delta_E^p(u,v,G)$) be the minimum cost of a p-vertex (resp., p-edge) multipath in G between u and v if one exists, and ∞ otherwise. We say that a subgraph H is a *p-vertex (resp., p-edge) multipath s-spanner* of G if $\delta_V^p(u,v,H) \leq s \cdot \delta_V^p(u,v,G)$ (resp., $\delta_E^p(u,v,H) \leq s \cdot \delta_E^p(u,v,G)$) for every two vertices u and v such that $\delta_V^p(u,v,G) \neq \infty$ (resp., $\delta_E^p(u,v,G) \neq \infty$).

Our interest in the disjoint multipath graph length stems from the need for multipath routing in networks. Using multiple paths between a pair of nodes is an obvious way to aggregate bandwidth. Additionally, a classical approach to quickly overcome link failures consists in pre-computing alternate paths which are disjoint from the primary paths [18, 24, 20]. Multipath routing can be used for traffic load balancing and for minimizing delays. It has been extensively studied in ad hoc networks for load balancing, fault-tolerance, higher aggregate bandwidth, diversity coding, minimizing energy consumption (see [19] for a quick overview). Considering only a subset of links is a practical concern in link state routing in ad hoc networks [17]. This raises the problem of computing spanners for the multipath graph length, a first step towards constructing compact multipath routing schemes.

Edge-disjoint multipath spanners were first introduced in [15], where the authors provided construction examples. More specifically, they showed how to construct a p-edge multipath spanner with $O(pn^{1+\frac{2}{s+1}})$ edges and stretch $s \cdot p$. It was also shown that the factor p in the stretch can be discarded by an ad-hoc construction for $p = 2$ and $s = 3$. In [16] the authors proved that for every weighted graph one can efficiently construct a p-vertex multipath spanner with $O(p^2 n^{1+\frac{2}{s+1}})$ edges, but with a large stretch for large values of p (about $(1 + p/s)^s$).

In this paper, we significantly improve the stretch bounds from the two previous papers, using the related concept of fault-tolerant s-spanners, introduced in [6] for general graphs. We show that the constructions of edge-fault tolerant s-spanners from [6] and vertex-fault tolerant s-spanners from [9] can yield p-multipath s-spanners, by fixing the right number of faults. Specifically, for edge multipath spanners we reduce the stretch from $s \cdot p$ to s, thus deriving edge multipath spanners with stretch-size tradeoff close to the best known bounds for standard spanners. For vertex multipath spanners we reduce the stretch bound from a function of both s and p to just s in the case of unweighted graphs, and to $s \cdot p$ in the case of weighted graphs, s being the stretch of the underlying fault-tolerant spanner.

More precisely, we show the following theorems.

Theorem 1. *Given a weighted graph $G = (V,E)$ with maximal edge weight $\hat{\omega}$ and minimal edge weight 1, odd integer s and integer p, one can efficiently construct a p-edge-multipath s-spanner with $O\left(sp^2 \cdot \log \hat{\omega} \cdot n^{1+\frac{2}{s+1}}\right)$ edges.*

Theorem 2. *Given an unweighted graph $G = (V, E)$, odd integer s and integer p, one can efficiently construct a p-vertex-multipath s-spanner with $O\left(\left(s^{6\alpha}p^{2\alpha} + (s \cdot p)^{4\alpha}\right) n^{1+\frac{2}{s+1}} \log n\right)$ edges, where $\alpha = 1 - \frac{1}{s+1}$.*

Theorem 3. *Given a weighted graph $G = (V, E)$, odd integer s and integer p, one can efficiently construct a p-vertex-multipath $(s \cdot p)$-spanner with $O\left((p \cdot s)^{2\alpha} \cdot n^{1+\frac{2}{s+1}} \log n\right)$ edges, where $\alpha = 1 - \frac{1}{s+1}$.*

2 Preliminaries

For a graph H and two vertices u and v, let $\delta(u, v, H)$ denote the distance between u and v in H. A subgraph H of G is a *spanner* of *stretch* s (or, an s-spanner) of G if $\delta(u, v, H) \leq s \cdot \delta(u, v, G)$ for every $u, v \in V$. For a path P and two vertices x and y on it, let $P[x, y]$ denote the subpath of P from x to y.

A graph H is an *r-vertex (resp., edge) fault-tolerant s-spanner* of G if for any set $F \subseteq V$ (resp., $F \subseteq E$) of size at most r, the subgraph $H \setminus F$ is an s-spanner of $G \setminus F$, where $H' \setminus F'$ for a subgraph H' and set of vertices F' is the graph obtained by removing all vertices F' from H' together with their edges. Similarly $H' \setminus F'$ for a subgraph $H' = (V', E')$ and set of edges F' is the graph $(V', E' \setminus F')$.

For a subgraph H, let $\mathbf{cost}(H)$ denote the sum of weights of the edges in H.

For a path P, let $V(P)$ be the set of vertices on P and let $E(P)$ be the set of edges on P. Similarly for a set of paths $\mathcal{S}$, let $V(\mathcal{S}) = \bigcup_{P \in \mathcal{S}} V(P)$ and $E(\mathcal{S}) = \bigcup_{P \in \mathcal{S}} E(P)$. Consider two paths P_1 and P_2 between the same two vertices x and y. We say that P_1 and P_2 are *internal vertex-disjoint* if $V(P_1) \cap V(P_2) = \{x, y\}$. We say that a path is an *s-path* if it is of length at most s.

3 Edge Disjoint Multipath Spanners

In this section we show that every edge fault-tolerant spanner is an edge multipath spanner with the same stretch, by fixing the right number of faults.

We begin by considering the unweighted case, and show later how to proceed with weighted graphs.

A set of paths $\mathcal{B}$ is called a set of *edge-disjoint-s-bypasses* of u and v if the paths in $\mathcal{B}$ are edge-disjoint s-paths between u and v.

Consider two vertices x and y and let $\mathcal{S} = \{Q_1, ..., Q_p\}$ be the set of p vertex-disjoint paths from x to y in G of minimal cost. Consider an edge $e = (u, v)$ such that $e \in E(\mathcal{S})$. The next lemma shows that G does not contain "many" edge-disjoint-s-bypasses, namely, edge disjoint s-paths between u and v, that intersect with the other paths of $\mathcal{S}$.

Lemma 1. *Consider two vertices x and y and let $\mathcal{S} = \{Q_1, ..., Q_p\}$ be the set of p edge-disjoint paths from x to y in G of minimal cost. Consider an edge $e = (u, v) \in Q_i$ for some $1 \leq i \leq p$. There are at most $2sp + 2p$ edge-disjoint-s-bypasses between u to v, that intersect with $E(\mathcal{S}) \setminus E(Q_i)$.*

Proof: Assume towards contradiction that there are more than $2sp + 2p$ edge-disjoint-s-bypasses between u and v that intersect with $E(\mathcal{S}) \setminus E(Q_i)$. Let $\mathcal{B}$ be the set of all these edge-disjoint-s-bypasses that intersect with $E(\mathcal{S}) \setminus E(Q_i)$. Let J be the set of indices j such that $1 \leq j \leq p$, $j \neq i$ and $E(Q_j) \cap E(\mathcal{B}) \neq \emptyset$. For an index $j \in J$, let $e_h^j = (u_h^j, v_h^j)$ be the edge in $E(Q_j) \cap E(\mathcal{B})$ closest to x in Q_j, and let $e_l^j = (u_l^j, v_l^j)$ be the edge in $E(Q_j) \cap E(\mathcal{B})$ closest to y in Q_j. Let $M_j = [u_h^j, v_l^j]$ for $j \in J$ and let $M = \bigcup_{j \in J} M_j$. We show that it is possible to replace the set of edges $E(M)$ with a set of edges E' such that the resulting graph contains p edge disjoint paths from x to y and its cost is less than $\mathbf{cost}(\mathcal{S})$ and thus derive a contradiction to the optimality of $\mathcal{S}$.

We now explain how to gradually build the set of edges E'. Roughly speaking, the set of edges E' is the union of some prefixes and suffixes of the paths M_j together with some edge-disjoint-s-bypasses from $\mathcal{B}$.

Let $p_j = e_h^j$ (resp., $s_j = e_l^j$) be the prefix (resp., suffix) of M_j. Our construction process will gradually add edges to these prefixes and suffixes.

Let $\text{tip}(p_j)$ (resp., $\text{tip}(s_j)$) be the edge in $E(p_j) \cap E(\mathcal{B})$ (resp., $E(s_j) \cap E(\mathcal{B})$) closest to y on Q_j (resp., to x on Q_j). We examine the set of tips of the prefixes and suffixes, and the set of edge-disjoint-s-bypasses $\mathcal{B}'$ in which they appear. Let X be the set of prefixes and suffixes, namely, $X = \bigcup_{j \in J} \{p_j, s_j\}$. The set $\mathcal{B}'$ is the set of edge-disjoint-s-bypasses containing one of the edges $\{\text{tip}(p_j), \text{tip}(s_j) \mid j \in J\}$. For an edge $e \in E(\mathcal{B}')$, let $B(e)$ be the edge-disjoint-s-bypass such that $e \in E(B)$. Note that there is exactly one such edge-disjoint-s-bypass since the edge-disjoint-s-bypasses are disjoint. We say that a path $P \in X$ is *clean* if the sub-path $B(\text{tip}(e))[\text{tip}(P), u]$ does not contain other edges from $E(X)$. For an edge-disjoint-s-bypass $B \in \mathcal{B}'$, let $P_{clean}(B)$ be the path $P \in X$ such that $\text{tip}(P) \in E(B)$ and P is clean; note that there is exactly one such path.

We say that a prefix $p_j \in X$ (resp., suffix s_j) is *complete* if $p_j \circ s_j = M_j$. We apply the following process until all paths in X are clean. Choose an unclean incomplete path and add edges to it until it becomes clean. By adding an edge to a prefix p_j (resp., suffix s_j) we mean adding the edge on M_j adjacent to $\text{tip}(p_j)$ (resp., $\text{tip}(s_j)$) closest to y (resp., to x).

Note that it could happen that during this process some clean path becomes unclean. Note also that edge-disjoint-s-bypasses are only added to $\mathcal{B}'$ (but never removed). Namely, $\mathcal{B}'(t_1) \subseteq \mathcal{B}'(t_2)$ for $t_1 \leq t_2$, where $\mathcal{B}'(t)$ is the set $\mathcal{B}'$ in the t'th step of this process. To see this, note that the process does not add edges to $P_{clean}(B)$ for any $B \in \mathcal{B}'$. Thus in any stage of this process, B contains $\text{tip}(P)$ for $P = P_{clean}(B)$. Hence, by definition, $B \in \mathcal{B}'$. Notice that the path $P_{clean}(B)$ itself may change (since the process might add an edge to another path and this edge could belong to the path from $\text{tip}(P)$ to u).

We claim that $\mathcal{B}'$ contains at most $2p$ edge-disjoint-s-bypasses. This follows directly from the fact that each edge-disjoint-s-bypass $B \in \mathcal{B}'$ contains a different path $P_{clean}(B)$ and that there are $2p$ paths in X.

We now show that it possible to substitute the paths in M with "cheaper" paths and thus derive a contradiction to the optimality of $\mathcal{S}$. For every incomplete prefix $p_j \in X$, let p'_j be the clean subpath $B(p_j)[\text{tip}(p_j), u]$. Similarly, let s'_j be the clean subpath $B(s_j)[u, \mathrm{tip}(s_j)]$. For every index $j \in J$, if p_j is complete then set $Q'_j = Q_j$, otherwise set $Q'_j = Q_j[x, u^j_h] \circ p_j \circ p'_j \circ s'_j \circ s_j \circ Q_j[v^j_l, y]$.

Let $D' = \bigcup_{j \in J} E(p'_j \circ s'_j)$ and $D = \bigcup_{j \neq i} E(M_j) \setminus (E(p_j) \cup E(s_j))$. Let $\mathcal{S}' = \{Q'_1, ..., Q'_p\}$. Note that $|E(\mathcal{S}')| = |E(\mathcal{S})| + |D'| - |D|$. It is not hard to verify that the paths Q'_j are disjoint and each of them leads from x to y. Moreover, M intersects with at least $2sp + 2p + 1$ edge-disjoint-s-bypasses, and the set $E(p_j) \cup E(s_j)$ intersects with at most $2p$ edge-disjoint-s-bypasses, thus $|D| > 2sp + 1$. In addition, $|D'| \le 2sp$. We get that $|D| > |D'|$ and thus $\mathbf{cost}(\mathcal{S}') < \mathbf{cost}(\mathcal{S})$, contradiction. ∎

We now show that edge fault-tolerant spanners constructed by the algorithm of [6] are also edge multipath spanners with the same stretch, by fixing the right number of faults. For completeness, we outline the algorithm and its properties that are essential for our needs. The algorithm operates in q iterations. Initially, set $H = (V, \emptyset)$. In each iteration i, consider the graph $G_i = (V, E \setminus E(H))$, and construct an s-spanner H_i for this graph, (say, using the algorithm of [1]), and add the edges of H_i to the subgraph H.

We summarize the required properties for our purposes in the following lemma.

Lemma 2. *[6] For every graph $G = (V, E)$, odd integer s and integer q, one can efficiently construct in polynomial time a collection of edge disjoint subgraphs $\{H_1, ..., H_q\}$ with the following properties. Let H be the union of the subgraphs $\{H_1, ..., H_q\}$.*

(1) The number of edges in H is at most $O(q \cdot n^{1+\frac{2}{s+1}})$.
(2) For every edge $e = (u, v) \in E$, either $e \in E(H)$ or each H_i contains a path from u to v of length at most s.

Theorem 4. *Given a graph $G = (V, E)$, odd integer s and integer p, one can efficiently construct a p-multipath s-spanner with $O\left(sp^2 \cdot n^{1+\frac{2}{s+1}}\right)$ edges.*

Proof: Construct the collection of subgraphs $\{H_1, ..., H_{2sp^2+2p^2+p}\}$ of Lemma 2 with parameters s and $q = 2sp^2 + 2p^2 + p$. Let H be the union of all subgraphs $\{H_1, ..., H_{2sp^2+2p^2+p}\}$. Consider two vertices x and y and let $\mathcal{S} = \{Q_1, ..., Q_p\}$ be the set of p edge-disjoint paths from x to y in G of minimal cost. We now show how to find a set of edge-disjoint paths $\mathcal{S}' = \{Q'_1, ..., Q'_p\}$ from x to y such that $E(\mathcal{S}') \subseteq E(H)$ and $\mathbf{cost}(Q'_i) \le s \cdot \mathbf{cost}(Q_i)$. Let $T_i = \{H_j \mid (2sp + 2p + 1) \cdot (i - 1) + 1 \le j \le (2sp + 2p + 1) \cdot i\}$ for $1 \le i \le p$. Note that $E(T_i)$ contains $2sp + 2p + 1$ edge-disjoint paths from u to v for every edge $(u, v) \notin E(H)$. Moreover, the sets $E(T_i)$ are disjoint for $1 \le i \le p$.

The path Q'_i is constructed as follows. For every edge $e \in E(Q_i) \cap E(H)$ add e to Q'_i. For every edge $e = (u, v) \in E(Q_i) \setminus E(H)$, consider the set $\mathcal{B}_i$ with the maximum number of edge-disjoint-s-bypasses from u to v in T_i. By Lemma 1, there are at most $2sp + 2p$ edge-disjoint-s-bypasses in $\mathcal{B}_i$ that intersect with $E(\mathcal{S}) \setminus E(Q_i)$. Since $\mathcal{B}_i$ contains at least $2sp + 2p + 1$ edge-disjoint-s-bypasses, at least one edge-disjoint-s-bypass $B(e) \in \mathcal{B}_i$ does not intersect with $E(\mathcal{S}) \setminus E(Q_i)$. Add $B(e)$ to Q'_i instead of e.

We claim that (1) the paths Q'_i for $1 \leq i \leq p$ are edge-disjoint, and (2) $\mathbf{cost}(\mathcal{S}') \leq s \cdot \mathbf{cost}(\mathcal{S})$.

To see claim (1), consider an edge $e = (u, v)$ such that $e \in E(Q'_i)$ for some $1 \leq i \leq p$. We consider two cases. The first case is when $e \in E(Q_i)$. Note that e does not appear in any $E(Q_j)$ for $i \neq j$, since the paths in $\mathcal{S}$ are disjoint. Moreover, e does not appear in any $B(e')$ for $e' \in E(Q_j)$ for some $j \neq i$. To see this, recall that $B(e')$ does not intersect with $E(\mathcal{S}) \setminus E(Q_j)$. The second case is when $e \in B(\tilde{e})$ for some $\tilde{e} \in E(Q_i)$. The edge-disjoint-s-bypass $B(\tilde{e})$ does not intersect with $E(\mathcal{S}) \setminus E(Q_i)$. Moreover, the edge-disjoint-s-bypass $B(\tilde{e})$ does not intersect with any $B(e')$ for $e' \in E(Q_j)$ for some $j \neq i$. To see this, recall that $E(B(\tilde{e})) \subseteq E(T_i)$, $E(B(e')) \subseteq E(T_j)$, and $E(T_i) \cap E(T_j) = \emptyset$. It follows that the paths Q'_i are edge-disjoint for $1 \leq i \leq p$.

To see claim (2), note that for every edge $e \in E(Q_i)$, either e itself or an alternative path of length s is added to $E(Q'_i)$. We get that $\mathbf{cost}(Q'_i) \leq s \cdot \mathbf{cost}(Q_i)$. Claim (2) follows. ∎

Weighted Graphs. We now show the modifications needed for weighted graphs. Assume the minimal edge weight is 1 and let $\hat{\omega}$ be the maximal edge weight. We now describe the algorithm for constructing p-edge multipath s-spanner. Initially, set $H = (V, \emptyset)$. Consider the graphs $G_i = (V, E_i)$ such that $E_i = \{e \in E \mid 2^{i-1} \leq \omega(e) \leq 2^i\}$ for every $1 \leq i \leq \lceil \log \hat{\omega} \rceil$. Construct the collection of subgraphs $F_i = \{H_1, ..., H_{4sp^2+2p^2+p}\}$ of Lemma 2 for parameters s and $q = 4sp^2 + 2p^2 + p$ on the graph G_i. Add $E(F_i)$ to H.

We claim that H is a p-edge multipath s-spanner. The analysis is very similar to the unweighted case. We now outline the slight changes.

Here we call a set of paths $\mathcal{B}$ a set of *edge-disjoint-s-bypasses* of two nodes u and v that are connected by an edge if the paths in $\mathcal{B}$ are edge-disjoint paths between u and v of length at most $s \cdot \omega(u, v)$ each.

Consider two vertices x and y and let $\mathcal{S} = \{Q_1, ..., Q_p\}$ be the set of p edge-disjoint paths from x to y in G of minimal cost. Consider an edge $e = (u, v) \in Q_i$ for some $1 \leq i \leq p$. Let i be the index such $2^{i-1} \leq \omega(e) \leq 2^i$. In Lemma 1 we prove that the graph $(V, E(F_i))$ contains at most $4sp + 4p$ edge-disjoint-s-bypasses $\mathcal{B}$ from u to v. Note that since the weight of the edges in the edge-disjoint-s-bypasses $\mathcal{B}$ could be half the weight $\omega(e)$, we double the factor of sp^2 in the number of edge-disjoint-s-bypasses. The rest of the proof of Lemma 1 is similar to the unweighted case.

The proof of Theorem 4 is also similar to the unweighted case, where for each edge $e = (u, v) \in Q_j$ for some $1 \leq j \leq p$ such that $e \notin E(H)$, we pick an edge-disjoint-s-bypass from F_i that does not intersect $E(\mathcal{S}) \setminus E(Q_j)$, for i such that $2^{i-1} \leq \omega(e) \leq 2^i$.

We thus conclude with Theorem 1.

4 Vertex Disjoint Multipath Spanners

In this section we show that every vertex fault-tolerant s-spanner is a vertex multipath spanner with the same stretch, by fixing the right number of faults.

Note that it is unclear how to generalize the analysis from the previous section to vertex disjoint multipath spanners. To see this, recall that in the previous section we consider a set $\mathcal{S} = \{Q_1, ..., Q_p\}$ of p vertex-disjoint paths from x to y in G of minimal cost. We claim that every edge $e \in E(Q_i)$ does not contain too many bypasses that intersect with the other paths of $\mathcal{S}$. To prove this claim, we substitute sub-paths of each Q_j with some E-bypasses of the edge $e = (u, v)$. All these E-bypasses are edge disjoint but not vertex-disjoint. Specifically, all these E-bypasses contain the nodes u and v. Therefore, it is unclear how to use these E-bypasses to substitute multiple sub-paths and stay with vertex disjoint paths. We thus present a different analysis for vertex disjoint multipath spanners at the price of slightly increasing the size of the spanner. Moreover, our analysis for vertex disjoint multipath spanners works only for unweighted graphs. We later show a simple construction for weighted vertex multipath spanners with stretch sp (instead of s).

A subgraph H is *q-vertex-resilient with stretch s* if for every edge $e = (x, y) \in E$, either $e \in E(H)$ or H has at least q internal vertex-disjoint s-paths between x and y. For a path P between two nodes x and y and a vertex $v \in V(P)$, let **index**(v, P) be the distance (number of hops) between x and v in P. Two paths are said to intersect if they have at least one common vertex.

A set of paths $\mathcal{B}$ is called a set of *vertex-disjoint-s-bypasses* of u and v if the paths in $\mathcal{B}$ are internal vertex-disjoint s-paths between u and v. The next lemma shows that every vertex fault-tolerant s-spanner H has "many" vertex-disjoint-s-bypasses between u and v for every edge $e = (u, v)$ in $E \setminus E(H)$.

Lemma 3. *Every r-fault tolerant s-spanner is $\lfloor r/(s-1) \rfloor$-vertex-resilient with stretch s.*

Proof: Consider an r-fault tolerant s-spanner H. Consider an edge $e = (u, v) \in E \setminus E(H)$. We need to show that H contains $\lfloor r/(s-1) \rfloor$ vertex-disjoint-s-bypasses. Assume towards contradiction that H contains only k vertex-disjoint-s-bypasses between u and v such that $k < \lfloor r/(s-1) \rfloor$. Let $\mathcal{B}$ be the set of these k vertex-disjoint-s-bypasses. Note that $V(\mathcal{B}) \setminus \{u, v\}$ contains at most $(s-1) \cdot k < r$ vertices. Fix the set of vertices $F = V(\mathcal{B}) \setminus \{u, v\}$ to be faulty. Since the subgraph H is an r-fault tolerant s-spanner, by definition $H \setminus F$ contains an s-path between u and v. Therefore H contains more than k vertex-disjoint-s-bypasses between u and v, contradiction. ∎

Throughout, we consider a graph $G(V, E)$. Consider two vertices x and y and let $\mathcal{S} = \{Q_1, ..., Q_p\}$ be a set of p vertex-disjoint paths from x to y in G of minimal cost. Consider an edge $e = (u, v)$ in one of the paths of $\mathcal{S}$. The next lemma shows that G does not contain "many" vertex-disjoint-s-bypasses, namely, internal vertex disjoint s-paths between u and v, that intersect with the other paths of $\mathcal{S}$.

Lemma 4. *Consider two vertices x and y and let $\mathcal{S} = \{Q_1, ..., Q_p\}$ be a set of p vertex-disjoint paths from x to y in G of minimal cost. Consider an edge $e = (u, v) \in Q_i$ and $\notin H$ for some $1 \leq i \leq p$. There are at most $2sp(p-1) + 2p(p-1)$ vertex-disjoint-s-bypasses between u and v that intersect with $V(\mathcal{S}) \setminus V(Q_i)$.*

Proof: Assume towards contradiction that there are more than $2sp(p-1)+2p(p-1)$ such vertex-disjoint-s-bypasses that intersect $V(\mathcal{S}) \setminus V(Q_i)$. By the Pigeonhole principle, there exists a path Q_j for some $j \neq i$ such that at least $2sp+2p+1$ of these vertex-disjoint-s-bypasses intersect with Q_j. Let $\mathcal{B}$ be the set of all these vertex-disjoint-s-bypasses that intersect Q_j. For every vertex-disjoint-s-bypass $A \in \mathcal{B}$, let $\mathbf{top}(A)$ be the earliest vertex of A on Q_j, i.e., the vertex in $V(A) \cap V(Q_j)$ with minimal $\mathbf{index}(\mathbf{top}(A), Q_j)$, and let $\mathbf{bottom}(A)$ be the last vertex of A on Q_j, i.e., the vertex in $V(A) \cap V(Q_j)$ with maximal $\mathbf{index}(\mathbf{top}(A), Q_j)$. Let $\mathcal{B}_h$ be the set of p vertex-disjoint-s-bypasses $A \in \mathcal{B}$ with minimal $\mathbf{index}(\mathbf{top}(A), Q_j)$ and let $\mathcal{B}_l$ be the set of p vertex-disjoint-s-bypasses $A \in \mathcal{B}$ with maximal $\mathbf{index}(\mathbf{bottom}(A), Q_j)$. Let $A_h \in \mathcal{B}_h$ be the vertex-disjoint-s-bypass with maximal $\mathbf{index}(\mathbf{top}(A_h), Q_j)$ and let $q_h = \mathbf{top}(A_h)$. Let $A_l \in \mathcal{B}_l$ be the vertex-disjoint-s-bypass with minimal $\mathbf{index}(\mathbf{bottom}(A_l), Q_j)$ and let $q_l = \mathbf{bottom}(A_l)$. Let $M = Q_j[q_h, q_l]$ (i.e., the subpath of Q_j from q_h to q_l). See Figure 1 for illustration.

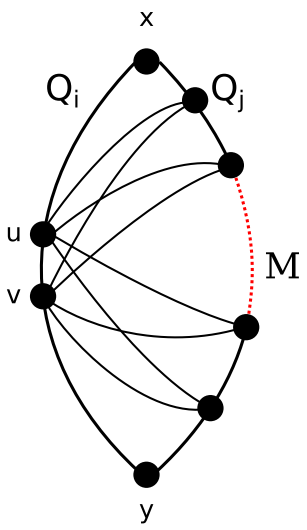


Fig. 1. Illustration of the sets $\mathcal{B}_h$, $\mathcal{B}_l$ $(p=2)$ and the path M

We claim that (1) the subgraph $H'(V, \mathcal{S}')$ for $\mathcal{S}' = (E(\mathcal{S}) \setminus E(M)) \cup E(\mathcal{B}_h) \cup E(\mathcal{B}_l)$ contains p vertex-disjoint paths from x to y and (2) $\mathbf{cost}(\mathcal{S}') \leq \mathbf{cost}(\mathcal{S})$.

To prove claim (1) we use Menger's theorem. We show that there is no set F of $p-1$ vertices such that x and y are disconnected in $\mathcal{S}' \setminus F$. Consider a set F of at most $p-1$ vertices. If F fails to intersect a path $Q_r \in \mathcal{S}$, for some $r \neq j$, then clearly x and y are connected in $\mathcal{S}' \setminus F$. So suppose the set F disconnects every path $Q_r \in \mathcal{S}$, for $r \neq j$, hence F contains exactly one vertex from each path $Q_r \in \mathcal{S}$ for every $r \neq j$. In particular, F contains only one vertex of Q_i. Therefore, one of u or v is not in F. Assume without loss of generality that $u \notin F$. Note that both sets $\mathcal{B}_h$ and $\mathcal{B}_l$ contain p vertex-disjoint-s-bypasses.

Since F contains at most $p-1$ vertices, there must be a vertex-disjoint-s-bypass $B_h \in \mathcal{B}_h$ and a vertex-disjoint-s-bypass $B_l \in \mathcal{B}_l$ whose internal vertices are not in F. Let $x_1 = \mathbf{top}(B_h)$ and $y_1 = \mathbf{bottom}(B_l)$. Note that the subpaths $Q_j[x, x_1]$ and $Q_j[y_1, y]$ do not contain any vertex from F, as $F \cap V(Q_j) = \emptyset$. Moreover, the vertex-disjoint-s-bypasses B_h and B_l contain subpaths $B_h[x_1, u]$ and $B_l[u, y_1]$ that do not intersect F. Concatenating all these paths together, we get a path $Q_j[x, x_1] \circ B_h[x_1, u] \circ B_l[u, y_1] \circ Q_j[y_1, y]$ from x to y. We thus conclude that H' contains p vertex-disjoint paths from x to y, establishing (1). Next, we show claim (2). Recall that $\mathcal{B}$ contains at least $2sp + 2p + 1$ vertex-disjoint-s-bypasses where each of which intersects Q_j. Moreover, each of the subpaths $Q_j[x, q_h]$ and $Q_j[q_l, y]$ intersect with exactly p vertex-disjoint-s-bypasses from $\mathcal{B}$. We get that the remaining part of Q_j, namely, the path $M = Q_j[q_h, q_l]$, intersects with at least $2sp + 2p + 1 - 2p = 2sp + 1$ vertex-disjoint-s-bypasses from $\mathcal{B}$. Thus, the length of M is at least $2sp + 1$. In contrast, the number of edges in the edge collection $E(\mathcal{B}_h) \cup E(\mathcal{B}_l)$ that replaced M in $\mathcal{S}$ is at most $2ps$. Hence, $\mathbf{cost}(\mathcal{S}') < \mathbf{cost}(\mathcal{S})$.

Finally parts (1) and (2) of the claim imply a contradiction to the optimality of $\mathcal{S}$. The lemma follows. ∎

Let $f = ((4s + 2)(p - 1)s + 1 + 2sp(p - 1) + 2p(p - 1))$.

Lemma 5. *Every f-vertex-resilient subgraph H is a p-vertex disjoint multipath spanner.*

Proof: Consider two vertices x and y and let $\mathcal{S} = \{Q_1, ..., Q_p\}$ be the set of p vertex-disjoint paths from x to y in G of minimal cost. Consider an edge $e = (u, v) \in E(Q_i)$ such that $e \notin E(H)$ for some $1 \leq i \leq p$. By definition of H, H contains f vertex-disjoint-s-bypasses between u to v. By Lemma 4, there are at most $2sp(p - 1) + 2p(p - 1)$ vertex-disjoint-s-bypasses between u to v, that intersect $V(\mathcal{S}) \setminus V(Q_i)$. We thus get that there exists a set $\mathbf{Bypasses}(e)$ of at least $f - 2sp(p - 1) - 2p(p - 1)$ vertex-disjoint-s-bypasses between u to v in H, that do not intersect $V(\mathcal{S}) \setminus V(Q_i)$. We now show how to select for each edge $e \in Q_i$ for some i a vertex-disjoint-s-bypass $B_e \in \mathbf{Bypasses}(e)$, such that B_e is vertex disjoint with any $B_{e'}$ for any $e' \in Q_j$ such that $e' \notin H$ and $j \neq i$.

Consider an edge $e = (u, v) \in Q_i$ such that $e \notin E(H)$. Let E_e be the set of edges e' such that $e' \in E(Q_j) \setminus E(H)$ and $V(B) \cap V(B') \neq \emptyset$ for some $j \neq i$, $B \in \mathbf{Bypasses}(e)$ and $B' \in \mathbf{Bypasses}(e')$. Towards proving Lemma 5, we first prove the next auxiliary lemma.

Lemma 6. *For every edge $e \in E(\mathcal{S}) \setminus E(H)$, the set E_e contains at most $(4s + 2)(p - 1)$ edges*

Proof: Assume, towards contradiction, that $|E_e| \geq (4s + 2)(p - 1) + 1$. By the Pigeonhole principle, there is a path Q_j (for $j \neq i$) such that $|E(Q_j) \cap E_e| > 4s+2$. Let $e_h = (u_h, v_h)$ be the edge in $E(Q_j) \cap E_e$ closest to x in Q_j, and let $e_l = (u_l, v_l)$ be the edge in $E(Q_j) \cap E_e$ closest to y in Q_j. Let $h_1, h_2 \in \mathbf{Bypasses}(e)$, $h_3 \in \mathbf{Bypasses}(e_h)$ and $h_4 \in \mathbf{Bypasses}(e_l)$ such that h_1 and h_3 intersect and h_2 and h_4 intersect (it could be that $h_1 = h_2$). Let M be the subpath $Q_j[v_h, v_l]$.

We now claim that (1) the subgraph $H''(V, \mathcal{S}'')$ for $\mathcal{S}'' = E(\mathcal{S}) \setminus E(M) \cup E(h_1) \cup E(h_2) \cup E(h_3) \cup E(h_4)$ contains p vertex disjoint paths from x to y and (2) $\mathbf{cost}(\mathcal{S}'') < \mathbf{cost}(\mathcal{S})$.

We show claim (1) by using Menger's theorem to establish that $\mathcal{S}''$ contains p vertex-disjoint paths from x to y. Consider a set F of at most $p-1$ vertices. If F fails to intersect a path $Q_r \in \mathcal{S}$, for some $r \neq j$, then clearly x and y are connected in $\mathcal{S}'' \setminus F$. So suppose the set F disconnects every path $Q_r \in \mathcal{S}$, for $r \neq j$, hence F contains exactly one vertex from each path $Q_r \in \mathcal{S}$ for every $r \neq j$. Note that all vertices in h_3 and h_4 are not in F as h_3 and h_4 are disjoint from all Q_r for $r \neq j$ and we assume that F contains only nodes from Q_r for some $r \neq j$. Since F contains exactly one vertex in Q_i then one of u and v are not in F, assume w.l.o.g. that $u \notin F$. Let r_3 be a vertex in $h_3 \cap h_1$ and let r_4 be a vertex in $h_4 \cap h_2$. Note that the subgraph $(E(Q_j) \setminus E(M)) \cup E(h_3)$ contains a path from x to r_3 that does not intersect F. Similarly, the subgraph $(E(Q_j) \setminus E(M)) \cup E(h_4)$ contains a path from r_4 to y that does not intersect F. The cycle $h_1 \cup \{(u, v)\}$ contains at most one vertex in F and thus there is a path from r_3 to u that does not contain vertices from F. Similarly, $h_1 \cup \{(u, v)\}$ contains a path from r_4 to u that does not contain vertices from F. Concatenating all these paths together we get a path from x to y. Claim (1) follows. Next, we show claim (2) and thus derive a contradiction to the optimality of $\mathcal{S}$. the path Q_j contains at least $4s+3$ edges from E_e and since the path M contains all these edges but two (e_h and e_l), the length of M is at least $4s+1$. In contrast, the number of edges in the edge collection $E(h_1) \cup E(h_2) \cup E(h_3) \cup E(h_4)$ that replaced M in $\mathcal{S}$ is at most $4s$. We thus get that $\mathbf{cost}(\mathcal{S}'') < \mathbf{cost}(\mathcal{S})$. This implies a contradiction to the optimality of $\mathcal{S}$. The lemma follows. ∎

Consider the edges $e \in E(\mathcal{S}) \setminus E(H)$ one by one. For each edge $e \in E(Q_i) \setminus E(H)$, choose a vertex-disjoint-s-bypass B_e that does not intersect with any $B_{e'}$ for an edge e' that was already considered by this process and such that $e' \in E(Q_r) \setminus E(H)$ for some $r \neq i$. We claim that this process never gets stuck, namely, each time we consider an edge $e \in E(Q_i) \setminus E(H)$, there is at least one vertex-disjoint-s-bypass in $\mathbf{Bypasses}(e)$ that does not intersect with the other vertex-disjoint-s-bypasses selected so far. Let $\tilde{E}_e \subseteq E_e$ be the set of edges that were considered before e by this process. Note that $|\tilde{E}_e| \leq (4s+2)(p-1)$ by Lemma 6. Moreover, note that each path $B_{e'}$ for some $e' \in \tilde{E}_e$ intersects with at most s vertex-disjoint-s-bypasses in $\mathbf{Bypasses}(e)$. Since there are more than $(4s+2)(p-1)s$ vertex-disjoint-s-bypasses in $\mathbf{Bypasses}(e)$, at least one of these vertex-disjoint-s-bypasses does not intersect with any of $B_{e'}$ for $e' \in \tilde{E}_e$.

For each path $Q_r \in \mathcal{S}$ for $1 \leq r \leq p$, construct a path $\tilde{Q}_r$ as follows. For every edge $e \in E(Q_r)$, if $e \in E(H)$ then add e to $\tilde{Q}_r$, otherwise add B_e to $\tilde{Q}_r$. It is not hard to verify that $V(Q_i) \cap V(Q_j) = \emptyset$ for any $i \neq j$ and that each Q_r is a path from x to y such that $\mathbf{cost}(\tilde{Q}_r) \leq s \cdot \mathbf{cost}(Q_r)$. The lemma follows. ∎

The following theorem was shown by Dinitz and Krauthgamer in [9].

Theorem 5. *[9] For every graph $G = (V, E)$, odd integer s and integer r, one can construct in polynomial time with high probability a r-vertex fault-tolerant s-spanner with $O\left(r^{2-\frac{2}{s+1}} n^{1+\frac{2}{s+1}} \log n\right)$ edges.*

Combining Theorem 5, Lemma 3 and Lemma 5, we get Theorem 2.

We note that for weighted graphs, every p-vertex-resilient H with stretch s is a p-vertex multipath $(s \cdot p)$-spanner. To see this, consider two vertices x and y and let $\mathcal{S} = \{Q_1, ..., Q_p\}$ be the set of p vertex-disjoint paths from x to y in G of minimal cost. We now show how to construct a subgraph $H'(V, \mathcal{S}')$ such that (1) H' contains p vertex-disjoint paths from x to y and (2) $\mathbf{cost}(\mathcal{S}') \leq s \cdot \mathbf{cost}(\mathcal{S})$.

Initially, set $\mathcal{S}' = \emptyset$. For every edge $e = (u, v) \in E(Q_r)$, if $e \in E(H)$ then add e to $\mathcal{S}'$, otherwise add to $\mathcal{S}'$ a set $\mathcal{B}(e)$ of V-bypasses such that $|\mathcal{B}(e)| = p$ and $E(\mathcal{B}(e)) \subseteq E(H)$. Note that such a set $\mathcal{B}(e)$ exists since H is p-vertex-resilient with stretch s.

To prove claim (1) we use Menger's theorem. We show that there is no set F of $p-1$ vertices such that x and y are disconnected in $\mathcal{S}' \setminus F$. Consider a set F of at most $p-1$ vertices. Note that, for every edge $e \in (E(Q_r) \setminus E(H))$, there exists a V-bypass $B(e) \in \mathcal{B}(e)$ that is internal vertex disjoint from F. To see this, recall that $\mathcal{B}(e)$ contains p V-bypasses. In addition, there exists at least one path $Q_r \in \mathcal{S}$ such that $V(Q_r) \cap F = \emptyset$ for some $1 \leq r \leq p$. Construct the path Q'_r as follows. For every edge $e = (u, v) \in E(Q_r)$, if $e \in E(H)$ then add e to Q'_r, otherwise add the alternative path $B(e)$ to Q'_r. It is not hard to verify that Q'_r leads from x to y, $E(Q'_r) \subseteq E(H)$, and $V(Q'_r) \cap F = \emptyset$. Claim (1) follows.

To see claim (2), note that for every edge $e \in E(\mathcal{S})$, either e itself or an alternative path of length s is added to $E(\mathcal{S}')$. We get that $\mathbf{cost}(\mathcal{S}') \leq s \cdot \mathbf{cost}(\mathcal{S})$.

We note that Lemma 3 can be generalized to weighted graphs. This can be done by invoking a construction for s-spanner, in the algorithm for constructing vertex fault-tolerant spanners of Dinitz and Krauthgamer in [9], that satisfies the following property, defined in [16]. Every edge e is either included in the spanner H or H contains an alternative path to e of length at most $s \cdot \omega(e)$ and with at most s hops. The details are omitted.

Combining with Theorem 5, we conclude Theorem 3.

References

1. Althöfer, I., Das, G., Dobkin, D., Joseph, D., Soares, J.: On sparse spanners of weighted graphs. Discrete & Computational Geometry 9, 81–100 (1993)
2. Awerbuch, B., Berger, B., Cowen, L., Peleg, D.: Near-linear cost sequential and distributed constructions of sparse neighborhood covers. In: Proc. 34th IEEE FOCS, pp. 638–647 (1993)
3. Barenboim, L., Elkin, M.: Deterministic distributed vertex coloring in polylogarithmic time. In: Proc. 29th ACM PODC, pp. 410–419 (2010)
4. Baswana, S., Kavitha, T.: Faster algorithms for approximate distance oracles and all-pairs small stretch paths. In: Proc. 47th IEEE FOCS, pp. 591–602 (2006)
5. Bollobás, B., Coppersmith, D., Elkin, M.: Sparse distance preservers and additive spanners. In: Proc. 14th ACM-SIAM SODA, pp. 414–423 (2003)

6. Chechik, S., Langberg, M., Peleg, D., Roditty, L.: Fault-tolerant spanners for general graphs. In: Proc. 41st ACM STOC, pp. 435–444 (2009)
7. Cowen, L.: Compact routing with minimum stretch. J. Algo. 38, 170–183 (2001)
8. Cowen, L., Wagner, C.: Compact roundtrip routing in directed networks. J. Algo. 50, 79–95 (2004)
9. Dinitz, M., Krauthgamer, R.: Fault-Tolerant Spanners: Better and Simpler. In: Proc. 30th ACM PODC, pp. 169–178 (2011)
10. Dubhashi, D., Mei, A., Panconesi, A., Radhakrishnan, J., Srinivasan, A.: Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. J. Computer and System Sciences 71, 467–479 (2005)
11. Elkin, M.: Computing almost shortest paths. ACM Tr. Algo. 1, 283–323 (2005)
12. Elkin, M.: A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In: Proc. 26th ACM PODC, pp. 185–194 (2007)
13. Elkin, M., Zhang, J.: Efficient algorithms for constructing $(1 + \epsilon, \beta)$-spanners in the distributed and streaming models. In: Proc. 23rd ACM PODC, pp. 160–168 (2004)
14. Farley, A.M., Proskurowski, A., Zappala, D., Windisch, K.: Spanners and message distribution in networks. Discrete Applied Mathematics 137(2), 159–171 (2004)
15. Gavoille, C., Godfroy, Q., Viennot, L.: Multipath Spanners. In: Patt-Shamir, B., Ekim, T. (eds.) SIROCCO 2010. LNCS, vol. 6058, pp. 211–223. Springer, Heidelberg (2010)
16. Gavoille, C., Godfroy, Q., Viennot, L.: Node-Disjoint Multipath Spanners and Their Relationship with Fault-Tolerant Spanners. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 143–158. Springer, Heidelberg (2011)
17. Jacquet, P., Viennot, L.: Remote spanners: what to know beyond neighbors. In: Proc. 23rd IEEE IPDPS (2009)
18. Kushman, N., Kandula, S., Katabi, D., Maggs, B.M.: R-BGP: Staying connected in a connected world. In: Proc. 4th NSDI (2007)
19. Mueller, S., Tsang, R.P., Ghosal, D.: Multipath Routing in Mobile Ad Hoc Networks: Issues and Challenges. In: Calzarossa, M.C., Gelenbe, E. (eds.) MASCOTS 2003. LNCS, vol. 2965, pp. 209–234. Springer, Heidelberg (2004)
20. Nasipuri, A., Castañeda, R., Das, S.R.: Performance of multipath routing for on-demand protocols in mobile ad hoc networks. Mobile Networks and Applications 6(4), 339–349 (2001)
21. Peleg, D., Schǎffer, A.A.: Graph spanners. J. Graph Theory, 99–116 (1989)
22. Peleg, D., Ullman, J.D.: An optimal synchronizer for the hypercube. SIAM J. Computing 18(4), 740–747 (1989)
23. Pettie, S.: Low Distortion Spanners. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 78–89. Springer, Heidelberg (2007)
24. Pan, P., Swallow, G., Atlas, A.: Fast Reroute Extensions to RSVP-TE for LSP Tunnels. RFC 4090 (Proposed Standard) (2005)
25. Roditty, L., Thorup, M., Zwick, U.: Roundtrip spanners and roundtrip routing in directed graphs. ACM Trans. Algorithms 3(4), Article 29 (2008)
26. Thorup, M., Zwick, U.: Compact routing schemes. In: Proc. SPAA, pp. 1–10 (2001)
27. Thorup, M., Zwick, U.: Approximate distance oracles. JACM 52, 1–24 (2005)
28. Woodruff, D.P.: Lower bounds for additive spanners, emulators, and more. In: Proc. 47th IEEE FOCS, pp. 389–398 (2006)

Constant Thresholds Can Make Target Set Selection Tractable[⋆]

Morgan Chopin[1,⋆⋆], André Nichterlein[2], Rolf Niedermeier[2], and Mathias Weller[2,⋆⋆⋆]

[1] LAMSADE, Université Paris-Dauphine, France
chopin@lamsade.dauphine.fr

[2] Institut für Softwaretechnik und Theoretische Informatik, TU Berlin, Germany
{andre.nichterlein,rolf.niedermeier,mathias.weller}@tu-berlin.de

Abstract. Target Set Selection, which is a prominent NP-hard problem occurring in social network analysis and distributed computing, is notoriously hard both in terms of achieving useful approximation as well as fixed-parameter algorithms. The task is to select a minimum number of vertices into a "target set" such that all other vertices will become active in course of a dynamic process (which may go through several activation rounds). A vertex, which is equipped with a threshold value t, becomes active once at least t of its neighbors are active; initially, only the target set vertices are active. We contribute further insights into islands of tractability for Target Set Selection by spotting new parameterizations characterizing some sparse graphs as well as some "cliquish" graphs and developing corresponding fixed-parameter tractability and (parameterized) hardness results. In particular, we demonstrate that upper-bounding the thresholds by a constant may significantly alleviate the search for efficiently solvable, but still meaningful special cases of Target Set Selection.

1 Introduction

The NP-hard graph problem Target Set Selection (TSS) is defined as follows: Given an undirected graph $G = (V, E)$ where each vertex $v \in V$ is assigned a positive integer threshold value $\text{thr}(v)$, the task is to find a minimum-cardinality target set $S \subseteq V$. A vertex set $S \subseteq V$ is called a *target set* of G if it "activates" *all* vertices in G in a dynamic process where a vertex v gets activated once at least $\text{thr}(v)$ many of its neighboring vertices are activated. Initially, only the vertices in S are active. TSS generalizes well-known graph problems such as Dominating Set with thresholds [14], Vector Dominating Set [24], k-Tuple Dominating Set [18] (all these variants allow for only one "activation round"), Vertex Cover [5] (where the threshold value equals the vertex degree), Irreversible k-Conversion Set [10] and r-Neighbor Bootstrap

[⋆] Major part of this work was done while all authors were at TU Berlin.
[⋆⋆] Main work done during a three-month visit of TU Berlin supported by DAAD.
[⋆⋆⋆] Supported by the DFG, project DARE (NI 369/11).

Percolation [1] (where the threshold of each vertex is k or r, respectively), and so-called dynamic monopolies [23] (where the threshold of a vertex v with degree $\deg(v)$ equals $\lceil \deg(v)/2 \rceil$—in the following this condition is referred to as *majority* thresholds). Besides being a problem of considerable graph-theoretic interest, TSS is also motivated by applications in areas such as social network analysis [5, 17] and distributed computing [23]. Indeed, different research communities using different names for describing the same or similar concepts, some work has been done independently from each other.

Since previous work has shown that TSS is computationally very hard [2, 5, 21], it is a natural approach to search for practically relevant, but computationally tractable special cases. We contribute to this line of research by starting from the following: While TSS is linear-time solvable both on trees [5] and on cliques [21, 25], it turns hard if the treewidth is unbounded [2] (more specifically, it is W[1]-hard with respect to the parameter treewidth of the graph) and it is NP-hard on graphs with diameter two [21] (cliques are exactly the diameter-one graphs). In this work, we focus on parameterizations measuring the distance from being a tree or forest and parameterizations measuring the distance from being a clique or cluster graph. Along these lines, we are particularly interested in the role of the allowed thresholds and one of our main conclusions is that bounding the thresholds by a constant (independent of the vertex degree) may be decisive in order to gain (fixed-parameter) tractability. This is of interest since in several applications (such as influence spreading in social networks) it is conceivable that constant thresholds suffice to model the underlying application scenarios.[1]

Previous Work. We focus on previous results that have direct relation to our work and refrain from discussing the history of work on TSS (a more thorough review of previous work can be found in [21]).

Chen [5] showed hardness of approximation for TSS, and Ben-Zwi et al. [2] obtained parameterized hardness results, particularly with respect to the parameter "treewidth".[2] Recently, further parameterized complexity studies for the structural graph parameters "diameter", "cluster editing number", "vertex cover number", and "feedback edge set number" have been undertaken [21]. Moreover, polynomial-time algorithms for TSS restricted to special graph classes including chordal graphs and block-cactus graphs have been developed [4, 6, 25].

The role of the threshold values resp. threshold functions has been studied in some publications. For instance, Dreyer and Roberts [10] showed NP-hardness for TSS when all vertices have a threshold of t, $t \geq 3$. Centeno et al. [4] and Chiang et al. [6] exploited threshold values being upper-bounded by two to develop polynomial-time algorithms for TSS on chordal graphs. Most interesting in our context, however, is the result of Ben-Zwi et al. [2] who showed that TSS is

[1] For instance, independent of my total number of friends it may suffice that at least five of my friends (that is, neighbor vertices) in a social network buy a certain product to convince me about the product's usefulness.

[2] Indeed, they also showed that TSS is polynomial-time solvable on constant-treewidth graphs. However, the degree of the polynomial depends on the treewidth.

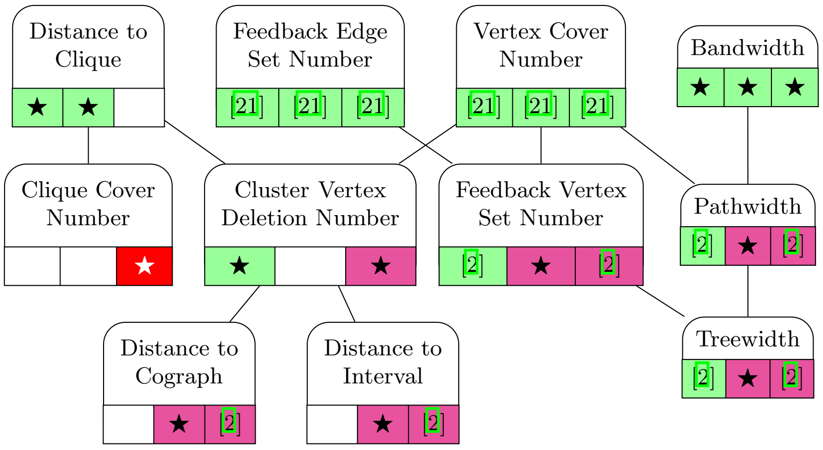


Fig. 1. Overview of the relations between structural graph parameters and our results for TSS. An edge from a parameter α to a parameter β below of α means that β can be upper-bounded in a linear function in α. The three rectangles below each parameter indicate the known results for TSS with (from left to right:) constant, majority, and general threshold function. The white star at the "clique cover number" means NP-hard for constant values of this parameter, dark violet background means W[1]-hard, light green background means FPT, and white background indicates an open question. Results marked with a star are obtained in this paper.

W[1]-hard with respect to the treewidth in case of unbounded threshold values whereas they showed it to be fixed-parameter tractable for the same parameter once the threshold values are bounded by a constant.

Our Contributions. Starting from the efficient solvability of TSS on trees and cliques [5, 21, 25], we also investigate to what extent efficient algorithms can be obtained for more general graph classes. On the one hand, we generalize from trees, thus in a way following previous work [2, 21] and consider further parameters measuring tree-likeness or sparseness, and, on the other hand we spot several parameters measuring distance to "cliquish" graphs. In both lines, we put particular emphasis on how the allowed threshold functions (arbitrary versus majority versus constant) influence computational complexity.[3] Our findings, which are pictorially presented (including the relations between parameters) in Figure 1, read as follows.

We start with the "sparse setting". For majority thresholds, we show that W[1]-hardness results for parameters such as "feedback vertex set number" and "pathwidth" for general threshold functions (which are due to Ben-Zwi et al. [2])

[3] Notably, all our positive results for constant thresholds generalize to the case that the maximum threshold $t_{\max}$ is given as an additional parameter. However, to keep matters simple and in accordance with previous work we focus on constant thresholds.

extend to the case of majority thresholds. Conversely, the very same parameterizations lead to fixed-parameter tractability results in case of constant threshold values [2]. Further, we briefly indicate that TSS is fixed-parameter tractable for the parameter bandwidth[4] even in case of arbitrary threshold functions.

Our main results are related to the "cliquish setting". The most central result here is that, with respect to the parameter "cluster vertex deletion number" (that is, the minimum number of vertices to delete from a graph to transform it into a union of disjoint cliques [15]), TSS is W[1]-hard for general thresholds but becomes fixed-parameter tractable in case of constant thresholds (the case of majority thresholds is open). For the larger (thus "weaker" [19]) parameter "distance to clique" (the minimum number of vertices to delete to make a graph a clique), however, TSS is fixed-parameter tractable for both constant and majority thresholds whereas this is open for general thresholds. Finally, for the parameter "clique cover number" (the minimum number of cliques needed to cover all vertices of a graph) we show NP-hardness even for parameter value two (rendering fixed-parameter tractability very unlikely) whereas the (parameterized) complexity is open in case of majority and constant thresholds.

Several proof details (particular concerning hardness proofs) are deferred to a full version of the paper.

2 Preliminaries and Parameter Identification

Preliminaries. We use standard graph-theoretic notation. For graphs $G = (V, E)$, we use $n := |V|$ and $m := |E|$. We omit the index of the neighborhood $N_G(v)$ or degree $\deg_G(v)$ of a vertex v if G is clear from the context. To formally define Target Set Selection, consider a graph $G = (V, E)$ and a function $\text{thr} : V \to \mathbb{N} \cup \{0\}$. For a vertex set $S \subseteq V$, we define the set of vertices that are *activated by S in the ith round* as $\mathcal{A}^i_{G,\text{thr}}(S)$ with

$$\mathcal{A}^0_{G,\text{thr}}(S) := S \text{ and}$$
$$\mathcal{A}^{j+1}_{G,\text{thr}}(S) := \mathcal{A}^j_{G,\text{thr}}(S) \cup \{v \in V \mid |N(v) \cap \mathcal{A}^j_{G,\text{thr}}(S)| \geq \text{thr}(v)\}.$$

We call $r(S) := \max\{i \mid \mathcal{A}^{i-1}_{G,\text{thr}} \neq \mathcal{A}^i_{G,\text{thr}}\}$ the *number of activation rounds* and say that S is a *target set* for (G, thr) if $\mathcal{A}^{r(S)}_{G,\text{thr}}(S) = V$. We can now formally define the central problem of this work:

Target Set Selection (TSS)
Input: An undirected graph $G = (V, E)$, a threshold function $\text{thr} : V \to \mathbb{N} \cup \{0\}$ and an integer $k \geq 0$.
Question: Is there a target set $S \subseteq V$ for G with $|S| \leq k$?

We denote the maximum threshold of an instance (G, thr) by $t_{\max}(G, \text{thr}) := \max\{\text{thr}(v) \mid v \in V(G)\}$. Again, we omit (G, thr) if it is clear from the context.

[4] A graph with *bandwidth* k has a linear arrangement of its vertices $v_1, \dots, v_n$ such that the length $|i - j|$ of each edge $\{v_i, v_j\}$ is at most k.

Parameterized Complexity. One dimension of a parameterized problem is the input size s, and the other one is the *parameter* (usually a positive integer). A parameterized problem is called *fixed-parameter tractable* (fpt) with respect to a parameter k if it can be solved in $f(k) \cdot s^{O(1)}$ time, where f is a computable function only depending on k. A core tool in the development of fixed-parameter algorithms is polynomial-time preprocessing by *data reduction* [3, 13]. Here, the goal is to transform a given problem instance I with parameter k in polynomial time into an equivalent instance I' with parameter $k' \leq k$ such that the size of I' is upper-bounded by some function g only depending on k. If this is the case, we call I' a (problem) *kernel* of size $g(k)$. Usually, this is achieved by applying polynomial-time executable data reduction rules. We call a data reduction rule $\mathcal{R}$ *correct* if the new instance I' that results from applying $\mathcal{R}$ to I is a yes-instance if and only if I is a yes-instance. An instance is called *reduced* with respect to some data reduction rule if further application of this rule has no effect on the instance. The whole process is called *kernelization*.[5]

Downey and Fellows [9] developed a parameterized theory of computational complexity to show (presumable) fixed-parameter intractability by means of *parameterized reductions*. A parameterized reduction from a parameterized problem P to another parameterized problem P' is a function that, given an instance (x, k), computes in $f(k) \cdot s^{O(1)}$ time an instance (x', k') (with k' only depending on k) such that (x, k) is a yes-instance of P if and only if (x', k') is a yes-instance of P'. The basic complexity class for fixed-parameter intractability is called $W[1]$ and there is good complexity-theoretic reason to believe that $W[1]$-hard problems are not fpt [9, 12, 22].

Parameter Identification. Fixed-parameter algorithms work best if the parameter they are designed for is small in practice. TSS having many applications on social networks [11], it is natural to extract small parameters from typical properties of social networks.

When the network models friendships for example, we expect the network to be made up of multiple cliques (or otherwise dense substructures) that overlap. This motivates considering the number of cliques needed to cover all vertices [16] (the "clique cover number") or the number of vertices to remove to obtain a clique (the "distance to clique"). As the latter parameter is somewhat restrictive, we also considered the number of vertices to delete in order to obtain a collection of disjoint cliques (the "cluster vertex deletion number"). Recently, the cluster vertex deletion number was also used to parameterize problems related to coloring and hamiltonicity [8].

In some applications, we deal with very sparse social networks, for instance networks modeling sexual contacts [11, Chap. 2, Fig. 2.7]. In these cases, parameters related to the sparseness of the input graph are interesting. Among them, we consider the number of vertices to remove to obtain an edgeless graph ("vertex cover number"), the number of edges or vertices to remove to obtain a forest

[5] It is well-known that a parameterized problem is fixed-parameter tractable if and only if it has a problem kernel.

("feedback edge set number" and "feedback vertex set number") as well as some graph width parameters (treewidth, pathwidth, bandwidth). For definitions of treewidth, pathwidth, cographs, and interval graphs see for example Diestel's book [7].

Data Reduction. We use the following two data reduction rules throughout this work.

If the threshold of a vertex exceeds its degree, it cannot be activated by its neighbors and, hence, the vertex is part of any target set. Moreover, we consider threshold-0 vertices as already active.

Reduction Rule 1 ([21, Reduction Rule 1]). *Let $G = (V, E)$ and $v \in V$. If $\operatorname{thr}(v) > \deg(v)$, then delete v, decrease the threshold of all its neighbors by one and decrease k by one. If $\operatorname{thr}(v) = 0$, then delete v and decrease the thresholds of all its neighbors by one.*

If the instance is reduced with respect to Reduction Rule 1, then every degree-one vertex has threshold one. Thus, considering an arbitrary degree-one vertex we do not choose it to be in the target set as choosing its neighbor is at least as good. Hence, this vertex does not help in activating any other vertex. This is formalized in the next data reduction rule.

Reduction Rule 2 ([21, Reduction Rule 5]). *Let $(G = (V, E), \operatorname{thr}, k)$ be an instance of TSS reduced with respect to Reduction Rule 1 and let $v \in V$ with $\operatorname{thr}(v) = \deg(v) = 1$. Then delete v from G.*

3 Parameters Related to Sparse Structures

In this section, we consider parameters that measure the sparseness of the input graph. Since trees are the most sparse connected graphs and TSS is polynomial time solvable on trees [5], parameters measuring the distance to trees are most interesting. Canonical candidates for this are the treewidth, the pathwidth, and the feedback vertex set of the input graph. Notably, a fixed-parameter algorithm of Ben-Zwi et al. [2] for the parameter treewidth tw can solve TSS in $t_{\max}^{O(\mathrm{tw})} n^{O(1)}$ time, implying fixed-parameter tractability for the three parameters mentioned above if the maximum threshold $t_{\max}$ is some constant. We extend this result by showing W[1]-hardness for them if we, instead of limiting the maximum threshold by a constant, require the thresholds to respect the majority condition. The proof even shows hardness for the combination of the feedback vertex set, the pathwidth, the vertex-deletion-distance to cograph, and the vertex-deletion-distance to interval graph. The proof is based on and further extends the W[1]-hardness reduction presented by Ben-Zwi et al. [2].

Theorem 1. Target Set Selection *with majority threshold is $W[1]$-hard even with respect to the combination of the following parameters: feedback vertex set, distance to cograph, distance to interval graph, and pathwidth.*

Bandwidth. Another possible measure for sparseness is the bandwidth of the input. Here, our result is of more positive nature.

Theorem 2. Target Set Selection *is fixed-parameter tractable with respect to the parameter bandwidth.*

Proof. Let $(G = (V, E), \mathrm{thr}, k)$ be an instance of TSS. First, exhaustively apply Reduction Rule 1 to get a new equivalent instance $(G' = (V', E'), \mathrm{thr}', k')$. Let bw denote the bandwidth of G'. It is not hard to show that $\mathrm{thr}'(v) \leq \deg_{G'}(v) \leq 2\,\mathrm{bw}$ for all $v \in V'$. Moreover, Ben-Zwi et al. [2] gave a $(t_{\max})^{O(\mathrm{tw})}n$-time algorithm for solving TSS, where tw is the treewidth of the input graph and $t_{\max}$ is the maximum threshold value. It follows that this algorithm applied to G runs in time $(2\,\mathrm{bw})^{O(\mathrm{bw})}n$ since $\mathrm{tw} \leq 2\,\mathrm{bw}$. □

4 Parameters Related to Dense Structures

In contrast to the previous section, we now consider TSS with respect to parameters related to the denseness of the input graph. Since cliques are the most dense graphs and TSS is polynomial time solvable on cliques [21], parameters measuring the distance to cliques are most interesting. In particular, we consider the vertex deletion distance to clique and to a collection of disjoint cliques (also called cluster vertex deletion set size), and the clique cover number.

Unrestricted Thresholds. Here, we research the general TSS setting without constraints on the thresholds of the input. As hinted in the introduction, these variants are hard with respect to the denseness measures employed.

Theorem 3. Target Set Selection *is W[1]-hard with respect to the parameter "cvd number".*

Theorem 4. Target Set Selection *is NP-hard and $W[2]$-hard with respect to the parameter "target set size" k, even on graphs with clique cover number two.*

Restricted Thresholds. In the spirit of researching the influence of bounded thresholds to TSS, we consider the parameters distance to clique and cluster vertex deletion number (cvd number). Recall that we showed W[1]-hardness for the parameter cvd number (for unbounded thresholds) in the previous paragraph. By presenting an exponential-size problem kernel, we show that the problem becomes tractable for this parameter if the maximum threshold is a constant.

First, we show that TSS with majority thresholds or constant thresholds is fixed-parameter tractable with respect to the parameter distance ℓ to clique. We can even show fixed-parameter tractability for less restrictive threshold functions. To this end let $\mathcal{P}(V)$ be the power set of V.

Theorem 5. Target Set Selection *is fixed-parameter tractable with respect to the parameter distance ℓ to clique when the threshold function* thr *fulfills the restriction "$\mathrm{thr}(v) > g(\ell) \Rightarrow \mathrm{thr}(v) = f(N(v))$" for all vertices $v \in V$ and arbitrary functions $f : \mathcal{P}(V) \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$.*

Proof. We prove the theorem by giving a fixed-parameter algorithm computing a minimum-size target set for (G, thr). To this end, we introduce some notations. Let $X \subset V$, $|X| = \ell$, denote a set of vertices such that $G[V \setminus X]$ is a clique. We define a non-standard "twins" equivalence relation $\equiv$ by

$$u \equiv v \iff (N[u] = N[v]) \wedge (\text{thr}(u) = \text{thr}(v)) \wedge (u \in X \iff v \in X).$$

Since the thresholds and neighborhoods of all vertices in an equivalence class Z are equal, we can denote this threshold and this neighborhood by $\text{thr}(Z)$ and $N[Z]$, respectively. Let $Z_1, Z_2, \ldots, Z_s$ be a list of all nonempty equivalence classes of $\equiv$. Since $G[V \setminus X]$ is a clique, we know that for all $u, v \in V \setminus X : N[u] = N[v]$ if and only if $N_{G[X]}[u] = N_{G[X]}[v]$. Due to the condition $\text{thr}(v) > g(\ell) \Rightarrow \text{thr}(v) = f(N(v))$, for each subset $X' \subseteq X$, there are at most $g(\ell) + 1$ equivalence classes disjoint from X whose neighborhood in X is exactly X'. Hence, $s \leq 2^\ell(g(\ell) + 1) + \ell$.

Let S be a minimum-size target set for (G, thr). With S, we can define r_i as the number of the first activation round in which all vertices of Z_i are active. More formally, $r_i := \min\{j \mid Z_i \subseteq \mathcal{A}^j_{G,\text{thr}}(S)\}$. Let $r := \max\{r_i \mid 1 \leq i \leq s\}$.

In the following, we upper-bound r by s. We do this by showing that for each $1 \leq j \leq r$, there is an $1 \leq i \leq s$ such that $r_i = j$. Assume this was false, that is, there is some activation round j such that none of the equivalence classes gets activated in round j. Since $j \leq r$, there is some vertex v that gets activated in round j. Let Z_i denote the equivalence class of v. Since $j \geq 1$, we know that $|N(v) \cap \mathcal{A}^{j-1}_{G,\text{thr}}(S)| \geq \text{thr}(v)$. Since for each vertex $u \in Z_i$, $\text{thr}(u) = \text{thr}(v)$ and $N(u) = N(v)$, we conclude that $Z_i \subseteq \mathcal{A}^j_{G,\text{thr}}(S)$, contradicting the assumption that $r_i \neq j$.

Now we describe our algorithm. In the first phase, we guess the correct values of r_i for all $1 \leq i \leq s$. There are at most $r^s \leq s^s$ possibilities to do so.

In the second phase of the algorithm, we use an ILP formulation to solve the problem. Each variable x_i in the ILP represents the number of vertices in the equivalence class i that are in the target set S. We use the constraints to model the activation process: For each equivalence class Z_i, the number of active neighbors in round r_i exceeds $\text{thr}(Z_i)$. Two types of active neighbors are considered. First, the vertices in $N[Z_i] \cap S$. Second, the vertices in all equivalence classes $Z_j \subseteq N[Z_i]$ that are active in round i, that is, $r_j < r_i$. More formally,

$$\begin{aligned} \text{Minimize: } & \sum_{i=1}^{s} x_i \\ \text{subject to: } & \forall 1 \leq i \leq s : \text{thr}(Z_i) \leq \sum_{\substack{Z_j \subseteq N[Z_i] \\ r_j \geq r_i}} x_j + \sum_{\substack{Z_j \subseteq N[Z_i] \\ r_j < r_i}} |Z_j| \\ & \forall 1 \leq i \leq s : x_i \in \{0, 1\}. \end{aligned}$$

By the discussion above, a solution to this ILP corresponds to a minimum-size target set for (G, thr). Since the ILP formulation has s variables, a result by Lenstra [20] implies that solving it is fixed-parameter tractable with respect to s.

Since at most s^s such ILPs have to be solved and $s \leq 2^\ell(g(\ell)+1)+\ell$, fixed-parameter tractability with respect to ℓ follows. □

Clearly, Theorem 5 is a pure complexity classification result. Since the majority thresholds and constant thresholds both satisfy the restrictions required in Theorem 5, the next corollary immediately follows.

Corollary 1. TARGET SET SELECTION *with majority thresholds and constant thresholds is fixed-parameter tractable with respect to the parameter distance to clique.*

Next, we show fixed-parameter tractability for TSS with constant thresholds with respect to the parameter "cvd number". In the following, we assume that an optimal cvd set X of the input graph is given. If this is not the case, then one might instead use a simple factor-3 approximation.[6] Either way, we abbreviate $\ell := |X|$.

In this section we use the notation of "critical cliques". Here, a clique K in a graph is *critical* if all its vertices have the same closed neighborhood and K is maximal with respect to this property.

First, we present a data reduction rule allowing us to bound the number of vertices with the same open or closed neighborhood by the maximum threshold $t_{\max}$.

Reduction Rule 3. *Let $I := (G = (V, E), \mathrm{thr}, k)$ be an instance of TSS that is reduced with respect to Reduction Rule 1 and let $v_1, v_2, \ldots, v_{t_{\max}+1} \in V$ be vertices such that either*

$$N(v_1) = N(v_2) = \ldots = N(v_{t_{\max}+1}) \text{ or } N[v_1] = N[v_2] = \ldots = N[v_{t_{\max}+1}].$$

Furthermore, let v_1 be the vertex with the highest threshold, that is, for all $1 \leq i \leq t_{\max}+1$ it holds that $\mathrm{thr}(v_1) \geq \mathrm{thr}(v_i)$. Then delete v_1.

Lemma 1. *Reduction Rule 3 is correct and can be applied exhaustively in $O(n+m)$ time.*

In the following we assume that the input graph G is reduced with respect to Reduction Rule 1 and Reduction Rule 3. Thus, $G[V \setminus X]$ consists of disjoint cliques, each of size at most $2^\ell t_{\max}$. Hence, in order to show a problem kernel it remains to bound the number of cliques in $G[V \setminus X]$. To this end, we introduce the following notation:

Definition 1. *Let $I := (G = (V, E), \mathrm{thr}, k)$ be an instance of TSS, let $X \subseteq V$ be a cvd set, and let $S \subseteq V$. Let $C_1, C_2 \subseteq V$ be two clusters in $G[V \setminus X]$. We call C_1 and C_2* equivalent with respect to X*, denoted by $C_1 \equiv_X C_2$, if there exists a bijection $f : C_1 \to C_2$ such that for every $v \in C_1$ it holds that $\mathrm{thr}(v) = \mathrm{thr}(f(v))$*

[6] A graph is a cluster graph if and only if it contains no induced P_3, that is, an induced path of three vertices. Using this characterization, the factor 3-approximation simply deletes all vertices occurring in an induced P_3.

and $N(v) \cap X = N(f(v)) \cap X$*. Furthermore, we call* C_1 *and* C_2 equivalent with respect to X and S*, denoted by* $C_1 \equiv_X^S C_2$*, if the bijection* f *additionally fulfills* $v \in S \iff f(v) \in S$ *for all* $v \in C_1$*.*

Note that $\equiv_X$ is an equivalence relation on the clusters in $G[V \setminus X]$ with at most $(t_{\max}+1)^{2^\ell t_{\max}}$ equivalence classes. To see this, observe that each equivalence class is uniquely determined by 2^ℓ (possibly empty) sequences of thresholds. One for each subset of X. Since G is reduced with respect to Reduction Rule 3, each such sequence contains between 0 and $t_{\max}$ thresholds. Since each threshold is at most $t_{\max}$, the number of equivalence classes is at most

$$\left(\sum_{i=0}^{t_{\max}} t_{\max}^i\right)^{2^\ell} \leq \left((t_{\max}+1)^{t_{\max}}\right)^{2^\ell} = (t_{\max}+1)^{2^\ell t_{\max}} .$$

In the following, our goal is to bound the number of cliques in one equivalence class in a function depending only on $t_{\max}$ and ℓ. Note that once we achieve this goal, we have a problem kernel with respect to the parameter "cvd number". The next lemma is a first step towards this goal.

Lemma 2. *Let* $I := (G = (V,E), \mathrm{thr}, k)$ *be an instance of TSS, let* $X \subseteq V$ *be a cvd set for* G*, and let* $S \subseteq V$*,* $|S| \leq k$*, be a target set for* G*. Furthermore let* $C_1, C_2, \ldots, C_{t_{\max}+1} \subseteq V$ *be clusters in* $G[V \setminus X]$ *that are pairwise equivalent with respect to* X *and* S*. Then,* $S \setminus C_1$ *is a target set for* $G[V \setminus C_1]$*.*

Proof. Let $S' = S \setminus C_1$ and $G' = G[V \setminus C_1]$. We prove the lemma by contradiction: Assume that S' is not a target set for G'. Let $Y \subseteq V \setminus C_1$ be the set of vertices that are activated in G in some round i but are not activated in G' in the round i. Formally, $Y := \{v \in V \setminus C_1 \mid \exists i \geq 1 : v \in \mathcal{A}^i_{G,\mathrm{thr}}(S) \wedge v \notin \mathcal{A}^i_{G',\mathrm{thr}}(S')\}$. Since S' is not a target set for G', the set Y is not empty. In particular, Y contains all vertices in G' that are not activated by S'. Let $v \in Y$ be the vertex that is activated first in G, that is, for all $u \in Y$ it holds that $u \in \mathcal{A}^i_{G,\mathrm{thr}}(S) \Rightarrow v \in \mathcal{A}^i_{G,\mathrm{thr}}(S)$, $1 \leq i$.

Since $v \in Y$ and $Y \subseteq V \setminus C_1$, it holds that $v \notin S$. Let $i \geq 1$ be the round in which v becomes active in G, that is, $v \in \mathcal{A}^i_{G,\mathrm{thr}}(S) \setminus \mathcal{A}^{i-1}_{G,\mathrm{thr}}(S)$. Thus, $|N_G(v) \cap \mathcal{A}^{i-1}_{G,\mathrm{thr}}(S)| \geq \mathrm{thr}(v)$. Since v is in G' not activated by S', it follows that $|N_{G'}(v) \cap \mathcal{A}^{i-1}_{G',\mathrm{thr}}(S')| < \mathrm{thr}(v)$. From the selection of v it follows that $Y \cap \mathcal{A}^{i-1}_{G,\mathrm{thr}}(S) = \emptyset$. Thus, $\mathcal{A}^{i-1}_{G,\mathrm{thr}}(S) \setminus \mathcal{A}^{i-1}_{G',\mathrm{thr}}(S') \subseteq C_1$. Since $N_G(v) \setminus N_{G'}(v) \subseteq C_1$, it follows that $N_G(v) \cap \mathcal{A}^{i-1}_{G,\mathrm{thr}}(S) \cap C_1 \neq \emptyset$ and $v \in X$. Let $u \in N_G(v) \cap \mathcal{A}^{i-1}_{G,\mathrm{thr}}(S) \cap C_1$. Note that C_1 and C_j, $1 < j \leq t_{\max}+1$, are equivalent with respect to X and S and, hence, there is a bijection f_j as described in Definition 1. Thus, it is easy to see that $u \in \mathcal{A}^{i-1}_{G,\mathrm{thr}}(S) \Rightarrow f_j(u) \in \mathcal{A}^{i-1}_{G,\mathrm{thr}}(S)$. Moreover, since $u \in N_G(v)$ it follows that $f_j(u) \in N_G(v)$ and, thus, $f_j(u) \in N_{G'}(v)$. Hence, $f_j(u) \in N_{G'}(v) \cap \mathcal{A}^{i-1}_{G',\mathrm{thr}}(S')$ for all $2 \leq j \leq t_{\max}+1$ and thus $|N_{G'}(v) \cap \mathcal{A}^{i-1}_{G',\mathrm{thr}}(S')| \geq t_{\max}$. Hence, $\mathrm{thr}(v) > |N_{G'}(v) \cap \mathcal{A}^{i-1}_{G',\mathrm{thr}}(S')| \geq t_{\max}$, a contradiction. □

Since we do not know the target set S for G, two problems have to be solved in order to convert this lemma into a data reduction rule: The first problem is to find out by how much we have to decrease k, or, equivalently, how to compute $|S \cap C_1|$ in polynomial time? The second problem is that we do not know the target set S. As we show in the following, the key in overcoming these two problems is to increase the number of equivalent clusters C_j in the assumption of the lemma.

To this end, we first compute a lower and upper bound on the size of the target set for G. Let G^X be the graph that results from activating all vertices in X and applying Reduction Rule 1 exhaustively. Let $C_1^X, C_2^X, \ldots, C_\zeta^X$ denote the maximal cliques of G^X. Clearly, for each clique C^X of G^X there is a cluster C in $G[V \setminus X]$ such that $C^X \subseteq C$. Let $S^X \subseteq V$ be an optimal solution for G^X. Note that S^X can be computed in linear time [21, 25]. By construction of G^X it is clear that $|S^X|$ is a lower bound for the size of any target set for G. Furthermore, $S^X \cup X$ is a target set for G. Hence, if $k < |S^X|$ we can immediately answer no and if $k \geq |S^X| + |X| = |S^X| + \ell$ we can answer yes. Thus, we assume in the following that $|S^X| \leq k < |S^X| + \ell$. Besides these general bounds on the target set size we can also derive bounds for the number of vertices in a target set for each cluster C in $G[V \setminus X]$: If there is a (uniquely determined) clique C^X in G^X such that $C^X \subseteq C$, then set $\min(C) = |S^X \cap C^X|$. In case there is no such clique in G^X, set $\min(C) = 0$. Finally, set $\max(C) = \min\{t_{\max}, \min(C) + \ell\}$. Clearly, $\min(C)$ and $\max(C)$ are lower resp. upper bounds on the number of vertices of C that are in an optimal target set for G. Note that if two clusters C_1 and C_2 in $G[V \setminus X]$ are equivalent with respect to X, then $\min(C_1) = \min(C_2)$. Furthermore, having $\ell + 1$ clusters $C_1, \ldots, C_{\ell+1}$ in $G[V \setminus X]$ that are equivalent with respect to X, we can conclude that for any optimal target set S there is a cluster C_i, $1 \leq i \leq \ell + 1$, having exactly $\min(C_1)$ vertices in the target set, since otherwise, the solution $S^X \cup X$ for G contains fewer vertices than S. Likewise, if there are $\ell + r$ clusters $C_1, \ldots, C_{\ell+r}$ that are equivalent with respect to X, then it is clear that for any optimal target set S at least r of these clusters contain exactly $\min(C_1)$ vertices of S. Hence, increasing the number of equivalent clusters to at least $\ell + t_{\max} + 1$ solves the first problem.

We overcome the second problem by relaxing the condition for the clusters $C_1, \ldots, C_{t_{\max}} \subseteq V$ to "equivalent with respect to X" instead of "equivalent with respect to X and S" and increase the number of equivalent clusters: We can assume that, out of each cluster C, at most $\max(C) \leq t_{\max}$ vertices are in a target set. Thus, there are at most $t_{\max}^{2^\ell}$ possibilities for choosing $t_{\max}$ vertices from a cluster to be in a target set: Choose at most $t_{\max}$ vertices with the highest threshold from each of the at most 2^ℓ critical cliques of the cluster.[7] Thus, when increasing the number of clusters that have to be equivalent with respect to X to $\ell + t_{\max}^{2^\ell}(t_{\max} + 1)$ we can conclude with the pigeonhole principle that there are clusters $C_{i_1}, \ldots, C_{i_{t_{\max}+1}}$ that are equivalent with respect to X and S for

[7] Having a set of vertices with the same closed neighborhood and the task is to choose s of them to be in a target set, it is best to choose the s vertices with the highest thresholds [21, Observation 7].

any target set S and each cluster C_{i_j} contains $\min(C_{i_j})$ vertices of S. Hence, applying Lemma 2 to this set we arrive at the following reduction rule.

Reduction Rule 4. *Let $I := (G = (V, E), \mathrm{thr}, k)$ be an instance of TSS that is reduced with respect to Reduction Rule 1 and let $X \subseteq V$ be a cvd set of size ℓ. Let $C_1, C_2, \ldots, C_\alpha \subset V$ be disjoint clusters in $G[V \setminus X]$ such that $\alpha = \ell + t_{\max}^{2^\ell}(t_{\max} + 1)$ and for each pair C_i, C_j, $1 \leq i, j \leq \alpha$, it holds that $C_i \equiv_X C_j$. Then delete C_1 and reduce k by $\min(C_1)$.*

The correctness of the data reduction rule follows from Lemma 2 and the above discussion. As to the running time, note that Reduction Rule 4 can be exhaustively applied in $O(n^2)$ time. Since we require that the cvd set X is given, we can compute the clusters in $G[V \setminus X]$ in linear time. Then, we sort the vertices in these clusters by neighborhood and threshold. This can be done in $O(n \log(n))$ time. After this sorting the check whether two clusters are equivalent with respect to X can be done in linear time: Simply iterate over the sorted vertices and check whether the current vertices in both clusters have the same neighborhood and threshold. Overall, iterating over all clusters in $G[V \setminus X]$, determining the equivalent clusters, and deleting the respective clusters can be done in $O(n^2)$ time.

With these data reduction rules we now arrive at the following theorem.

Theorem 6. Target Set Selection *admits a problem kernel with $t_{\max}^{O(2^\ell t_{\max})}\ell$ vertices, where ℓ is the size of a cvd set and $t_{\max}$ the maximum threshold. The problem kernel can be found in $O(n^2)$ time.*

Proof. Let $I := (G = (V, E), \mathrm{thr}, k)$ be an instance of TSS that is reduced with respect to Reduction Rules 1, 3, and 4. Furthermore let $X \subseteq V$ be a cvd set and let $\ell = |X|$.

Since I is reduced with respect to Reduction Rule 3, the clusters in $G[V \setminus X]$ have size at most $2^\ell t_{\max}$. Hence, there are at most $(t_{\max} + 1)^{2^\ell t_{\max}}$ clusters in $G[V \setminus X]$ that are all pairwise *not* equivalent with respect to X. Furthermore, since I is reduced with respect to Reduction Rule 4, each equivalence class of $\equiv_X$ contains at most $\ell + t_{\max}^{2^\ell}(t_{\max}+1)$ clusters. Thus, the number of clusters in $G[V \setminus X]$ is bounded by $(\ell + t_{\max}^{2^\ell}(t_{\max} + 1))(t_{\max} + 1)^{2^\ell t_{\max}}$, each of these clusters contains at most $2^\ell t_{\max}$ vertices. Overall this gives $t_{\max}^{O(2^\ell t_{\max})}\ell$ vertices in $G[V \setminus X]$ and, thus, G contains at most $t_{\max}^{O(2^\ell t_{\max})}\ell$ vertices. The Reduction Rules 1 and 3 can both be applied exhaustively in $O(n+m)$ time and Reduction Rule 4 can be applied exhaustively in $O(n^2)$. Overall, the kernelization runs in $O(n^2)$ time. $\square$

Clearly this problem kernel implies that TSS is fixed-parameter tractable with respect to the combined parameter $(t_{\max}, \ell)$. This leads to our next corollary considering TSS with constant thresholds.

Corollary 2. Target Set Selection *with constant thresholds is fixed-parameter tractable with respect to the parameter "cvd number".*

5 Conclusion

We showed that constant threshold values, as can be assumed in several real-world applications of TSS, can help to find efficient algorithms that exactly solve TSS. This extends previous work of Ben-Zwi et al. [2] where this observation was made for the parameter treewidth. A question left open in our work is whether or not TSS is fixed-parameter tractable with respect to the parameter cluster vertex deletion number for majority thresholds. A second open question arising from our work is whether or not TSS is fixed-parameter tractable with respect to the parameter "distance to clique" for general thresholds. Indeed, these two cases are part of the more general open question whether, in terms of computational complexity, TSS with majority thresholds is basically as hard as for general thresholds but significantly easier for constant thresholds[8]—the results we achieved in this paper may be interpreted as directing to a corresponding conjecture. Considering the practical relevance of TSS, it would be interesting to incorporate further natural parameters into the search for islands of tractability; among these we clearly have "graph diameter" (note, however, that this parameter needs to be combined with others since TSS is already hard on diameter-two graphs [21]) and "number of activation rounds" (the case of only activation round—that is, the non-dynamic setting—leads to variants of domination [14, 18, 24]; again, in order to lead to tractability results, this parameter needs to be combined with others [21]).

References

[1] Balogh, J., Bollobás, B., Morris, R.: Bootstrap percolation in high dimensions. Combinatorics, Probability & Computing 19(5-6), 643–692 (2010)
[2] Ben-Zwi, O., Hermelin, D., Lokshtanov, D., Newman, I.: Treewidth governs the complexity of target set selection. Discrete Optimization 8(1), 87–96 (2011)
[3] Bodlaender, H.L.: Kernelization: New Upper and Lower Bound Techniques. In: Chen, J., Fomin, F.V. (eds.) IWPEC 2009. LNCS, vol. 5917, pp. 17–37. Springer, Heidelberg (2009)
[4] Centeno, C.C., Dourado, M.C., Penso, L.D., Rautenbach, D., Szwarcfiter, J.L.: Irreversible conversion of graphs. Theoretical Computer Science 412(29), 3693–3700 (2011)
[5] Chen, N.: On the approximability of influence in social networks. SIAM Journal on Discrete Mathematics 23(3), 1400–1415 (2009)
[6] Chiang, C.-Y., Huang, L.-H., Li, B.-J., Wu, J., Yeh, H.-G.: Some results on the target set selection problem. Journal of Combinatorial Optimization (2012)
[7] Diestel, R.: Graph Theory, 4th edn. Graduate Texts in Mathematics, vol. 173. Springer (2010)
[8] Doucha, M., Kratochvíl, J.: Cluster Vertex Deletion: A Parameterization between Vertex Cover and Clique-Width. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 348–359. Springer, Heidelberg (2012)

[8] Recall that majority thresholds are of particular interest in distributed computing [23].

[9] Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer (1999)
[10] Dreyer Jr., P.A., Roberts, F.S.: Irreversible k-threshold processes: Graph-theoretical threshold models of the spread of disease and of opinion. Discrete Applied Mathematics 157, 1615–1627 (2009)
[11] Easley, D., Kleinberg, J.: Networks, Crowds, and Markets: Reasoning about a Highly Connected World. Cambridge University Press (2010)
[12] Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer (2006)
[13] Guo, J., Niedermeier, R.: Invitation to data reduction and problem kernelization. ACM SIGACT News 38(1), 31–45 (2007)
[14] Harant, J., Pruchnewski, A., Voigt, M.: On dominating sets and independent sets of graphs. Combinatorics, Probability and Computing 8(6), 547–553 (1999)
[15] Hüffner, F., Komusiewicz, C., Moser, H., Niedermeier, R.: Fixed-parameter algorithms for cluster vertex deletion. Theory of Computing Systems 47(1), 196–217 (2010)
[16] Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press (1972)
[17] Kempe, D., Kleinberg, J., Tardos, É.: Maximizing the spread of influence through a social network. In: Proc. 9th ACM KDD, pp. 137–146. ACM Press (2003)
[18] Klasing, R., Laforest, C.: Hardness results and approximation algorithms of k-tuple domination in graphs. Information Processing Letters 89(2), 75–83 (2004)
[19] Komusiewicz, C., Niedermeier, R.: New Races in Parameterized Algorithmics. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 19–30. Springer, Heidelberg (2012)
[20] Lenstra, H.W.: Integer programming with a fixed number of variables. Mathematics of Operations Research 8, 538–548 (1983)
[21] Nichterlein, A., Niedermeier, R., Uhlmann, J., Weller, M.: On tractable cases of target set selection. Social Network Analysis and Mining (2012)
[22] Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press (2006)
[23] Peleg, D.: Local majorities, coalitions and monopolies in graphs: a review. Theoretical Computer Science 282, 231–257 (2002)
[24] Raman, V., Saurabh, S., Srihari, S.: Parameterized Algorithms for Generalized Domination. In: Yang, B., Du, D.-Z., Wang, C.A. (eds.) COCOA 2008. LNCS, vol. 5165, pp. 116–126. Springer, Heidelberg (2008)
[25] Reddy, T., Krishna, D., Rangan, C.: Variants of Spreading Messages. In: Rahman, M. S., Fujita, S. (eds.) WALCOM 2010. LNCS, vol. 5942, pp. 240–251. Springer, Heidelberg (2010)

Faster Variance Computation for Patterns with Gaps

Fabio Cunial[⋆]

College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
fabio.cunial@gatech.edu

Abstract. Determining whether a pattern is statistically overrepresented or underrepresented in a string is a fundamental primitive in computational biology and in large-scale text mining. We study ways to speed up the computation of the expectation and variance of the number of occurrences of a pattern with rigid gaps in a random string. Our contributions are twofold: first, we focus on patterns in which groups of characters from an alphabet Σ can occur at each position. We describe a way to compute the exact expectation and variance of the number of occurrences of a pattern w in a random string generated by a Markov chain in $O(|w|^2)$ time, improving a previous result that required $O(2^{|w|})$ time. We then consider the problem of computing expectation and variance of the *motifs* of a string s in an IID text. Motifs are rigid gapped patterns that occur at least twice in s, and in which at most one character from Σ occurs at each position. We study the case in which s is given offline, and an arbitrary motif w of s is queried online. We relate computational complexity to the structure of w and s, identifying sets of motifs that are amenable to $o(|w| \log |w|)$ time online computation after $O(|s|^3)$ preprocessing of s. Our algorithms lend themselves to efficient implementations.

Keywords: gapped patterns, variance, convolution, tiling motifs.

1 Introduction and State of the Art

Given a string $w \in \Sigma^+$ and a random text $Z \in \Sigma^+$, the statistical properties of the occurences of w as a *substring* of Z have been extensively studied and repeatedly applied to biological sequences (see e.g. [1] and references therein). Quantities of interest are typically the number of occurrences, the waiting time before the first occurrence, *r-scans* (the distance between an occurrence and the r-th next one), and corresponding quantities applied to higher-order structures, like *renewals* and *clumps* (maximal sets of overlapping occurrences). Traditionally, the focus has been on producing exact closed forms of the distribution and moments of such quantities or of corresponding asymptotic approximations, and bounds on approximation error. Comparatively little is known about the *algorithmic* aspects of computing such quantities.

[⋆] Current affiliation: Helsinki Institute for Information Technology (HIIT), Department of Computer Science, University of Helsinki, Finland.

When Z is generated by an IID source, the expected value and variance of the number of occurrences of all prefixes of w in Z can be derived in overall $O(|w|)$ time, by embedding the computation in a landmark string searching algorithm for constructing the longest border of all prefixes of w [2]. This technique, combined with the linear-time construction of the suffix tree of a string s, allows to score and discover all significantly overrepresented and underrepresented substrings of s in overall $O(|s|)$ time, assuming that the measure of statistical significance f satisfies $w \equiv wx \Rightarrow f(w) \leq f(wx)$ for any $x \in \Sigma^+$, where $\equiv$ means left-equivalence in s [3, 4]. Similar dynamic programming schemes apply to strings with *mismatches*: the expected number of occurrences of a string w with up to k mismatches in an IID text Z can be computed in $O(k^2)$ time after a $O(k|w|)$ preprocessing of w [5]. A related algorithm allows to compute the expectation of all substrings of w with prescribed length in $O(k|w|)$ time, both for IID and for Markov sources [5, 6].

Measuring the statistical significance of strings with *gaps* or *don't cares* is a core primitive in computational biology [7]. Some natural ways to model gaps are requiring a pattern to occur as a *subsequence* of a given text with local [8] or global [9] flexibility constraints, representing a pattern as a *regular expression* [10], and allowing gaps to be flexible but forcing them to occur only in the middle of the pattern [11]. Closed-form, fast approximations have been proposed for the expected number of distinct *rigid maximal motifs* with length ℓ, with k solid characters, and with exactly n occurrences in an IID string Z [12]. Intuitively, a rigid maximal motif (to be defined in Section 4) is a pattern with rigid gaps that occurs at least two times in Z, and that cannot be made more specific without losing support [13]. Most significance scores are monotonically nondecreasing with respect to motif specification, thus maximal motifs usually have the greatest significance among all motifs with the same support, and at the same time they embed any motif with exactly the same support and score (see e.g. [14] and references therein). Approximations like those in [12], or even simpler ones, back popular motif discovery tools (see e.g. [15, 16]), but crucially rely on the assumption that the occurrences of a motif w in Z are independent.

Independence is waived in [17], which gives exact formulas for the expectation and variance of the number of occurrences of a rigid gapped pattern with symbols in $\Gamma \subset 2^{\Sigma}$ in a random string generated by a Markov chain. These formulas will be detailed in Section 2; here we just remark that they are used at the core of a popular algorithm that discovers transcription factor binding sites in DNA [18], and that the kernels of such computations iterate over a number of strings that grows exponentially in the length of the pattern, thus limiting this approach to very small queries. In Section 3 we describe a simple observation that brings the running time of the formulas in [17] from $O(2^{|w|})$ to $O(|w|^2)$. In the IID case, the time to compute such formulas is dominated by convolution, and thus belongs to $O(|w| \log |w|)$. Given a string s provided offline, Section 4 studies the problem of computing expectation and variance for an arbitrary *motif* w of s

provided *online*. We relate computational complexity to the structure of w and to the basis of *tiling motifs* of s, and we identify sets of rigid gapped motifs whose variance can be computed in less than $O(|w| \log |w|)$ time. The key idea behind our construction is reusing a suitable set of convolutions performed offline.

2 Notation and Problem Definition

In this section we summarize the algorithm described in [17], highlighting its computational kernels. For clarity of presentation, our notation differs slightly from [17]. Let Σ be a finite alphabet, and let $\Gamma \subset 2^{\Sigma} \setminus \emptyset$ be such that $\{a\} \in \Gamma$ for all $a \in \Sigma$. In what follows, we will assume that $|\Sigma|$ and $|\Gamma|$ are bounded by a constant. We call *pattern* any string $s \in \Gamma^{+}$, and we say that a position i in s is a *gap* if $s[i] = \Sigma$. Given patterns s and t, we write $s \otimes_k t$ to mean a pattern of length $k+|t|$ with set $s[i] \cap t[i-k]$ at position i. We postulate $s[i] = \Sigma$ for $i \notin [0, |s|-1]$, and we set $s \otimes_k t = \varepsilon$ if $s[i] \cap t[i-k] = \emptyset$ for some i. With $w \dashv s$ we indicate that pattern $w \in \Gamma^{|s|}$ is a copy of pattern s in which every nonsingleton set $G \in \Gamma$ that occurs in s, where $G \neq \Sigma$, has been transformed into a corresponding character $c \in G$: we call w an *instantiation* of s. In other words, an instantiation of pattern s forces all positions of s, except for those occupied by a gap, to equal a symbol in Σ. With $w \prec s$ we indicate that *string* $w \in \Sigma^{|s|}$ is a copy of pattern s in which every set $G \in \Gamma$ that occurs in s has been transformed into a corresponding character $c \in G$, including gaps.

Given a pattern s, we call *selector* a diagonal square matrix $\mathbf{I}(s,i)$ with $|\Sigma|^d$ rows, such that every diagonal element corresponding to a string $w \in \Sigma^d$ with $w \prec s[i, i+d-1]$ is equal to one, and all other elements are zero. We overload the term *selector* to include vectors as well: $\mathbf{e}(s,i)$ is a vector with $|\Sigma|^d$ components, such that every component corresponding to a string $w \in \Sigma^d \ : \ w \prec s[i, i+d-1]$ is one, and all other components are zero. Whether we will be referring to matrices or vectors will be clear from the context. Clearly $\mathbf{e}(s,i)$ (respectively, $\mathbf{e}(s,i)'$) is a right (respectively, left) eigenvector of $\mathbf{I}(s,i)$ associated with eigenvalue 1.

Recall that our purpose is computing expectation and variance of the number of occurrences of a pattern in a random string. Consider thus a Markov chain of order d with matrix of transition probabilities $\mathbf{P} \in [0,1]^{|\Sigma|^d \times |\Sigma|^d}$ and stationary distribution $\mathbf{p} \in [0,1]^{|\Sigma|^d}$. In what follows, we will treat d as a constant. Let $\mathbf{v} \in [0,1]^{|\Sigma|^d}$ be a vector of a priori probabilities for d-mers, let Z be a string generated by the Markov chain, and let s be a pattern with $|s| \leq |Z|$. With $p(s|\mathbf{v})$ we denote the probability that s occurs at position $0 \leq i \leq |Z| - |s|$ of Z in the form of one or more of its instantiations, assuming that $\mathbf{v}$ is the probability distribution of d-mers at i. If Z is long enough, it is safe to set $\mathbf{v} = \mathbf{p}$ independent of i (see e.g. [17] and references therein). Given a pattern w, we define the random variable X_w to be the number of occurrences of w in Z, and we set the indicator random variable $X_{w,i}$ to be one iff w occurs at position i in Z in the form of one or more of its instantiations. The expectation of X_s is clearly:

$$\mathbb{E}(X_s) = \sum_{w \in A(s)} (|Z| - |w| + 1) \cdot p(w|\mathbf{p}) \tag{1}$$

where $A(s) = \{w \dashv s\}$ and $p(w|\mathbf{p}) = \mathbf{p}' \cdot \mathbf{I}(w,0) \cdot \left[\prod_{i=1}^{|w|-d} \mathbf{PI}(w,i)\right] \cdot \mathbf{e}(w, |w|-d)$. After standard manipulations, computing the variance of X_s reduces to Equation 1 and to the two following kernels, that relate to overlapping and nonoverlapping occurrences of s, respectively:

$$\begin{aligned}\sum_{i=0}^{|Z|-|s|-1} \sum_{j=i+1}^{|Z|-|s|} \mathbb{E}(X_{s,i}X_{s,j}) &= \sum_{i=0}^{|Z|-|s|-1} \sum_{l=1}^{m} \sum_{v \dashv s} \sum_{w \dashv s} \mathbb{E}(X_{v,i}X_{w,i+l}) \\ &= \sum_{w \in B(s)} (|Z| - |w| + 1) \cdot p(w|\mathbf{p}) \end{aligned} \tag{2}$$

$$\begin{aligned}\sum_{i=0}^{|Z|-2|s|} \sum_{j=i+|s|}^{|Z|-|s|} \mathbb{E}(X_{s,i}X_{s,j}) &= \sum_{i=0}^{|Z|-2|s|} \sum_{l=0}^{|Z|-2|s|-i} \sum_{v \dashv s} \sum_{w \dashv s} \mathbb{E}(X_{v,i}X_{w,i+|s|+l}) \\ &= \sum_{w \in C(s)} (|Z| - |w| + 1) \cdot p(w|\mathbf{p}) \end{aligned} \tag{3}$$

where $m = \min\{|s|-1, |Z|-i-|s|\}$, $B(s) = \{v \otimes_l w \mid v \dashv s, w \dashv s, 1 \le l < |s|\} \setminus \{\varepsilon\}$ is the set of all valid overlaps of two instantiations of s, and $C(s) = \{v \otimes_{|s|+l} w \mid v \dashv s, w \dashv s, 0 \le l \le |Z|-2|s|\}$ is the set of all spaced concatenation of two instantiations of s. Sets $A(s)$, $B(s)$ and $C(s)$ are enumerated explicitly in [17], thus computing Equations 1, 2 and 3 requires time $O(|s| \cdot |\Sigma|^{2d+|s|})$, $O(|s|^2 \cdot |\Sigma|^{2d+|s|})$, and $O(|Z|^2 \cdot |\Sigma|^{2d+|s|})$, respectively, which become $O(|s| \cdot 2^{|s|})$, $O(|s|^2 \cdot 2^{|s|})$, and $O(|Z|^2 \cdot 2^{|s|})$ if we assume $|\Sigma|$ and d to be constants. The exponential dependency of running time on $|s|$ is not a problem in the application domain of [17], where patterns have length approximately 20 and $|\Sigma| = 4$, however it does not allow to scale this approach to longer patterns. In Section 3 we describe a way to compute Equations 1, 2 and 3 in $O(|s|^2)$ time.

3 Gapped Patterns

Equations 1, 2 and 3 can be computed without explicitly iterating over sets $A(s)$, $B(s)$ and $C(s)$. Avoiding the explicit construction of such sets brings both an asymptotic speedup, and the practical advantage of removing string operations altogether from the implementation of the corresponding equations.

Lemma 1. *Let s be a pattern, and let $\mathbf{v}$ be a vector of d-mer probabilities. Then, $p(s|\mathbf{v}) = \mathbf{v}'\mathbf{I}(s,0) \cdot \mathbf{PI}(s,1) \cdot \mathbf{PI}(s,2) \cdots \mathbf{Pe}(s, |s|-d)$.*

Proof. Clearly $\mathbf{I}(s,i) = \sum_{w \dashv s[i,i+d-1]} \mathbf{I}(w,0)$, and the matrices in this sum select disjoint subsets of Σ^d. Similarly, $\mathbf{e}(s,i) = \sum_{w \dashv s[i,i+d-1]} \mathbf{e}(w,0)$, and the vectors in the sum select disjoint subsets of Σ^d. Thus, $\mathbf{v}'\mathbf{I}(s,0)\cdot\mathbf{PI}(s,1)\cdot\mathbf{PI}(s,2)\cdots\mathbf{Pe}(s,|s|-d)$ can be written as:

$$\begin{aligned}&\mathbf{v}'\left(\sum_{w \dashv s[0,d-1]}\mathbf{I}(w,0)\right)\cdot\mathbf{P}\left(\sum_{w\dashv s[1,d]}\mathbf{I}(w,0)\right)\cdots\mathbf{P}\cdot\left(\sum_{\substack{w\dashv s[|s|-d,\\ |s|-1]}}\mathbf{e}(w,0)\right)\\ =&\sum_{\substack{w_0\dashv s[0,d-1],\\ \vdots\\ w_{|s|-1}\dashv s[|s|-d,|s|-1]}}\mathbf{v}'\mathbf{I}(w_0,0)\cdot\mathbf{PI}(w_1,0)\cdots\mathbf{P}\cdot\mathbf{e}(w_{|s|-d},0)\end{aligned}$$

Two selectors $\mathbf{I}(w_i,0)$ and $\mathbf{I}(w_j,0)$ in the sum above are called *incompatible* if either $i<j<i+d$ and $w_i\otimes_{j-i}w_j=\varepsilon$, or $j<i<j+d$ and $w_j\otimes_{i-j}w_i=\varepsilon$. By the structure of $\mathbf{P}$, products containing incompatible selectors do not contribute to the sum, and set $\{(w_0,w_1,\ldots,w_{|s|-d})\mid w_i\otimes_{j-i}w_j\neq\varepsilon\ \forall\ i<j\}$ can be put in one-to-one correspondence with $A(s)$. □

Using Lemma 1, Equation 1 reduces to:

$$(|Z|-|s|+1)\cdot\mathbf{p}'\mathbf{I}(s,0)\cdot\mathbf{PI}(s,1)\cdot\mathbf{PI}(s,2)\cdots\mathbf{Pe}(s,|s|-d) \tag{4}$$

which can be computed in $O(|s|)$ time assuming $|\Sigma|$ and d to be constants. Applying Lemma 1 to Equation 3, we get:

$$\mathbf{p}'\mathbf{I}(s,0)\cdots\mathbf{PI}(s,|s|-d)\cdot\left(\sum_{i=0}^{|Z|-2|s|}\sum_{l=0}^{|Z|-2|s|-i}\mathbf{P}^{l+d}\right)\mathbf{I}(s,0)\cdots\mathbf{Pe}(s,|s|-d)$$

After using the expression for sums of powers of stochastic matrices given in [19], this becomes:

$$\begin{aligned}\mathbf{q}'\mathbf{QP}^{|Z|-2|s|+3}\mathbf{QP}^{d-1}\mathbf{r}-\mathbf{q}'\mathbf{QP}^2\mathbf{QP}^{d-1}\mathbf{r}&+\\ +(|Z|-2|s|+1)\mathbf{q}'\mathbf{QP1p}'\mathbf{P}^{d-1}\mathbf{r}-(|Z|-2|s|+1)\mathbf{q}'\mathbf{QP}^{d}\mathbf{r}&+\\ +\left(\frac{|Z|^2}{2}+2|s|^2-2|s|\cdot|Z|+1+\frac{3}{2}|Z|-3|s|\right)\mathbf{q}'\mathbf{1p}'\mathbf{P}^{d-1}\mathbf{r}&\end{aligned} \tag{5}$$

where $\mathbf{q}'=\mathbf{p}'\mathbf{I}(s,0)\cdot\mathbf{PI}(s,1)\cdots\mathbf{PI}(s,|s|-d)$, $\mathbf{r}=\mathbf{I}(s,0)\cdot\mathbf{PI}(s,1)\cdots\mathbf{Pe}(s,|s|-d)$, $\mathbf{1}$ is the vector of $|\Sigma|^d$ ones, and $\mathbf{Q}=(\mathbf{P}-\mathbf{I}+\mathbf{1p}')^{-1}$ as defined in [19]. Assuming $|\Sigma|$ and d to be constants, this equation can be computed in $O(|Z|)$ time in the current form, or in constant time if we postulate $|Z|\gg|s|$ and approximate $\mathbf{P}^{|Z|-2|s|+3}$ with $\mathbf{1p}'$, as done in [17]. Equation 2 can be similarly computed in $O(|s|^2)$ time:

$$\sum_{\substack{w=s\otimes_k s,\\ 1\le k<|s|}}(|Z|-|s|-k+1)\cdot\mathbf{p}'\mathbf{I}(w,0)\cdot\mathbf{PI}(w,1)\cdot\mathbf{PI}(w,2)\cdots\mathbf{Pe}(w,|w|-d) \tag{6}$$

It is also easy to show that Lemma 1 can speed up the few remaining kernels of [17] that depend on the specific domain of transcription factor binding sites.

When Z is generated by an IID source, Equation 6 becomes the bottleneck of the whole computation, and it assumes the following form:

$$\sum_{\substack{w=s\otimes_{|s|-b}s,\\ b\in\beta(s)}} (|Z|-2|s|+b+1)\cdot \prod_{i=0}^{|s|-b-1}\mathbb{P}(s[i])\cdot \prod_{i=|s|-b}^{|s|-1}\mathbb{P}(w[i])\cdot \prod_{i=b}^{|s|-1}\mathbb{P}(s[i]) \quad (7)$$

where $\beta(s)$ is the set of *borders* of s, i.e. the set of all integers b such that $s = wu = vw'$, $w \otimes_0 w' \neq \varepsilon$, and $|w| = |w'| = b$. The second and fourth terms in the sum can be accessed in constant time after a $O(|s|)$ preprocessing of s. The third term is clearly convolutional, thus it can be accessed in constant time after an overall $O(|s|\log|s|)$ preprocessing of s based on the landmark match-count algorithm by Fischer and Paterson [20] (or by using one of its randomized variants, e.g. [21]). In the next section, we study ways to bring the complexity of Equation 7 below $O(|s|\log|s|)$ when s is a *motif* of a string provided offline.

4 Motifs

In many applications we are given a fixed text $s \in \Sigma^+$ provided offline, and we are asked to compute the expectation and variance of *arbitrary* patterns provided *online*. This scenario models popular websites that allow to search for biologically significant patterns in genomes and proteomes (e.g. [22]), and it captures the post-processing stage of most pattern-discovery algorithms, which rank their results according to statistical significance (e.g. [15]). In what follows we will focus on computing Equation 7 in the IID case. The main intuition behind performing less than $O(|w|\log|w|)$ operations for a pattern w given online consists in moving some convolutions offline, and in reusing such convolutions at query time with the help of suitable data structures.

Given a string $w \in \Gamma^+$, we define $\sharp_{w,a}[i]$ to be the number of positions in which $w \otimes_i w$ equals $a \in \Gamma$, $0 \leq i < |w|$. We define $\sharp_{w,a}[i,j,k]$ to be the number of positions in which $w[i,i+k-1]\otimes_0 w[j,j+k-1]$ equals $a \in \Gamma$, $0 \leq i,j \leq |w|-k$. Finally, we define $\sharp_{s,t,a}[i]$ to be the number of positions in which $s \otimes_i t$ equals $a \in \Gamma$, $-|t|+1 \leq i < |s|$. To simplify notation, we use symbol $\bullet$ to denote set Σ, and we indicate with $||w||$ the number of positions in which w is different from $\bullet$. We first study ways in which the convolution of w can be reused to compute the convolution of its prefixes and suffixes.

Lemma 2. *Let $w \in \Sigma(\Sigma \cup \{\bullet\})^*\Sigma$, and assume that $\sharp_{w,a}[i]$ is known for every $1 \leq i < |w|$ and every $a \in \Sigma$. Then, the value of $\sharp_{w_k,a}[i]$ for $1 \leq i \leq k$ and $a \in \Sigma$ can be computed for all prefixes $w_k = w[0,k]$ of w (similarly, the value of $\sharp_{w_k,a}[i]$ for $1 \leq i < |w|-k$ and $a \in \Sigma$ can be computed for all suffixes $w_k = w[k,|w|-1]$ of w) in overall optimal $O(|w|^2)$ time and space and in overall $O(||w||\cdot|w|)$ arithmetic operations.*

	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	-1	-1	0	0	-1	1	-1	-1
2		0	0	-2	-1	0	1	-2	0	-1	-1
3			0	-2	-2	0	1	-1	0	-1	-1
4				-1	-2	0	0	-1	1	-2	-1
5					-1	0	0	-1	1	-1	-2
6						0	0	-2	0	-1	-1
7							0	-2	0	-1	-1
8								-1	0	-2	-1
9									0	-2	-2
10										-1	-2
11											-1

Fig. 1. Matrix $\mathbf{T}_a$ of Lemma 2 for string $ab\bullet\bullet baa\bullet babaa$. Light gray highlights column 11 and its shifts.

Proof. For a generic offset i, $\sharp_{w[0,|w|-2],a}[i] = \sharp_{w,a}[i] + \tau_{a,i,|w|-2}$, where:

$$\tau_{a,i,j} = -\sharp_{w,a}[j+1,j+1,1] - \sharp_{w,a}[j+1,j-i+1,1] + \sharp_{w,a}[j-i+1,j-i+1,1]$$

In particular, $w[j+1] = \bullet$ implies $\tau_{a,i,j} = 0$ for all i and $a \in \Sigma$. Let $\mathbf{T}_a$ be an upper-triangular matrix with $|w|-2$ rows and columns, indexed starting from one, in which row i corresponds to offset i, column j corresponds to prefix $w[0,j]$, and $\mathbf{T}_a[i,j] = \tau_{a,i,j}$. Matrix $\mathbf{T}_a$ is filled in column-major order, starting from column $|w|-2$. It is easy to see that $\mathbf{T}_a$ has a regular structure (Figure 1). First, as mentioned above, since $w[|w|-1] \in \Sigma$, the fact that $\mathbf{T}_a[i,|w|-2] = -2$ implies that $\mathbf{T}[k,|w|-i-2] = 0$ for all $1 \leq k \leq |w|-i-2$. Second, let $j_a = \max\{j : w[j] = a, 0 \leq j < |w|\}$; then, every column $\mathbf{T}_a[:,j]$ such that $w[j+1] = a$ equals column $\mathbf{T}[:,j_a-1]$ shifted up by $j_a - j - 1$ cells. Third, there is only one other type of column in $\mathbf{T}_a$, not considering shifts and columns that are identically zero: the column corresponding to symbols different from a, which contains only zeros and ones, and appears sequentially shifted up (for $j < j_a - 1$) and down (for $j > j_a - 1$) as described above. We can thus compute these two types of column in $O(|w|)$ time and space, and store them rather than $\mathbf{T}_a$ itself in practice. To compute $\sharp_{w[0,k],a}$ for every k, traverse the columns of $\mathbf{T}_a$ from right to left, keeping a running sum of the cells associated with every row i. If $\mathbf{T}_a[i,j] = 0$, then the corresponding $\sharp_{w[0,j],a}[i]$ is just copied from $\sharp_{w[0,j+1],a}[i]$. $\square$

Lemma 2 generalizes easily to suffixes and to strings in Γ^+. Given a string w, in what follows we will call $\mathcal{P}_{w,a}$ the upper-triangular matrix with $|w|-2$ rows and columns, indexed starting from one, in which row i corresponds to offset i, column j corresponds to prefix $w[0,j]$, and $\mathcal{P}_{w,a}[i,j] = \sharp_{w[0,j],a}[i]$. We similarly define $\mathcal{S}_{w,a}$ for suffixes. $\mathcal{P}_{w,a}$ and $\mathcal{S}_{w,a}$ can be used as indexes to answer questions on arbitrary substrings of w in linear time[1].

Lemma 3. *Let $w \in \Gamma^+$. After $O(|w|^2)$ preprocessing, we can compute the following quantities: (1) $\sharp_{v,a}[k]$ for all $1 \leq k < |v|$ and $a \in \Gamma$ in $O(|v|)$ time, for any substring v of w; (2) $\sharp_{w_1,w_2,a}[i]$ for all $-|w_2|+1 \leq i < |w_1|$ and $a \in \Gamma$ in $O(|w_1|+|w_2|)$ time, for any pair of substrings w_1 and w_2 of w.*

Proof. (1) Build $\mathcal{P}_{w,a}$ in $O(|w|^2)$ time. Then, build the suffix tree $\mathcal{T}_w$ of w, and assign to every internal node the starting position of one of the suffixes in its subtree. This can be done in overall $O(|w|)$ time. Given a substring v of w, find its proper locus in $\mathcal{T}_w$, extract the associated starting position i in w, set $j = i + |w| - 1$, and apply the following identity to every $a \in \Gamma$ (see Figure 2a):

$$\sharp_{w[i,i+|w|-1],a}[k] = \sharp_{w[0,i+|w|-1],a}[k] - \sharp_{w[0,i+k-1],a}[k] + 2 \cdot \sharp_{w,a}[i,i,k] \tag{8}$$

The first two terms in the right-hand side can be computed from $\mathcal{P}_{w,a}$, and the last term can be accessed in constant time after $O(|w|)$ preprocessing of w. The equation above could be set up for using $\mathcal{S}_{w,a}$ rather than $\mathcal{P}_{w,a}$. Notably, just one of $\mathcal{P}_{w,a}$ and $\mathcal{S}_{w,a}$ suffices to answer queries on single substrings. (2) Preprocess w as above. Let i_1 and i_2 be the starting positions of w_1 and w_2 in w, respectively. For clarity of presentation, we describe the formula for the case $0 < k < i_2 - i_1 - |w_1|$ (see Figure 2b), leaving the general case to the reader. Let $k' = i_2 - i_1 - k$ and $i^* = i_1 + |w_1|$. Then:

$$\begin{aligned}
\sharp_{w_1,w_2,a}[k] = \sharp_{w,a}[k'] + & \\
& -\sharp_{w[0,i_2-1],a}[k'] + \sharp_{w,a}[i_1+k, i_1+k, k'] + \sharp_{w,a}[i_1, i_1, k] + && (9)\\
& -\sharp_{w[i_1+|w_1|,|w|-1],a}[k'] + \sharp_{w,a}[i_1+|w_1|, i_1+|w_1|, k'] + && (10)\\
& +\sharp_{w,a}[i^*+k', i^*+k', i_2+|w_2|-i^*-k'] && (11)
\end{aligned}$$

where all terms can be accessed in constant time by either querying $\mathcal{P}_{w,a}$ and $\mathcal{S}_{w,a}$, or by accessing values that have been precomputed in $O(|w|)$ time. □

In what follows, we will also be interested in computing $\sharp_{v,a}[i]$ for a string v that is *less specific* than a known string w. Given a string $w \in \Gamma^+$, we say that string $v \in \Gamma^{|w|}$ is *less specific* than w (or, equivalently, that v is a *sparsification* of w) if $w[i] \subseteq v[i]$ for all $0 \leq i < |w|$, and if $w[i^*] \subset v[i^*]$ in at least one position i^*. When the maximum distance between two sparsified positions of v is bounded by a sublinear function of $|w|$, computing the convolution of v by exploiting the convolution of w is asymptotically faster than computing the convolution of v with no prior information.

Lemma 4. *Let w be a string in $(\Sigma \cup \{\bullet\})^+$, and let $v \in (\Sigma \cup \{\bullet\})^{|w|}$ be a sparsification of w such that $v[i] = w[i]$ for all $0 \leq i < |w|$, except for a set of positions $V = \{i_0, i_1, \ldots, i_{k-1}\}$ where $w[i_j] \subset v[i]$, $0 \leq j < k$. We can compute $\sharp_{v,a}[i]$ from $\sharp_{w,a}[i]$ for all offsets i in overall $O(|v| \log(i_{k-1} - i_0))$ time.*

[1] They thus recall the *match matrix* described in [23].

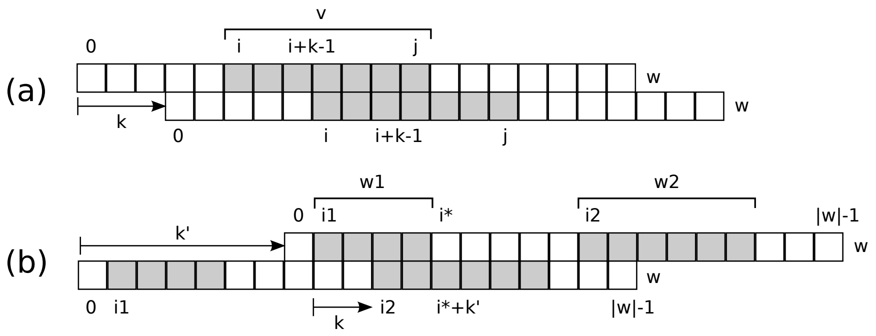


Fig. 2. Illustrating Lemma 3. Reusing the convolution of a string w to compute the convolution of a substring v (a) and of two substrings w_1 and w_2.

Proof. By applying the convolution strategy of the match-count algorithm.

Lemma 4 can be easily generalized to strings in Γ^+, and it can be extended to *pairs* of strings w_1, w_2 with different sparsifications, at cost $O((|w_1| + |w_2|)\log(|w_1| + |w_2|))$. Another natural way to constrain the sparsification of w is forcing sparsified positions to occur inside a contiguous interval.

Lemma 5. *Let w be a string in $(\Sigma \cup \{\bullet\})^+$, and let $v \in (\Sigma \cup \{\bullet\})^{|w|}$ be a sparsification of w such that $v[i] = w[i]$ for all $0 \leq i < |w|$, except possibly at positions $V = \{d, d+1, \ldots, d+k-1\}$ where $v[d+i] = \bullet$, $0 \leq i < k$. After $O(||w|| \cdot |w|)$ preprocessing, we can compute $\sharp_{v,a}[i]$ for all i in $O(|v|)$ time.*

Proof. Let $\overrightarrow{\sharp}_{w,a}[x,y,z]$ be the number of positions in $w[x, x+z-1] \otimes_0 w[y, y+z-1]$ that have an a in $w[x, x+z-1]$ and a gap in $w[y, y+z-1]$. Such values can be computed for w using convolution, then they can be propagated to all prefixes and suffixes of w following a strategy similar to the proof of Lemma 2. The resulting matrices support substring queries as described in Lemma 3, thus enabling the computation of the following correction in constant time:

$$\sharp_{v,a}[i] = \sharp_{w,a}[i] - \sharp_{w,a}[d+i, d, k-i] - \overrightarrow{\sharp}_{w,a}[d+k-i, d+k, i] - \overrightarrow{\sharp}_{w,a}[d, d-i, i]$$

□

Recall that the purpose of this section is preprocessing a given text $s \in \Sigma^+$ provided offline to compute the expectation and variance of arbitrary patterns $w \in \Gamma^+$ provided *online*. From now on, we will restrict to a specific class of patterns, called *motifs*. Given a string $s \in \Sigma^+$, a motif is a string $w \in \Sigma(\Sigma \cup \{\bullet\})^*\Sigma$ that occurs at least two times in s [13]. The number of distinct motifs in a string s grows exponentially with $|s|$. Among all motifs of s, a notable subset cannot be intuitively made "more specific" without losing support.

Definition 1 (Maximal motif [13]). *Let w be a motif occurring at positions $\mathcal{L}(w) = \{i_0, i_1, \ldots, i_{n-1}\}$ in a string $s \in \Sigma^+$, $n \geq 2$. We say that w is* maximal in composition *if no other motif $v \neq w$ of s has $\mathcal{L}(v) = \mathcal{L}(w)$ and $v[i] \subseteq w[i]$ for all $i \in \{0, \dots, |w|-1\}$. We say that w is* maximal in length *if no other motif $v \neq w$ of s is such that $|\mathcal{L}(v)| = |\mathcal{L}(w)|$ and w is a substring of v. We say that w is a* maximal motif *of s if it is both maximal in composition and maximal in length.*

Unfortunately, even the number of maximal motifs can grow exponentially in $|s|$. A landmark result in pattern discovery states that the subset of *tiling* maximal motifs is bounded by a linear function of $|s|$ [13, 24].

Definition 2 (Tiling motif [24]). *A maximal motif w of a string s is* tiled *is there exist maximal motifs $w_0, w_1, \dots, w_{n-1}$ of s ($w_i \neq w \; \forall \; i$) and integers $d_0, d_1, \dots, d_{n-1}$ such that $\mathcal{L}(w) = \bigcup_{i=0}^{n-1} \mathcal{L}(w_i) + d_i$. We call* tiling *a maximal motif of s that is not tiled.*

The set of tiling motifs of s, together with their occurrence lists, contains sufficient information to generate any other maximal motif in s and its occurrences, without knowing s itself [13, 25]. It is thus standard to call this set a *basis*: in what follows, we will denote it with $\mathcal{B}_s$. We are interested here in the mechanism by which the basis generates a motif of s.

Fact 1 ([25]). *The motifs of s are all and only the strings in $\Sigma(\Sigma \cup \{\bullet\})^*\Sigma$ that can be obtained as follows: (1) take a substring of a tiling motif that starts and ends with a character in Σ; (2) replace an arbitrary set of solid characters (excluding the first and last ones) with gaps.*

This fact, combined with the sparsification tools described above, will be the core of our construction. Before describing the main result of this section, however, we need one last piece of notation.

Definition 3. *The* tiling factorization *of a string $w \in (\Sigma \cup \{\bullet\})^+$ induced by a string $s \in \Sigma^+$ is the decomposition $w = u_0 \bullet^{d_0} u_1 \bullet^{d_1} \cdots \bullet^{d_{k-2}} u_{k-1}$, $d_i > 0 \; \forall \; 0 \leq i < k-1$, where each u_i starts with a solid character, and is the shortest prefix of $u_i \bullet^{d_i} u_{i+1} \bullet^{d_{i+1}} \dots u_{k-1}$ that* matches *a maximal substring v of $\mathcal{B}_s$. By "matching" we mean that $v[j] \subseteq u_i[j]$ for all $0 \leq j < |v|$.*

We are now ready to state our main theorem.

Theorem 1. *Let $s \in \Sigma^+$, let w be a motif of s provided online, and let $w = u_0 \bullet^{d_0} u_1 \bullet^{d_1} \cdots \bullet^{d_{k-2}} u_{k-1}$ be the tiling factorization of w induced by s. After a $O(|s|^3)$ offline preprocessing of s, we can compute $\sharp_{w,a}[i]$ for all $i \in \{1, \dots, |w|-1\}$ and all $a \in \Sigma$ in worst-case time:*

$$O\left(\sum_{i=0}^{k-1}\sum_{j=i+1}^{k-1}(|u_i|+|u_j|)\log(|u_i|+|u_j|) + \sum_{i=0}^{k-1}|u_i|\log|u_i|\right)$$

Proof. Build $\mathcal{B}_s = \{t_0, t_1, \dots, t_{|\mathcal{B}_s|-1}\}$ in $O(|s|^2 \log |s|)$ time [24]: the result is a set of $O(|s|)$ tiling motifs of length $O(|s|)$ each [24]. For all $i \in \{0, \dots, |\mathcal{B}_s| - 1\}$, compute the convolution of t_i with itself in $O(|s|^2 \log |s|)$ time overall. Build matrices $\mathcal{P}_{t_i,a}$ and $\mathcal{S}_{t_i,a}$ for every $a \in \Sigma$ using Lemma 2, in overall $\sum_{i=0}^{|\mathcal{B}_s|} ||t_i|| \cdot |t_i| \in O(|s|^3)$ time and space. At the same cost, build the matrices used by Lemma 5. Then, build in $O(|s|^2)$ time the generalized suffix tree $\mathcal{T}_s$ of the strings in $\mathcal{B}_s$, treating $\bullet$ as different from every other symbol in Σ. In what follows, we will decorate the nodes of $\mathcal{T}_s$ with additional information that will help answering online queries. For clarity, given a tree $\mathcal{T}$, we will denote with $\mathcal{T}[\alpha]$ the value stored at node α of $\mathcal{T}$. First, we initialize a digital search tree $\mathcal{Q}_s$ with height two. For every tiling motif $t_i \in \mathcal{B}_s$, let $\mathcal{T}_i$ be its corresponding suffix tree. We set $\mathcal{T}_i[\alpha] = j$ for every node α in $\mathcal{T}_i$, where j is a position at which the substring of t_i associated with α occurs in t_i. This can be done in $O(|s|)$ time. Then, we mark all nodes α of $\mathcal{T}_s$ that correspond to nodes of $\mathcal{T}_i$ in $O(|s^2|)$ time, by traversing $\mathcal{T}_i$ and $\mathcal{T}_s$ top-down in parallel. Let α be a node of $\mathcal{T}_s$ that corresponds to node $\bar{\alpha}$ in $\mathcal{T}_i$: we set $\mathcal{T}_s[\alpha] = (i, \mathcal{T}_i[\bar{\alpha}])$. Similarly, let α and β be two nodes of $\mathcal{T}_s$ that correspond to nodes $\bar{\alpha}$ and $\bar{\beta}$ in $\mathcal{T}_i$, respectively. Then, we add to $\mathcal{Q}_s$ strings $\alpha\beta$ and $\beta\alpha$, and we store at the corresponding leaves of $\mathcal{Q}_s$ the triplet $(i, \mathcal{T}_i[\bar{\alpha}], \mathcal{T}_i[\bar{\beta}])$. This can be done in $O(|s|^2)$ time. $\mathcal{T}_i$ is then discarded and we proceed to the next i. The overall preprocessing of s thus takes $O(|s|^3)$ time and space.

Let now w be a motif of s provided online. Follow w in $\mathcal{T}_s$: if w is a substring of $\mathcal{B}_s$, then it has a proper locus α in $\mathcal{T}_s$, and $\mathcal{T}_s[\alpha]$ is sufficient to compute $\sharp_{w,a}[i]$ for all i and a in $O(|w|)$ time using Lemma 3. If w is not a substring of $\mathcal{B}_s$, then there is a position $0 \leq i < |w|$ such that $w[i] = \bullet$ and one of the two following cases occurs: (1) the current position in the suffix tree lies inside an edge, but $\bullet$ is not the next symbol in the suffix tree: in this case, we continue matching w; (2) the current position in the suffix tree is a node: this node corresponds to string v_0, the shortest maximal substring of $\mathcal{B}_s$ that matches prefix u_0 of w. We then continue reading from the next solid character of w starting from the root of $\mathcal{T}_s$, thus finding substrings $v_1, v_2, \dots, v_k$ of $\mathcal{B}_s$ that match $u_1, u_2, \dots, u_k$, respectively. In the worst case, the value of $\sharp_{u_i,a}[j]$ for $1 \leq j < |u_i|$ can be computed in $O(|u_i| \log |u_i|)$ time from the information stored at the node of $\mathcal{T}_s$ that corresponds to v_i, using Lemma 4. To compute $\sharp_{w,a}[i]$ we also need to know $\sharp_{u_i,u_j,a}[h]$ for $0 \leq i < j < k$. Let α_i and α_j be the nodes of $\mathcal{T}_s$ that correspond to v_i and v_j, respectively. Fact 1 and the structure of $\mathcal{T}_s$ guarantee that there is at least one string in $\mathcal{B}_s$ in which both v_i and v_j occur, thus string $\alpha_i\alpha_j$ must occur in $\mathcal{Q}_s$: using the information returned by $\mathcal{Q}_s$ and Lemma 3, we can thus access $\sharp_{v_i,v_j,a}[h]$ in constant time for any h. The value of $\sharp_{u_i,u_j,a}[h]$ can then be derived using natural adaptations of Lemmas 4 and 5 to pairs of strings. □

For arbitrary motifs, the worst-case online running time of Theorem 1 is never asymptotically faster than the convolution of motif w with itself: for example, if k is bounded by a constant, the worst-case running time is $O(|w| \log |w|)$; if k is $O(\log |w|)$ the worst-case running time is $O(|w|(\log |w|)^3)$; and if k is $O(|w|)$, the worst-case running time is $O(|w|^2)$. However, Theorem 1 establishes a previously unknown connection between the time to compute $\sharp_{w,a}[i]$ for a motif w of a string

s ($a \in \Sigma$, $1 \leq i < |w|$) and the structure of w and s. The following corollary, that derives immediately from Theorem 1 and Lemma 5, identifies families of motifs that are amenable to $o(n\log(n))$ online processing, where n is the length of the motifs in such families.

Corollary 1. *Let s be a string, let w be a motif of s, let $w = u_0 \bullet^{d_0} u_1 \bullet^{d_1} \cdots \bullet^{d_{k-2}} u_{k-1}$ be the tiling factorization of w induced by s, and let $v_0, v_1, \ldots, v_{k-1}$ be the corresponding substrings of $\mathcal{B}_s$ that match the factors of w. A* block *in a factor u_i, $0 \leq i < k$, is a substring $u_i[d, d+\ell-1]$ such that $(u_i[j] = \bullet) \wedge (v_i[j] \neq \bullet)$ for $j = d$ and $j = d+\ell-1$, and such that $u_i[j] = \bullet$ for $d < j < d+\ell-1$. A block is* maximal *if it is not contained in any other block. If b_i, the number of maximal blocks in factor u_i, is bounded by a constant for all i, then we can compute $\sharp_{w,a}[i]$ for all $i \in \{1, \ldots, |w|-1\}$ and all $a \in \Sigma$ in $O(k|w|)$ time after $O(|s|^3)$ preprocessing. Similarly, if k is bounded by a constant, then we can compute $\sharp_{w,a}[i]$ for all $i \in \{1, \ldots, |w|-1\}$ and all $a \in \Sigma$ in $O(|w|\sum_{i=0}^{k-1} b_i)$ time after $O(|s|^3)$ preprocessing.*

5 Discussion and Extensions

Speeding up the computation of statistical properties of gapped patterns is crucial in large-scale molecular biology and text mining. The implicit technique used in Section 3 to bring the computational complexity of expectation and variance of a gapped pattern w from $O(2^{|w|})$ to $O(|w|^2)$ has the desirable practical effect of limiting string operations to the construction of selectors, thus keeping only matrix and vector operations in the kernels. This likely allows to take better advantage of existing software libraries and hardware in a practical implementation of such equations. The offline preprocessing of Theorem 1 is clearly amenable to multiple levels of parallelization and tuning, which could help reducing its cubic running time in practice. Similarly, the factorization of a motif w provided online defines a formal way to process w in parallel and to aggregate partial results.

From an algorithmic standpoint, Lemma 1 is likely applicable to the *conditional* expectation and variance of a pattern given the occurrences of others (see e.g. [26]), and to the expectation and variance of a *set* of patterns allowed to overlap each other. The setup in Theorem 1 heavily relies on having string s available offline: assuming that even s is given online would be more realistic in applications like security and logging, and it would probably require a completely different set of data structures and algorithms. The tiling factorization of a motif seems also a notion of independent interest, and resonates with ideas in conditional algorithmic information and data compression (e.g. [27]): it would be interesting to push this relationship further, for example by explicitly relating the time to compute the variance of a motif to a measure of mutual algorithmic information between the motif and the basis. Finally, embedding the efficient computation of expectation and variance into existing algorithms that produce *all* motifs of a string from its tiling basis (e.g. [28]) could be another stimulating extension of this work.

Acknowledgements. Partially funded by the Academy of Finland under grant 250345.

References

[1] Reinert, G., Schbath, S., Waterman, M.: Probabilistic and statistical properties of words: an overview. Journal of Computational Biology 7, 1–46 (2000)

[2] Apostolico, A., Bock, M., Xu, X.: Annotated statistical indices for sequence analysis. In: Proceedings of the Compression and Complexity of Sequences, Sequences 1997, pp. 215–229. IEEE Computer Society, Washington, DC (1997)

[3] Apostolico, A., Bock, M., Lonardi, S.: Monotony of surprise and large-scale quest for unusual words. In: Proceedings of the Sixth Annual International Conference on Computational Biology, RECOMB 2002, pp. 22–31. ACM, New York (2002)

[4] Apostolico, A., Bock, M., Lonardi, S., Xu, X.: Efficient detection of unusual words. Journal of Computational Biology 7(1), 71–94 (2000)

[5] Apostolico, A., Pizzi, C.: Monotone Scoring of Patterns with Mismatches. In: Jonassen, I., Kim, J. (eds.) WABI 2004. LNCS (LNBI), vol. 3240, pp. 87–98. Springer, Heidelberg (2004)

[6] Pizzi, C., Bianco, M.: Expectation of Strings with Mismatches under Markov Chain Distribution. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 222–233. Springer, Heidelberg (2009)

[7] Ferreira, P., Azevedo, P.: Evaluating deterministic motif significance measures in protein databases. Algorithms for Molecular Biology 2(1), 16 (2007)

[8] Flajolet, P., Guivarc'h, Y., Szpankowski, W., Vallée, B.: Hidden Pattern Statistics. In: Yu, Y., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 152–165. Springer, Heidelberg (2001)

[9] Gwadera, R., Atallah, M., Szpankowski, W.: Reliable detection of episodes in event sequences. In: Knowledge and Information Systems, pp. 67–74 (2004)

[10] Nicodème, P., Salvy, B., Flajolet, P.: Motif statistics. Theoretical Computer Science 287, 593–617 (2002)

[11] Robin, S., Daudin, J.J., Richard, H., Sagot, M.F., Schbath, S.: Occurrence probability of structured motifs in random sequences. Journal of Computational Biology, 761–774 (2002)

[12] Stolovitzky, G., Califano, A.: Statistical significance of patterns in biosequences. IBM research report (1998)

[13] Parida, L., Rigoutsos, I., Floratos, A., Platt, D., Gao, Y.: Pattern discovery on character sets and real-valued data: linear bound on irredundant motifs and an efficient polynomial time algorithm. In: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2000, pp. 297–308. Society for Industrial and Applied Mathematics, Philadelphia (2000)

[14] Apostolico, A., Comin, M., Parida, L.: Conservative extraction of over-represented extensible motifs. Bioinformatics 21, i9–i18 (2005)

[15] Califano, A.: SPLASH: structural pattern localization analysis by sequential histograms. Bioinformatics 16, 341–357 (2000)

[16] Rigoutsos, I., Floratos, A.: Combinatorial pattern discovery in biological sequences: the TEIRESIAS algorithm. Bioinformatics 14(1), 55–67 (1998)

[17] Sinha, S., Tompa, M.: A statistical method for finding transcription factor binding sites. In: Proc. Int. Conf. Intell. Syst. Mol. Biol., vol. 8, pp. 344–354 (2000)

[18] Sinha, S., Tompa, M.: Discovery of novel transcription factor binding sites by statistical overrepresentation. Nucleic Acids Research 30(24), 5549–5560 (2002)
[19] Kleffe, J., Borodovsky, M.: First and second moment of counts of words in random texts generated by Markov chains. Bioinformatics/Computer Applications in the Biosciences 8, 433–441 (1992)
[20] Fischer, M., Paterson, M.: String-matching and other products. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1974)
[21] Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing, STOC 2002, pp. 592–601. ACM, New York (2002)
[22] Sigrist, C., Cerutti, L., de Castro, E., Langendijk-Genevaux, P., Bulliard, V., Bairoch, A., Hulo, N.: PROSITE, a protein domain database for functional characterization and annotation. Nucleic Acids Research 38, 161–166 (2010)
[23] Apostolico, A., Parida, L.: Incremental paradigms of motif discovery. Journal of Computational Biology 11, 15–25 (2004)
[24] Pisanti, N., Crochemore, M., Grossi, R., Sagot, M.-F.: A Basis of Tiling Motifs for Generating Repeated Patterns and Its Complexity for Higher Quorum. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 622–631. Springer, Heidelberg (2003)
[25] Pisanti, N., Crochemore, M., Grossi, R., Sagot, M.: Bases of motifs for generating repeated patterns with wildcards. IEEE/ACM Transactions on Computational Biology and Bioinformatics 2(1), 40–50 (2005)
[26] Blanchette, M., Sinha, S.: Separating real motifs from their artifacts. Bioinformatics 17(1), S30–S38 (2001)
[27] Lempel, A., Ziv, J.: On the complexity of finite sequences. IEEE Transactions on Information Theory 22(1), 75–81 (1976)
[28] Parida, L., Rigoutsos, I., Platt, D.: An Output-Sensitive Flexible Pattern Discovery Algorithm. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 131–142. Springer, Heidelberg (2001)

Enhancing the Computation of Distributed Shortest Paths on Real Dynamic Networks*

Gianlorenzo D'Angelo[1], Mattia D'Emidio[2], Daniele Frigioni[2], and Daniele Romano[2]

[1] MASCOTTE Project INRIA/I3S(CNRS/UNSA) 2004 Route Des Lucioles, 06902 Sophia–Antipolis Cedex, France
gianlorenzo.d_angelo@inria.fr

[2] Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Via Gronchi 18, I–67100, L'Aquila, Italy
{mattia.demidio,daniele.frigioni}@univaq.it, daniele.romano.vis@gmail.com

Abstract. The problem of finding and updating shortest paths in distributed networks is considered crucial in today's practical applications. In the recent past, there has been a renewed interest in devising new efficient distance-vector algorithms as an attractive alternative to link-state solutions for large-scale Ethernet networks. In this paper we present *Distributed Computation Pruning* (DCP), a new technique, which can be combined with every distance-vector algorithm based on shortest paths, allowing to reduce the total number of messages sent by that algorithm and its space occupancy per node. To check its effectiveness, we combined DCP with DUAL (Diffuse Update ALgorithm), one of the most popular distance-vector algorithm in the literature, and with the recently introduced LFR (Loop Free Routing) which has been shown to have good performances on real networks. We give experimental evidence that these combinations lead to a significant gain both in terms of number of messages sent and memory requirements per node.

1 Introduction

The problem of computing and updating shortest paths in a distributed network whose topology dynamically changes over the time is a core functionality of today's communication networks. This problem has been widely studied in the literature, and the solutions found are classified as *distance-vector* and *link-state*. Distance-vector algorithms require that a node knows the distances from each of its neighbors to every destination and stores them in a data structure called *routing table*; a node uses its own routing table to compute the distance and the next node in the shortest path to each destination. Most of the known distance-vector solutions (see e.g. [3,5]) are based on the classical Distributed Bellman-Ford method (DBF), which is still used in real networks and implemented in

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

the RIP protocol [15]. DBF has been shown to converge to the correct distances if the link weights stabilize and all cycles have positive lengths [2]. However, the convergence can be very slow (possibly infinite) due to the well-known *looping* phenomenon. Link-state algorithms, as for example the *Open Shortest Path First (OSPF)* protocol used in the Internet [11], require a node to know the entire network topology, to compute its distance to any destination, usually running the Dijkstra's algorithm, thus requiring quadratic space per node. Link-state algorithms are free of looping, however each node needs to receive and store up-to-date information on the entire network topology after a change. This is achieved by broadcasting each change of the network topology to all nodes [11] and by using a dynamic centralized algorithm for shortest paths, as for example that in [8].

Related Works. In the last years, there has been a renewed interest in devising new efficient light-weight distributed shortest paths solutions for large-scale Ethernet networks (see, e.g., [4,6,12,14,16]), where distance-vector algorithms seem to be an attractive alternative to link-state solutions when scalability and reliability are key issues or when the memory power of the nodes of the network is limited. Notwithstanding this increasing interest, the most important distance vector algorithm is still DUAL (Diffuse Update ALgorithm) [9], which is free of looping and is part of CISCO's widely used EIGRP protocol, although it requires a quite big space occupancy per node. Another distance vector algorithm, named LFR (Loop Free Routing), has been recently proposed in [7]. Compared with DUAL, LFR has the same theoretical message complexity but it uses an amount of data structures per node which is much smaller than that of DUAL. Moreover, LFR has been experimentally shown to be very effective in terms of both messages sent and memory requirements per node in some real-world networks. Recently, in [6] a general strategy named DLP (Distributed Leaf Pruning) has been introduced which can be combined with every distance vector algorithm with the aim of reducing the number of messages sent by that algorithm. In [6] the effectiveness of DLP has been confirmed by combining it with DUAL and by running experiments on both real-world and artificial instances.

Results of the Paper. We provide a new technique, named *Distributed Computation Pruning* (DCP), which is a generalization of DLP and can be combined with every distance-vector algorithm with the aim of overcoming some of their main limitations in real-world networks (high number of messages sent, high space occupancy per node, etc.). DCP has been designed to be efficient mainly in networks following a power-law node degree distribution, which from now on will be referred as *power-law networks*. Power-law networks includes many of the currently implemented communication infrastructures, like the Internet, the World Wide Web, some social networks, and so on [1]. The main idea underlying DCP rely on the fact that a power-law network with n nodes typically has average node degree much smaller than n and a high number of nodes with small degree (less than 3). Nodes with small degree often do not provide any useful information for the distributed computation of shortest paths, that is there are many topological situations in which these nodes should neither perform nor be involved in any kind of distributed computation, as their shortest paths depend on those of higher degree nodes.

In order to check the effectiveness of DCP, we combined it with DUAL and LFR by obtaining two new algorithms named DUAL-DCP and LFR-DCP, respectively. Then, we implemented the two new algorithms in the OMNeT++ simulation environment [13], a network simulator widely used in the literature. We also implemented DUAL, LFR, DUAL-DLP and LFR-DLP, where the last two algorithms are the combination of DUAL and LFR with DLP [6]. As input to the algorithms, we considered the power-law Internet topologies of the *CAIDA IPv4 topology dataset* [10]. The results of our experiments can be summarized as follows: the combination of DUAL and LFR with DCP provides a huge improvement in the number of messages sent with respect to DUAL and LFR, respectively. In particular, the number of messages sent by DUAL-DCP is always between 3% and 16% that of DUAL, while the number of messages sent by LFR-DCP is always between 10% and 26% that of LFR. The gain is significant also with respect to DUAL-DLP and LFR-DLP. We observed also an improvement in the maximum space occupancy per node of DUAL-DCP and LFR-DCP, and in the average space occupancy per node of DUAL-DCP. This is due to the fact that nodes with small degree do not need to store some of the data structures implemented by DUAL and LFR, respectively.

2 Preliminaries

We consider a network made of processors linked through communication channels that exchange data using a message passing model. We are interested in the practical case of networks whose topologies dynamically change over the time due to update operations on the edges (weight increase, weight decrease, insert, delete).

Graph Notation. We represent a network by an undirected weighted graph $G = (V, E, w)$, where V is a finite set of n nodes, one for each processor, E is a finite set of m edges, one for each communication channel, and w is a weight function $w : E \to \mathbb{R}^+$ that assigns to each edge a real value representing the optimization parameter associated to the corresponding channel. An edge in E that links nodes $u, v \in V$ is denoted as $\{u, v\}$. Given $v \in V$, $N(v)$ denotes the set of neighbors of v, and $deg(v) = |N(v)|$ denotes the degree of v. A path P in G between nodes u and v is denoted as $P = \{u, ..., v\}$. The *weight* of P, denoted as $w(P)$ is the sum of the weights of the edges in P. A *shortest path* between nodes u and v is a path from u to v with the minimum weight. The *distance* $d(u, v)$ from u to v is the weight of a shortest path from u to v. Given two nodes $u, v \in V$, the *via* from u to v is the set of neighbors of u that belong to a shortest path from u to v. Formally: $via(u, v) \equiv \{z \in N(u) \mid d(u, v) = w(u, z) + d(z, v)\}$. Given a time t, we denote as $w^t()$, $d^t()$, and $via^t()$ the edge weight, the distance, and the via at time t, respectively. We denote a sequence of update operations on the edges of G by $\mathcal{C} = (c_1, c_2, ..., c_k)$. Assuming $G_0 \equiv G$, we denote as G_i, $0 \le i \le k$, the graph obtained by applying c_i to G_{i-1}. We consider the case of weight increase and weight decrease operations.

Distance-Vector Algorithms. Given a graph $G = (V, E, w)$, distance-vector routing algorithms based on shortest-paths usually share a set of common features. In detail, a generic node v of G: (i) knows the identity of every other node of G, the identity of all its neighbors and the weights of the edges incident to it; (ii) maintains and updates its own routing table that has one entry for each $s \in V$, which consists of at least two fields: $\mathtt{D}[v, s]$, the estimated distance between v and s, and $\mathtt{VIA}[v, s]$, the neighbor used to forward data from v to s; (iii) handles edge weight changes either by a single procedure (see, e.g., [9]), which we denote as WeightChange, or separately (see, e.g., [4,7]) by two procedures, which we denote as WeightIncrease and WeightDecrease; (iv) asks information to its neighbors through a message denoted as *query* and receives reply through a message denoted as *reply*. If the routing information on a node changes, such a variation is propagated as follows: if v is performing WeightChange, then it sends to its neighbors a message, from now on denoted as *update*; a node that receives this kind of message executes a procedure named Update; if v is performing WeightIncrease or WeightDecrease, then it sends to its neighbors message *increase* or *decrease*, resp.; a node that receives *increase*/*decrease* executes a procedure named Increase/Decrease, respectively.

3 Dynamic Scenarios

In this section, we introduce some preliminary definitions that are useful to capture the dynamic scenarios typical of power-law networks and we show some properties of shortest paths in these scenarios. Given an undirected weighted graph $G = (V, E, w)$, we classify the nodes of G with respect to their degree as follows. A node $v \in V$ is: *central* if $deg(v) \geq 3$; *peripheral* if $deg(v) = 1$; *semiperipheral* if $deg(v) = 2$. An edge $\{u, v\}$ of G is: *central* if both u and v are central; *peripheral* if either u or v is peripheral; *semiperipheral* if either u or v is semiperipheral and neither of them is peripheral. A path $P = \{v_0, v_1, ..., v_{j-1}\}$ in G is: *central* if it is made only of central edges; *peripheral* if it contains exactly one peripheral edge and exactly one central node; the unique central node of P is called *owner* of P; *semiperipheral* if it is formed only by semiperipheral edges. If v_0 and v_{j-1} are two distinct central nodes, then they are called *semiowners* of P. If $v_0 \equiv v_{j-1}$, then P is a *semiperipheral cycle*, and $v_0 \equiv v_{j-1}$ is called the *cycleowner* of P. The following lemmata, which proofs are quite straightforward, introduce some basic relationships between central and non-central shortest paths of the network.

Lemma 1 (Peripheral shortest paths). *Given a graph $G = (V, E, w)$, let $P = \{v, p_1, ..., p_{j-1}\}$ be a peripheral path of G whose owner is node v, and let $P' = \{v, p_1, ..., p_i\}$, $1 \leq i \leq j - 1$, be a sub-path of P containing v. Then, for each $x \in V \setminus \{p_1, ..., p_{j-1}\}$, $d(x, p_i) = d(x, v) + w(P')$.*

Lemma 2 (Semi-peripheral shortest paths). *Given a graph $G = (V, E, w)$, let $S = \{u, sp_1, ..., sp_{j-2}, v\}$ be a semiperipheral path of G whose semiowners* *are nodes u and v, and let $S' = \{u, sp_1, ..., sp_i\}$, $1 \leq i \leq j-2$, and $S'' = \{sp_i, ..., sp_{j-2}, v\}$ be two sub-paths of S, containing u and v, respectively. Then, for each $x \in V \setminus \{sp_1, ..., sp_{j-2}\}$, $d(x, sp_i) = \min\{d(x,u)+w(S'), d(x,v)+w(S'')\}$.*

Lemma 3 (Semi-peripheral cycle shortest paths). *Given a graph $G = (V, E, w)$, let $C = \{u, c_1, ..., c_{j-1}, u\}$ be a semiperipheral cycle of G whose cycle-owner is node u, and let $C' = \{u, c_1, ..., c_i\}$ and $C'' = \{c_i, ..., c_{j-1}, u\}$, $1 \leq i \leq j-1$, be two sub-paths of C. Then, for each $x \in V \setminus \{c_1, ..., c_{j-1}\}$, $d(x, c_i) = \min\{d(x,u) + w(C'), d(x,u) + w(C'')\}$.*

Some useful additional relationships can be derived introducing time instants in the above Lemmata. In particular, by Lemma 1 we know that, if between the time instants t_i and t_{i+1} the weight of the edge $\{p_1, p_2\}$ between two nodes belonging to a peripheral path $P = \{v, ..., p_1, p_2, ..., p_{j-1}\}$ changes, that is $w^{t_i}(p_1, p_2) \neq w^{t_{i+1}}(p_1, p_2)$, then for each $x \in V$ that does not belong to P, the distance from p_1 to x does not change, while the distance from p_2 to x changes as follows:

$$d^{t_{i+1}}(p_2, x) = d^{t_i}(p_2, x) + w^{t_{i+1}}(p_1, p_2) - w^{t_i}(p_1, p_2) \tag{1}$$

By Lemma 2 we know that, if between the time instants t_i and t_{i+1} the weight of the edge $\{sp_1, sp_2\}$ between two nodes belonging to a semiperipheral path $S = \{u, ..., sp_1, sp_2, ..., v\}$ changes, that is $w^{t_i}(sp_1, sp_2) \neq w^{t_{i+1}}(sp_1, sp_2)$, then for each $x \in V$, both the distances from sp_1 to x and from sp_2 to x change as follows:

$$d^{t_{i+1}}(sp_1, x) = \min_{z \in N(sp_1)} \{d^{t_{i+1}}(z, x) + w^{t_{i+1}}(sp_1, z)\} \tag{2}$$

$$d^{t_{i+1}}(sp_2, x) = \min_{z \in N(sp_2)} \{d^{t_{i+1}}(z, x) + w^{t_{i+1}}(sp_2, z)\} \tag{3}$$

Let us assume that, between t_i and t_{i+1}, the weight of the edge $\{c_1, c_2\}$ between two nodes belonging to a semiperipheral cycle $C = \{u, ..., c_0, c_1, c_2, c_3, ..., u\}$ changes, that is $w^{t_i}(c_1, c_2) \neq w^{t_{i+1}}(c_1, c_2)$. If we denote as $C_0 = (u, ..., c_0, c_1)$, $C_1 = (c_1, c_2, ..., u)$, $C_2(u, ..., c_1, c_2)$ and $C_3 = (c_2, c_3, ..., u)$ then by Lemma 3, for each $x \in V$, the distances from c_1 to x and from c_2 to x change as follows:

$$d^{t_{i+1}}(c_1, x) = d^{t_i}(x, u) + \min\{\sum_{\{l,q\} \in C_0} w^{t_{i+1}}(l, q), \sum_{\{l,q\} \in C_1} w^{t_{i+1}}(l, q)\} \tag{4}$$

$$d^{t_{i+1}}(c_2, x) = d^{t_i}(x, u) + \min\{\sum_{\{l,q\} \in C_2} w^{t_{i+1}}(l, q), \sum_{\{l,q\} \in C_3} w^{t_{i+1}}(l, q)\} \tag{5}$$

If the distance between a generic node $x \in V$ and a central node c changes between the time instants t_i and t_{i+1} (that is, $d^{t_{i+1}}(x, c) \neq d^{t_i}(x, c)$), then the following relationships hold:

- by Lemma 1, for each peripheral path $P = \{c, ..., \overline{p}, ..., p_{j-1}\}$ with owner c, and for each $\overline{p} \in P$:

$$d^{t_{i+1}}(x, \overline{p}) = d^{t_i}(x, \overline{p}) + d^{t_{i+1}}(x, c) - d^{t_i}(x, c), \tag{6}$$

- by Lemma 2, for each semiperipheral path $S = \{c, ..., \overline{sp}, ..., d\}$ with semiowners c and d and for each $\overline{sp} \in S$ such that c belongs to the shortest path from $\overline{sp}$ to x at time t_i, if we denote as $D = (d, ..., \overline{sp})$ the sub-path of S from d to $\overline{sp}$, and by $k = \sum_{\{l,q\} \in D} w^{t_i}(l, q)$, then:

$$d^{t_{i+1}}(x, \overline{sp}) = \min\{d^{t_{i+1}}(x, \overline{sp}) + d^{t_{i+1}}(x, c) - d^{t_i}(x, c), d^{t_i}(x, d) + k\} \tag{7}$$

- by Lemma 3, for each cyclic path $C = \{c, c_1, ..., \overline{c}, ..., c_{j-1}, c\}$ with cycleowner c, and for each $\overline{c} \in C$:

$$d^{t_{i+1}}(x, \overline{c}) = d^{t_i}(x, \overline{c}) + d^{t_{i+1}}(x, c) - d^{t_i}(x, c), \tag{8}$$

4 The New Technique

Distributed Computation Pruning (DCP) has been designed to be efficient mainly in power-law networks, by forcing the distributed computation to be carried out only by the central nodes, which are few in practice. The non-central nodes, which are the great majority in power-law networks, receive updates about routing information passively from the respective owners, without starting any kind of distributed computation. Then, the larger is the set of non-central nodes of the network, the bigger is the improvement in the pruning of the distributed computation and, consequently, in the global number of messages sent.

Data Structures. Given a generic distance-vector algorithm **A**, DCP requires that a generic node of G stores some additional information with respect to those required by **A**. In particular, a node v needs to store and update information about non-central paths of G. To this aim, v maintains a data structure called *ChainPath*, denoted as $\texttt{CHP}_v$, which is an array containing one entry $\texttt{CHP}_v[s]$, for each central node s. $\texttt{CHP}_v[s]$ stores the list of all edges, with the corresponding weight, belonging to the non-central paths containing s. A central node is obviously not present in any list of $\texttt{CHP}_v$. A peripheral node is present in exactly one list $\texttt{CHP}_v[s]$, where $s \in V$ is its owner. A semi-peripheral node is present in exactly two lists $\texttt{CHP}_v[v_0]$ and $\texttt{CHP}_v[v_{j-1}]$, if it belongs to a semi-peripheral path (v_0 and v_{j-1} are its semiowners), while it is present in a single list $\texttt{CHP}_v[v_0]$, if it belongs to a semi-peripheral cycle (v_0 is its cycleowner). The space occupancy overhead per node due to *ChainPath* can be quantified using the following observations: the *ChainPath* contains at most as many entries as the number of the central nodes; the sum of the sizes of all the lists in the *ChainPath* is twice the number of non-central edges of G in the worst case; the number of non-central edges of G is $O(n)$, as they belong to paths in which every node has degree at most two. Hence, the space overhead per node due to $\texttt{CHP}_v$ is $O(n)$. Note that, despite the overhead due to the *ChainPath* data structure, the use of DCP can induce a decrease in the space occupancy per node required by **A** for the following observations: (i) in most of the cases nodes do not ask and do not need to store information received from non-central nodes; (ii) computations which involves the whole network are performed only with respect to central destinations.

Distributed Computation Pruning. The combination of DCP with a distance vector algorithm **A** induces a new algorithm denoted as **A**-DCP. The behavior of **A**-DCP can be summarized as follows. While in a classic routing algorithm every node performs the same code thus having the same behavior, in **A**-DCP central and non-central nodes have different behaviors. In particular, central nodes detect changes concerning all kind of edges, while semiperipheral, peripheral and cyclic nodes detect changes concerning only semiperipheral, peripheral and cyclic edges, respectively.

If the weight of a central edge $\{u, v\}$ changes, then node u (v, resp.) performs the procedure provided by **A** for the distributed computation of the shortest paths, only with respect to central nodes. During this computation, if u (v) needs information by its neighbors, it asks only to central neighbors or, if u (v) is the semiowner of one or more semiperipheral paths, it asks information also to the other semiowner of each semiperipheral path, by means of a strategy we called *Mod*-DBF. In detail, node u (v) sends to each semiperipheral neighbor a *query*DBF message, whose aim is to traverse the semiperipheral path, in order to get information by the other semiowner. The *query*DBF message contains just one field, the source object of the computation. A semiperipheral node, which receives a *query*DBF message from one of its two neighbors, simply performs a store-and-forward step and sends a *query*DBF message to the other neighbor. A central node, which receives a *query*DBF message, simply replies the information that *query*DBF is asking for. Once u (v) has updated its own routing information, it propagates the variation to all its neighbors through the *update*, *increase* or *decrease* messages of **A**. When a generic node x receives an *update*, *increase* or *decrease* message, it stores the current value of $\mathtt{D}[x, s]$ in a temporary variable.

Now, if x is a central node, then it handles the change and updates its routing information toward s, by using the proper procedure of **A** (Update, Increase, or Decrease) and propagates the new information to its neighbors. Otherwise, if x is a peripheral, semiperipheral or cyclic node, it handles the change and updates its routing information toward s by using Lemmata 1–3, and the data provided by its owner, semiowner and cycleowner, respectively. At the end, x verifies whether the routing table entry of s is changed or not and, in the affermative case, it updates the routing information about the non-central neighbors of s, if they exist, by implementing Equations 6–8. Note that, node x uses the data contained in CHP in order to properly update its routing information towards the non-central nodes of s, if they exist.

If a weight change occurs on a peripheral edge $\{u, v\}$, then nodes u and v both send a *p_change*$(u, v, w(u, v))$ message to each of their neighbors. When a generic node x receives message *p_change*, it first verifies whether the update has been already processed or not, by comparing the new value of $w(u, v)$ with the one stored in its CHP. In the first case the message is discarded. Otherwise, node x updates its CHP with the updated value of $w(u, v)$ and its routing information by using Equation 1. Then, it propagates the change by a flooding algorithm to forward the message over the network.

If the weight of a semiperipheral edge $\{u, v\}$ changes, then node u (v resp.) sends two kind of messages: a *sp_change*$(u, v, w(u, v))$, to each of its owners, and a *sp_update*$(s, \mathtt{D}[u, s])$ (*sp_update*$(s, \mathtt{D}[v, s])$) to v (u), for each s such that $\mathtt{VIA}[u, s] \neq v$ ($\mathtt{VIA}[v, s] \neq u$). When a generic node x receives message *sp_change*, it first verifies whether the update has been already processed or not, by comparing the new value of $w(u, v)$ with the one stored in its CHP. In the first case the message is discarded. Otherwise, node x simply updates its CHP with the updated value of $w(u, v)$. When a generic node x receives a *sp_update*$(s, \mathtt{D}[u, s])$ message from a neighbor u, two cases can occur. If x is a central node, it simply performs procedure *update* of **A**. Otherwise, it updates routing information towards s by using Equations 2–3. Note that, in this case, node x uses the information contained in CHP in order to verify whether it belongs or not to the same semiperipheral path of s and to properly update its routing information.

If the weight of a cyclic edge $\{u, v\}$ changes, nodes u and v both send a *cy_update*$(u, v, w(u, v))$ message to each of their neighbors. When a generic node x receives message *cy_update*, it first verifies whether the update has been already processed or not, by comparing the new value of $w(u, v)$ with the one stored in its CHP. In the first case the message is discarded. Otherwise, node x updates its CHP with the updated value of $w(u, v)$ and its routing information by using Equations 4–5. Then, it propagates the change by a flooding algorithm to forward the message over the network.

5 Experimental Analysis

Our experiments have been performed on a workstation equipped with a Quad-core 3.60 GHz Intel Xeon X5687 processor, with 12MB of internal cache and 24 GB of main memory, and consist of simulations within the OMNeT++ 4.0p1 environment [13].

Executed Tests. For the experiments we used the power-law networks of the *CAIDA IPv4 topology dataset* [10]. We parsed the files provided by CAIDA to obtain a weighted undirected graph, denoted as G_{IP}, where a node represents an IP address in the dataset (both source/destination hosts and intermediate hops), edges represent links among hops and weights are given by Round Trip Times. As the graph G_{IP} consists of almost 35000 nodes, we could not use it for the experiments, due to the big amount of memory required to store the routing tables for DUAL. Hence, we performed our tests on connected subgraphs of G_{IP} induced by the settled nodes of a breadth first search starting from a node taken at random. We denoted a h nodes subgraph of G_{IP} with G_{IP-h}. We generated a set of different tests, each test consists of a subgraph of G_{IP} and a set of k edge updates, where k assumes values in $\{5, 10, \ldots, 200\}$. An edge update consists of

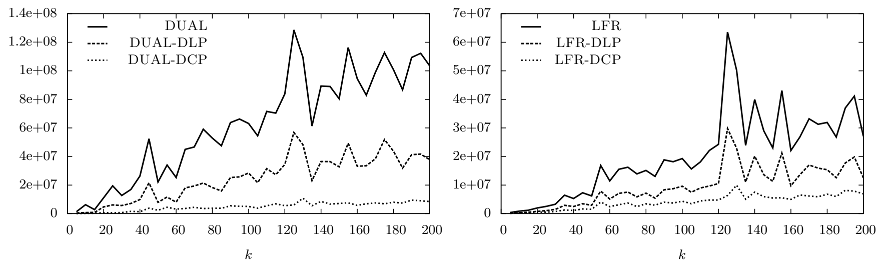


Fig. 1. Number of messages sent by DUAL, DUAL-DLP and DUAL-DCP (left) and by LFR, LFR-DLP and LFR-DCP (right) on $G_{IP-8000}$

multiplying the weight of a random selected edge by a percentage value randomly chosen in $[50\%, 150\%]$. For each test configuration (a graph with a fixed value of k) we performed 5 different experiments (for a total of 200 runs) and we report average values.

Analysis. We ran simulations on CAIDA instances with different number of nodes $n \in \{1200, 5000, 8000\}$. The results of our experiments on the different instances are similar, hence we report those on the bigger instances, which has 8000 nodes and 11141 edges. In particular, in Figures 1(left) and 1(right) we report the number of messages sent by DUAL, DUAL-DLP and DUAL-DCP and by LFR, LFR-DLP and LFR-DCP, respectively, on $G_{IP-8000}$. Note that, $G_{IP-8000}$ has average node degree equal to 2.8, a percentage of degree 1 nodes approximately equal to 38.5%, and a percentage of degree 2 nodes approximately equal to 33%. The figures show that the combinations of DUAL and LFR with DCP provide a huge improvement in the global number of messages sent. The gain is significant also with respect to DUAL-DLP and LFR-DLP. In the tests of Fig. 1(left) the ratio between the number of messages sent by DUAL-DCP and DUAL is within 0.03 and 0.16 which means that DUAL-DCP sends a number of messages which is between 3% and 16% that of DUAL. Similarly, the ratio between the number of messages sent by DUAL-DCP and DUAL-DLP is within 0.11 and 0.40. In the tests of Fig. 1(right) the ratio between the number of messages sent by LFR-DCP and LFR is within 0.10 and 0.26 which means that the number of messages sent by LFR-DCP is always between 10% and 26% that of LFR. Similarly, the ratio between the number of messages sent by LFR-DCP and LFR-DLP is within 0.21 and 0.58.

To conclude our analysis, we consider the space occupancy per node of each algorithm. The results are summarized in Table 1 where we report the maximum and the average space occupancy per node, in Bytes, of each algorithm on $G_{IP-8000}$. We also report the ratio between the space occupancy per node of the algorithms integrating DLP and DCP and that of the original algorithms, for each test instance. Note that, since the space occupancy per node of LFR, LFR-DLP and LFR-DCP depends on the number of weight change operations, we show median values for each of these algorithms. Our experiments show that the use of DCP induces, in most of the cases, a clear improvement also in

Table 1. Space occupancy per node of the implemented algorithms

Graph	**Algorithm**	**MAX**		**AVG**	
		Bytes	**Ratio**	**Bytes**	**Ratio**
$G_{IP-8000}$	DUAL	8 320 000	1	311 410	1
	DUAL-DLP	5 161 984	0.62	240 754	0.77
	DUAL-DCP	2 517 680	0.30	252 625	0.81
$G_{IP-8000}$	LFR	549 170	1	192 871	1
	LFR-DLP	421 862	0.77	204 675	1.06
	LFR-DCP	392 658	0.72	295 930	1.53

the space requirements per node. In particular, DUAL-DCP (LFR-DCP) requires a maximum space occupancy per node which is 0.30 (0.72) times that of DUAL (LFR). Notice that, the improvement is more evident in the case of DUAL, as its maximum space occupancy per node is by far higher than that of LFR. Concerning DUAL, this behavior is confirmed also in the average case, where DUAL-DCP requires 0.81 times the average space occupancy per node of DUAL. On the contrary, our data show that the average space occupancy per node of LFR-DCP is slightly greater than that of LFR and that the use of DCP induces an overhead in the average space occupancy per node which is equal to 53%. This is due to the fact that the average space occupancy of LFR is quite low by itself and that, in this case, the space overhead needed to store the *ChainPath* is greater than the space occupancy reduction induced by the use of DCP.

References

1. Albert, R., Barabási, A.-L.: Emergence of scaling in random networks. Science 286, 509–512 (1999)
2. Bertsekas, D., Gallager, R.: Data Networks. Prentice Hall International (1992)
3. Cicerone, S., D'Angelo, G., Di Stefano, G., Frigioni, D.: Partially dynamic efficient algorithms for distributed shortest paths. Theoretical Computer Science 411, 1013–1037 (2010)
4. Cicerone, S., D'Angelo, G., Di Stefano, G., Frigioni, D., Maurizio, V.: Engineering a new algorithm for distributed shortest paths on dynamic networks. Algorithmica (to appear, 2012), doi: 10.1007/s00453-012-9623-9
5. Cicerone, S., Stefano, G.D., Frigioni, D., Nanni, U.: A fully dynamic algorithm for distributed shortest paths. Theoretical Comp. Science 297(1-3), 83–102 (2003)
6. D'Angelo, G., D'Emidio, M., Frigioni, D., Maurizio, V.: A Speed-Up Technique for Distributed Shortest Paths Computation. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011, Part II. LNCS, vol. 6783, pp. 578–593. Springer, Heidelberg (2011)
7. D'Angelo, G., D'Emidio, M., Frigioni, D., Maurizio, V.: Engineering a New Loop-Free Shortest Paths Routing Algorithm. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 123–134. Springer, Heidelberg (2012)
8. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully dynamic algorithms for maintaining shortest paths trees. Journal of Algorithms 34(2), 251–281 (2000)

9. Garcia-Lunes-Aceves, J.J.: Loop-free routing using diffusing computations. IEEE/ACM Trans. on Networking 1(1), 130–141 (1993)
10. Hyun, Y., Huffaker, B., Andersen, D., Aben, E., Shannon, C., Luckie, M., Claffy, K.: The CAIDA IPv4 routed/24 topology dataset, http://www.caida.org/data/active/ipv4_routed_24_topology_dataset.xml
11. Moy, J.T.: OSPF: Anatomy of an Internet routing protocol. Addison-Wesley (1998)
12. Myers, A., Ng, E., Zhang, H.: Rethinking the service model: Scaling ethernet to a million nodes. In: ACM SIGCOMM HotNets. ACM Press (2004)
13. OMNeT++. Discrete event simulation environment, http://www.omnetpp.org
14. Ray, S., Guérin, R., Kwong, K.-W., Sofia, R.: Always acyclic distributed path computation. IEEE/ACM Trans. on Networking 18(1), 307–319 (2010)
15. Rosen, E.C.: The updating protocol of arpanet's new routing algorithm. Computer Networks 4, 11–19 (1980)
16. Zhao, C., Liu, Y., Liu, K.: A more efficient diffusing update algorithm for loop-free routing. In: 5th International Conference on Wireless Communications, Networking and Mobile Computing (WiCom 2009), pp. 1–4. IEEE Press (2009)

Experimental Analysis of Rumor Spreading in Social Networks

Benjamin Doerr[1], Mahmoud Fouz[2], and Tobias Friedrich[3]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] Rocket Internet, Dubai, U.A.E.
[3] Friedrich-Schiller-Universität Jena, Germany

Abstract Randomized rumor spreading was recently shown to be a very efficient mechanism to spread information in preferential attachment networks. Most interesting from the algorithm design point of view was the observation that the asymptotic run-time drops when memory is used to avoid re-contacting neighbors within a small number of rounds.

In this experimental investigation, we confirm that a small amount of memory indeed reduces the run-time of the protocol even for small network sizes. We observe that one memory cell per node suffices to reduce the run-time significantly; more memory helps comparably little. Aside from extremely sparse graphs, preferential attachment graphs perform faster than all other graph classes examined. This holds independent of the amount of memory, but preferential attachment graphs benefit the most from the use of memory. We also analyze the influence of the network density and the size of the memory. For the asynchronous version of the rumor spreading protocol, we observe that the theoretically predicted asymptotic advantage of preferential attachment graphs is smaller than expected. There are other topologies which benefit even more from asynchrony.

We complement our findings on artificial network models by the corresponding experiments on crawls of popular online social networks, where again we observe extremely rapid information dissemination and a sizable benefit from using memory and asynchrony.

1 Introduction

Randomized rumor spreading is a class of simple randomized distributed algorithms, all building on the paradigm that nodes of a network contact random neighbors to exchange information. Despite being very simple protocols, they proved to be very efficient both in theoretical investigations [14, 15, 23, 26–32, 36] and in practical applications [19, 33].

In a recent work [22], the authors analyzed the performance of the classical phone call model of Karp et al. [32] on networks following the preferential attachment model suggested by Barabási and Albert [1] to model real-world networks. The model assumes that new vertices attach to already-present vertices with a probability proportional to their degree. The problem of rumor spreading on these networks was first considered by Chierichetti, Lattanzi, and Panconesi [16] who showed that $\mathcal{O}(\log^2 n)$ rounds suffice with high probability. In [22], an asymptotically tight rumor spreading time of $\Theta(\log n)$ was proven, which is the same order of magnitude as for many other network topologies including complete networks, hypercubes and many classical random graph classes. Surprisingly, this run-time drops to the again tight order of $\Theta(\log n/\log\log n)$ when the protocol is modified so that contacting the same neighbor twice in a row is avoided. This observation is important from the viewpoint of algorithm design, since such a mechanism is very simple to implement. However, so far such fine-tuning has rarely led to provably better algorithms.

The aim of this work is to use an experimental investigation in order to (a) better understand the performance of randomized rumor spreading protocols on preferential attachment networks; in the long run, this might help in the design of efficient communication networks; and (b) to better understand the advantage of equipping nodes with a small amount of memory, which is used to avoid contacting a constant number of previous contactees; this is interesting from the viewpoint of algorithm design.

In summary, our main findings are the following. Generally, rumor spreading is very fast in preferential attachment networks, significantly faster than in random-attachment networks and hypercubes (which are much denser), and faster than in complete networks (unless the density is very small). There is a clearly visible advantage of keeping track of the most-recently-contacted neighbor (using a one-item memory) in preferential attachment networks, particularly if the density is small. There is less to be gained from memory on random attachment networks and almost no gain in complete networks and hypercubes. Additional memory is of some benefit, but not very much.

For communication in social networks in particular, it makes sense to consider an asynchronous version of the rumor spreading protocol with nodes acting at exponentially distributed times (with expectation one). For random graphs with a given expected degree distribution that follows a power law with exponent in $(2,3)$, Fountoulakis et al. [29] showed very recently that the push-pull protocol becomes much faster in the asynchronous setting. A recent theoretical analysis [23] proves a reduced time of $\mathcal{O}(\sqrt{\log n})$ in preferential attachment graphs and argues that random-attachment graphs, complete graphs and hypercubes keep their $\Theta(\log n)$ times. Our experiments show that the asynchronous model is faster on all graph classes, but a clearly greater advantage for preferential attachment graphs is not visible.

We conducted similar experiments on crawls of the Twitter and Orkut online social networks. Interestingly, we observe an even faster information dissemination than in preferential attachment graphs of corresponding size and density. These experiments also confirm that tracking one neighbor (one-item memory) cell leads to a significant improvement, whereas using additional memory to track more neighbors does not produce significant gains.

Rumor Spreading Protocols

When talking about rumor spreading, in this paper we generally refer to the *random phone call* model introduced by Karp et al. [32]. This is a push-pull protocol, meaning that information is exchanged between initiator and recipient of a call in both directions. A push-protocol with only the caller sending information to the recipient has also been widely discussed in the literature [27, 30], in particular, for the application of making replicated databases consistent [19, 33]. As shown in [16], however, the push protocol has very poor performance in preferential attachment networks.

The random phone call model is a synchronized protocol. In each round, each node of the network calls a random neighbor and both exchange information with each other. If one of the communication partners was informed at the beginning of the round, then both will be at the end of the round. An asynchronous analog of the protocol is discussed in Section 5.

It is interesting to enhance the random phone call model by excluding recently-contacted neighbors. When allowing a memory of size k, each node v chooses his next communication partner uniformly at random from all his neighbors except the previous $\min\{k, \deg(v) - 1\}$ contactees. Note that nodes with degree $d(v) \leq k+1$ act as in the quasirandom model of Doerr, Friedrich, and Sauerwald [20] with randomly chosen lists.

Network Models

We are mainly interested in the *preferential attachment* (PA) model of Barabási and Albert [1]. The density of the resulting graph is controlled by a single parameter m. The model iteratively adds new vertices, which are connected to m already present vertices with a probability proportional to their degree. See Bollobás, Riordan, Spencer, and Tusnády [8, 9] for a precise description of this random graph model. It can be easily seen that for $m = 1$ the graph is disconnected with high probability. We therefore focus on $m \geq 2$. Under this assumption, the diameter is $\Theta(\log(n)/\log\log n)$ with high probability [8]. Besides various other typical properties of social networks[3, 6, 7, 18, 28], it also has been shown that the degree distribution follows a power law [9].

In addition to the PA model, we shall also include random-attachment networks in our investigation. In this network model, also known as the m-out model [5], each node chooses m other nodes as neighbors uniformly at random; finally, this neighbor relation is made symmetric and multiple edges are removed. Consequently, we obtain a random graph with average degree close to $2m$ and minimum degree at least m. These graphs form a good point of comparison with preferential attachment graphs with density parameter m, where nodes also choose m random neighbors, but according to the preferential attachment paradigm.

Related Work

For many network topologies, the random phone call model very quickly distributes a piece of information initially only present at one node to all other nodes. In addition, due to its randomized nature, this process is highly robust against transmission failure. Karp et al. [32] show that in complete networks (any node can talk to any other node), $(1+o(1))\log_3(n)$ rounds suffice to spread a rumor in the whole network. Similarly, Elsässer [25] proved a bound of $\Theta(\log n)$ rounds for Erdős-Rényi random graphs $G_{n,p}$ with $p \geq \text{polylog}(n)/n$. For hypercubes, a $\Theta(\log n)$ bound follows from the $\mathcal{O}(\log n)$ bound of Feige et al. [27] for the push protocol together with the trivial lower bound stemming from the logarithmic diameter of the hypercube.

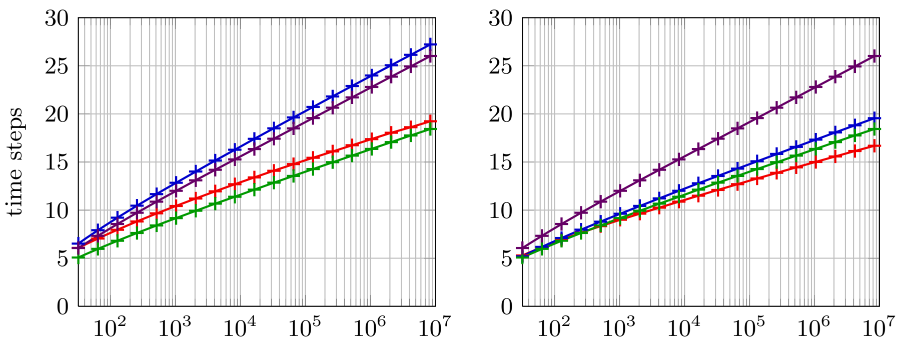
In a recent paper [22], the authors proved that the random phone call protocol spreads a rumor to all vertices of a preferential attachment graph in $\Theta(\log n)$ rounds as well. This improves over the previous $\mathcal{O}(\log^2 n)$ bound by Chierichetti et al. [16], but falls short of showing that these graphs, which are often used to model social networks, support rumor spreading better than classical network topologies. This is achieved in some sense in [22]. If we slightly alter the protocol such that a node chooses its communication partner uniformly at random from all neighbors excluding the one contacted in the previous round, then the rumor spreading time reduces to $\mathcal{O}(\log n / \log\log n)$, which is a tight bound because it is the diameter of these graphs [8].

Note that excluding previous contactees has almost no effect on classical network topologies. By checking the proofs of the results cited above, we see that also when excluding a constant number of previous contactees, the $\Theta(\log n)$ bound remains valid for complete graphs, hypercubes and random graphs. The quasirandom protocol of Doerr et al. [20] is a way of excluding all previous contactees. It has been investigated only in the push model, where again many known $\Theta(\log n)$ run time bounds have been verified. An experimental investigation [21] revealed that the quasirandom protocol is faster than the independent one, minimally for complete networks, but noticeably for sparser ones like random graphs and hypercubes. Unfortunately, our current results cannot be compared to these, because the latter are based only on push protocols. Baumann et al. [2] observed that the behavior of the quasirandom protocol changes significantly if the nodes known which of their neighbors already received the rumor.

2 Fast Broadcasting in Preferential Attachment Graphs, Influence of Graph Density

The result of [22] shows that rumor spreading in the random phone call model with memory size at least one has an asymptotically faster run-time of $\Theta(\log n / \log\log n)$ in preferential attachment graphs, in contrast to the $\Theta(\log n)$ time observed (i) for the no-memory version on preferential attachment graphs and (ii) regardless of memory on most classical graphs like complete graphs, hypercubes, and random attachment graphs. Since in [22] only asymptotic results were proven, it is not clear if the proven differences are apparent for reasonable graph sizes. This is the focus of the current section of this paper. We have examined the average time needed to inform all vertices, starting from a random vertex, for different graphs.

In Figure 1, we show the broadcast times observed for complete graphs, hypercubes, and preferential and random attachment graphs with density parameters $m = 2$ and $m = 10$, with one-item memory. We observe that rumor



(a) density parameter $m = 2$. **(b)** density parameter $m = 10$.

Fig. 1. Comparison of synchronous rumor spreading with one-item memory on preferential attachment graph (🟥), random-attachment graph (🟦), complete graph (🟩), and hypercube (🟪). The two charts show different density parameters of the preferential and random-attachment graph. The results for complete graphs and hypercubes are equivalent in both charts; they are given for comparison. The x-axis corresponds to the number of vertices $n = 2^5 \dots 2^{23}$. The y-axis corresponds to the run-time to inform all vertices, averaged over 10,000 runs.
For $m = 10$ the preferential attachment graph performs faster than all other graph classes. For not too large ($n \le 2^{23}$) and very sparse case ($m = 2$) considered, the complete graph is even faster than the preferential attachment graph.

spreading is quite fast in preferential attachment graphs (🟥), faster than in hypercubes (🟪) and random-attachment graphs (🟦) for both density parameters $m = 2$ and $m = 10$, and even faster than in complete graphs (🟩) for $m = 10$. Hence only the very sparse preferential attachment graphs with $m = 2$ are outperformed by complete graphs for $n \le 10^7$. As for $n \ge 10^4$ the two last-mentioned charts constantly get closer, we expect that for sufficiently large graphs, information spreading is also faster on sparse preferential attachment graphs than on complete graphs.

We also observed structurally different behavior of the information spreading process on the different graphs. To be precise, let us consider graphs with $n = 10^6$ vertices and $m = 2$, averaged over 10,000 runs. Then on average 57% of the nodes of a random attachment graph are informed with a pull operation (and 43% via push). On the other hand, in preferential attachment graphs 73% of the nodes are informed by a pull operation. Moreover, on average such a pull operation transfers the rumor from a high degree node (with degree 66 on average) to a node with low degree (with degree 3 on average). This matches the structure used in the proofs of [16, 22, 29].

The path by which a piece of information is spread in a preferential attachment graph seems to differ from the typical paths in a random attachment graph. We measured the number of hops a piece of information needed to inform a node and compared this to its graph distance. In general, it is preferable to have a good correlation between the two measures [34]. The graph distance from the source gives a lower bound for the number of hops needed to inform a node. We call the difference between the number of hops needed and the graph distance the *delay*. If the delay is small, the information is spread on nearly-shortest paths. On random attachment graphs we observed that vertices which are less than six steps away from the source have a delay of less than one on average. On preferential attachment graphs, nodes with distance between two and six from the source have on average a delay of four. This shows that on preferential attachment graphs the information is *not* spread via shortest paths, but via detours. This again has been used in the theoretical analyses of [16, 22, 29].

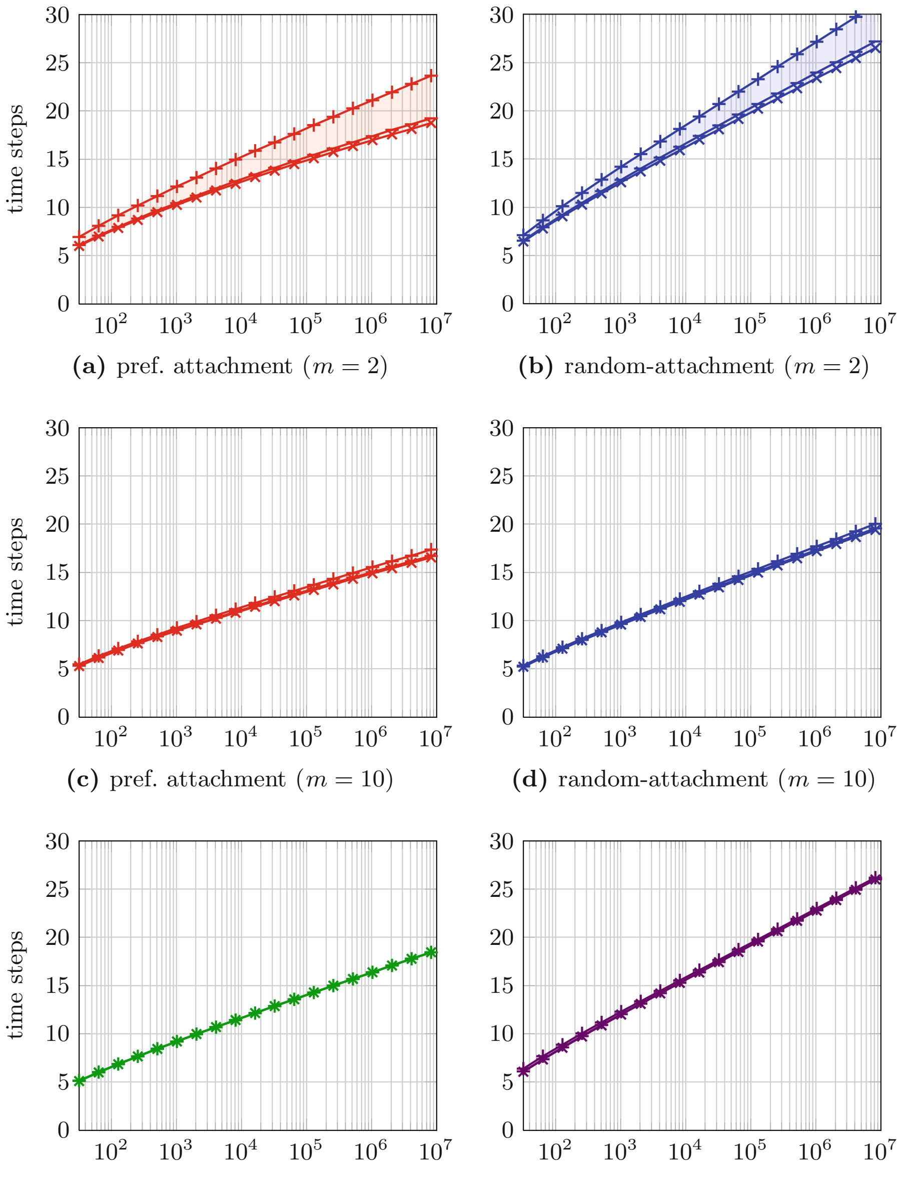
3 The Effect of Short-Term Memory

Perhaps the most surprising finding of [22] is that keeping track of a certain small number of recently-contacted neighbors, and avoiding selecting any of these when randomly choosing the next communication partner, significantly reduces the time needed to inform all nodes of preferential attachment networks. More precisely, it was shown that for the classical random phone call model, this time is $\Theta(\log n)$. If the communication partners are chosen uniformly at random from all neighbors except the one called in the previous round (one-item memory), then this time decreases to $\Theta(\log n/\log\log n)$. Using additional memory to track more than one recent contactee, however, does not yield times better than $\Theta(\log n/\log\log n)$.

In this section, we experimentally investigate this phenomenon. Figure 2 shows the average time needed to inform all nodes. We first discuss the results on preferential attachment graphs with $m = 2$ shown in Figure 2 (a). As expected, we observe a significant improvement between no exclusion (marked with $+$) and exclusion of one neighbor (marked with $-$). In fact, for all graph sizes, one-item memory leads to nodes becoming informed between 14% and 21% faster than no memory. Observing the curves for different graph sizes also suggests that we have a $\Theta(\log n)$ broadcast time in the no-memory case and an $o(\log n)$ time with memory of any non-zero size. We do observe additional but very small improvements if we increase the memory to a size larger than the run-time, that is, when avoiding all previous contactees (marked with $\times$). For the graph sizes considered, the improvement of unbounded memory compared to memory of only one item is around 2%. The advantage of memory for preferential attachment graphs gets smaller for larger m, as shown in Figure 2 (c).

The results on random-attachment graphs are similar, just generally slower. Figure 2 (b) shows that the difference for $m = 2$ between no memory and one-item memory is between 10% and 13%, while the additional improvement of unbounded memory is again around 2%. Theoretical consideration suggests that these gains can be at most by constant factors[1], and our experiments show that this can be at most a small constant.

[1] It is known that these graphs have a diameter of $\Theta(\log n)$, so this is a natural lower bound. On the other hand, with high probability each pair of vertices is connected by a path such that the sum of the degrees of the vertices on the path is at most $\mathcal{O}(\log n)$. Consequently, with probability $1 - o(n^{-1})$, $\mathcal{O}(\log n)$ rounds suffice to transmit a rumor along such a path. This yields an upper bound of $\mathcal{O}(\log n)$ for the broadcast time on random attachment graphs.



(a) pref. attachment ($m = 2$) **(b)** random-attachment ($m = 2$)

(c) pref. attachment ($m = 10$) **(d)** random-attachment ($m = 10$)

(e) complete graph **(f)** hypercube

Fig. 2. Comparison of synchronous rumor spreading without memory (marked with +), one-item memory (marked with −), and unbounded memory (marked with ×) on different graphs. The x-axis corresponds to the number of vertices $n = 2^5 \dots 2^{23}$. The y-axis corresponds to the run-time to inform all vertices, averaged over 10,000 runs. The benefit of remembering more than one neighbor is very limited for all graphs. The benefit of one-item memory compared to no memory is the largest for the sparse preferential and random-attachment graphs. The complete graph and hypercube benefit very little from additional memory.

Table 1. Comparison of the average time needed to inform a certain fraction of the vertices on the Orkut network depending on the amount of memory. For each combination, the average and standard deviation of 100,000 runs is given. With regard to the time needed to inform all vertices, we observe a large difference between excluding none and excluding the one most recently contacted. If only a 90% or 99% fraction should be informed, the gap is significantly smaller.

	90% informed	**99% informed**	**100% informed**
memory=0	15.74±0.99	16.87±1.00	23.13±2.28
memory=1	15.51±0.98	16.60±1.00	20.97±1.59
memory=2	15.47±0.98	16.55±0.99	20.31±1.30
memory=3	15.45±0.98	16.54±0.99	20.18±1.22
memory=25	15.45±0.97	16.54±0.99	20.11±1.13

In contrast, for other network topologies we see little advantage from using memory. For complete graphs, we observe in Figure 2 (e) barely any advantage even with unbounded memory. The difference between no memory and unbounded memory is less than 1% for complete graphs of all sizes. Because of the large vertex degrees, little benefit was expected; however, this is a notable difference from the results of using a pure push protocol without pull. Here, [21] observed at least a small advantage for the quasirandom protocol, which, when used with random lists, is equivalent to random choices with excluded previous contactees. The results of Figure 2 (f) for hypercubes show a similarly small impact of memory. For graphs with more than a few thousand nodes, the difference between no memory and unbounded memory is smaller than 2%.

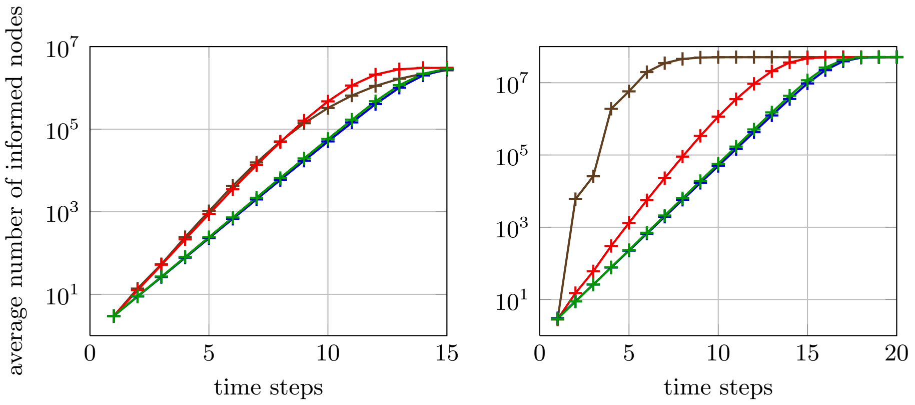
The benefit of a small amount of memory can also be observed on real-world graphs. We examined the time needed to spread a rumor on a crawl of the Orkut network (for details on the network see Section 4). Table 1 shows a large difference between no memory and one-item memory for the time needed to inform all vertices. It is clearly visible that (a) more memory is of very little benefit and (b) this difference vanishes when considering the time needed to inform only a fraction of the vertices.

In summary, we also observe in experiments that a small amount of memory helps a lot for preferential and random attachment graphs, but much less for classical network topologies like complete graphs and hypercubes.

4 Real-World Social Networks

Most previous statistics were based on mathematically-defined graph models. To support our claim that news spreads very fast on social networks in general, we have also simulated the rumor spreading process on crawls of the *Twitter* and *Orkut* social networks.

Twitter is a social networking site which allows users to send and read short messages (so-called "tweets") of up to 140 characters. It is currently one of the



(a) Orkut network **(b)** Twitter network

Fig. 3. Comparison of synchronous rumor spreading with one-item memory on two real networks (—) with preferential attachment graph (—), random-attachment graph (—), and complete graph (—) of same size and density (where applicable). The Orkut network in (a) has $n = 3,072,441$ vertices and density parameter $m = 38$, the Twitter network in (b) has $n = 51,217,936$ vertices and density parameter $m = 32$. The Orkut network behaves very similarly to the corresponding preferential attachment graph. The Twitter network is even faster than the corresponding preferential attachment graph. The complete and random-attachment graphs are significantly slower.

top ten most visited sites on the Web[2]. We performed our experiments on a snapshot of the Twitter network that was crawled in September 2009 by Cha, Haddadi, Benevenuto, and Gummadi [13], available from [4]. It consists of 51,217,936 nodes and 1,963,263,821 directed edges. By making all edges undirected and considering the largest connected component, we obtained a connected graph with 51,161,011 nodes and 1,613,927,450 undirected edges. The preprocessing step of making all edges undirected might change the network structure, but the resulting network is still a typical social power law network.

Orkut is a social networking site operated by Google Inc. It is one of the top ten most visited websites in India and Brazil[2]. We used the data crawled in October and November 2006 by Mislove, Marcon, Gummadi, Druschel, and Bhattacharjee [35], which can be downloaded from [4]. The crawled graph contains 3,072,441 nodes and 117,185,083 edges. The edges are undirected, since Orkut requires consent from both users before a link between the two is created. At the time of the crawl, new users had to be invited by an existing user; therefore, the graph consists of a single component. The data covers roughly 11% of the total user population. The technical reason for this is that Orkut limits the rate at which a single IP address can download information. As a result, it took more than a month to crawl even this currently available part of the graph.

We chose these online social networks because of the available network data and because we feel that their structure might be similar to that of other real-world social networks. We are aware of the fact that interactions in Twitter

[2] See "Top 500 Sites on the web" at `www.alexa.com`.

and Orkut are more complex than in our simple randomized rumor spreading model.

We ran the protocol with one-item memory on these real-world graphs and, for comparison, on preferential attachment, random-attachment and complete graphs with size and density as close as possible to the corresponding values of the real-world graph, that is, $m = 32$ for Twitter network and $m = 38$ for the Orkut network. The numbers shown in Figure 3 are averages of 500 runs[3] for the Twitter network and 100,000 runs for the Orkut network.

Figure 3 shows that news spreads much faster in the real-world networks (▬) and the preferential attachment graphs (▬) than in the complete (▬) and random-attachment graphs (▬). Interestingly, rumor spreading in the Orkut network and the comparable preferential attachment graph proceeds very similarly, whereas the Twitter network leads to much faster rumor propagation.

5 Asynchronous Rumor Spreading

So far, we have considered only the synchronized model where all nodes take action simultaneously at discrete time steps. Depending on the circumstances, this assumption may not be plausible. In fact, the assumption of a common centralized time clock contradicts the idea of a distributed self-organized broadcasting protocol [10, 21]. Boyd et al. [10] proposed an *asynchronous time model* with a continuous time line. There, each node has its own clock that ticks at the increments of a rate 1 Poisson process independent from all other clocks, which implies that the time between two ticks is exponentially distributed with parameter 1. In the asynchronous rumor spreading protocol, every node contacts a neighbor whenever its own clock ticks, and both exchange their information. Until last year, rumor spreading in the asynchronous model has received much less attention. Very recently, the authors have studied asynchronous rumor spreading theoretically on preferential attachment graphs [23], while Fountoulakis et al. [29] studied it on Chung-Lu random graphs [17] with a given expected degree distribution. Note that Chung-Lu graphs are quite different from preferential attachment graphs, e.g., their average diameter is $\Theta(\log \log n)$ [17], whereas it is $\Theta(\log n/ \log \log n)$ [24] for preferential attachment graphs.

It is not surprising that asynchronous rumor spreading can be slow to inform all vertices. Note that it takes $\Theta(\log n)$ time until every node has performed at least one action. For this reason, in Figure 4 we consider times needed to inform 99% of the nodes. Note, however, that the times needed to inform 100% were also lower for the asynchronous model compared to the synchronous one. The charts clearly show a substantial speedup. Interestingly, for $n = 2^{23}$, the speedup for preferential (▬) and random-attachment graphs (▬) is slightly smaller

[3] The reason for the relatively small number of runs is that the Twitter network has more than one billion edges and we needed more than 50 GB of main memory to process it. A single simulation of the process required a run-time of several hours on a Hewlett Packard DL980 G7 server with eight eight-core Intel Xeon X7560 processors and 2048 GB of main memory.

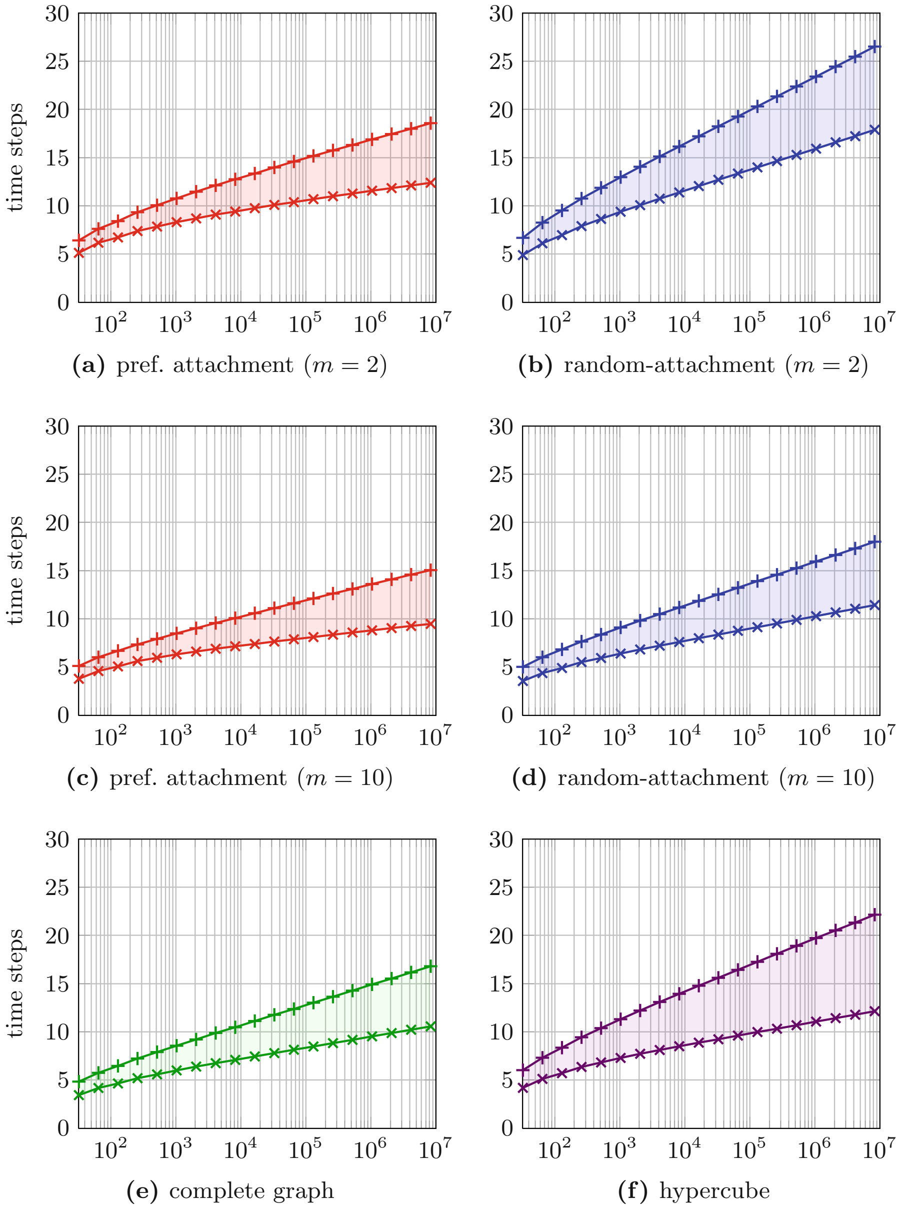
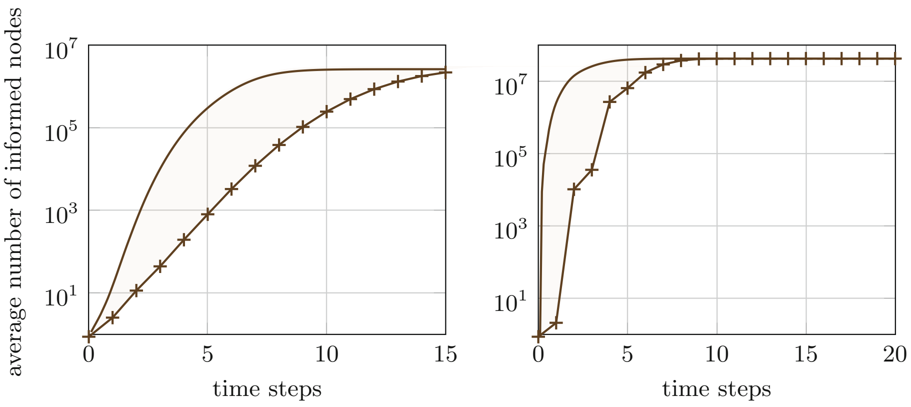


Fig. 4. Comparison of the average number of time steps needed to inform 99% of the vertices with synchronous (marked with +) and asynchronous (marked with $\times$) rumor spreading without memory on different graphs. The x-axis corresponds to the number of vertices $n = 2^5 \ldots 2^{23}$. The y-axis corresponds to the run-time to inform 99% of the vertices, averaged over 10,000 runs.
The asynchronous protocol spreads information faster than the synchronous protocol on all graphs. The difference is of the same order of magnitude for all graphs.



(a) Orkut network **(b)** Twitter network

Fig. 5. Comparison of synchronous (─┼─) and asynchronous (──) rumor spreading without memory on two real social networks. The x-axis corresponds to the time steps (in the synchronous setting) or the time (in the asynchronous setting). The y-axis corresponds to the number of informed vertices after this time, averaged over 1000 runs for the Orkut network and 50 runs for the Twitter network.
In both cases, the asynchronous counterparts spread the rumor significantly faster than the synchronous models.

(48-50% for $m = 2$ and 58-59% for $m = 10$) than for complete graphs (──) and hypercubes (──), which are 59% and 82%, respectively.

These empirical observations for moderately sized graphs are surprising given the theoretical findings on the expected asymptotic behavior. For the preferential graph, it has been shown that the time to inform $n - o(n)$ vertices without memory decreases from $\Theta(\log n)$ for the synchronous model without memory to $\mathcal{O}(\sqrt{\log n})$ for the corresponding asynchronous model [23]. On the other hand, it has been argued that random-attachment graphs, complete graphs and hypercubes keep their $\Theta(\log n)$ times, while our experiments show that the asynchronous model is faster on all graph classes. An asymptotic advantage for preferential attachment graphs is not apparent. We expect that the theoretically proven asymptotic behavior can be observed only for very large graphs. For the real-world social networks Orkut and Twitter, Figure 5 shows that, especially at the beginning, the asynchronous protocol (──) performs much faster than its synchronous counterpart (─┼─). (For a comparison between the logarithmically scaled y-axis of Figure 5 (a) and the second row of Table 1, note that after 15 time steps the synchronous protocol only informed 84% of the nodes and the asynchronous protocol informed 99.99%.) This matches well with the theoretical finding that asynchrony speeds up rumor spreading on different models of social networks [23, 29].

6 Discussion

We have empirically studied several classical rumor spreading protocols on different artificial and real-world networks. As theoretically predicted, we observed that in preferential attachment networks rumors spread significantly faster than in all other examined network models. This confirms that the structure of social networks apparently allows spreading news very efficiently. This is remarkable as social networks evolve in a random and decentralized manner and are *not* designed with this purpose in mind.

The experiments also gave a much more detailed picture than possible purely theoretically. It has been demonstrated that in order to design a fast rumor-propagation algorithm on social networks, modeled by preferential attachment graphs, one needs small memory that helps to decide which node to contact next. This again seems to be specific to such networks as memory helps other network topologies much less. We also observed that a surprisingly small amount of memory is sufficient.

While theoretical results for models of social networks predicted a large speed-up when allowing asynchronous communication, we observed that other network topologies can benefit even more. The difference between synchronous and asynchronous propagation is very apparent for the two real-world networks Orkut and Twitter. We also observed that the speed of information spreading is very similar in the Orkut network and a preferential attachment graph of comparable density. Future work should include other rumor spreading protocols (e.g. [2, 11, 12]), more artificial graphs (e.g. [17]), and preferably even larger real-world networks like Facebook, which has close to one billion nodes.

References

[1] Barabási, A.-L., Albert, R.: Emergence of scaling in random networks. Science 286, 509–512 (1999)

[2] Baumann, H., Fraigniaud, P., Harutyunyan, H.A., de Verclos, R.: The Worst Case Behavior of Randomized Gossip. In: Agrawal, M., Cooper, S.B., Li, A. (eds.) TAMC 2012. LNCS, vol. 7287, pp. 330–345. Springer, Heidelberg (2012)

[3] Berger, N., Borgs, C., Chayes, J.T., Saberi, A.: On the spread of viruses on the Internet. In: 16th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 301–310 (2005)

[4] Bhattacharjee, B., Druschel, P., Gummadi, K., et al.: Online social networks research at the Max Planck Institute for Software Systems, http://socialnetworks.mpi-sws.org

[5] Bohman, T., Frieze, A.M.: Hamilton cycles in 3-out. Random Structures & Algorithms 35, 393–417 (2009)

[6] Bollobás, B., Riordan, O.: Robustness and vulnerability of scale-free random graphs. Internet Mathematics 1, 1–35 (2003)

[7] Bollobás, B., Riordan, O.: Coupling scale-free and classical random graphs. Internet Mathematics 1, 215–225 (2003b)

[8] Bollobás, B., Riordan, O.: The diameter of a scale-free random graph. Combinatorica 24, 5–34 (2004)
[9] Bollobás, B., Riordan, O., Spencer, J., Tusnády, G.: The degree sequence of a scale-free random graph process. Random Structures & Algorithms 18, 279–290 (2001)
[10] Boyd, S., Ghosh, A., Prabhakar, B., Shah, D.: Randomized gossip algorithms. IEEE Transactions on Information Theory 52, 2508–2530 (2006)
[11] Censor-Hillel, K., Shachnai, H.: Fast information spreading in graphs with large weak conductance. In: 22nd ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 440–448 (2011)
[12] Censor-Hillel, K., Haeupler, B., Kelner, J.A., Maymounkov, P.: Global computation in a poorly connected world: Fast rumor spreading with no dependence on conductance. In: 44th ACM Symposium on Theory of Computing (STOC), pp. 961–970 (2012)
[13] Cha, M., Haddadi, H., Benevenuto, F., Gummadi, P.K.: Measuring user influence in Twitter: The million follower fallacy. In: 4th International AAAI Conference on Weblogs and Social Media, ICWSM (2010)
[14] Chierichetti, F., Lattanzi, S., Panconesi, A.: Rumour spreading and graph conductance. In: 21st ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1657–1663 (2010)
[15] Chierichetti, F., Lattanzi, S., Panconesi, A.: Almost tight bounds for rumour spreading with conductance. In: 42nd ACM Symposium on Theory of Computing (STOC), pp. 399–408 (2010)
[16] Chierichetti, F., Lattanzi, S., Panconesi, A.: Rumor spreading in social networks. Theoretical Computer Science 412, 2602–2610 (2011)
[17] Chung, F.R.K., Lu, L.: The average distance in a random graph with given expected degrees. Internet Mathematics 1, 91–113 (2003)
[18] Cooper, C., Frieze, A.M.: The cover time of the preferential attachment graph. Journal of Combinatorial Theory, Series B 97, 269–290 (2007)
[19] Demers, A.J., Greene, D.H., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H.E., Swinehart, D.C., Terry, D.B.: Epidemic algorithms for replicated database maintenance. Operating Systems Review 22, 8–32 (1988)
[20] Doerr, B., Friedrich, T., Sauerwald, T.: Quasirandom rumor spreading. In: 19th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 773–781 (2008)
[21] Doerr, B., Friedrich, T., Künnemann, M., Sauerwald, T.: Quasirandom rumor spreading: An experimental analysis. In: 10th Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 145–153 (2009)
[22] Doerr, B., Fouz, M., Friedrich, T.: Social networks spread rumors in sublogarithmic time. In: 43rd ACM Symposium on Theory of Computing (STOC), pp. 21–30 (2011)
[23] Doerr, B., Fouz, M., Friedrich, T.: Asynchronous Rumor Spreading in Preferential Attachment Graphs. In: Fomin, F.V., Kaski, P. (eds.) SWAT 2012. LNCS, vol. 7357, pp. 307–315. Springer, Heidelberg (2012)
[24] Dommers, S., van der Hofstad, R., Hooghiemstray, G.: Diameters in preferential attachment models. J. of Statistical Physics 139, 72–107 (2010)
[25] Elsässer, R.: On the communication complexity of randomized broadcasting in random-like graphs. In: 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 148–157 (2006)
[26] Elsässer, R., Sauerwald, T.: On the power of memory in randomized broadcasting. In: 19th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 218–227 (2008)

[27] Feige, U., Peleg, D., Raghavan, P., Upfal, E.: Randomized broadcast in networks. Rand. Struct. & Algo. 1, 447–460 (1990)
[28] Flaxman, A.D., Frieze, A.M., Vera, J.: Adversarial deletion in a scale-free random graph process. Comb., Probab. & Comput. 16, 261–270 (2007)
[29] Fountoulakis, N., Panagiotou, K., Sauerwald, T.: Ultra-fast rumor spreading in social networks. In: 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1642–1660 (2012)
[30] Frieze, A.M., Grimmett, G.R.: The shortest-path problem for graphs with random arc-lengths. Discrete Applied Mathematics 10, 57–77 (1985)
[31] Giakkoupis, G.: Tight bounds for rumor spreading in graphs of a given conductance. In: 28th International Symposium on Theoretical Aspects of Computer Science (STACS), pp. 57–68 (2011)
[32] Karp, R., Schindelhauer, C., Shenker, S., Vöcking, B.: Randomized rumor spreading. In: 41st IEEE Symposium on Foundations of Computer Science (FOCS), pp. 565–574 (2000)
[33] Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: 44th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 482–491 (2003)
[34] Kempe, D., Kleinberg, J.M., Demers, A.J.: Spatial gossip and resource location protocols. J. ACM 51(6), 943–967 (2004)
[35] Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In: 7th ACM SIGCOMM Conference on Internet Measurement (IMC), pp. 29–42 (2007)
[36] Pittel, B.: On spreading a rumor. SIAM Journal on Applied Mathematics 47, 213–223 (1987)

A Randomised Approximation Algorithm for the Partial Vertex Cover Problem in Hypergraphs

Mourad El Ouali[1], Helena Fohlin[2], and Anand Srivastav[1]

[1] Department of Computer Science, University of Kiel. Germany
{meo,asr}@informatik.uni-kiel.de
[2] Department of Clinical and Experimental Medicine, Linköping University, Sweden
Helena.Fohlin@lio.se

Abstract. In this paper we present an approximation algorithm for the k-partial vertex cover problem in hypergraphs. Let $\mathcal{H} = (V, \mathcal{E})$ be a hypergraph with set of vertices $V, |V| = n$ and set of (hyper-)edges $|\mathcal{E}|, |\mathcal{E}| = m$. The k-partial vertex cover problem in hypergraphs is the problem of finding a minimum cardinality subset of vertices in which at least k hyperedges are incident. It is a generalisation of the fundamental (partial) vertex cover problem in graphs and the hitting set problem in hypergraphs. Let $l, l \geq 2$ be the maximum size of an edge, Δ be the maximum vertex degree and D be maximum edge degree. For a constant $l, l \geq 2$ a non-approximabilty result is known: an approximation ratio better than l cannot be achieved in polynomial-time under the unique games conjecture (Khot and Rageev 2003, 2008). On the other hand, with the primal-dual method (Gandhi, Khuller, Srinivasan 2001) and the local-ratio method (Bar-Yehuda 2001), the l-approximation ratio can be proved. Thus approximations below the l-ratio for large classes of hypergraphs, for example those with constant D or Δ are interesting. In case of graphs ($l = 2$) such results are known. In this paper we break the l-approximation barrier for hypergraph classes with constant D resp. Δ for the partial vertex cover problem in hypergraphs. We propose a randomised algorithm of hybrid type which combines LP-based randomised rounding and greedy repairing. For hypergraphs with arbitrary l, $l \geq 3$, and constant D the algorithm achieves an approximation ratio of $l(1 - \Omega(1/(D+1)))$, and this can be improved to $l(1 - \Omega(1/\Delta))$ if Δ is constant and $k \geq m/4$. For the class of l-uniform hypergraphs with both l and Δ being constants and $l \leq 4\Delta$, we get a further improvement to a ratio of $l\left(1 - \frac{l-1}{4\Delta}\right)$. The analysis relies on concentration inequalities and combinatorial arguments.

Keywords: Combinatorial optimization, approximation algorithms, hypergarphs, vertex cover, probabilistic methods.

1 Introduction

A hypergraph $\mathcal{H} = (V, \mathcal{E})$ consists of a set V, say $|V| = n$, and a set $\mathcal{E}$ of subsets of V, $|\mathcal{E}| = m$. We call the elements of V vertices and the elements of $\mathcal{E}$

(hyper-)edges. The k-partial vertex cover in hypergraphs can be stated as follows. A set $X \subseteq V$ is called a k-partial vertex cover for $\mathcal{H}$ if at least k edges of $\mathcal{H}$ are incident in X. The (unweighted) k-partial vertex cover problem for hypergraphs is to find a k-vertex cover of minimum cardinality. If k is equal to the number of hyperedges, we have the well-known hitting set problem (or vertex cover problem) in hypergraphs. For graphs it is the classical vertex cover problem in combinatorial optimization, whose approximation complexity has been studied for nearly 4 decades.

Among the motivations to study hypergraphs are not only the natural question of generalising graph theory to hypergraphs (C. Berge [4]), but also relevant applications in areas where hypergraphs are most natural, e.g. data structures in computational geometry like ϵ-nets [20], which are a kind of hitting sets, or discrepancy theory [19].

Previous Works. Consider a minimisation problem. For $\rho \geq 1$ we say that a (polynomial-time) algorithm achieves an ρ-approximation or an approximation ratio of ρ, if it computes for all instances a solution of value at most ρOpt, where Opt is the value of an optimal solution to the problem.

For the k-partial vertex cover problem in *graphs*, it has been an open question for which graphs the 2-approximation can be improved. For graphs with maximum vertex degree at most Δ, Gandhi, Khuller and Srinivasan [10] gave the first algorithm with approximation ratio smaller than 2. Improvements have been obtained by Halperin and Srinivasan [12]. For hypergraphs, the hitting set and the set cover problem have been investgated intensively in the context of polynomial time approximations [5, 6, 11, 13, 16, 17] and complexity of non-approximability [1, 7, 18, 23]. For the hitting set problem in l-uniform hypergraphs with constant l, an approximation with a ratio better than l cannot be achieved in polynomial time under the unique games conjecture (UGC) [15]. Since the hitting set problem in hypergraphs is a special case of the k-vertex cover problem in hypergraphs with $k = m$, this hardness of approximation holds also for the k-vertex cover problem in hypergraphs. On the other hand, for the k-partial vertex cover problem in hypergraphs with edge size at most l (l not necessarily constant), Bar-Yehuda [3] gave an algorithm based on the local-ratio method with approximation guarantee is l. Later Gandhi, Khuller and Srinivasan [10] achieved the same ratio, using a primal/dual approach. Thus polynomial-time approximations below the l-ratio for significant classes of hypergraphs are complexity-theoretic and algorithmically interesting and would extend the approximation theory for the k-partial vertex cover problem from graphs to hypergraphs.

Our Contribution. In this paper we present a randomised algorithm for the partial vertex cover problem in hypergraphs which achieves approximation ratios below l for hypergraphs with constant D or constant Δ. First we consider hypergraphs with edge size at most l, $l \geq 3$, not necessarily constant: in case that the maximum edge degree is at most a constant D an approximation ratio of $l(1 - \Omega(1/(D+1)))$ can be proved. In case that the vertex degree is at most a constant Δ, we can show an improved ratio of $l(1 - \Omega(1/\Delta))$, provided that $k \geq m/4$. As we are interested in hypergraphs, and graphs have already been considered in previous work, the (technical) restriction $l \geq 3$ is of less concern. However, we believe that an optimization of constants in this paper may cover graphs as well. Finally, with a different analysis we show for l-uniform hypergraphs , $l \geq 2$, with constant Δ, l and $l \leq 4\Delta$, the further improved approximation ratio of $l\left(1 - \frac{l-1}{4\Delta}\right)$. As Δ can be assumed to be at least 2, the result automatically covers all l-uniform hypergraphs with $2 \leq l \leq 8$ and constant Δ.

Outline of the Paper. The paper is organised as follows: In Section 2 we give definitions and probabilistic tools. In Section 3 we present our randomised algorithm for the k-partial vertex cover for hypergraphs and state estimations of expectations and variance. In Section 4 resp. 5 we analyse the approximation ratio for hypergraphs with constant D resp. Δ. In Section 6 we analyse the algorithm for l-uniform hypergraphs where l and Δ are constants.

2 Preliminaries and Definitions

Graph-theoretical Notions. Let $\mathcal{H} = (V, \mathcal{E})$ be a hypergraph, with V and $\mathcal{E}$ its set of vertices and edges. For $v \in V$ we define $d(v) = |\{E \in \mathcal{E}; v \in E\}|$ and $\Delta = \max_{v \in V}\{d(v)\}$. Here $d(v)$ is the vertex-degree of v and Δ is the maximum vertex degree of $\mathcal{H}$. For a edge $E \in \mathcal{E}$, let $\deg(E)$ be the (edge-) degree of E and D the maximum edge degree of $\mathcal{H}$, i. e. $\deg(E) = |\{F \in \mathcal{E}; E \cap F \neq \emptyset\}|$ and $D = \max_{E \in \mathcal{E}} \deg(E)$. Further for a set $X \subseteq V$ we denote by $\Gamma(X) := \{E \in \mathcal{E}\,;\, X \cap E \neq \emptyset\}$ the set of edges incident to the set X. Let $l \in \mathbb{N}$ be given constant, we call $\mathcal{H}$ l-uniform resp. l-bounded , if $|E| = l$ resp. $|E| \leq l$ for all $E \in \mathcal{E}$. It is convenient to order the vertices and hyperedges, $V = \{v_1, \dots, v_n\}$ and $\mathcal{E} = \{E_1, \dots, E_m\}$, and to identify vertices and edges with their indices.

Concentration Inequalities. For the one-sided deviation the following Chebychev-Cantelli inequality will be frequently used:

Theorem 1 ([2]). *Let X be a non-negative random variable with finite mean $\mathbb{E}(X)$ and variance $\mathrm{Var}(X)$. Then for any $a > 0$ we have*

$$\Pr(X \geq \mathbb{E}(X) + a) \leq \frac{\mathrm{Var}(X)}{\mathrm{Var}(X) + a^2}.$$

A further useful concentration result is the independent bounded differences inequality theorem:

Theorem 2 (see [21]). *Let $X = (X_1, X_2, ..., X_n)$ be a family of independent random variables with X_k taking values in a set A_k for each k. Suppose that the real-valued function f defined on $\Pi_{k=1}^{n} A_k$ satisfies $|f(x) - f(x')| \leq c_k$ if the vector x and x' differ only in the k-th coordinate. Let $\mathbb{E}(X)$ be the expected value of the random variable $f(X)$. Then for any $t > 0$ it holds*

$$\Pr(f(X) \leq \mathbb{E}(f(X)) - t) \leq \exp\left(\frac{-2t^2}{\sum_{k=1}^{n} c_k^2}\right).$$

The following estimate on the variance of a sum of dependent random variables can be proved as in Alon, Spencer.

Lemma 1 (see [2]). *Let X be the sum of finitely many $0/1$ random variables, i.e. $X = \sum_{i=1}^{n} X_i$, and let $p_i = \mathbb{E}(X_i)$ for all $i = 1, \dots, n$. For a pair $i, j \in \{1, \dots, n\}$ we write $i \sim j$, if X_i and X_j are dependent. Let Γ be the set of all unordered dependent pairs i, j and $\gamma = \sum_{\{i,j\} \in \Gamma} \mathbb{E}(X_i X_j)$, then it holds*

$$\mathrm{Var(X)} \leq \mathbb{E}(\mathrm{X}) + 2\gamma.$$

3 Randomised Algorithm for Partial *k*-Vertex Cover

The input is a l-bounded hypergraph $\mathcal{H}$, $l \geq 2$. At the moment we do not assume D or Δ to be constants. An integer linear programming formulation of the k-partial vertex cover in $\mathcal{H}$ is the following:

$$\begin{aligned}
&(\mathrm{ILP}-k-\mathrm{VC}) \ \min \sum_{j=1}^{n} x_j \\
&\sum_{j=1}^{n} x_j \geq z_i \quad \text{for all } i \in [m] := \{1, \dots, m\} \\
&\sum_{i=1}^{m} z_i \geq k \\
&x_j, z_i \in \{0, 1\} \quad \text{for all } i \in [m], j \in [n].
\end{aligned}$$

Its linear programming relaxation, denoted by LP-k-VC, is given by relaxing the integrality constraints to $X_j, Z_i \in [0, 1]$ $\forall i \in [m], j \in [n]$. Let Opt resp. Opt* be the value of optimal solution to ILP-k-VC resp. LP-k-VC. Let x^* and z^* be an optimal solution of LP-k-VC. Let $\epsilon \in [0, 1]$, we set $\lambda := l(1-\epsilon)$, $S_1 := \{j \in [n] \mid x_j^* = 1\}$, $S_{\geq} := \{j \in [n] \mid 1 \neq x_j^* \geq \frac{1}{\lambda}\}$ and $S_{\leq} := \{j \in [n] \mid 0 \neq x_j^* < \frac{1}{\lambda}\}$.

Algorithm 1. VC-$\mathcal{H}$

```
Input  : A hypergraph $\mathcal{H} = (V, \mathcal{E})$ and an integer $k$
Output: A partial vertex cover $C$
1. Initialise $C := \emptyset$.
2. Solve the LP relaxation of ILP-$k$-VC
3. Take all vertices of $S_1$ and $S_{\geq}$ into the cover $C$. Set $V := V \setminus S_1$ and
   $\mathcal{E} := \mathcal{E} \setminus \Gamma(S_1)$.
4. (Randomised Rounding) For all vertices $j \in S_{\leq}$ include the vertex $j$ in the cover
   $C$, independently for all such $j$ with probability $x_j^* \lambda$.
5. (Repairing) Repair the cover $C$ (if necessary) as follows:
   a) If $|\{E \in \mathcal{E} \mid E \cap C \neq \emptyset\}| \geq k$, then return $C$.
   b) If $|\{E \in \mathcal{E} \mid E \cap C \neq \emptyset\}| < k$, then pick at most $k - |C|$ additional vertices
      from arbitrary not covered edges in the cover.
6. Return the cover $C$.
```

The algorithm VC-$\mathcal{H}$ extends the randomised algorithm of Gandhi, Khuller and Srinivasan [10] from graphs to hypergraphs. While the extension is quite natural, the analysis needs efforts beyond [10], for example variance computation for dependent sums of random variables, stronger concentration inequalities adapted to situations for bounded edge resp. vertex degree (section 4 and 5) or a novel approach for the estimation of the expectation of the objective function *including* the greedy repairing step (section 6).

Computation of Expection and Variance
Let $X_1, ..., X_n$ be $\{0,1\}$-random variables defined as follows:

$$X_j = \begin{cases} 1 & \text{if the vertex } v_j \text{ was picked into the cover after the rounding step} \\ 0 & \text{otherwise.} \end{cases}$$

Note that the $X_1, ..., X_n$ are independent. For all $i \in [m]$ we define the $\{0,1\}$-random variables Z_i as follows

$$Z_i = \begin{cases} 1 & \text{if the the hyperedge } E_i \text{ is covered after the rounding step} \\ 0 & \text{otherwise.} \end{cases}$$

Then $Y := \sum_{j=1}^{n} X_j$ is the cardinality of the cover after the randomised rounding step in the algorithm and $W = \sum_{j=1}^{m} Z_j$ is the number of covered hyperedges after this step.

For the expected size of the cover we have the following upper bound:

$$\mathbb{E}(|C|) \leq \mathbb{E}(Y) + \mathbb{E}(\max\{k - W, 0\}) \tag{1}$$

For the computation of the expectation of W we need the following lemma that gives the exact solution of a constrained optimization problem (See Lemma 2.2 [22]).

Lemma 2. *For all $n \in \mathbb{N}$, $\lambda > 0$ and $x_1, \cdots, x_n, z \in [0,1]$ with $\sum_{i=1}^{n} x_i \geq z$ and $\lambda x_i < 1$ for all $i \in \mathbb{N}$, we have $\prod_{i=1}^{n}(1 - \lambda x_i) \leq (1 - \lambda \frac{z}{n})^n$, and this bound is the tight maximum.*

For the analysis of the algorithm we need also the following lemma

Lemma 3. *Let D and Δ be as above, not assumed to be constants and let $\epsilon > 0$.*

(i) $(1 - (1-\epsilon)x)^2 \leq 1 - x(1 - \epsilon^2)$ *For all* $x \in [0,1]$.
(ii) $\mathbb{E}(W) \geq (1 - \epsilon^2)k$.
(iii) $\mathrm{Var}(W) \leq (D+1)\mathbb{E}(W)$.
(iv) $\mathrm{Opt}^* \leq \mathbb{E}(Y) \leq \lambda \mathrm{Opt}^*$.
(v) $\mathrm{Opt}^* \geq \frac{k}{\Delta} \geq \frac{k}{D+1}$.
(vi) *Let $\mathcal{H} = (V, \mathcal{E})$ be a uniform hypergraph with edge size l. Then* $\sum_{v \in V} d(v) = l|\mathcal{E}|$

Proof. (i). By straightforward calculations.

(ii)Let $i \in [m]$ and $|E_i| = r$. If there is a $j \in E_i$ with $\lambda x_j \geq 1$ then $\Pr(Z_i = 0) = 0$, else we have

$$\begin{aligned} P(Z_i = 0) &= \prod_{j \in \mathcal{E}_i} (1 - \lambda x_j^*) \underset{\text{Lem 2}}{\leq} \left(1 - \frac{\lambda z_i}{r}\right)^r \leq \left(1 - \frac{\lambda z_i^*}{l}\right)^r \\ &\leq (1 - (1-\epsilon) z_i^*)^2 \underset{\text{Lem 3 (i)}}{\leq} 1 - z_i^*(1 - \epsilon^2) \end{aligned}$$

and we get

$$\begin{aligned} \mathbb{E}(W) &= \sum_{i=1}^{m} \Pr(Z_i = 1) = \sum_{i=1}^{m} (1 - \Pr(Z_i = 0)) \\ &\geq \sum_{i=1}^{m} (1 - (1 - z_i^*(1 - \epsilon^2)) = (1 - \epsilon^2) \underbrace{\sum_{i=1}^{m} z_i^*}_{\geq k} \geq (1 - \epsilon^2) k. \end{aligned}$$

(iii) Let $\varGamma$ and γ like in Lemma 1. Furthermore for every $E_i, E_j \in \mathcal{E}, Z_i, Z_j$ are dependent iff the hyperedges E_i and E_j have non-empty intersection. Thus, for a fixed E_i, there are at the most D random variables Z_j depending on Z_i. Furthermore it holds for every $E_i, E_j \in \mathcal{E}$:

$\mathbb{E}(Z_i Z_j) = \Pr(Z_i = 1 \wedge Z_j = 1) \leq \min\{\Pr(Z_i = 1), \Pr(Z_j = 1)\} \leq \frac{\Pr(Z_i=1)+\Pr(Z_j=1)}{2}$.
Moreover

$$\begin{aligned} \gamma &= \sum_{\{E_i, E_j\} \in \varGamma} \mathbb{E}(Z_i Z_j) \leq \sum_{\{E_i, E_j\} \in \varGamma} \frac{\Pr(Z_i = 1) + \Pr(Z_j = 1)}{2} \\ &\leq \sum_{i=1}^{m} \frac{D}{2} \Pr(Z_i = 1) = \frac{D}{2} \sum_{i=1}^{m} \Pr(Z_i = 1) = \frac{D}{2} \mathbb{E}(W) \end{aligned}$$

so with Lemma 1 $\mathrm{Var}(W) \leq \mathbb{E}(W) + 2\mathbb{E}(Z_i Z_j) \leq (D+1)\mathbb{E}(W)$.

(iv) By using the LP relaxation and the definition of the sets S_1, $S_{\geq}$ and $S_{\leq}$, and since $\lambda \geq 1$, we get

$$\mathrm{Opt}^* \leq \underbrace{|S_1| + |S_{\geq}| + \lambda \mathrm{Opt}^*(S_{\leq})}_{= \mathbb{E}(|C|)} \leq \underbrace{|S_1|}_{= \mathrm{Opt}^*(S_1)} + \underbrace{|S_{\geq}|}_{\leq \lambda \mathrm{Opt}^*(S_{\geq})} + \lambda \mathrm{Opt}^*(S_{\leq}) \leq \lambda \mathrm{Opt}^*.$$

(v)Let $d(v_j)$ the degree of the vertex v_j. With the ILP constraints we have
$k \leq \sum_{i=1}^{m} z_i^* \leq \sum_{i=1}^{m} \sum_{j \in E_i} x_j^* = \sum_{j=1}^{n} d(v_j) x_j^* \leq \varDelta \cdot \mathrm{Opt}^* \leq (D+1) \cdot \mathrm{Opt}^*$

(vi) Let $\mathcal{H} = (V, \mathcal{E})$ be a hypergraph and r_j the size of the hyperedge E_j for $j \in [m]$. Then $\sum_{v \in V} d(v) = \sum_{E_j \in \mathcal{E}} r_j$ and since $\mathcal{H}$ is uniform with edge size l, the assertion holds. □

4 Analysis for Bounded Edge Degree

In this section we consider hypergraphs with arbitrary $l \in \mathbb{N}, l \geq 3$ (l is not necessarily assumed to be a constant), but with bounded edge degree (so D is constant).

Theorem 3. *Let $\mathcal{H}$ be a hypergraph with edge size at most l, $l \in \mathbb{N}, l \geq 3$, and bounded edge degree D. The algorithm k-VC returns a k-partial vertex cover C such that*

$$|C| \leq l\left(1 - \Omega\left(\frac{1}{D+1}\right)\right)\text{Opt} \quad \textit{with probability at least} \quad \frac{3}{5}.$$

***Proof*:** First we choose

$$\epsilon := \frac{\text{Opt}^*(1+\beta)}{k} \quad \text{for} \quad \beta = \frac{1}{3(D+1)}. \tag{2}$$

We can assume that

$$\epsilon \leq \frac{1+\beta}{l-\eta}, \quad \text{where} \quad \eta = \frac{l}{6(D+1)}, \tag{3}$$

because otherwise it follows from the definition of ϵ in (2) that $\text{Opt}^* \geq \frac{k}{l-\eta}$, hence $l(1-\frac{\eta}{l})\text{Opt}^* \geq k$. Since a partial cover of size k can be trivially found by picking k arbitrary hyperedges and taking one vertex from each of them pairwise distinct, we can get a $l(1-\frac{\eta}{l})$-approximation —i.e. a ratio strictly smaller than l— in this case.

Since $l \geq 3$ and $D \geq 1$, it is straightforward to check that (3) implies $\epsilon \leq \frac{1}{2}$, so $\lambda = l(1-\epsilon) \geq \frac{l}{2} > 1$.

Claim 1.

$$\Pr\left(W \leq k(1-\epsilon^2) - 2\sqrt{k(D+1)}\right) \leq \frac{1}{5}$$

Proof of Claim 1. First we consider the function:

$$f\colon \mathbb{R}_{>D+1} \to \mathbb{R}, f(x) = x - 2\sqrt{(D+1)x}.$$

Since $f'(x) = 1 - \sqrt{\frac{D+1}{x}} > 0$, f is monotonely increasing. For $k < 4(D+1)$, it holds:

$$k(1-\epsilon^2) - 2\sqrt{k(D+1)} \leq \underbrace{k - 2\sqrt{k(D+1)}}_{=f(k)} < f(4(D+1)) = 0,$$

and because $W \geq 0$ also $\quad \Pr\left(W \leq k(1-\epsilon^2) - 2\sqrt{k(D+1)}\right) = 0$.

Let now $k \geq 4(D+1)$. We set $\mu := \mathbb{E}(W)$. Then, as $\epsilon \leq \frac{1}{2}$

$$k(1-\epsilon^2) \geq 4(D+1)(1-\epsilon^2) > 4(D+1)(1-\frac{1}{4}) = 3(D+1) > D+1,$$

so by Lemma 3(ii) $f(k(1-\epsilon^2)) \leq f(\mu)$. Furthermore,

$$\begin{aligned}\Pr\left(W \leq k(1-\epsilon^2) - 2\sqrt{k(D+1)}\right) &\leq \Pr\left(W \leq k(1-\epsilon^2) - 2\sqrt{k(1-\epsilon^2)(D+1)}\right)\\ &\leq \Pr\left(W \leq \mu - 2\sqrt{(D+1)\mu}\right)\\ &\underset{\text{Th 1}}{\leq} \frac{1}{1+\frac{4\mu(D+1)}{\mathrm{Var}(W)}} \underset{\text{Lem 3}(iii)}{\leq} \frac{1}{1+\frac{4\mu(D+1)}{\mu(D+1)}} = \frac{1}{5}.\end{aligned}$$

This concludes the proof of Claim 1.

Claim 2. For $\beta = \frac{1}{3(D+1)}$ it holds $\Pr(Y \geq l \cdot \mathrm{Opt}^*(1-\epsilon)(1+\beta)) < \frac{1}{5}$.

Proof of Claim 2. W.l.o.g. we may assume for a constant $\alpha > 16$ that

$$k \geq \alpha(D+1)^5. \tag{4}$$

Otherwise, if $k < \alpha(D+1)^5$, we would be able to solve the problem in polynomial time: since $\mathrm{Opt}^* \leq k$, our assumption of D being a constant allows to find the optimal solution by enumerating all subsets of V of size at most k in polynomial time. Furthermore we have:

Since the resulting random variables $X_1, ..., X_n$ after the rounding step are independent, the Chernoff bound shows that

$$\Pr(Y \geq l(1-\epsilon)(1+\beta)\mathrm{Opt}^*) \underset{\text{Lem 3}(iv)}{\leq} \Pr(Y \geq \mathbb{E}(Y)(1+\beta)) \leq \exp\left(-\frac{\beta^2\mathbb{E}(Y)}{3}\right).$$

On the other hand we have

$$\frac{\mathbb{E}(Y)\beta^2}{3} \underset{\text{Lem 3}(iv)}{\geq} \frac{\mathrm{Opt}^*}{27(D+1)^2} \underset{\text{Lem 3}(v)}{\geq} \frac{k}{27(D+1)^3}.$$

Since $k \geq \alpha(D+1)^5$ and $\alpha \geq 16$ we finally get:

$$\Pr(Y \geq l(1-\epsilon)(1+\beta)\mathrm{Opt}^*) \leq \exp\left(-\frac{16(D+1)^2}{27}\right) \leq \exp\left(-\frac{64}{27}\right) < \frac{1}{5}.$$

This concludes the proof of Claim 2.

By Claim 1 and 2 we get with probability at least $1 - (\frac{1}{5} + \frac{1}{5}) = \frac{3}{5}$ an upper bound for the final cover:

$$|C| \leq \underbrace{l(1-\epsilon)(1+\beta)\mathrm{Opt}^* + k\epsilon^2}_{(*)} + \underbrace{2\sqrt{k(D+1)}}_{(**)}.$$

It holds

$$\begin{aligned}(*) \quad &= \quad l\left((1+\beta)(1-\epsilon) + \frac{\mathrm{Opt}^*(1+\beta)^2}{lk}\right)\mathrm{Opt}^*\\ &\underset{\text{Lem 3}(v)}{\leq} l(1+\beta)\left(1 - \frac{(l-1)(1+\beta)}{l(D+1)}\right)\mathrm{Opt}^* = l\left(1+\beta - \frac{(l-1)(1+\beta)^2}{l(D+1)}\right)\mathrm{Opt}^*.\end{aligned}$$

On the other hand we can easily check, using $l \geq 3$, that $\frac{(l-1)(1+\beta)^2}{l(D+1)} - \beta \geq \frac{1}{3(D+1)}$, therefore

$$l(1-\epsilon)(1+\beta)\mathrm{Opt}^* + k\epsilon^2 \leq l\left(1 - \frac{1}{3(D+1)}\right)\mathrm{Opt}^*.$$

Next, by Lemma 3 (v) and inequality (4) we have: $\mathrm{Opt}^* \geq \frac{k}{D+1} \geq \alpha(D+1)^4$ and thus get:

$$\begin{aligned} 2\sqrt{k(D+1)} = 2\sqrt{\frac{k}{D+1}}(D+1) &\underset{\text{Lem 3}(v)}{\leq} 2\sqrt{\mathrm{Opt}^*}(D+1) \\ &\underset{\text{Ineq (4) and Lem 3}(v)}{\leq} 2\mathrm{Opt}^* \frac{1}{\sqrt{\alpha}(D+1)}. \end{aligned}$$

Finally, the sum of $(*)$ and $(**)$ is

$$(*) + (**) \underset{\alpha \geq 16 \text{ and } l \geq 3}{\leq} l\left(1 - \frac{1}{6(D+1)}\right)\mathrm{Opt}^*.$$

The randomised algorithm returns with probability at least $\frac{3}{5}$ a cover C of cardinality at most $l\left(1 - \Omega\left(\frac{1}{(D+1)}\right)\right)\mathrm{Opt}^*$. □

5 Analysis for Constant Vertex Degree

In this section we consider hypergraphs with edge size at most l, $l \in \mathbb{N}, l \geq 3$ (l is not necessarily assumed to be a constant), but with bounded vertex degree (so Δ is constant).

Theorem 4. *Let $\mathcal{H}$ be a hypergraph with edge size at most l, $l \in \mathbb{N}, l \geq 3$, and bounded vertex degree Δ. For $k \geq \frac{m}{4}$ the algorithm k-VC returns a k-partial vertex cover C such that*

$$|C| \leq l\left(1 - \Omega\left(\frac{1}{\Delta}\right)\right)\mathrm{Opt} \quad \textit{with probability at least} \quad \frac{3}{5}.$$

This is an improvement over the last section, at least for $k \geq m/4$, as always $\Delta \leq D$.

Proof. First we assume that $\mathcal{H}$ is a uniform hypergraph with edge size l. As mentioned above, l is not necessarily assumed to be a constant. Let us choose

$$\epsilon := \frac{\mathrm{Opt}^*(1+\beta_1)}{k} \quad \text{for} \quad \beta_1 = \frac{1}{3\Delta}. \tag{5}$$

We can assume that

$$\epsilon \leq \frac{1+\beta_1}{l-\eta}, \quad \text{where} \quad \eta = \frac{l}{6\Delta}. \tag{6}$$

The rest of the proof is similar to the proof of Theorem 3, except Claim 1. We replace Claim 1 in the proof of Theorem 3 by the following claim:

Claim 4.

$$\Pr\left(W \leq k(1-\epsilon^2) - 2\sqrt{kl\Delta}\right) \leq \frac{1}{5}.$$

Proof of Claim 4. Let f be the function defined as follows:

$$f : \{0,1\}^n \longrightarrow \mathbb{N} \quad f(X_1, ..., X_n) = \sum_{j=1}^{m} Z_j.$$

Then f is component-wise Lipschitz bounded:

$$|f(X_1, .., X_k, .., X_n) - f(X_1, .., X_k^{'}, .., X_n)| \leq d(v_k).$$

Since the $X_1, ..., X_n$ are chosen independently at random, by Theorem 2 we get for any $t \geq 0$

$$\Pr(f(X) - \mathbb{E}(f(X)) \leq -t) \leq \exp\left(\frac{-2t^2}{\sum_{v\in V} d(v)^2}\right). \tag{7}$$

We choose $t = 2\sqrt{\Delta lk}$. Since $\mathcal{H}$ is uniform with size edge l, by Lemma 3 (ii) we have

$$\begin{aligned}\Pr\left(f(X) \leq k(1-\epsilon^2) - 2\sqrt{kl\Delta}\right) &= \Pr\left(W \leq k(1-\epsilon^2) - 2\sqrt{kl\Delta}\right)\\ &\underset{\text{Lem 3}\,(ii)}{\leq} \Pr\left(W \leq \mathbb{E}(W) - 2\sqrt{kl\Delta}\right)\\ &\underset{\text{Ineq (7)}}{\leq} \exp\frac{-8l\Delta k}{\sum_{v\in V} d(v)^2} \underset{k\geq\frac{m}{4}}{<} \frac{1}{5}.\end{aligned}$$

This concludes the proof of Claim 4.

Claim 5. For $\beta_1 = \frac{1}{3\Delta}$ it holds $\Pr\left(Y \geq l \cdot \text{Opt}^*(1-\epsilon)(1+\beta_1)\right) < \frac{1}{5}$.

Proof of Claim 5. W.l.o.g. we may assume for an constant $\alpha > 16$ that $k \geq \alpha\Delta^5$, than the Claim 5 holds on the same maner as Claim 2.
Hence by Claim 4 and 5 we get an upper bound for the final cover with probability at least $1 - (\frac{1}{5} + \frac{1}{5}) \geq \frac{3}{5}$:

$$|C| \leq \underbrace{l(1-\epsilon)(1+\beta_1)\text{Opt}^* + k\epsilon^2}_{(*)} + \underbrace{2\sqrt{kl\Delta}}_{(**)}.$$

As in proof of theorem 3 it holds

$$(*) \leq l\left(1 - \frac{1}{3\Delta}\right)\text{Opt}^* \text{ and } (**) \leq 2l\text{Opt}^* \frac{1}{\sqrt{l\alpha\Delta}}.$$

Hence

$$(*) + (**) \leq l\text{Opt}^*\left(1 - \frac{1}{3\Delta}\right) + l\text{Opt}^* \frac{2}{\sqrt{l\alpha\Delta}} \underset{\alpha\geq 16 \text{ and } l\geq 3}{\leq} l\left(1 - \frac{2-\sqrt{3}}{6\Delta}\right)\text{Opt}^*.$$

The randomised algorithm returns with probability at least $\frac{3}{5}$ a cover C with cardinality at most $l\left(1 - \Omega\left(\frac{1}{\Delta}\right)\right)\mathrm{Opt}^*$.

The algorithm VC-$\mathcal{H}$ can be extended to hypergraphs with edge size at most l, by adding $l - |E|$ dummy vertices to every edge E. It is obvious that the new hypergraph is uniform with edge size l and the other assumptions being unchanged, thus Theorem 4 holds. □

6 Analysis for Constant l and Δ

Instead bounding the error probability of the randomised rounding step and the repairing step separately as above, in this section we consider the expected size of the cover including repairing, and then use concentration inequalities, a more elegant and efficient way of dealing with coupled random variables. This will lead to the better approximation ratio $l\left(1 - \frac{l-1}{4\Delta}\right)\mathrm{Opt}$. This ratio requires $l \leq 4\Delta$. As Δ is assumed to be constant, l is constant as well. That is the reason why (at the moment) we cannot transfer this approach to the more general setting in section 4 and 5, where l is not necessarily constant.

Now, for the moment, if not specified otherwise, we consider l-uniform hypergraphs with bounded vertex degree.

For a set $S \subset \{1, ..., n\}$ let $\mathrm{Opt}^*(S) := \sum_{j\in S} x_j^*$. By (1) it holds

$$\begin{aligned}\mathbb{E}(|C|) \leq &\quad \mathrm{Opt}^*(S_1) + \lambda\left(\mathrm{Opt}^*(S_\geq) + \mathrm{Opt}^*(S_\leq)\right) + k\epsilon^2 \\ = &\quad \mathrm{Opt}^*(S_1) + l(1-\epsilon)(\mathrm{Opt}^*(S_\geq) + \mathrm{Opt}^*(S_\leq)) + k\epsilon^2\end{aligned} \tag{8}$$

We consider the function

$$f\colon [0,1] \to \mathbb{R}, \epsilon \mapsto \mathrm{Opt}^*(S_1) + l(1-\epsilon)(\mathrm{Opt}^*\,(S_\geq) + \mathrm{Opt}^*(S_\leq) + k\epsilon^2.$$

f attains its minimum for

$$\epsilon = \frac{l(\mathrm{Opt}^*\,(S_\geq) + \mathrm{Opt}^*(S_\leq))}{2k}. \tag{9}$$

Moreover we can assume that $\frac{l\left(\mathrm{Opt}^*(S_\geq)+\mathrm{Opt}^*(S_\leq)\right)}{2k} \in [0,1]$. Otherwise, if $\frac{l\left(\mathrm{Opt}^*(S_\geq)+\mathrm{Opt}^*(S_\leq)\right)}{2k} > 1$ then $\frac{l}{2}\mathrm{Opt}^* \geq \frac{l}{2}\left((\mathrm{Opt}^*(S_\geq) + \mathrm{Opt}^*(S_\leq)\right) > k$. Since any k-partial vertex cover of cardinality k can be found trivially, this approximates the optimum within a factor of $\frac{l}{2} < l$.

Let $S_f := S_\geq \cup S_\leq \backslash \{j \in [n] | x_j^* = 0\}$. Plugging in ϵ from (9) into (8), we get

$$\mathbb{E}(|C|) \leq \mathrm{Opt}^*(S_1) + l\left(1 - \frac{l\mathrm{Opt}^*(S_f)}{4k}\right)\mathrm{Opt}^*(S_f). \tag{10}$$

We observe here that the LP-based sparsening of the instance becomes relevant. For the variance of the cover we have,

Lemma 4. *Let $X_1, \ldots, X_n$ be the 0/1-random variables returned by algorithm VC-$\mathcal{H}$. Then we have* $\mathrm{Var}(|C|) \leq l\Delta\mathbb{E}(|C|)$.

Proof. The proof is along the lines of the proof of Lemma 3 (iii). □

Let $\tilde{\mathcal{H}} = (\tilde{V}, \tilde{\mathcal{E}})$ be the sub-hypergraph of $\mathcal{H}$ constructed in step 3 of the algorithm VC-$\mathcal{H}$ with $|\tilde{V}| = \tilde{n}$ and $|\tilde{\mathcal{E}}| = \tilde{m}$. We denote by $\tilde{l}$ and $\tilde{\Delta}$ the maximum size of all edges and the maximum vertex degree in $\tilde{\mathcal{H}}$. We consider the LP relaxation of the ILP formulation of the partial vertex cover problem in $\tilde{\mathcal{H}}$ for a covering factor $\tilde{k} := k - |\Gamma(S_1)|$ which we denote by LP($\tilde{\mathcal{H}}$). By Opt$^*(\tilde{\mathcal{H}})$ we denote the value of the optimal solution of LP($\tilde{\mathcal{H}}$). The optimal LP solution for $\mathcal{H}$ is Opt*. Then the following holds.

Lemma 5. $\mathrm{Opt}^*(\tilde{\mathcal{H}}) = \mathrm{Opt}^* - |S_1|$ *and* $\mathbb{E}(|C|) \le |S_1| + \mathbb{E}(|\tilde{C}|)$.

Lemma 6. *Let $\mathcal{H}$ be a l-uniform hypergraph. Then it holds*

$$\mathbb{E}(|C|) \le l\left(1 - \frac{l}{4\Delta}\right)\mathrm{Opt}^*. \tag{11}$$

Proof. As there are no 1's in the solution $(\tilde{x}_1, \ldots, \tilde{x}_{\tilde{n}})$, there is no tight LP($\tilde{\mathcal{H}}$)-variable, using (8) we get (8)

$$\mathbb{E}(|\tilde{C}|) \le \tilde{l}\left(1 - \frac{\tilde{l}\mathrm{Opt}^*(\tilde{\mathcal{H}})}{4\tilde{k}}\right)\mathrm{Opt}^*(\tilde{\mathcal{H}}) \underset{\text{Lem 3}(iii)}{\le} \tilde{l}\left(1 - \frac{\tilde{l}}{4\tilde{\Delta}}\right)\mathrm{Opt}^*(\tilde{\mathcal{H}}).$$

By Lemma 5 and since $\lambda \ge 1$ we have $\mathbb{E}(|C|)\tilde{l}\left(1 - \frac{\tilde{l}}{4\tilde{\Delta}}\right)\mathrm{Opt}^*$, and because $\mathcal{H}$ is uniform and $\Delta \ge \tilde{\Delta}$ we conclude that $\mathbb{E}(|C|) \le l\left(1 - \frac{l}{4\Delta}\right)\mathrm{Opt}^*$. □

Lemma 6 and Lemma 4 imply the following theorem using the Chebyshev-Cantelli inequality and standard calculations.

Theorem 5. *Let l, Δ be constants and let $\mathcal{H}$ be an l-uniform hypergraph with bounded vertex degree Δ. We further assume that $l \le 4\Delta$. Then the algorithm VC-$\mathcal{H}$ returns a k-partial vertex cover C such that*

$$|C| \le l\left(1 - \frac{l-1}{4\Delta}\right)\mathrm{Opt}^* \textit{ with probability at least } \frac{3}{4}.$$

Proof. W.l.o.g. we may assume that $k \ge 16\Delta^5$. Otherwise, we can solve the problem by enumeration. We have

$$\Pr\left(|C| \ge l\left(1 - \frac{l-1}{4\Delta}\right)\mathrm{Opt}^*\right) \underset{\text{Th 6}}{\le} \Pr\left(|C| \ge \mathbb{E}(|C|) + \frac{l\mathrm{Opt}^*}{4\Delta}\right) \le \frac{1}{1 + \frac{\left(\frac{l\mathrm{Opt}^*}{4\Delta}\right)^2}{\mathrm{Var}(|C|)}}.$$

Furthermore for $k \ge 16\Delta^5$ we get,

$$\frac{\left(\frac{l\mathrm{Opt}^*}{4\Delta}\right)^2}{\mathrm{Var}(|C|)} \underset{\text{Lem 4}}{\ge} \left(l\frac{(\mathrm{Opt}^*)^2}{16\Delta^3\mathbb{E}(|C|)}\right) \underset{\mathbb{E}(|C|)\le l\mathrm{Opt}^*}{\ge} \frac{\mathrm{Opt}^*}{16\Delta^3} \underset{\text{Lem 3}(iii)}{\ge} \frac{\Delta k}{16\Delta^5} \ge \Delta.$$

Therefore we get $\Pr\left(|C| \ge l\left(1 - \frac{l-1}{4\Delta}\right)\mathrm{Opt}^*\right) \le \frac{1}{1+\Delta} \underset{\Delta\ge 3}{\le} \frac{1}{4}$. □

Remark 1. Note that Theorem 5 automatically covers all uniform hypergraphs with constant Δ and $2 \leq l \leq 8$ as we can (trivially) assume $\Delta \geq 2$. This result is thus a natural generalisation of known approximations below l for graphs to hypergraphs, whenever the (hyper-)graph vertex degree is a constant.

In the same manner as in section 5, Theorem 5 can be extended to l-bounded hypergraphs with constant l.

7 Further Work

It would be interesting to give a better approximation for hypergraphs with other kind of sparseness conditions. Another challenge is the derandomisation of this and other hybrid algorithms combining randomised rounding and greedy heuristics.

References

[1] Alon, N., Moshkovitz, D., Safra, S.: Algorithmic construction of sets for k-restrictions. ACM Trans. Algorithms (ACM) 2, 153–177 (2006)
[2] Alon, N., Spencer, J.: The probabilistic method, 2nd edn. Wiley Interscience (2000)
[3] Bar-Yehuda, R.: Using homogeneous weights for approximating the partial cover problem. Journal of Algorithms 39(2), 137–144 (2001)
[4] Berge, C.: Hypergraphs- combinatorics of finite sets. North Holland Mathematical Library (1989)
[5] Chvátal, V.: A greedy heuristic for the set covering problem. Math. Oper. Res. 4(3), 233–235 (1979)
[6] Duh, R., Fürer, M.: Approximating k-set cover by semi-local optimization. In: Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC 1997), pp. 256–264 (May 1997)
[7] Feige, U.: A treshold of $\ln n$ for approximating set cover. Journal of the ACM 45(4), 634–652 (1998)
[8] Feige, U., Langberg, M.: Approximation algorithms for maximization problems arising in graph partitioning. Journal of Algorithms 41(2), 174–201 (2001)
[9] Frieze, A., Jerrum, M.: Improved approximation algorithms for max k-cut and max bisection. Algorithmica 18, 67–81 (1997)
[10] Gandhi, R., Khuller, S., Srinivasan, A.: Approximation Algorithms for Partial Covering Problems. J. Algorithms 53(1), 55–84 (2004)
[11] Halperin, E.: Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. In: ACM-SIAM Symposium on Discrete Algorithms, vol. 11, pp. 329–337 (2000)
[12] Halperin, E., Srinivasan, A.: Improved Approximation Algorithms for the Partial Vertex Cover Problem. In: Jansen, K., Leonardi, S., Vazirani, V.V. (eds.) APPROX 2002. LNCS, vol. 2462, pp. 161–174. Springer, Heidelberg (2002)
[13] Hochbaum, D.S.: Approximation algorithms for the set covering and vertex cover problems. SIAM J. Computation 11(3), 555–556 (1982)
[14] Jäger, G., Srivastav, A.: Improved approximation algorithms for maximum graph partitioning problems. Journal of Combinatorial Optimization 10(2), 133–167 (2005)

[15] Khot, S., Regev, O.: Vertex cover might be hard to approximate to within 2-epsilon. J. Comput. Syst. Sci. 74(3), 335–349 (2008)
[16] Krivelevich, J.: Approximate set covering in uniform hypergraphs. J. Algorithms 25(1), 118–143 (1997)
[17] Lovász, L.: On the ratio of optimal integral and fractional covers. Discrete Math. 13, 383–390 (1975)
[18] Lund, C., Yannakakis, M.: On the hardness of approximating minimization problems. J. Assoc. Comput. Mach. 41, 960–981 (1994)
[19] Matousek, J.: Geometric Discrepancy. Algorithms and Combinatorics, vol. 18. Springer, Heidelberg (2010)
[20] Matousek, J., Wagner, U.: New Constructions of Weak epsilon-Nets. Discrete & Computational Geometry 32(2), 195–206 (2004)
[21] McDiarmid, C.: On the method of bounded differences. Surveys in Combinatorics, Norwich, pp. 148–188. Cambridge Univ. Press, Cambridge (1989)
[22] Peleg, D., Schechtman, G., Wool, A.: Randomized approximation of bounded multicovering problems. Algorithmica 18(1), 44–66 (1997)
[23] Raz, R., Safra, S.: A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In: Proc. 29th ACM Symp. on Theory of Computing, pp. 475–484 (1997)

Simulation-Based Analysis of Topology Control Algorithms for Wireless Ad Hoc Networks

Fabian Fuchs, Markus Völker, and Dorothea Wagner

Institute of Theoretical Informatics
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{fabian.fuchs,markus.voelker,dorothea.wagner}@kit.edu

Abstract. Topology control aims at optimizing throughput and energy consumption of wireless networks by adjusting transmission powers or by restricting the communication to a well-chosen subset of communication links. Over the years, a variety of topology control algorithms have been proposed. However, many of these algorithms have been mainly studied from a theoretical point of view. On the other hand, existing simulation-based studies often only compare few approaches based on rather simple simulations, e.g., abstracting from communication protocols.

In this paper, we present a thorough study of a variety of topology control algorithms based on the methodology of algorithm engineering. To analyze achievable performance improvements for communication according to the IEEE 802.11g standard we use the ns-3 network simulator. In addition to analyzing the communication throughput, we also study the effects of topology control on the energy demand in the network. Based on our simulation results, we then identify properties of the computed topologies that are essential for the achieved improvements. The gained insights are finally used to motivate an extension of the well-known XTC algorithm, which enables significant performance improvements in the considered application scenario.

Keywords: topology control, algorithms, wireless communication, wireless ad hoc network, network simulator, ns-3, IEEE 802.11g, energy consumption, throughput.

1 Introduction

As the amount of data that is sent through wireless ad hoc networks increases, network structures that can serve this increased demand are needed. However, as energy is a limited resource in most ad hoc networks, this must be done while also achieving low energy consumption. The goal of topology control is to increase the network throughput, for example by reducing interference between concurrent transmissions, while simultaneously decreasing the energy consumption. To achieve this goal, nodes can adjust their transmission powers and restrict communication to a subset of their neighbors.

Although many different strategies have been proposed how the communication links that are used in the network topology should be selected, there is still no real consensus about what distinguishes good topologies from bad ones. A minimum requirement of almost all approaches is that the computed topology should be connected. Additionally, most algorithms try to compute rather sparse topologies. This strategy is motivated by the assumption that interference is minimized if every node has only few neighbors in the communication graph. However, so far there exists no real indication whether this strategy is really advantageous under realistic circumstances.

To verify the aforementioned assumption and to identify general properties of good communication topologies, we present in this work a broad comparison of various topology control algorithms. While existing simulation studies usually use rather simple simulations to compare two or three approaches with each other, in this work we use the powerful ns-3 network simulator [1] to analyze a broad variety of topology control algorithms with respect to their effect on network throughput and energy demand. We chose to use the ns-3 network simulator, which is a standard tool in the networking community, as it already implements a variety of standard communication protocols. This allows us to identify and study additional effects of topology control in connection with communication according to the IEEE 802.11g standard that are usually not modeled in simple self-implemented simulations.

We then try to interpret performance differences between the considered algorithms based on fundamental properties of the computed topologies, e.g., the average number of intermediate nodes that are necessary to enable communication between distant nodes. Based on the gained insights, we propose a simple extension of the well-known XTC algorithm. This extension simply consists in adding all links that exceed a certain link quality to the original XTC topology. According to our simulation results, this simple modification already results in significant performance improvements in comparison to the standard XTC approach. The modified algorithm, XTC_{RLS}, outperforms all other approaches in our simulations. At the same time, it preserves the advantages of XTC, i.e., it produces a connected topology based solely on local information about received signal strengths.

The rest of this paper is structured as follows: In Section 2 we define the topology control problem and introduce some basic terminology. Related work is discussed in Section 3. The topology control algorithms that are analyzed in this work are then introduced in some more detail in Section 4. The simulation setup is described in Section 5 and in Section 6 the simulation results are presented. Section 7 concludes this work.

2 Problem Definition and Terminology

Networks can naturally be modeled as graphs by mapping network nodes to graph nodes and possible communication links $l = (u, v)$ between two nodes u and v to edges between the corresponding nodes in the graph. Using this notation, the problem of topology control can be seen as a selection of edges such that some desired properties are achieved. For a graph $G = (V, E)$, a subgraph

$G' = (V, E')$ with $E' \subseteq E$ must be computed such that efficient communication, high throughput, maximal network lifetime or similar criteria are achieved. Depending on the considered application, the desired properties may differ and hence several different quality criteria are commonly considered. Typical examples for such quality criteria are connectivity, symmetry, stretch factors, sparseness, throughput, and planarity [2].

The *length* of a link $l = (u, v)$ denotes the Euclidean distance d_{uv} between nodes u and v. If node u sends a signal to node v with transmission power P_{uv}, the strength of the signal usually decreases on the way from u to v with increasing distance to u. Additionally, obstacles and interference effects result in additional signal attenuation. All these effects are subsumed in the *link gain* γ_{uv}, which defines how much the signal decreases on its way from u to v. If node u sends with transmission power P_{uv}, the signal is received by v with power $\gamma_{uv} P_{uv}$.

In wireless networks, the maximum transmission power that a node can use is usually limited. In consequence, nodes can only communicate directly within a certain communication radius. In order to allow communication between more distant nodes, multi-hop communication has to be used in which additional nodes relay the messages. If a message is routed on a path $(n_1, n_2, \ldots, n_k)$ from node n_1 to node n_k, each direct communication between two consecutive nodes along the path is called a *hop*. Given a communication graph G, the *hop-distance* between two nodes u and v is defined as the number of edges of a shortest path from u to v in G.

3 Related Work

Topology control has been a very vivid research area in the past years. The topology control problem has been considered isolated as well as combined with other aspects of wireless communication such as scheduling, clustering or routing. For both directions, a wide variety of algorithms has been proposed. In this work, we focus on pure topology control algorithms, as this enables us to analyze the effects of topology control apart from other influences.

Early approaches to the topology control problem were often graph-based and featured well-known graph-theoretic algorithms. Examples are a local variant of the minimum spanning tree (LMST) [3], the Gabriel graph (GG) [2], or the Relative Neighborhood graph (RNG) [2]. The cone-based topology control (CBTC) method [4] and the Yao graph (YG) [5] are other approaches that are based on angular separations between nodes and signal strengths of communication links. In the COMPOW protocol [6], all nodes try to find a minimal but common transmission power while preserving network connectivity. The XTC algorithm [7] computes a topology that is similar to the RNG; however, it is capable of achieving this without knowledge about node positions but based on signal strengths. More recently, kTC [8] and Inclusive Directed RNG (IDRNG) [9] have been proposed as improvements on XTC and the Directed RNG topology control algorithms. Another approach is k-Neigh, which locally selects up to k neighbors for each node [10]. Surveys that give a more extensive overview on existing topology control algorithms can be found in [11,12]. In Section 4, we will give a more detailed description of those topology control algorithms that are studied in this paper.

Most topology control algorithms have been mainly analyzed on a theoretical basis, e.g., by proving upper bounds on node degrees or proving certain spanner properties. Experimental and simulation-based comparisons are only seldom given. In the rest of this section, we will give an overview on the major real-world and simulation-based studies.

In [13], Jeong *et al.* studied the throughput and the energy consumption of several variants of the k-Neigh protocol using Mica2dot nodes. They showed that dynamic determination of the transmission power can increase the throughput and decrease the energy consumption in comparison to fixed transmission powers. In [14], Duràn *et al.* apply topology control algorithms such as the Gabriel graph, Relative Neighborhood graph, Yao graph and the Delaunay triangulation to multi-hop cellular networks. Based on data from a real-world network, they found that topology control can help to achieve significantly higher signal-to-interference-plus-noise ratios (SINR). In their experiments, the RNG algorithm achieved the best link quality.

In [15], Xu *et al.* use the ns-2 network simulator to study a newly proposed topology control algorithm in comparison to the original network without topology control. In [16], Gao *et al.* propose the MaxSR topology and compare the throughput of the proposed topology with LMST, CBTC, and the original topology. The energy consumption of the LMST and the R&M protocol has been considered in [17]. The authors found that if each node sends only one packet to a sink node then the LMST topology achieves the lowest energy consumption.

The work that is most similar to our work is [18], where Blough *et al.* use the GTNetS simulator, a network simulator similar to ns-3, to study the throughput that can be achieved with topology control in IEEE 802.11 networks. They consider a minimal spanning tree topology, two variants of the k-Neigh algorithm, CBTC, a common power topology and a max-power topology. Blough *et al.* conclude from their simulations that on the one hand the common power topology does not increase the throughput and the minimum spanning tree even decreases the throughput while on the other hand topology control algorithms such as k-Neigh and CBTC can improve the throughput significantly.

4 Examined Algorithms

As there exists a large variety of topology control algorithms (many of which are very similar), it is not possible to study all of them in detail. For this reason, we chose for our simulations a set of well-known algorithms that cover most of the different approaches towards topology control. In the following, we give a brief overview and short descriptions of the algorithms and topologies that are covered in our simulation study.

The first topology that we consider is the topology one gets when every sender uses the maximum transmission power and when all possible communication links are used. We call this topology *All-Links-Graph (ALG)*. Of course, all other topologies are a subset of the ALG topology. In *XTC* [7], a link between two nodes is used if there is no third node within reach that has equal or higher signal strength to both of the nodes. The *kTC* [8] algorithm searches triangles in the one-hop neighborhood of a node and discards the longest edge of such triangles if this edge is k times longer than the shortest edge in the triangle. This is similar to the approach in XTC. However, XTC discards each longest edge within such a triangle. The kTC algorithm is more robust towards errors in the perception of the received signal strength due to the parameter k and a local consensus on the signal strength. The parameter k is chosen as $1.41 \leq \sqrt{2}$, so that kTC is still a subset of the Gabriel graph and the distinction towards XTC is maximal. In the *Inclusive Directed RNG (IDRNG)* [9], communication links are selected in two steps: First the DRNG algorithm is executed, which discards communication links that do not have an empty relative neighborhood. Then each node determines the transmission power it has to use in order to reach all neighbors and adds all neighbors that can be reached using this transmission power to the topology. The *Localized Euclidean MST (LMST)* [3] is based on a local computation of Prim's algorithm in order to compute a minimum spanning tree. Each node computes the minimum spanning tree in its one-hop neighborhood and selects the communication links that are used in this tree. In the *Gabriel Graph (GG)*, a communication link from node u to node v is discarded if the circle with diameter $\text{dist}(u, v)$ that has u and v on its boundary is not empty. The area that must be empty is smaller than the relative neighborhood considered in some of the previous algorithms. The *Yao Graph (YG)* [5] divides the surroundings of each node in $c = 6$ cones of equal angle and adds a communication link to the nearest neighbor in each cone. If there are two or more nearest neighbors, one neighbor is chosen arbitrarily. Bi-directionality of the constructed topology is ensured by forcing uni-directional edges to be bi-directional. The *k-Neighborhood Protocol (k-Neigh)* [10] computes a topology consisting of the k nearest neighbors of each node. k-Neigh does not necessarily yield a connected topology for lower values of k. We use $k = 8$, which should easily suffice to ensure connectivity [10]. Uni-directional edges are discarded from the topology.

4.1 XTC$_{\mathbf{RLS}}$

The algorithms discussed so far all aim at rather sparse topologies. To validate whether it is really advantageous to exclude many links from the communication, we also implemented a strategy that only excludes very weak links and keeps all links for which the received signal strength exceeds some threshold. In our simulations, the resulting topologies usually resulted in very good network performance. However, this approach has one major drawback: Depending on the considered network, it sometimes is necessary to use certain very weak links in order to guarantee connectivity of the communication topology. Thus, it sometimes happened that the computed topologies were not connected.

To deal with this problem efficiently, we propose the following extension of XTC: First, the XTC algorithm is used to create a sparse topology that is guaranteed to be connected. Afterwards, each node additionally adds all those communication links whose signal strengths exceed a given threshold. We experimentally determined -86 dBm to be a good value for the threshold and use this value in our simulations. In the following, we will refer to the described extension of XTC as XTC_{RLS}, where the RLS reminds of the restricted link strength. Note that similar to XTC, the $\mathrm{XTC_{RLS}}$ topology can be easily computed in a distributed fashion based solely on local information about received signal strengths.

5 Simulation Setup

For our simulations we used version 3.13 of the network simulator ns-3 [1], which is designed for network related research and implements a wide variety of communication standards and protocols. In each simulation run, 60 nodes that are equipped with a wireless communication device according to the IEEE 802.11g standard are distributed randomly in a square-shaped deployment area. To adjust the node density, we vary the base lengths of the deployment area between 100 m and 600 m. Among the 60 nodes, 18 source-target pairs are randomly selected and each source node must transmit 5 MB of data to the corresponding target node. Note that due to the multi-hop communication this accumulates to up to 900 MB of data that must be transmitted across the network, depending on the average number of hops that are necessary for communication between distant nodes (see Section 6.1).

For the communication, we use end-to-end TCP connections in a CSMA/CA[1]-based network using the Open Link State Routing (OLSR) protocol. As ns-3 does not offer a standard framework to support topology control mechanisms, e.g., by restricting communication to a given subset of communication links, we adjusted the OLSR routing protocol such that it only uses the links that are selected by the considered topology control algorithm.

To model signal decay in our simulations, we use the standard log-distance path loss model as implemented in ns-3. According to the log-distance model, the path loss L in dB is given as

$$L = L_0 + 10 \cdot \alpha \cdot \log_{10}\left(\frac{d}{d_0}\right) \quad ,$$

where L_0 is the reference path loss at distance d_0, α is the path loss exponent, and d is the distance between sender and receiver. In our simulations we use $L_0 = 46.6777, d_0 = 1$, and $\alpha = 3$, which correspond to an average free space environment with some obstacles.

In each simulation run we measure the time needed to finish all transmissions and the energy that is consumed during the transmissions, as well as the time the

[1] Carrier Sense Multiple Access with Collision Avoidance; implements collision avoidance in IEEE 802.11 wireless networks.

network devices spent in the TX, RX, IDLE and CCA_BUSY[2] states to allow the subsequent application of different energy models to our simulation results. Additionally we analyze basic properties of the computed topologies such as average node degrees, average sender-receiver distances, and the average number of hops that are necessary to allow communication between the source and target nodes. Our plots depict median values based on 50 independent simulation runs. The time limit of the simulations was set to 5000 seconds.

Energy Model. While properties such as the achieved throughput or the average number of hops are rather unambiguous, the consumed energy strongly depends on assumptions about the used communication hardware. Choosing an energy model that applies for a wide variety of available wireless network devices is a difficult task. We decided to adjust the energy consumption in our simulations to the energy consumption of the Roving Networks RN-174, which has a relatively low energy demand. Table 1 states some of the relevant specifications as stated in the corresponding data sheet.

Table 1. Energy consumption of an RN-174, measured at 3.3 V DC [20]. The transmission and the reception states are abbreviated by TX and RX.

State	Idle	RX	TX (dBm)						
			0	2	4	6	8	10	12
Current (mA)	40	40	135	150	190	200	210	225	240

As we need the energy consumption for a wider range of transmission powers, we use a linear least squares fit to interpolate the current that is drawn from the battery for arbitrary transmission powers. This gives

$$\text{TxCurrent}(\text{TxPower}) = \begin{cases} 8.66 \cdot \text{TxPower} + 140.89 & \text{if TxPower} \geq -10, \\ 54.28 & \text{else,} \end{cases}$$

where TxPower is the transmission power in dBm and the function returns the current in mA. We consider the network to be active even after all transmissions are finished. Hence, we subtract the idle power consumption from the power consumption of the other states (i.e., computationally eliminate the idle power consumption). Note that due to this only transmission time and transmission power affect the energy consumption. We will see in Section 6 that this does not change the results of our analysis as the energy consumption calculated according to this measure still correlates to the time needed to finish the transmissions.

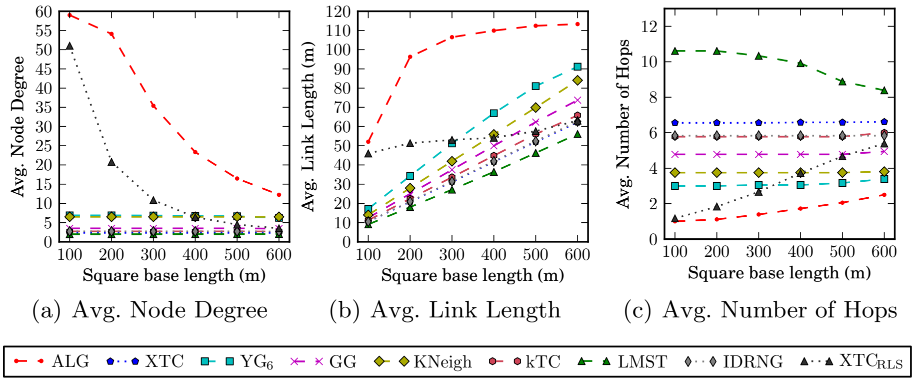
[2] We assume, according to [19], that CCA_BUSY requires the same amount of energy as the IDLE state.

6 Experiments

6.1 Basic Topology Properties

Although the main focus of our simulations lies on the effects of different topologies on the throughput and the energy consumption in wireless networks, we first want to briefly examine some basic properties of the computed topologies. This will allow us later to gain a deeper understanding of why certain topologies perform better than others.

First we take a look at the average node degree of the computed topologies, which is depicted in Figure 1(a). Apparently, all algorithms but ALG and XTC_{RLS} produce topologies with very low node degrees. Additionally, for those algorithms the average node degree is almost independent of the node density. In contrast, for high node densities ALG and XTC_{RLS} have rather high average node degrees. For example, for the deployment area of 100 m × 100 m almost all nodes can communicate directly in the ALG topology. Not surprisingly, for networks with low node density the average node degree of XTC_{RLS} approaches the one of XTC.



(a) Avg. Node Degree (b) Avg. Link Length (c) Avg. Number of Hops

Fig. 1. Basic properties of the computed topologies in dependence of the deployment area

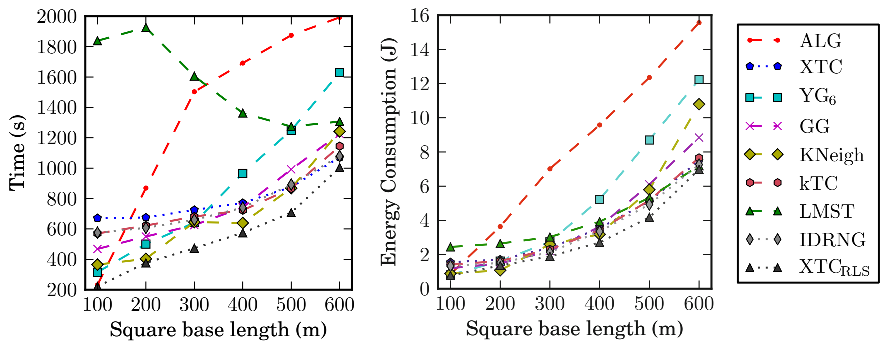
Concerning the average link length, i.e., the average distance between pairs of nodes that are allowed to communicate directly with each other, for high node densities ALG and XTC_{RLS} result in a higher average link length than the other approaches (cf. Figure 1(b)). However, while the average link length of ALG increases up to more than 100 m for large deployment areas, the average link length of XTC_{RLS} only approaches the one of XTC and is thus lower than the link lengths of most other approaches. The reason for this of course is that XTC_{RLS} only adds links with relatively high link gain, which means that links that exceed about 50 m are only chosen if they are also chosen by XTC.

Figure 1(c) depicts the average number of hops that are necessary to enable communication between the randomly selected source-target pairs. For most of the topologies the number of hops is almost independent of the node density, whereas in the ALG and XTC_{RLS} topologies the average number of hops starts close to one and rises with decreasing node density. In the following, we will see that both the required time to complete all transmissions and the consumed energy are strongly correlated with the average number of required hops. This can be explained by the fact that each message has to be sent once for each hop. Thus, the overall number of packets that are transmitted within the network also depends on the average number of hops per source-target pair.

6.2 Topology Control with Uniform Transmission Powers

In this section we study the influence of topology control when all senders use the maximum available transmission power. In this setting, performance improvements can only be achieved by avoiding weak links and by keeping the interference in the network low. As we saw in Section 6.1, most of the considered topology control algorithms try to achieve this by using only few links with very high link gain.

Communication Throughput. First we examine the time that is needed to finish all transmissions, which is shown in Figure 2(a). We observe that ALG, the original topology without topology control, is only competitive for the deployment area of 100 × 100 meters. Once the nodes are deployed on an area



(a) Required Time (b) Consumed Energy

Fig. 2. Influence of topology control on the required time to finish all transmissions and on the consumed energy

of 200 × 200 meters or larger, almost all other topologies can finish faster, i.e., achieve higher throughput. This presents a strong motivation for the use of topology control.

The ALG topology works well for small deployment areas as in the resulting networks most source-target pairs can communicate directly (mostly with a reasonable data rate). Since the ALG topology does not discard weak links, for larger deployment areas the OLSR routing algorithm selects more links that achieve a relatively low signal strength and hence can only communicate with low data rate and maybe even with a high packet error rate. The very poor performance of LMST can be explained in terms of bottlenecks that emerge when all communication is limited to a tree-like backbone. Surprisingly, for larger deployment areas the throughput of LMST actually improves. The reason is that the local approximation to a minimum spanning tree on average is worse and hence uses more edges on large deployment areas.

All other topologies show a similar tendency that the throughput slowly decreases with increasing deployment area. According to Figure 1(c), for most topologies this can not be explained in terms of a higher number of hops between source nodes and target nodes. Instead, this tendency is most likely caused by the fact that the average link length increases for larger deployment areas (cf. Fig. 1(b)). A higher link length also means a lower link gain, which in IEEE 802.11g networks finally causes that the achievable data rate over the link is lower.

By comparing Fig. 1(c) with Fig. 2(a), one can see that especially for small deployment areas the time that is needed to complete all transmissions is correlated to the average number of communication hops. Those topologies that require more hops usually also require more time to finish all transmissions. With increasing deployment area, however, this effect decreases as the data rate that can be achieved over the links becomes increasingly important. For networks that use a fixed data rate, the average number of hops of the topology is expected to be even more relevant for the throughput performance.

Note that our simulation results contradict the usual assumption that sparser topologies increase the throughput in the network or lower the energy consumption. Especially when comparing XTC with $\mathrm{XTC}_{\mathrm{RLS}}$, it clearly seems to pay off to include additional links as long as they provide sufficiently high link gain. However, a comparison of $\mathrm{XTC}_{\mathrm{RLS}}$ with ALG also reveals that it really helps to avoid links with low link gain.

Energy Consumption. Regarding the energy consumption, which is shown in Fig. 2(b), we observe that for deployment areas above $200\,\mathrm{m} \times 200\,\mathrm{m}$ all considered topologies are more energy-efficient than the original ALG topology. At the first moment this might seem a little surprising. We did not adjust transmission powers, which implies that transmissions over long distances use the same transmission power as short-range transmissions. However, in the considered IEEE 802.11g communication protocol another factor that often is not considered comes into play: For transmissions over short distances the link gain is usually higher, which means that a higher data rate is used. This implies that short-range transmissions are finished faster and the wireless network devices spend less time in the energy-consuming TX state.

The rapid increase in energy consumption of the ALG topology for larger deployment areas can thus easily be explained by the use of long-range communication links that only allow for very low data rates. Interestingly, although the

LMST topology uses only very short links, it results in the highest energy consumption of all topologies for dense networks. This observation can be explained by the high number of hops that are on average necessary to transfer data from one node to another (cf. Fig. 1(c)). Concerning $\mathrm{XTC}_{\mathrm{RLS}}$, it not only produces the topology that allows for the highest throughput in the considered scenario, but it also results in the lowest energy consumption for most of the considered node densities.

6.3 Topology Control in Combination with Power Control

In addition to restricting the communication to a well-chosen subset of all possible links, it is also sometimes proposed that the network performance can be additionally improved by reducing the transmission power that is used for communication between nodes that are located close together. In this section, we examine the influence of such an additional optimization.

Communication Throughput. As transmission power reduction can help to reduce interference effects between concurrent transmissions, it seems reasonable to assume that throughput improvements are possible if short-range transmissions use lower transmission powers. However, existing studies on this topic came to different conclusions: In [13], the authors stated that for *k*-Neigh on Mica2dot nodes transmission power reduction does not increase the throughput performance but at most achieves similar throughput. In contrast, the authors of [18] showed with simulations using the GTNetS simulator that an additional transmission power reduction can produce significant throughput improvements for various topology control algorithms.

To further study this topic, we conducted similar experiments as before (i.e., multi-hop communication between 18 random source-target pairs in a network of 60 nodes) but reduced the transmission powers for transmissions over short distances. Again, the time that passes until all transmissions are finished is analyzed. We consider three different reduction thresholds: the transmission power of each node is reduced such that the communication partner with the lowest received signal strength can receive the signal with −80 dBm, −85 dBm and −90 dBm, respectively. If the received signal strength of a communication link is already lower than this threshold, the transmission power is not reduced. The times needed to finish all transmissions are depicted for the three considered thresholds in Figure 3.

We observe that the reduction of transmission powers such that the received signal strengths do not exceed −85 dBm or −90 dBm actually results in lower throughput. This is due to the data rate management in IEEE 802.11 wireless communication, which determines the used bitrate based on the measured signal-to-interference-plus-noise ratio (SINR). The higher the SINR, the higher the possible data rate. Thus, especially links with high link gain are negatively affected by the transmission power reduction. However, if the transmission powers are only reduced so far that the received signal strengths still equal −80 dBm, most

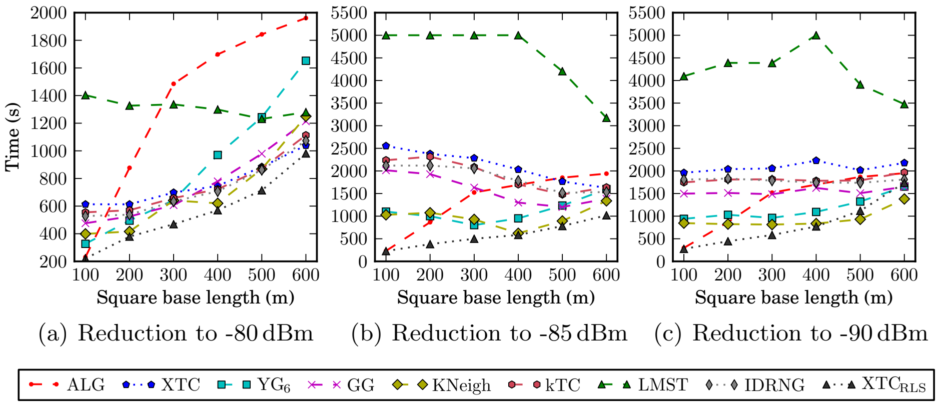
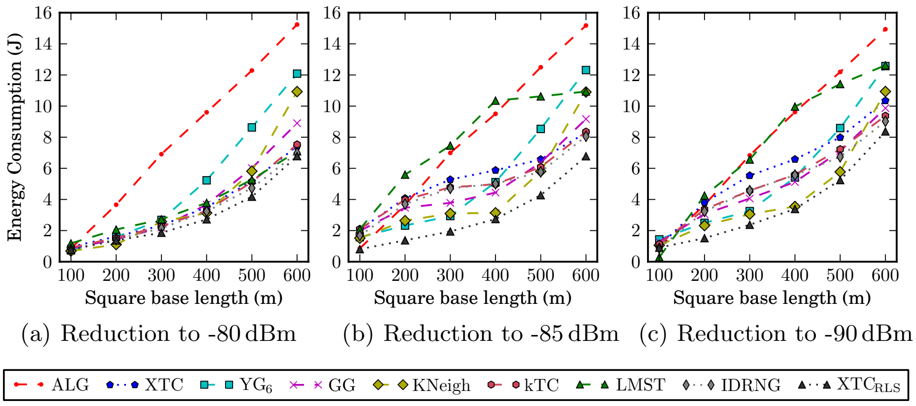


Fig. 3. Time needed to finish the transmissions for different reductions of the transmission power. Note the different scales of the y-axis.

topologies show small throughput improvements (cf. Fig. 2(a) and Fig. 3(a)). The largest improvements are achieved for the LMST topology in networks with high node density.

Energy Consumption. Especially when the power consumption of a wireless device increases with increasing transmission power, as it is assumed in this work, it seems natural that using lower transmission powers also decreases the energy consumption. It was shown, for example, in [13] that a reduction of the transmission powers yields a lower energy consumption for Mica2dot nodes. In our simulations, however, we found that this can not be stated in such a general way. Again we consider the three power control strategies that reduce the transmission powers until the received signal strength equals -80 dBm, -85 dBm, or -90 dBm. The average energy consumptions per node for the three considered setups are depicted in Figure 4. The energy consumption per node using topology control without transmission power reduction, which is shown in Figure 2(b), is very similar to the energy consumption after a reduction to received signal strengths of -80 dBm, depicted in Figure 4(a). Only for very dense instances some sparse topologies such as the LMST, XTC, kTC, IDRNG and the Gabriel graph achieve a small improvement in the energy demand. This is probably due to the significant reduction of the transmission power they achieve.

For the scenarios where the transmission powers are reduced until the received signal strengths equal -85 dBm or -90 dBm, however, the consumed energy increases significantly for most topologies. Again, this can be explained with an adaption of the data rate to the measured SINR. As the data rate is reduced, the wireless devices require more time to transmit each single packet, which leads to a higher energy consumption.



(a) Reduction to -80 dBm (b) Reduction to -85 dBm (c) Reduction to -90 dBm

Fig. 4. Energy consumption for transmissions using topology control with transmission power reduction. The transmission powers are reduced until the received signal strengths correspond to the stated values.

6.4 Performance Comparison

To conclude this section, we present a brief comparison of the different topologies based on the achieved throughput and the energy consumption. Since the relative order of the topologies is similar for topology control with and without additional power control, we will focus on the results without power control, which are depicted in Figure 2.

For small deployment areas the ranking according to the throughput is strongly related to the average number of hops for communication between distant nodes, where $\mathrm{XTC}_{\mathrm{RLS}}$ and ALG perform best, followed by the Yao graph, k-Neigh, the Gabriel graph, IDRNG, kTC, XTC, and finally LMST. We can see that all topologies except ALG and LMST achieve somewhat comparable throughput. For larger deployment areas the performance of ALG decreases rapidly while $\mathrm{XTC}_{\mathrm{RLS}}$ still achieves the highest throughput, followed by IDRNG, XTC, kTC, the Gabriel graph, k-Neigh and LMST, which are all relatively close together. The lowest throughput is achieved by the Yao graph and ALG.

Regarding the energy consumption, most topologies achieve a similar performance. The ALG topology has the highest energy demand for most instances since links with weak signal strength are not discarded. Those links usually achieve a poor performance considering both the throughput and the energy efficiency. The LMST is worse than the other topologies (except partially ALG) for relatively dense instances, while the Yao graph and k-Neigh consume more energy than the other topologies for sparse instances. The proposed extension of XTC, $\mathrm{XTC}_{\mathrm{RLS}}$, achieves the best results for almost all instances. However, it is closely followed by most other topologies: IDRNG, Gabriel graph, kTC, XTC, k-Neigh, Yao graph and LMST. Only for the deployment area of 200×200 meters, k-Neigh achieves an energy consumption lower than that of $\mathrm{XTC}_{\mathrm{RLS}}$.

7 Conclusion

In this paper, we studied throughput and energy consumption of a variety of topology control algorithms. Especially in networks with rather low node density, all algorithms were able to improve the network throughput and the energy demand considerably. Our broad comparison made it then possible to relate the achieved performance improvements to basic properties of the computed topologies. Contrary to what is usually assumed, producing sparse topologies turned out to be not beneficial. In topologies with low average node degree usually more hops were necessary to allow communication between distant nodes. As each additional hop also means that the packets have to be relayed an additional time, sparse networks resulted in lower throughput and higher energy consumption.

Concerning power control, we showed that—contrary to what one might expect—reducing transmission powers does not necessarily result in power savings. The reason is that standard communication protocols usually adjust the data rate according to the measured SINR. Thus, lower transmission powers can result in longer transmissions and consequently in higher energy consumption. This effect is currently ignored in most studies of scheduling and routing protocols, even in those that particularly focus on energy consumption.

Motivated by the observation that denser topologies often allow for better performance and lower energy consumption, we proposed $\mathrm{XTC}_{\mathrm{RLS}}$, an extension of XTC that achieves connectivity by computing the XTC topology and that afterwards simply adds all links which provide a sufficiently high received signal strength. In our simulations, the $\mathrm{XTC}_{\mathrm{RLS}}$ algorithm outperformed all other approaches regarding both data throughput and energy efficiency.

Acknowledgments. This work was supported by the German Research Foundation (DFG) within the Research Training Group GRK 1194 ”Self-organizing Sensor-Actuator Networks”.

References

1. Network Simulator: ns-3 (2011), `http://www.nsnam.org`
2. Buchin, K., Buchin, M.: Topology Control. In: Wagner, D., Wattenhofer, R. (eds.) Algorithms for Sensor and Ad Hoc Networks. LNCS, vol. 4621, pp. 81–98. Springer, Heidelberg (2007)
3. Li, N., Hou, J.C., Sha, L.: Design and analysis of an MST-based topology control algorithm. IEEE Transactions on Wireless Communications 4, 1195–1206 (2005)
4. Wattenhofer, R., Li, L., Bahl, P., Wang, Y.: Distributed Topology Control for Power Efficient Operation in Multihop Wireless Ad Hoc Networks. In: Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001), pp. 1388–1397 (2001)
5. Yao, A.C.C.: On Constructing Minimum Spanning Trees in k-Dimensional Spaces and Related Problems. SIAM Journal on Computing 11, 721–736 (1982)

6. Narayanaswamy, S., Kawadia, V., Sreenivas, R.S., Kumar, P.R.: Power Control in Ad-Hoc Networks: Theory, Architecture, Algorithm and Implementation of the COMPOW Protocol. In: Proceedings of European Wireless 2002. Next Generation Wireless Networks: Technologies, Protocols, Services and Applications, pp. 156–162 (2002)
7. Wattenhofer, R., Zollinger, A.: XTC: A Practical Topology Control Algorithm for Ad-Hoc Networks. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium - Workshop 12 (IPDPS 2004), p. 216a (2004)
8. Schweizer, I., Wagner, M., Bradler, D., Mühlhäuser, M., Strufe, T.: kTC - Robust and Adaptive Wireless Ad-hoc Topology Control. In: Proceedings of the IEEE International Conference on Computer Communication Networks, ICCCN 2012 (2012)
9. Chu, X., Sethu, H.: A New Power-Aware Distributed Topology Control Algorithm for Wireless Ad Hoc Networks. In: Proceedings of the IEEE Global Telecommunications Conference, GLOBECOM 2011 (2011)
10. Blough, D.M., Leoncini, M., Resta, G., Santi, P.: The K-Neigh Protocol for Symmetric Topology Control in Ad Hoc Networks. In: Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2003), pp. 141–152. ACM, New York (2003)
11. Yick, J., Mukherjee, B., Ghosal, D.: Wireless sensor network survey. Computer Networks 52, 2292–2330 (2008)
12. Li, N., Hou, J.C.: Localized Topology Control Algorithms for Heterogeneous Wireless Networks. IEEE/ACM Transactions on Networking 13, 1313–1324 (2005)
13. Jeong, J., Culler, D., Oh, J.H.: Empirical Analysis of Transmission Power Control Algorithms for Wireless Sensor Networks. In: Proceedings of the 4th International Conference on Networked Sensing Systems (INSS 2007), pp. 27–32 (2007)
14. Duran, A., Toril, M., Ruiz, F., Solera, M., Navarro, R.: Analysis of Topology Control Algorithms in Multi-hop Cellular Networks. In: Proceedings of the 5th International Conference on Broadband and Biomedical Communications (IB2Com 2010), pp. 1–6 (2010)
15. Xu, L., Bo, H., Haixia, L., Mingqiang, Y., Mei, S., Wei, G.: Research and Analysis of Topology Control in ns-2 for Ad-hoc Wireless Network. In: Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems (CISIS 2008), pp. 461–465 (2008)
16. Gao, Y., Hou, J.C., Nguyen, H.: Topology Control for Maintaining Network Connectivity and Maximizing Network Capacity under the Physical Model. In: Proceedings of the 27th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2008), pp. 1013–1021 (2008)
17. Niewiadomska-Szynkiewicz, E., Kwaniewski, P., Windyga, I.: Comparative Study of Wireless Sensor Networks Energy-Efficient Topologies and Power Save Protocols. Journal of Telecommunications and Information Technology 3, 68–75 (2009)
18. Blough, D.M., Harvesf, C., Resta, G., Riley, G., Santi, P.: A Simulation-Based Study on the Throughput Capacity of Topology Control in CSMA/CA Networks. In: Proceedings of the 4th IEEE International Conference on Pervasive Computing and Communications, pp. 400–404 (2006)
19. Wu, H., Nabar, S., Poovendran, R.: An Energy Framework for the Network Simulator 3 (ns-3). In: Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools 2011), pp. 222–230 (2011)
20. Roving Networks: RN-174 data sheet (2012), `http://www.rovingnetworks.com/resources/download/14/RN_174`

Cache-Oblivious Dictionaries and Multimaps with Negligible Failure Probability

Michael T. Goodrich[1], Daniel S. Hirschberg[1],
Michael Mitzenmacher[2], and Justin Thaler[2]

[1] Dept. of Computer Science, University of California, Irvine
[2] School of Engineering and Applied Sciences, Harvard University

Abstract. A *dictionary* (or *map*) is a key-value store that requires all keys be unique, and a *multimap* is a key-value store that allows for multiple values to be associated with the same key. We design hashing-based indexing schemes for dictionaries and multimaps that achieve worst-case optimal performance for lookups and updates, with minimal space overhead and *sub-polynomial* probability that the data structure will require a rehash operation. Our dictionary structure is designed for the Random Access Machine (RAM) model, while our multimap implementation is designed for the cache-oblivious external memory (I/O) model. The failure probabilities for our structures are sub-polynomial, which can be useful in cryptographic or data-intensive applications.

1 Introduction

A *dictionary* (or *map*) is a key-value store that requires all keys be unique, and a *multimap* [3] is a key-value store that allows for multiple values to be associated with the same key. Key-value associations are used in many applications, and hash-based dictionary schemes are well-studied in the literature (e.g., see [12]). Multimaps [3] are less studied, although a multimap can be viewed as a dynamic *inverted file* or *inverted index* (e.g., see Knuth [21]). Given a collection, Γ, of documents, an inverted file is an indexing strategy that allows one to list, for any word w, all the documents in Γ where w appears. Multimaps also provide a natural representation framework for adjacency lists of graphs, with nodes being keys and adjacent edges being values associated with a key. For other applications, please see Angelino *et al.* [3].

Such structures are ubiquitous in the "inner-loop" computations involved in various algorithmic applications. Thus, we are interested in implementations of these abstract data types (ADTs) that are based on hashing and use a near-optimal amount of storage – ideally $(1+\epsilon)n$ words of storage, where n is the number of items in the dictionary or multimap and $\epsilon > 0$ is some small constant. In addition, because such solutions are used in real-time applications, we are interested in implementations that are *de-amortized*, meaning that they have asymptotically optimal worst-case lookup and update complexities, but may have small probabilities of overflowing their memory spaces.

Crucially, we additionally focus on two further design goals. The first is that we aim for our data structures to succeed with *overwhelming probability*, i.e. probability $1 - 1/n^{\omega(1)}$, rather than merely with high probability, i.e. probability $1 - 1/\operatorname{poly}(n)$, achieved by most previous constructions. While our aim of achieving structures that provide worst-case constant time operations with overwhelming probability instead of with high probability may seem like a subtle improvement, there are many applications where it is essential. In particular, it is common in cryptographic applications to aim for negligible failure probabilities. For example, cuckoo structures with negligible failure probabilities have recently found applications in oblivious RAM simulations [17]. Moreover, a significant motivation for de-amortized cuckoo hashing is to prevent timing attacks and clocked adversaries from compromising a system [4]. An inverse polynomial failure probability may render a structure unsuitable for these applications.

In addition, guarantees that hold with overwhelming probability allow us to handle super-polynomially long sequences of updates, as long as the total number of items resident in the dictionary is bounded by n at all times. This is useful in long-running or data-intensive applications. It is also crucial for applications in which it is not possible to anticipate certain parameters, such as the length of the sequence of operations to be handled, at the time the data structure is deployed.

Our final design goal is relevant for our solutions that operate in the *external-memory* (I/O) model. Specifically, we would like our external-memory solutions to be *cache-oblivious* [15], meaning that they should achieve their performance bounds without being tuned for the parameters of the memory hierarchy, like the size, B, of disk blocks, or the size, M, of internal memory. The advantage of cache-oblivious solutions is that one such algorithm can comfortably scale across all levels of the memory hierarchy and can also be a better match for modern compilers that perform predictive memory fetches.

Previous Related Work. Since the introduction of the cache-oblivious framework by Frigo *et al.* [15], several cache-oblivious algorithms have subsequently been presented, including cache-oblivious B-trees [6], cache-oblivious binary search trees [8], and cache-oblivious sorting [9]. Pagh *et al.* [27] describe a scheme for cache-oblivious hashing, which is based on linear probing and achieves $O(1)$ expected-time performance for lookups and updates, but it does not achieve constant time bounds for any of these operations in the worst case.

As mentioned above, the multimap abstract data type is related to the inverted file and inverted index structures, which are well-known in text indexing applications (e.g., see Knuth [21]) and are also used in search engines (e.g., see Zobel and Moffat [32]). Cutting and Pedersen [13] describe an inverted file implementation that uses B-trees for the indexing structure and supports insertions, but doesn't support deletions efficiently. More recently, Luk and Lam [24] describe an internal-memory inverted file implementation based on hash tables with chaining, but their method also does not support fast item removals. Lester *et al.* [22,23] and Büttcher *et al.* [11] describe external-memory inverted file implementations that support item insertions only. Büttcher and Clarke [10] consider trade-offs for allowing for both item insertions and removals, and Guo *et al.* [18] give a solution for performing such operations by using a B-tree variant. Finally, Angelino *et al.* [3] describe an efficient external-memory data structure for the multimap abstract data type, but like the above-mentioned work on inverted files, their method is not cache-oblivious.

Much prior work on de-amortized dictionaries use variants of *cuckoo hash tables*, which were presented by Pagh and Rodler [26] and studied by a variety of other researchers. In their basic form, these structures use a freedom to place each key-value pair in one of two hash tables to achieve worst-case constant-time lookups and removals and amortized constant-time insertions with high probability. Kirsch, Mitzenmacher, and Wieder [20] introduced the notion of a stash for cuckoo hashing, which allows the failure probability to be reduced to $O(1/n^{\alpha})$, for any constant $\alpha > 0$, by using a constant-sized adjunct memory to store items that wouldn't otherwise be able to be placed. The failure probability of this solution is still inverse-polynomial, and insertions take $O(1)$ *amortized time* rather than worst-case time. Kirsch and Mitzenmacher [19] and Arbitman *et al.* [4] study a method for de-amortizing cuckoo hashing, which achieves constant-time lookups, insertions, and deletions with high probability, and uses space $(2+\epsilon)n$ for any constant $\epsilon > 0$ (as is standard in cuckoo hashing). In a subsequent paper, Arbitman *et al.* [5] study a hashing method that achieves worst-case constant-time lookups, insertions, and removals with high probability while maintaining loads very close to 1. They accomplish this by using a two-level hashing scheme, where the first level uses a simple bin-based hash table, and the second uses the de-amortized cuckoo hashing scheme of [4]. Our dictionary construction uses their first level, but replaces their second level with a different structure based on the Q-heaps of Fredman and Willard [14].

A lower bound of Andersson, Bro Miltersen, Riis, and Thorup is also relevant [1]. They prove that even for *static* dictionaries, query time $\Theta\left(\sqrt{\log n/\log\log n}\right)$ is necessary and sufficient on AC^0 RAMs, a restriction of the RAM model in which the only operations permitted in unit time are those computable by polynomial-sized constant-depth circuits. This applies even if $n^{\mathrm{polylog}(n)}$ space is permitted for the dictionary. We clarify that the lower bound only applies for substantially super-polynomial sized universes (though stronger lower bounds are known for RAMs whose instruction set is restricted further [30]). Our algorithms do not contradict the Andersson *et al.* result because we work in the standard RAM model rather than the AC^0 RAM model (we remark that, like much prior work on dictionary data structures, the only place we use non-AC^0 operations is in the evaluation of sufficiently random hash functions).

An earlier version of this paper [16] with similar goals presented a more intricate two-level dictionary data structure, where both levels were implemented as de-amortized cuckoo hash tables. The present paper substantially simplifies the earlier dictionary structure, while achieving a smaller failure probability.

Contributions. Our contributions are two-fold. Our first contribution is a dictionary structure achieving worst-case optimal lookup and update complexities with *sub-polynomial* failure probability, while incurring minimal space overhead. Specifically, our structure requires $(1+\epsilon)n$ space for an *arbitrary* constant $\epsilon > 0$. The lookup and update complexities of our structure are given in Table 1.

To the best of our knowledge, ours is the first structure suitable for the Random Access Machine (RAM) model that achieves all of these goals assuming random hash functions that can be evaluated in constant time. We also discuss several solutions that work with hash functions that are realizable (although impractical) in the standard RAM model, while achieving slightly sub-optimal lookup and update complexities. These solutions partially address an open question raised by Arbitman *et al.* [5].

Our second contribution is to develop a multimap implementation suitable for the *external-memory* (I/O) model. Prior work [3] achieved a solution in this model with worst-case optimal update and lookup complexity, but their solution was *cache-aware*, requiring knowledge of the size, B, of disk blocks.

Table 1. Performance bounds for our dictionary and multimap implementations, which all hold in the worst-case with overwhelming probability, assuming truly random hash functions. These bounds are asymptotically optimal. We use B to denote the block size, k to denote an arbitrary key, v to denote an arbitrary value, and n_k to denote the number of items with key equal to k.

Method	**Dictionary I/O Performance**	**Multimap I/O Performance**
add(k, v)	$O(1)$	$O(1)$
containsKey(k)	$O(1)$	$O(1)$
containsItem(k, v)	$O(1)$	$O(1)$
remove(k, v)	$O(1)$	$O(1)$
get(k)/getAll(k)	$O(1)$	$O(1 + n_k/B)$
removeAll(k)	–	$O(1)$

2 Dictionary Data Structure

Our dynamic dictionary data structure is designed for the standard RAM model. The instruction set available will be arithmetic, bitwise logical, and comparison operations on $b = \Omega(\log n)$ bit words.

Recall that a collection H of functions $h : U \to V$ is k-wise independent if for any distinct $x_1, \ldots, x_k \in U$ and for any $y_1, \ldots, y_k \in V$ it holds that $Pr[h(x_1) = y_1 \wedge \cdots \wedge h(x_k) = y_k] = 1/|V|^k$. Throughout this section, we assume the existence of an n^α-wise independent family of hash functions (for some constant $\alpha > 0$) mapping the universe U to the set $\{0, \ldots, n-1\}$, that can be evaluated in constant time using $o(n)$ space. We present results on the validity of this assumption in Section 3.

We mention that, with the exception of hash function evaluation, all of the pieces in our construction can be made to run in the AC^0 RAM model [2].

Our dictionary data structure combines two pieces. First, we modify a dictionary construction due to Willard to achieve a data structure with optimal worst-case update times and failure probability just $1/n^{\mathrm{polylog}(n)}$. However, the resulting data structure uses $O(n)$ space, rather than $(1+\epsilon)n$ space. As our second step, we combine the result of Step 1 with the first level of the two-level hashing scheme of Arbitman *et al.* [5], to bring the space usage down to $(1+\epsilon)n$ words for any constant $\epsilon > 0$.

2.1 The First Piece

Willard [31] describes a simple dictionary data structure using $O(n\,\mathrm{polylog}(n))$ words of memory that supports worst-case constant time lookups and updates with failure probability $1/n^{\mathrm{polylog}(n)}$. The primary contribution of this subsection is to give a variant of his structure that achieves the same guarantees using $O(n)$ words of space.

Willard's construction is based on a variant of Fredman and Willard's *Q-heap*. Using $O(n)$ space and preprocessing time, the Q-heap supports constant-time insertions, deletions, member, and predecessor queries into sets of size $O(\log^{1/5} n))$. By using a Q*-heap [31], which is essentially a B-tree whose internal nodes are implemented as Q-heaps, one can in fact achieve worst-case constant time insertions, deletions, and lookups for sets of size $O(\log^c n)$ for an arbitrary constant $c > 0$ [31, Lemma 2].

With the Q*-heap functionality in hand, Willard proposes the following simple dictionary structure with failure probability $1/n^{\log^k(n)}$ for any constant $k > 0$. Let h be a hash function chosen at random from hash family H. Consider a hash table with n buckets, each implemented as a Q*-heap with capacity $\log^{k+2}(n)$. As long as no bucket overflows its capacity, this hash table ensures that a bucket can be searched in $O(1)$ time, and that inserts and deletions can be processed in $O(1)$ time. Moreover, as long as the hash family H is $\log^{k+2}(n)$-wise independent, the probability that any particular bucket overflows is at most $1/n^{\log^{k+1}(n)}$. A union bound then implies that *no* bucket overflows with probability at least $1 - 1/n^{\log^k(n)}$.

As mentioned above, a major downside of Willard's construction is that it uses $O(n \operatorname{polylog}(n))$ space. We now show a modification that brings the space usage down to $O(n)$ machine words.

Instead of using an array of n buckets, use an array of $n/\log^k(n)$ buckets for some constant $k > 1$. Implement each bucket with a Q*-heap of capacity $6\log^k(n)$. Assuming truly random hash functions, a suitable Chernoff bound [25, Theorem 4.4, Part 3] implies that the probability any individual bucket overflows is at most $1/2^{6\log^k(n)}$. By a union bound, *no* bucket overflows with probability at least $1 - 1/2^{5\log^k(n)}$.

Notice each "bad event" (namely a bucket overflowing) in the above analysis involves the hash values of a set S of at most $6\log^k(n)$ items, and as long as H is a $6\log^k(n)$-wise independent hash family, the values of h on S are fully independent. Thus, the same analysis applies as long as H is a $6\log^k(n)$-wise independent hash family. Notice our modified construction requires $O(n)$ words of memory.

Lemma 1. *Let $k > 1$ be any constant. Assume there exists a $\log^k(n)$-wise independent hash family H such that each $h \in H$ can be evaluated in constant time using $o(n)$ words of memory. Then there exists a dynamic dictionary scheme $\mathcal{A}$ using $O(n)$ words of memory that supports insertions, deletions, and membership queries in $O(1)$ worst-case time, with failure probability $1/n^{5\log^{k-1}(n)}$.*

2.2 The Second Piece

Arbitman *et al.* [5] present a novel two-level hashing scheme, which they call Backyard Cuckoo Hashing. The first level of their scheme uses $m = (1+\epsilon/2)n/d$ "bins" of size d, where d is a suitably chosen constant that depends on ϵ. In the simplest version of their scheme, lookups, insertions, and deletions into each bin can trivially be implemented in constant time (which depends on ϵ), because each bin has constant size. However, a constant fraction of items inserted into the first level may "overflow", and must be handled separately by the second level of their scheme, which they instantiate as a de-amortized cuckoo hash table.

We briefly remark that Arbitman *et al.* [5] also present a more involved scheme based on de-amortized perfect hashing that works for slightly subconstant values of ϵ; this variant can also be adapted to our setting, but we omit these details for brevity.

Our intention is to use the first level of their scheme to absorb all but a small fraction of the items in our table. We use the dictionary data structure described in Section 2.1 to handle the overflowing items. Details follow.

Inspection of the proof of [5, Lemma 3.2] shows that their scheme achieves the following guarantee.

Lemma 2. *Let α be any constant $0 < \alpha < 1$. Assume there exists an n^α-wise independent hash family H such that each $h \in H$ can be evaluated in constant time using $o(n)$ words of memory. Then for any constant $\epsilon > 0$, there exists a data structure using space $(1+\epsilon/2)n$ satisfying the following guarantees. For some $\beta > 0$, with probability $1 - 2^{-n^\beta}$, all but $\epsilon n/16$ items are successfully inserted into the data structure. Moreover, insertions (both successful and unsuccessful) and deletions take worst-case $O(1)$ time, and membership queries succeed in $O(1)$ time for any item successfully placed in the data structure.*

Thus, with probability $1 - 2^{-n^\beta}$, during any sequence of n insertions, at most $t := \epsilon n/16$ items overflow from the primary structure, and we can place these items in the data structure $\mathcal{A}$ described in Section 2.1.

The remaining issue is that, in order to guarantee that the second level never contains more than $\epsilon n/16$ items, we must move an item from the second level to the first level whenever an item is deleted from the first level. This issue is also encountered by Arbitman *et al.*, who suggest multiple ways to address it. One solution is to associate with each first-level bin a doubly-linked list pointing to all overflowing items from the bin (the doubly-linked lists in total require at most $c'\epsilon n/16$ space for some universal constant c'). This way, whenever an item is deleted from a first-level bin, we can replace it in constant time with one of the items that previously overflowed from the bin.

We thereby ensure that for any sequence of $\mathrm{poly}(n)$ insertions and deletions such that at most n items are actually stored in the data structure at any point in time, with probability $1 - 2^{-n^\beta}$ the second level never contains more than $\epsilon n/16$ items. Conditioned on this event, Lemma 1 implies that for any constant $k > 1$, the scheme $\mathcal{A}$ successfully supports worst-case constant time insertions, deletions, and lookups with probability $1 - 1/n^{5\log^{k-1}(n)}$ using space $c\epsilon n/16$ for some universal constant $c > 0$. Thus, our combined data structure supports worst-case constant time operations with failure probability $1/n^{5\log^{k-1}(n)} + 2^{-n^\beta} < 1/n^{\log^{k-1}(n)}$.

Setting $\epsilon = 16\epsilon'/(8 + c + c')$, the two levels of our data structure use $(1+\epsilon/2)n + (c+c')\epsilon n/16 = (1+\epsilon')n$ words of memory in total. Combined with the above analysis, we obtain the following theorem.

Theorem 1. *Let α be any constant $0 < \alpha < 1$. Assume there exists an n^α-wise independent hash family H such that each $h \in H$ can be evaluated in constant time using space $o(n)$. For any constants $\epsilon' > 0$ and $k > 1$, there exists a data structure using $(1+\epsilon')n$ words of memory and supports insertions, deletions, and membership queries in worst-case $O(1)$ time with probability $1 - 1/n^{\log^{k-1}(n)}$.*

3 Hash Families

The results of the previous section require an n^α-wise independent hash family H such that each $h \in H$ can be evaluated in constant time using space $o(n)$. We feel this assumption is supported in practice, for instance, by the fact that one of the most widely-used hash functions, SHA-1, can be implemented in $O(1)$ time even in the AC^0 RAM model. The aim of this section is to provide a careful treatment of the theoretical foundations of this assumption.

Siegel's Hash Functions. Although there are no known constructions of n^α-wise independent hash functions achieving the above desiderata, Siegel achieved a close (albeit impractical) approximation in an influential paper [29]. Assume for the moment that the universe size $|U|$ is at most n^r for some constant $r > 0$, and that the desired range of the hash function is V, where $|V|$ is a power of two. Siegel's construction makes use of a bipartite graph G with constant left-degree d. The left vertex set of G corresponds to the universe U; and the right vertex set is $\{0, \dots, n^\beta\}$ for some $0 < \beta < 1$. Each right vertex y is assigned a random value $M[y] \in V$ at initialization, and the hash value of element $x \in U$ is defined as $h(x) = \bigoplus_{(x,y) \in E(G)} M[y]$, where $\bigoplus$ denotes the bitwise XOR operation (if $|V|$ is not a power of two, we can replace $\bigoplus$ with any commutative group operation). Using a peeling argument, Siegel proves that as long as G has suitable *vertex expansion*, then the resulting hash family is n^α-wise independent for some $0 < \alpha < 1$.

Naively, storing the adjacency information of G would require space $\Omega(|U|) = \Omega(n^r)$, which is unacceptably large. To avoid this, Siegel uses a tensoring operation to turn a small expander graph (which can be stored explicitly) into larger expander. This approach increases the left degree and hence the evaluation time, but it remains constant as long as the universe size $|U|$ is polynomial in n.

From our usage standpoint, there are two potential sources of "failure" in Siegel's construction. The first is that there are currently no known explicit constructions of unbalanced vertex expanders that are sufficient to guarantee n^α-wise independence of Siegel's hash family. As a result, Siegel's hash family must either be *non-uniform*, with a suitable expander hardwired into the hashing algorithm, or the graph G must be generated at random. Siegel shows that for any constant $c > 0$, a suitable random graph will satisfy the requisite expansion properties with probability $1 - 1/n^c$.

The second source of failure comes into play if the universe size is super-polynomial in n. In this case, the universe should first be "mapped down" to a set U' of size n^r by a hash function h' from an almost-universal hash family H', before applying Siegel's construction to U'. For any set $T \subseteq U$ of size n^r, the resulting hash function will be fully independent on T, conditioned on the event that no elements in T collide under h', i.e., conditioned on the event that for all distinct elements $w, x \in T$, $h'(x) \neq h'(w)$. Unfortunately, any two elements in T will collide under h' with probability $1/n^r$.

In our applications, we cannot tolerate inverse-polynomial failure probabilities, and so neither source of failure is acceptable. Addressing these sources of failure was posed as an open question by Arbitman *et al.* [5]. In what follows, we give partial remedies to these sources of failure.

Obtaining Expanders. For polynomial-sized universes, Siegel's construction does yield *non-uniform* families of n^α-wise independent hash functions with $O(1)$ evaluation time and $o(n)$ space usage, by hardwiring in a suitable expander. However, if one requires a uniform algorithm, Siegel chooses the graph at random, generating a hash family H

that is n^α-wise independent only with probability $1 - 1/n^c$ (this is the probability that a randomly generated graph will satisfy the requisite expansion properties).

The most natural approach to eliminate this failure probability is to obtain *explicit constructions* of suitable expanders. Sadly, we do not resolve this question here: it remains an intriguing open question. Instead, we specify partial solutions to the problem.

Our first solution relies on the following observation: the probability a random graph G fails to satisfy the requisite expansion condition is dominated by the probability that *small sets* of vertices fail to satisfy the condition. Thus, one can obtain sub-polynomial failure probability by randomly generating a graph and exhaustively checking the vertex expansion of all sufficiently small sets. If a non-expanding set is found, the graph is rejected and a new graph is generated. This requires quasi-polynomial pre-processing time, but this may be acceptable in certain applications as the expensive phase need not happen online.

Theorem 2. *Assume the universe U has size n^r for some constant $r > 0$. Then for some α, for every pair of constants $k, r' > 0$, there is a set V of size $n^{r'}$ and a uniform algorithm $\mathcal{A}$ outputting a collection H of functions $h : U \to V$ achieving the following guarantees.*

1. *With probability $1 - 1/n^{\log^k(n)}$ over the internal coin tosses of $\mathcal{A}$, H is an n^α-wise independent family of hash functions.*
2. *$\mathcal{A}$ runs in $n^{\Theta\left(\log^{k+1}(n)\right)}$ time.*
3. *Any function $h \in H$ can be represented with $o(n)$ bits and evaluated in $O(1)$ time.*

We omit the proof because of space constraints; it can be found in the full version of the paper on the arXiv.

For polynomial-sized universes, we can instantiate the hash functions required in Section 2 using the algorithm of Theorem 2. This introduces an additional additive $1/n^{\log^k(n)}$ failure probability into our dictionary structure, which does not affect our results. We remark that the algorithm of Theorem 2 is implementable in the AC^0 RAM model, not just in the standard RAM model.

For polynomial-sized universes, a second partial solution is to avoid expensive preprocessing phase at the expense of slightly super-constant evaluation runtime. A first approach is to achieve this by simply increasing the degree of the randomly generated graph to $d(n)$, where d is some very slow-growing function of n. This reduces the probability that the graph fails to satisfy the requisite expansion properties to $\frac{1}{n^{\Omega(d(n))}}$. A problem with this approach is that the tensoring operation used by Siegel to blow up a small expander into a larger one will cause the larger graph to have degree $\omega(d(n))$.

A superior approach is described next.

Arbitrary Universe Sizes. We give a partial solution for achieving sub-polynomial failure probabilities with arbitrary universe sizes.

Theorem 3. *There exists some α, such that for every pair of constants $r, r' > 0$, and any function $k(n) = \omega(1)$, there is a set V of size $n^{r'}$ and a uniform algorithm $\mathcal{A}$ outputting a collection H of functions $h : U \to V$ achieving the following guarantees.*

1. *$\mathcal{A}$ runs in polynomial time.*

2. *Any function $h \in H$ can be represented with $o(n)$ bits and evaluated in time $\Theta(k(n))$.*
3. *For any set $T \subseteq U$ of size n^r, it holds that with probability $1 - 1/n^{k(n)}$ over the internal coin tosses of $\mathcal{A}$, the distribution $(h(x_1), \ldots, h(x_{n^\alpha}))$ is uniform over V^{n^α} for any distinct $x_1, \ldots, x_{n^\alpha} \in T$.*

Proof. Recall that in this setting Siegel gives a uniform algorithm $\mathcal{A}'$ generating a family of hash functions H such that, with high probability over the internal coin tosses of $\mathcal{A}'$, for any set $T \subseteq U$ of size n^r, H is fully independent on all subsets of T of size at most n^α. The algorithm $\mathcal{A}'$ works by first mapping the universe U down into a smaller universe U' of size n^r by a hash function h' from an universal hash family H', and then applying Siegel's expander-based hash function to U' using a randomly-generated expander G. As long as h' is one-to-one on T and G is an $(n, \epsilon, d, n^\alpha)$-weak expander (see the proof of Theorem 2), then the hash family H is fully independent on all subsets S of T of size at most n^α. However, both the universe-reduction step and the expander generation step introduce inverse-polynomial probabilities that the hash family H will *not* be fully independent on all such subsets S of T.

The idea to reduce the failure probability is to evaluate $k(n)$ independent instances of Siegel's hash function and XOR the results together (if $|V|$ is not a power of two, we replace XOR with any commutative group operation). For any set S at size at most n^α, as long as at least one of the k individual hash functions is fully independent on S, the result will be fully independent on S.

Formally, let $H_1, \ldots, H_{k(n)}$ be hash families generated by $k(n)$ independent runs of Siegel's algorithm $\mathcal{A}'$. We define our final hash family H to be $\{h : h(x) = \bigoplus_{i=1}^{k(n)} h_i(x), (h_1, \ldots, h_{k(n)}) \in H_1 \times \cdots \times H_{k(n)}\}$. Thus, a random element of h corresponds to randomly picking a hash function h_i from each hash family H_i, and XORing the results together.

Let $T \subseteq U$ be any set of size n^r. An easy calculation shows that with probability $1 - n^{\Omega(k(n))}$, the universe-reduction step is one-to-one on T for at least $k(n)/2$ runs of $\mathcal{A}'$. For each such run of $\mathcal{A}'$, with probability $1 - 1/n^c$, the expander-generation step successfully produces a graph with the requisite expansion properties for H_i to be n^α-wise independent on the "mapped down" universe U'.

Thus, with probability at least $1 - n^{\Omega(k(n))}$, it holds that for at least one run i of $\mathcal{A}'$, the hash family H_i is fully independent on all subsets of T of size at most n^α. That is, for any set $T \subseteq U$ of size n^r, with probability $1 - 1/n^{\Omega(k(n))}$, the distribution $(h_i(x_1), \ldots, h_i(x_{n^\alpha}))$ is uniform over V^{n^α} for any distinct $x_1, \ldots, x_{n^\alpha} \in T$. This is easily seen to imply that our final hash family H satisfies the same property for any distinct $x_1, \ldots, x_{n^\alpha} \in T$. □

Theorem 3 can be used to obtain sub-polynomial failure probabilities for super-polynomially long sequences of data structure updates, as long as at most n items reside in the data structure at a time. Theorem 3 is applied on a step-by-step basis, where the set T at each step corresponds to the n items extant in the structure. For example, setting $k(n) = \Theta(\log \log n)$, we conclude that the data structure operation at any particular step can be performed in $O(\log \log n)$ time with probability $1/n^{\log \log n}$. If there are $n^{(1/2) \log \log n}$ steps, then all steps succeed in $O(1)$ worst-case time with probability $1 - 1/n^{\Omega(\log \log n)}$.

4 Cache-Oblivious Multimaps

In this section, we describe our cache-oblivious implementation of the multimap ADT. To illustrate the issues that arise in the construction, we first give a simple implementation for a RAM, and then give an improved (cache-oblivious) construction for the external memory model. Specifically, we describe an amortized cache-oblivious solution and then we describe how to de-amortize this solution.

In the implementation for the RAM model, we maintain two dictionary data structures, as described in Section 2. The first table enables fast containsItem(k, v) operations; this table stores all the (k, v) pairs using each entire key-value pair as the key, and the value associated with (k, v) is a pointer to v's entry in a linked list $L(k)$ containing all values associated with k in the multimap. The second table ensures fast containsKey(k), getAll(k), and removeAll(k) operations: this table stores all the unique keys k, as well as a pointer to the head of $L(k)$.

Operations in the RAM Implementation

1. containsKey(k): We perform a lookup for k in Table 2.
2. containsItem(k, v): We perform a lookup for (k, v) in Table 1.
3. add(k, v): We add (k, v) to Table 1 using the insertion procedure of Section 2. We perform a lookup for k in Table 2, and if k is not found we add k to Table 2. We then insert v as the head of the linked list corresponding to Table 2.
4. remove(k, v): We remove (k, v) from Table 1, and remove v from the linked list $L(k)$; if v was the head of $L(k)$, we also perform a lookup for k in Table 2 and update the pointer for k to point to the new head of $L(k)$ (if $L(k)$ is now empty, we remove k from Table 2.)
5. getAll(k): We perform a lookup for k in Table 2 and return the pointer to the head of $L(k)$.
6. removeAll(k): We lookup k in Table 2, and follow the pointer to $L(k)$. We walk through the linked list $L(k)$, and for each entry (k, v) of L_k, we remove (k, v) from Table 1. We also remove k from Table 2.

With the exception of the removeAll(k) operation, all operations above are performed in $O(1)$ time in the worst case with overwhelming probability by the results of Section 2. The removeAll(k) operation takes $O(1)$ *amortized* time with overwhelming probability, because each time we remove a pair (k, v) from Table 1, we can charge the operation to the corresponding insertion of the pair (k, v). We will explain how to de-amortize the removeAll(k) operation in Section 4.2

Two major issues arise in the above construction. First, the space-usage remains $O(n)$ only if we assume the existence of a garbage-collector for leaked memory, as well as a memory allocation mechanism, both of which must run in $O(1)$ time in the worst case. Without the memory allocation mechanism, inserting v into $L(k)$ cannot be done in $O(1)$ time, and without the garbage collector for leaked memory, space cannot be reused after remove and removeAll operations. Second, in order to extract the actual values from a getAll(k) operation, one must actually traverse the list $L(k)$. Since $L(k)$ may be spread all over memory, this suffers from poor locality.

We now present our cache-oblivious multimap implementation. Our implementation avoids the need for garbage collection, and circumvents the poor locality of the above getAll operation. We do require a cache-oblivious mechanism to allocate and deallocate power-of-two sized memory blocks with constant-factor space and I/O overhead; this assumption is justified by the results of Brodal *et al.* [7], who design a system for allocating and deallocating memory using constant time in the worst case, and sub-linear (indeed sub-polynomial, i.e. $o(n^{\delta})$ for any $\delta > 0$) auxiliary storage.

Amortized Cache-Oblivious Multimaps. As in the RAM implementation, we keep two dictionary data structures. In Table 1, we store all the (k, v) pairs using each entire key-value pair as the key. With each such pair, we store a count, which identifies an ordinal number for this value v associated with this key, k, starting from 0. For example, if the keys were (4, Alice), (4, Bob), and (4, Eve), then (4, Alice) might be pair 0, (4, Bob) pair 1, and (4, Eve) pair 2, all for the key, 4.

In Table 2, we store all the unique keys. For each key, k, we store a pointer to the array, A_k, that stores all the key-value pairs having key k, stored in order by their ordinal values from Table 1. With the record for a key k, we also store n_k, the number of pairs having the key k, i.e., the number of key-value pairs in A_k. We assume that each A_k is maintained as an array that supports amortized $O(1)$-time element access and addition, while maintaining its size to be $O(n_k)$.

Operations

1. containsKey(k): We perform a lookup for k in Table 2.
2. containsItem(k, v): We perform a lookup for (k, v) in Table 1.
3. add(k, v): After ensuring that (k, v) is not already in the multimap by looking it up in Table 1, we look up k in Table 2, and add (k, v) at index n_k of the array A_k, if k is present in this table. If there is no key k in Table 2, then we allocate an array, A_k, of initial constant size. Then we add (k, v) to $A_k[0]$ and add key k to Table 2. In either case, we then add (k, v) to Table 1, giving it ordinal n_k, and increment the value of n_k associated with k in Table 2. This operation may additionally require the growth of A_k by a factor of two, which would then necessitate copying all elements to the new array location and updating the pointer for k in Table 2.
4. remove(k, v): We look up (k, v) in Table 1 and get its ordinal count, i. Then we remove (k, v) from Table 1, and we look up k in Table 2, to learn the value of n_k and get a pointer to A_k. If $n_k > 1$, we swap $(k', v') = A_k[n_k - 1]$ and $(k, v) = A_k[i]$, and then remove the last element of A_k. We update the ordinal value of (k', v') in Table 1 to now be i. We then decrement the value of n_k associated with k in Table 2. If this results in $n_k = 0$, we remove k from Table 2. This operation may additionally require the shrinkage of the array A_k by a factor of 2, so as to maintain the $O(n)$ space bound.
5. getAll(k): We look up k in Table 2, and then list the contents of the n_k elements stored at the array A_k indexed from this record.
6. removeAll(k): For all entries (k, v) of A_k, we remove (k, v) from Table 1. We also remove k from Table 2 and deallocate the space used for A_k.

In terms of I/O performance, containsKey(k) and containsItem(k, v) clearly require $O(1)$ I/Os in the worst case. getAll(k) operations use $O(1 + n_k/B)$ I/Os in the worst case, because scanning an array of size n_k uses $O(\lceil n_k/B \rceil)$ I/Os, even though we don't know the value of B. removeAll(k) utilizes $O(n_k)$ I/Os in the worst-case with overwhelming probability, but these can be charged to the insertions of the n_k values associated with k, for $O(1)$ amortized I/O cost. add(k, v) and remove(k, v) operations also require $O(1)$ amortized I/Os with overwhelming probability; the bound is amortized because there is a chance this operation will require a growth or shrinkage of the array A_k, which may require moving all (k, v) values associated with k and updating the corresponding pointers in Table 1.

In the next sections, we explain how to deamortize add(k, v), remove(k, v), and removeAll(k) operations.

4.1 De-amortizing Add(k, v) and Remove(k, v) Operations

To de-amortize the array operations, we use a rebuilding technique, which is standard in de-amortization methods (e.g., see [28]).

We consider the operations needed for insertions to an array; the methods for deletions are similar. The main idea is that we allocate arrays whose sizes are powers of 2. Whenever an array, A_k, becomes half full, we allocate an array, A'_k, of double the size and start copying elements A_k in A'_k. In particular, we maintain a crossover index, i_{A_k}, which indicates the place in A_k up to which we have copied its contents into A'_k. Each time we wish to access A_k during this build phase, we copy two elements of A_k into A'_k, picking up at position i_{A_k}, and updating the two corresponding pointers in Table 1. Then we perform the access of A_k, as would otherwise, except that if we wish access an index $i < i_{A_k}$, then we actually perform this access in A'_k. When we are done building A'_k, we deallocate the memory used for array A_k. Since we copy two elements of A_k for every access, we are certain to complete the building of A'_k prior to our needing to allocate a new, even larger array, even if all these accesses are insertions. Thus, each access of our array will now complete in worst-case $O(1)$ time with overwhelming probability. It immediately follows that add(k, v) and remove(k, v) operations run in $O(1)$ worst-case time.

4.2 De-amortizing removeAll(k) Operations

We describe two solutions for de-amortizing the removeAll(k) operation. The first is conceptually simpler but induces a constant-factor increase in memory usage; the second avoids using more memory than the de-amortized scheme above.

At a high level, in order to ensure removeAll(k, v) runs in worst-case $O(1)$ time, we simply remove k from Table 2 and deallocate the space used for the array A_k. We do *not* update the corresponding pointers of (k, v) pairs in Table 1 however; this leaves "spurious" pointers in Table 1, which we define to be (k, v) pairs satisfying the property that removeAll(k) has been called after the most recent insertion of (k, v). We need to explain how to modify all the other operations to deal with the presence of these spurious pointers.

First Solution. Our first solution is to maintain a global clock t, which is initialized to zero and is incremented after every operation. Assuming there are poly(n) total operations, the global time t can always be stored using $O(1)$ words of memory. Whenever we insert a key k into Table 2, or a key-value pair (k, v) into Table 1, we store with it the value t at the time of insertion. These timestamps increase the space usage of Tables 1 and 2 by a constant factor.

Whenever an operation invokes a lookup of a key-value (k, v) pair in Table 1 and finds that it is present, we have to check whether (k, v) is spurious. We do this by looking up key k in Table 2. If k is not found, then we know (k, v) is spurious, and we remove (k, v) from Table 1 and proceed as if (k, v) was not found in Table 1. If k is found in Table 2, we compare the timestamp t associated with k in Table 2 to the timestamp t' associated with (k, v) in Table 1. (k, v) is spurious if and only if $t' < t$; in the former case we remove (k, v) from Table 1 and proceed as if (k, v) was not found in Table 1; in the latter case we proceed as if (k, v) was found in Table 1.

The final issue we must deal with is ensuring that the presence of spurious key-value pairs does not cause the dictionary structure used to implement Table 1 to fail or to overflow its $(1 + \epsilon)n$ space bound. Recall that our dictionary structure consists of two levels, where the first level is implemented as an array of constant-sized "bins", and the second level is implemented as an array of Q^*-heaps, where each Q^*-heap can store $\log^{k+2}(n)$ items. Whenever we go to insert a (k, v) pair into this data structure, we first check whether any items in its first-level bin are spurious, and delete any spurious items from the first-level bin – this can be done in $O(1)$ time because the first-level bin has constant size. If there is room in the first-level bin after deleting spurious items, we insert the (k, v) pair into the bin and return. This ensures that spurious items residing in the first layer of our dictionary structure never affect the capacity of the structure. If we fail to insert the item (k, v) into the first layer of our structure, we attempt to place it into the second level of the data structure.

Dealing with spurious items in the second layer of our structure is more complicated: because a Q^*-heap may contain $\log^{k+2}(n)$ many items, we do not have time to iterate through all the items in the Q^*-heap to which (k, v) is assigned and check if any of the items are spurious. Instead, we increase the capacity of each Q^*-heap from $\log^{k+2}(n)$ items to $2\log^{k+2}(n)$ items. We also maintain with each Q^*-heap Q a doubly-linked list $L(Q)$ containing all items in Q; in addition we maintain for each item in Q a pointer to its entry in $L(Q)$. These pointers into $L(Q)$ are used so that, when an item is deleted from Q, we can also remove its entry from $L(Q)$ in constant time. Doubling the capacity of all the Q^*-heaps, as well as maintaining the lists $L(Q)$ and the pointers into $L(Q)$ causes the space usage of the second layer of our dictionary structure to increase by a constant factor, which can be absorbed into the parameter ϵ.

Recall that in the absence of spurious items, with all but sub-polynomial probability, no Q^*-heap should ever contain more than $\log^{k+2}(n)$ items. So when a Q^*-heap Q surpasses $\log^{k+2}(n)$ items, we know (with all but sub-polynomial probability) that this is due to the presence of spurious items. At this point, every time an item is inserted into Q, we take two items from the front of the doubly-linked list $L(Q)$ and check if they are spurious. Each time we find a spurious item, we delete it from Q and from the list $L(Q)$; otherwise we move the item to the end of the list $L(Q)$. This ensures that there are never more than $\log^{k+2}(n)$ spurious items in any Q^*-heap at any one time, so with all but sub-polynomial probability, no Q^*-heap will overflow its $2\log^{k+2}(n)$ capacity.

Second Solution. Our second solution differs from our first only in the manner in which we check whether a (k, v) pair is spurious. Specifically, we can avoid the use of timestamps. As before, whenever an operation invokes a lookup of a key-value (k, v) pair in Table 1 and finds that it is present, we have to check whether (k, v) is spurious. To accomplish this, we first lookup key k in Table 2. If k is not found, then we know (k, v) is spurious, and we remove (k, v) from Table 1 and proceed as if (k, v) was not found in Table 1. If k is found in Table 1, then we need to determine whether or not (k, v) is actually a member of the array A_k.

To determine this, let i be the count associated with pair (k, v) in Table 1. Recall i is supposed to represent (k, v)'s position in the array A_k if (k, v) is not spurious. We check if $i < n_k$ (recall n_k is stored with k in Table 2 and gives the number of pairs having the key k); if not we know (k, v) is spurious. If $i < n_k$, we check whether $A_k[i] = v$. This equality holds if and only if (k, v) is not spurious. Finally, it is straightforward to modify this solution to work in the case where we are in the process of moving items from an old array A_k to a new array A'_k as in the description of the de-amortized add(k, v) and remove(k, v) operations.

All time bounds in Table 1 follow.

5 Conclusion

In this paper, we have studied dictionary and multimap algorithms that support worst-case constant-time operations with sub-polynomial failure probability. Such structures should prove useful in cryptographic applications, as well as in long-running applications or those in which the duration of deployment is not known in advance. Our multimap solution is suitable for the cache-oblivious I/O model, and is to the best of our knowledge the first dynamic multimap achieving asymptotically optimal performance using linear space in this model.

Several interesting questions remain for future work. Are there (mildly) explicit constructions of unbalanced bipartite expanders sufficient to implement Siegel's hash family? Combined with our results, for polynomial sized universes this would yield an algorithm in the AC^0 RAM model for maintaining a dynamic dictionary with sub-polynomial failure probability, $(1 + \epsilon)n$ space, polynomial preprocessing time, and worst-case constant time operations. More ambitiously, we ask whether dictionaries supporting worst-case constant time operations with sub-polynomial failure probabilities can be achieved in the standard RAM model with quasipolynomial sized universes?

Acknowledgments. We are grateful to Jon Ullman and Mikkel Thorup for enlightening discussions. This research was supported in part by the U.S. National Science Foundation, under grants 0713046, 0830403, and 0847968, and by an Office of Naval Research: Multidisciplinary University Research Initiative (MURI) Award, number N00014-08-1-1015. Justin Thaler is supported in part by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program, and an NSF Graduate Research Fellowship. Michael Mitzenmacher was supported in part by the U.S. National Science Foundation, under grants 0964473, 0915922, 0721491, and 1011840.

References

1. Andersson, A., Miltersen, P.B., Riis, S., Thorup, M.: Static Dictionaries on AC^0 RAMs: Query Time $\Theta(\sqrt{\log n / \log \log n})$ is Necessary and Sufficient. In: Proc. of FOCS, pp. 441–450 (1996)
2. Andersson, A., Miltersen, P.B., Thorup, M.: Fusion trees can be implemented with AC^0 instructions only. Theoretical Computer Science 215(1-2), 337–344 (1999)
3. Angelino, E., Goodrich, M.T., Mitzenmacher, M., Thaler, J.: External-Memory Multimaps. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 384–394. Springer, Heidelberg (2011)
4. Arbitman, Y., Naor, M., Segev, G.: De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 107–118. Springer, Heidelberg (2009)
5. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In: Proc. of FOCS, pp. 787–796 (2010)
6. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious b-trees. In: Proc. of FOCS, pp. 399–409 (2000)
7. Brodal, G.S., Demaine, E.D., Munro, I.: Fast allocation and deallocation with an improved buddy system. Acta Inf. 41, 273–291 (2005)
8. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache oblivious search trees via binary trees of small height. In: Proc. of SODA, pp. 39–48 (2002)
9. Brodal, G.S., Fagerberg, R., Vinther, K.: Engineering a cache-oblivious sorting algorithm. J. Exp. Algorithmics 12, 2.2:1–2.2:23 (2008)
10. Büttcher, S., Clarke, C.L.A.: Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In: Proc. of CIKM, pp. 317–318 (2005)
11. Büttcher, S., Clarke, C.L.A., Lushman, B.: Hybrid index maintenance for growing text collections. In: Proc. of SIGIR, pp. 356–363 (2006)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
13. Cutting, D., Pedersen, J.: Optimization for dynamic inverted index maintenance. In: Proc. of SIGIR, pp. 405–411 (1990)
14. Fredman, M.L., Willard, D.E.: Surpassing the information theoretic bound with fusion trees. J. Comput. System Sci. 47, 424–436 (1993)
15. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. of FOCS, pp. 285–298 (1999)
16. Goodrich, M.T., Hirschberg, D.S., Mitzenmacher, M., Thaler, J.: Fully de-amortized cuckoo hashing for cache-oblivious dictionaries and multimaps. CoRR, abs/1107.4378 (2011)
17. Goodrich, M.T., Mitzenmacher, M.: Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011)
18. Guo, R., Cheng, X., Xu, H., Wang, B.: Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In: Proc. of CIKM, pp. 751–760 (2007)
19. Kirsch, A., Mitzenmacher, M.: Using a queue to de-amortize cuckoo hashing in hardware. In: Proc. of 45th Allerton Conference, pp. 751–758 (2007)
20. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: cuckoo hashing with a stash. SIAM J. Comput. 39, 1543–1561 (2009)
21. Knuth, D.E.: Sorting and Searching. The Art of Computer Programming, vol. 3. Addison-Wesley, Reading (1973)

22. Lester, N., Moffat, A., Zobel, J.: Efficient online index construction for text databases. ACM Trans. Database Syst. 33, 19:1–19:33 (2008)
23. Lester, N., Zobel, J., Williams, H.: Efficient online index maintenance for contiguous inverted lists. Inf. Processing & Management 42(4), 916–933 (2006)
24. Luk, R.W., Lam, W.: Efficient in-memory extensible inverted file. Information Systems 32(5), 733–754 (2007)
25. Mitzenmacher, M., Upfal, E.: Probability and computing - randomized algorithms and probabilistic analysis. Cambridge University Press (2005)
26. Pagh, R., Rodler, F.: Cuckoo hashing. Journal of Algorithms 52, 122–144 (2004)
27. Pagh, R., Wei, Z., Yi, K., Zhang, Q.: Cache-oblivious hashing. In: Proc. of PODS, pp. 297–304 (2010)
28. Rao Kosaraju, S., Pop, M.: De-amortization of Algorithms. In: Hsu, W.-L., Kao, M.-Y. (eds.) COCOON 1998. LNCS, vol. 1449, pp. 4–14. Springer, Heidelberg (1998)
29. Siegel, A.: On universal classes of extremely random constant-time hash functions. SIAM J. Comput. 33(3), 505–543 (2004)
30. Thorup, M.: On AC^0 implementations of fusion trees and atomic heaps. In: Proc. of SODA, pp. 699–707 (2003)
31. Willard, D.E.: Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. SIAM J. Comput. 29, 1030–1049 (1999)
32. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. 38 (July 2006)

An Efficient Generator for Clustered Dynamic Random Networks*

Robert Görke, Roland Kluge, Andrea Schumm, Christian Staudt, and Dorothea Wagner

Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany

Abstract. A *planted partition graph* is an Erdős-Rényi type random graph, where, based on a given partition of the vertex set, vertices in the same part are linked with a higher probability than vertices in different parts. Graphs of this type are frequently used to evaluate *graph clustering* algorithms, i.e., algorithms that seek to partition the vertex set of a graph into densely connected clusters. We propose a self-evident modification of this model to generate sequences of random graphs that are obtained by *atomic updates*, i.e., the deletion or insertion of an edge or vertex. The random process follows a dynamically changing ground-truth clustering that can be used to evaluate dynamic graph clustering algorithms. We give a theoretical justification of our model and show how the corresponding random process can be implemented efficiently.

1 Introduction

The broad variety of network structures we encounter in many fields of science and everyday life can mostly be modelled as *graphs*, where entities are mapped to vertices and the observed relationships to edges. It is not so clear how to subdivide the vertices of these graphs into clusters. In fact, no single answer to this question exists but a broad variety of approaches has been proposed in the past [1,2]. What all these measures agree on, is that clusters are characterized as densely connected subgraphs by some means or other. Frequently, the networks under consideration evolve over time, as vertices and links between them may appear or disappear. This can be either caused by random fluctuations or the split or merge of communities.

Even though real-world dynamic clustered instances with a reference clustering exist – that is, we know the clustering of the data in advance – there are several reasons why we are interested in generating artificial test data: First, we would like to produce graphs with a set of predefined properties such as the distribution of vertex degrees or size of the clusters, enabling us to examine the behavior of dynamic clustering algorithms under almost arbitrary circumstances. Second, real-world instances are not numerous and, most often, subject to confidentiality agreements.

* This work was partially supported by the DFG under grant WA 654/19-1 and WA 654/15-1.

Numerous results on random graphs exist [3]. One of the oldest and most fundamental models is the one introduced by Gilbert [4], in which each possible edge is present with uniform probability. This model can be altered in a straightforward way to incorporate a *planted partition*, i.e., a partition of the vertex set such that vertices in the same part are connected with a higher probability than vertices in different parts [5,6,7]. These graphs are frequently used to evaluate static graph clustering algorithms [1]; the big advantage of this approach is that the clustering obtained by an algorithm can be compared to the planted partition or *ground-truth clustering* used by the generator, which yields a possibility to assess clusterings independently of any particular quality measure. Well-known examples of random graphs based on this concept are the GN benchmark introduced by Girvan and Newman [8] and the relaxed cavemen graphs [9].

Prominent and fundamental models of social networks include small world networks [10] and the Barabási-Albert model [11]. The latter can be seen as a dynamic model for graph growth according to a preferential attachment process. Numerous variations thereof exist, most of which are targeted in capturing more accurately properties observed in real world social networks [12,13]. These models typically only simulate network growth and do not incorporate a known reference clustering. An exception is the model of Bagrow [14], where, starting from a graph generated according to Barabási-Albert, edges are randomly rewired to incorporate a given planted partition. The well-known LFR benchmark introduced by Lancichinetti and Fortunato is generated in a similar way [15]. While these approaches combine a reference partitioning with a more realistic degree distribution, the inherently dynamic process is lost. Other modifications of the planted partition model include the generalization to weighted [16] and bipartite [17] graphs, as well as hierarchical [18] and overlapping [19] reference clusterings. Aldecoa and Marín [20] propose to interpolate between two graphs with a strong clustering structure by rewiring edges at random. This process does not keep track of an explicit reference clustering over time, however, the assumption is that intermediate clusterings should have low distance to both the initial and the resulting clustering. Brandes and Mader [21] use as data for their experiments exponential-family random graphs [22] as basis and stochastic actor-oriented models [23] to describe the evolution between two networks. Both steps rely on properties of real-world dynamic networks that are given as input.

Other models for dynamic graphs based on random evolution according to a given Markov chain include *edge-Markovian Dynamic Graphs* [24,25]. This model does not incorporate a reference clustering but uses two fixed parameters p and q that represent the *edge birth-rate* and *edge death rate* of each possible edge. In contrast to the model we consider, an arbitrary number of edge deletions and insertion can take place in each time step.

Our Contribution. In this work, we augment the planted partition model by allowing *dynamic events*; edge and vertex events add or delete an edge or vertex, whereas cluster events split or merge clusters. More formally, we generate a time series of random graphs $G_0, \ldots, G_n$, where G_t emerges from G_{t-1} via *atomic updates*, i.e., the insertion or deletion of an edge or vertex. Over the whole generation process, the generator keeps track of a (dynamic) ground truth clustering. The probability of atomic events is chosen in a way that adheres to this clustering, without losing randomness. Graph growth/shrinkage and cluster dynamics can be simulated, steered by input parameters. Together with the benefit of the reference clustering, this can be used to thoroughly evaluate dynamic graph clustering algorithms, i.e., algorithms that incrementally update the calculated clustering as new node/edge events occur. A preliminary version of our generator is documented in our technical report [26] and the dissertation of one of the authors [27], and has been used in [28]. The new generator documented here differs fundamentally in the data structures used, which allows for faster practical and worst case running time, as well as linear space complexity. As the random model and parameters used are taken from the old generator, their description closely adheres to the technical report. Additional illustrations and proofs can be found in the full version of this paper [29]. Our generator is free for use and can be downloaded as Java software from our project page[1].

Notation. Let $G = (V, E)$ be an undirected, unweighted, and simple graph, i.e., G is loopless and has no parallel edges[2]. If not otherwise stated, n and m will always denote the cardinality of the sets V and E, respectively. The *degree* $\deg v$ of a vertex v is the number of its adjacent vertices. The set of all possible undirected edges in G is denoted with $\binom{V}{2}$. For a given graph G its *complement graph* $\overline{G}$ is defined as $\overline{G} = (\overline{V} = V, \overline{E} = \binom{V}{2} \setminus E)$. The pairs of vertices in $\overline{E}$ are called *non-edges* of G and $\overline{m} := |\overline{E}|$. A *clustering* $\mathcal{C} = \{C_1, \ldots, C_k\}$ is a partition of V where each of the C_i is non-empty. If not defined otherwise, the variable k will always refer to the number of clusters in $\mathcal{C}$. $\mathcal{C}(v)$ denotes the cluster in $\mathcal{C}$ that contains vertex v. For a cluster C the graph $G(C) = (C, E(C))$ is the vertex induced subgraph of C, where $E(C)$ are called *intracluster edges* of C. We identify a cluster C with the set of vertices it constitutes and with its vertex-induced subgraph of G. $m(C)$ is the number of edges in $G(C)$ and $\overline{m}(C) = \left|\binom{V(C)}{2} \setminus E(C)\right|$ is the number of *intracluster non-edges* of C. Edges having endpoints in two distinct clusters are called *intercluster edges*; the number of intercluster edges will be denoted with m_{inter}.

A *Markov chain* is a pair $M = (S, P)$, where S is a finite set of *states* and P a row stochastic matrix containing transition probabilities between the states. $C \subseteq S$ is *closed* if for all $i \in C$ and $j \in S \setminus C$ the transition probability from i to j is 0. M is *irreducible*, if there is no proper closed subset of S. We call a distribution vector w *stationary* if w is a left eigenvector of P.

2 Static Model

Gilbert's model on the generation of random graphs with uniform edge probability [4] can be easily modified to incorporate a planted partition [6,7]. The idea

[1] `http://i11www.iti.uni-karlsruhe.de/en/projects/spp1307/dyngen`
[2] Throughout this work, we will only consider graphs with this property.

behind this random model, which we will call $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$, is that vertices in the same cluster should be linked with high probability p_{in}, whereas intercluster edges should be present with a lower probability p_{out}, i.e., we always assume $p_{\text{in}} > p_{\text{out}}$.

The parameter n denotes the number of vertices. Edges are added randomly according to the following process. The generation is based on a fixed *ground truth* clustering $\mathcal{C} = \{C_1, \dots, C_k\}$ of the vertices. We chose to set p_{out} to a single value, whereas p_{in} is a list of length k: $p_{\text{in}} = \big(p_{\text{in}}(C_1), \dots, p_{\text{in}}(C_k)\big)$. For two vertices u and v the probability for edge $e = \{u, v\}$ to exist in a graph created with $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ is called *edge probability* $p(e) = p(u, v)$, where

$$p(u, v) = \begin{cases} p_{\text{in}}(C_i) & u, v \in C_i \\ p_{\text{out}} & \text{otherwise} \end{cases}$$

The probability of a graph G according to this model is thus

$$p(G) = \prod_{e \in E} p(e) \cdot \prod_{e \in \overline{E}} (1 - p(e))$$

Our dynamic generator is strongly based on this concept, and the first graph in the generated sequence is a $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ graph. The number of clusters as well as a list of intracluster probabilities and the uniform intercluster probability p_{out} are input parameters. Cluster sizes can either be set manually or determined automatically by the generator. In the latter case, we choose the cluster of a vertex uniformly at random which entails a binomial distribution of the cluster sizes with mean $\frac{n}{k}$.

Furthermore we introduce a coefficient β which *skews* the binomial distribution as follows ($\beta = 1$ in the unskewed case): Each cluster C_i is assigned to a subinterval $\big[\frac{i-1}{k}, \frac{i}{k}\big)$ of $[0, 1)$. When searching for a cluster to add a new vertex to, we draw an integer $i \in [0, k-1]$. Now, we add the vertex to the cluster which is assigned to the surrounding interval of $\big(\frac{i}{k}\big)^{\beta}$. Examples for cluster size distributions with different values of β can be found in our technical report [26].

3 Edge Dynamics

Neglecting cluster dynamics for the moment, we describe the random process we use for edge dynamics, along with some theoretical properties of the random sequence generated. We further give details on how this process is implemented in our generator, together with worst case guarantees on running times.

3.1 Associated Markov Chain and Distribution

At first glance, we would like to have a random process that triggers an edge operation, i.e., insertion or deletion, in each time step such that the relative frequency of a graph G in this sequence follows its probability $p(G)$ in the

$\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ model. Unfortunately, such a random process does not have to exist in general: Consider for example the simple case that we have two vertices and the probability of the edge between these equals 0.1. Under these assumptions, the probability of the empty (complete) graph on two vertices equals 0.9 (0.1), respectively. On the other hand, there is only one possible edge operation in each state. Hence, each random sequence alternates between the states, which can never lead to the assumed probabilities.

There is however a simple random process that follows prescribed edge probabilities if we allow for repeated occurrences of a graph in the sequence. This process chooses in each step a pair of vertices u and v uniformly at random, deletes the edge $\{u, v\}$ if it exists and (re)inserts it with probability $p(u, v)$. There is a natural correspondence between this procedure and a Markov chain M' whose states represent all possible (labeled) graphs on n vertices. It is not hard to see that $G(n, p_{\text{in}}, p_{\text{out}})$ is the unique stationary distribution of this chain. Thus, if we choose our initial state according to this distribution, the expected relative frequency of a graph generated by this Markov chain follows $G(n, p_{\text{in}}, p_{\text{out}})$.

However, in the context of evaluating dynamic algorithms, we are typically interested in sequences of graphs such that consecutive graphs follow from each other by atomic changes. We therefore slightly modify M' such that each time step that does not change the graph is discarded and call the resulting Markov chain M. Let $P_E := \sum_{e \in E} \big(1 - p(e)\big)$ and $P_{\overline{E}} := \sum_{e \in \overline{E}} p(e)$. Then, the probability $p_{\text{del}}(u, v)$ that one step of M deletes an existing edge $\{u, v\}$ is

$$p_{\text{del}}(u, v) = \frac{\binom{n}{2} \cdot \big(1 - p(u, v)\big)}{\sum_{e \in E} \left[\binom{n}{2} \cdot \big(1 - p(e)\big)\right] + \sum_{e \in \overline{E}} \left[\binom{n}{2} \cdot p(e)\right]} = \frac{1 - p(u, v)}{P_E + P_{\overline{E}}}$$

and the probability $p_{\text{ins}}(u, v)$ of inserting a non-existing edge is

$$p_{\text{ins}}(u, v) = \frac{\binom{n}{2} \cdot p(u, v)}{\sum_{e \in E} \left[\binom{n}{2} \cdot \big(1 - p(e)\big)\right] + \sum_{e \in \overline{E}} \left[\binom{n}{2} \cdot p(e)\right]} = \frac{p(u, v)}{P_E + P_{\overline{E}}}.$$

Intuitively, we expect the relative frequency of unlikely states in the sequence generated by M to be slightly larger than in the sequence generated by M' and vice versa, as they are less often discarded. The following theorem, a proof of which can be found in [29], makes this intuition precise.

Theorem 1. *If we choose the initial graph G_0 according to $\mathcal{G}(n, p_{in}, p_{out})$, the expected relative frequency of a graph $G = (V, E)$ in a sequence R generated by M is*

$$p(G) \cdot \left[\frac{\sum_{e \in E} \big(1 - p(e)\big) + \sum_{e \in \overline{E}} p(e)}{2 \cdot \sum_{e \in \binom{V}{2}} p(e) \big(1 - p(e)\big)}\right]$$

Until now, we have assumed that the initial graph is a $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ graph and that the following time steps are obtained by mere edge dynamics. As motivated in the introduction, our generator also incorporates vertex and cluster dynamics, which we will introduce later on. The latter two types of dynamics disturb the probability distribution in a way that is difficult to analyze. However, as it is always possible to reach any graph from any other graph in a finite number of steps with a positive probability, M is irreducible. The expected relative frequency in Theorem 1 is therefore the only stationary distribution of M and we can expect the relative frequency of graphs to get close to this distribution after a sufficiently large number of edge operations following vertex or cluster events [30].

3.2 Data Structures and Implementation

After a brief description of how vertex pairs can be enumerated continuously, we introduce the dynamic data structures our generator builds upon. In the second part, we show how these data structures can be used to efficiently implement edge dynamics.

Enumerating Vertex Pairs. To simplify the process of drawing random edges, we will use a bijection between pairs of vertices and integers between 0 and $\binom{|V|}{2}$ proposed by Batagelj and Brandes [31] to enumerate potential edges. Figure 1 intuitively illustrates this bijection by using the adjacency matrix of the graph. As we only consider undirected graphs, potential edges correspond to the entries below the diagonal. These entries can be enumerated by traversing this sub matrix likewise. For given vertices u and v, the index $e(u, v)$ can be obtained by

$$e(u,v) = \sum_{k=0}^{u-1} k + v = \frac{1}{2}(u-1)u + v$$

Vice versa, given the edge index $e(u, v)$, the corresponding vertices u and v can be found as follows:

$$u = 1 + \left\lfloor -\frac{1}{2} + \sqrt{\frac{1}{4} + 2 \cdot e(u,v)} \right\rfloor$$

$$v = e(u,v) - \frac{1}{2}u(u-1)$$

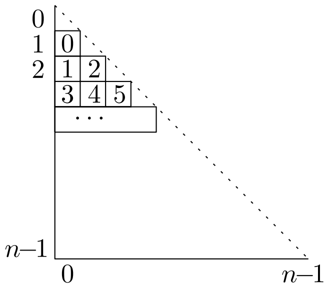


Fig. 1. Indexing scheme

Random Binary Selection Tree. In order to choose the next edge to be added or deleted, we need a data structure that allows us to efficiently draw an element from a weighted set $O = \{o_1, \ldots, o_n\}$ such that the probability to choose a certain element is proportional to its weight. Given such a data structure, a naïve generator could simply store each potential edge $\{u, v\}$ with weight $p(u, v)$ and each non-edge with weight $1 - p(u, v)$ and iteratively draw edges to add or delete. Later on, we will show that one entry for each cluster rather than for each edge is sufficient.

A simple solution for such a data structure is an array A storing prefix sums of the weights, i.e., $A[k] = \sum_{i=1}^{k-1} w(o_i)$, $1 \leq k \leq n+1$. Let W be the sum of all weights. Then we can draw a uniformly distributed random number x in $[0, W)$ and use binary search to find the index k such that $A[k] \leq x < A[k+1]$. It is easy to see that this process selects each element o_k with probability $w(o_k)/W$. For static sets this approach works well, however, we will need to update weights frequently, and to add and delete elements occasionally. Updating the prefix sums has linear worst-case complexity, which we would like to avoid.

We therefore use a complete binary tree to store the elements. We define each vertex of this tree to be a tuple $q_i = (o_i, w_i, l_i, r_i)$, where o_i is an element, w_i is its associated weight $w(o_i)$, l_i is the sum of the weights in the left subtree and r_i is the sum of weights in its right subtree. The weight l_m and r_m of a leaf q_m are simply 0. Contrary to the prefix sum array, inserting and deleting elements as well as weight updates can be done in logarithmic time. To keep the tree balanced, new elements are inserted as leaves with minimum distance to the root and deleted elements are replaced by a leaf element with maximum distance to the root. Afterwards, weight changes have to be propagated on the path(s) to the root.

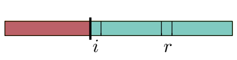
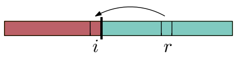
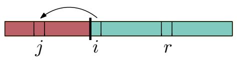
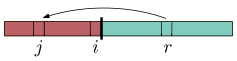
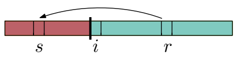
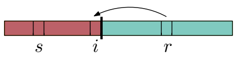
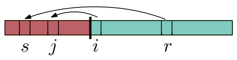
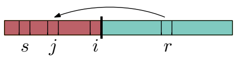
The procedure for the selection of an element starts at the root s by drawing a random number x from the interval $[0, l_s + w_s + r_s)$. Now there are three possible ranges for x: if $l_s \leq x < l_s + w_s$, the element is returned; if $x < l_s$, the carryover x is sent to the left subtree; and if $l_s + w_s \leq x$, the carryover $x - w_s - l_s$ is sent to the right subtree. The procedure continues recursively from there until an element is returned after at most $O(\log n)$ steps. The correctness of the selection process can be seen by constructing an array with prefix sums of the weights such that elements in the array are ordered according to an inorder traversal of the tree. Selecting an element in the tree is equivalent to a binary search in this array. An example for a random binary selection tree and the corresponding array can be found in the full version [29].

Virtual Fisher Yates Shuffle. As edges between pairs of vertices in the same cluster are equiprobable, using a binary tree to store all intracluster vertex pairs is quite inefficient. Instead, we use a modified *Fisher-Yates shuffle* [32] for this task. A Fisher-Yates shuffle is a simple method to uniformly sample without replacement from a given set of n elements. The elements are stored in an array of size n with indices from 0 to $n-1$ and the *border index* i of the shuffle is initially set to 0. In each step, a random number r between i and $n-1$ is drawn, the corresponding element at index r is marked as selected and swapped with the element at index i. Then, the border index i is increased by 1. It is easy to see that each element can only be chosen once and the probability to choose each subset of size k is the same.

A drawback of this approach is that we have to enumerate and store each element explicitly. For elements that can be easily enumerated it is more efficient to store an implicit representation of this array one of which is the *virtual Fisher-Yates shuffle* introduced by Batagelj and Brandes [31].

Let i be the number of elements drawn so far, L be the set of indices smaller than i that have not yet been drawn and H the set of indices at least i that have been drawn. In our view, the indices in L and H are "exceptions from the

Table 1. Illustrative figures for *select*. Index $r \in \{i, \ldots, n-1\}$ is drawn uniformly at random.

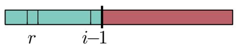
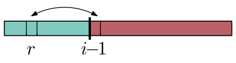
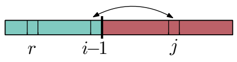
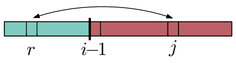
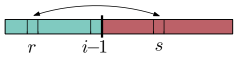
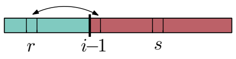
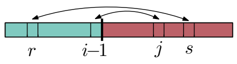
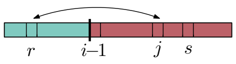
	Initial state	Final state
Case 1		
Case 2		
Case 3		
Case 4		

rule that small indices are selected and large indices unselected". The crucial observation is that the cardinality of L and H is equal. Hence, we can define a bijection from H to L and store it in a map `replace`. Similarly to the original Fisher-Yates shuffle, we can now iteratively draw a random number r between i and $n-1$. If `replace`$(r) = \perp$, i.e., if there is no entry for r in the map, index r has not yet been selected and we select the corresponding element. If `replace`$(r) = s$, we choose index s instead. This process guarantees that we draw in each step an unselected element with uniform probability.

After we have selected the element, we have to update the map such that it is still guaranteed that each element in L is assigned to a corresponding element in H. Depending on the previous state of the shuffle, we have to consider the four cases shown in Table 1. Entries of the form `replace`$(x) = y$ are depicted as arrows from x to y and the thick line marks the border between elements with index less than i and larger i. It is easy to see that in each case a constant number of lookup, delete and insert operations in `replace` suffices. If we assume that the data structure we use for the map `replace` is a binary search tree, these operations take logarithmic time. Hence, the time complexity of choosing the next element is in $O(\log n)$.

Unlike the generation of static random graphs, we also need to delete edges over time. To this end, we have to modify the virtual Fisher-Yates shuffle slightly to deselect already selected numbers, i.e., putting them once again in the set of selectable elements. This can be done by making the pointers bidirectional, i.e., for each entry of the form `replace`$(j) = i$, we add a corresponding entry `replace`$(i) = j$. Deselection now works analogously to selection: First, we draw a random number r in the interval $[0, i-1]$. If there is no entry for r in `replace`, r is a currently selected index. In this case, we undo the selection by setting `replace`$(i-1) = r$ and vice versa and move the border one index to the left. If r is an unselected element, we undo the selection of `replace`(r) instead. Special care has to be taken if the element at index $i-1$ is also unselected, i.e., if `replace`$(i-1) \neq \perp$. Figure 2 depicts all possible cases and the corresponding

Table 2. Illustrative figures for *deselect*. Index $r \in \{0, \ldots, i-1\}$ is drawn uniformly at random.

	Initial state	Final state
Case 1		
Case 2		
Case 3		
Case 4		

updates of the map. Deselection is therefore just as efficient as selection. For each selected element, we have to store at most two entries in the map, hence the space requirement is linear in the number i of currently selected elements.

Overview of Selection Procedure. In this section we explain how we use a combination between a random binary selection tree and virtual Fisher-Yates shuffles to efficiently implement edge dynamics according to the random model described in Section 3.1. For the sake of simplicity, we first assume that we only want to draw intracluster edges, i.e., $p_{\text{out}} = 0$. In this case, the following procedure can be used.

Each cluster has an associated shuffle that stores all intracluster edges in the cluster. This shuffle is used to uniformly select edges in the cluster to delete or to insert. To be able to use the enumeration scheme described above, each vertex v receives two ids, a global id that unambiguously identifies the vertex during the whole generation process and a local id in the range $[0, |\mathcal{C}(v)| - 1]$. The local id is used to enumerate intracluster edges in the individual shuffles.

On top of that, we store two randomized binary selection trees $\varGamma_{\text{ins}}$ and $\varGamma_{\text{del}}$ that each contain one entry for every cluster. The weight of each cluster C in $\varGamma_{\text{ins}}$ corresponds to the sum of the probability weight of edge insertions within the cluster, $\overline{m}(C) \cdot p_{\text{in}}(C)$. Similarly, its weight in $\varGamma_{\text{del}}$ is defined as $m(C) \cdot \big(1 - p_{\text{in}}(C)\big)$. The overall process of selecting the next edge operation is now divided into three steps:

1. As introduced in Section 3.1, let $P_E = \sum_{\{u,v\} \in E} \big(1 - p(u,v)\big)$ and $P_{\overline{E}} = \sum_{\{u,v\} \notin E} p(u,v)$. With probability $P_E/(P_E + P_{\overline{E}})$ we decide to delete and with probability $P_{\overline{E}}/(P_E + P_{\overline{E}})$ to insert an edge.
2. For edge deletions, we choose a cluster C in $\varGamma_{\text{del}}$ according to the stored weights. Similarly, we choose a cluster in $\varGamma_{\text{ins}}$ if we have decided to insert an edge.
3. Depending on the choice in the first step, we insert or delete an edge in the virtual Fisher-Yates shuffle associated with C.

Finally, the weight of C in Γ_{ins} and Γ_{del} has to be updated. A proof that this process inserts or deletes an edge with probabilities according to the random process described in Section 3.1 can be found in the full version [29]. It remains to describe how this procedure can be altered to be able to deal with intercluster edges.

Dealing with Intercluster Edges. In principle, it would be possible to handle intercluster edges analogously and just introduce a virtual Fisher-Yates shuffle containing all pairs of vertices in different clusters. As all these vertices exist with the same probability, this would be perfectly feasible. The problem with this approach is that it is not easy to consistently enumerate intercluster vertex pairs, as, due to vertex and cluster dynamics, the number of vertices in each cluster changes. For this reason, we introduce a shuffle for a *pseudocluster* containing all vertices in the graph. This pseudocluster gets an entry in Γ_{ins} with weight $\overline{m} \cdot p_{\text{out}}$. As the pseudocluster contains all vertices, the associated shuffle also contains intracluster vertex pairs. Hence, it is possible to draw intracluster edges either in the shuffle of the corresponding cluster or in the shuffle of the pseudocluster, which overestimates the probability of choosing such edges. To correct this, we exploit our assumption that $p_{\text{in}}(C) > p_{\text{out}}$ for each cluster C and decrease the weight associated with C in Γ_{ins} to $\overline{m}(C) \cdot (p_{\text{in}}(C) - p_{\text{out}}) \geq 0$.

For edge deletions, this trick cannot be used as $1 - p_{\text{out}}$ is larger than $1 - p_{\text{in}}(C)$. This is why we introduce an additional array storing the global id of all intercluster edges and draw a random edge in this array in case we decide to delete an intercluster edge. The respective edge is then both deleted in the array and in the shuffle of the pseudocluster. This procedure guarantees that we do not erroneously delete intracluster edges by choosing from the pseudocluster, which is why the weight of the true clusters in Γ_{del} remains unchanged. The weight of the pseudocluster in Γ_{del} corresponds to $m_{\text{inter}}(1 - p_{\text{out}})$. Whenever an intracluster edge is inserted or deleted, either in the pseudocluster or in its own cluster, the corresponding entry in the other shuffle has to be updated.

Time and Space Requirement. Deciding whether to insert or delete an edge is done in constant time. For both operations, choosing the cluster in the respective cluster tree takes time $O(\log k)$, where k denotes the current number of clusters. Similarly, selection or deselection in the corresponding virtual Fisher-Yates shuffle takes time logarithmic in the size of the shuffle. Hence, the expected time complexity for both operations is in $O(\log n)$. As the total size of the shuffles is asymptotically upper bounded by the number of edges of the current graph, it is easy to verify that the total space requirement is linear in the graph size. Vertex dynamics are steered by a parameter p_χ that specifies the probability that instead of an edge operation, we delete or insert a vertex. If a vertex operation is to be performed, another parameter p_ν determines the probability that this operation is a vertex insertion. Choosing p_ν to be smaller or larger than 0.5 gives the opportunity to simulate graph growth or shrinkage. For deletion, a vertex is chosen uniformly at random and all incident edges are deleted. To adhere to the initial cluster size distribution, new vertices are assigned to clusters according to expected cluster sizes, similar to the generation of the initial clustering. To dampen the effect of additional edge deletions in the course of vertex deletions and to stay closer to the $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ model, new vertices are immediately connected to other vertices according to the prescribed edge probabilities. Naïvely, this takes $O(n)$ time, however, it is possible to use the *geometric method* introduced by Fan et al. [33] and used by Batagelj and Brandes [31] to reduce the running time to $O(\deg v)$, where $\deg v$ is the resulting degree of the new vertex v.

It remains to explain what has to be done to update the data structures. Updating the affected entry in the cluster trees takes $O(\log k)$ time. If a new vertex is inserted or deleted, the index space of the shuffle of its cluster and of the pseudocluster has to be adapted. For insertion, it suffices to assign the highest vertex id in the cluster to the new vertex and increase the index space of the shuffle accordingly. If a vertex is deleted, we have to guarantee that the index space is still continuous. We do this by relabeling the vertex v_f with the previously largest local id in the shuffle of the deleted vertex u by the id of u. The same procedure has to be performed for the vertex w_f with the largest local id in the pseudocluster. Figure 2 illustrates this case.

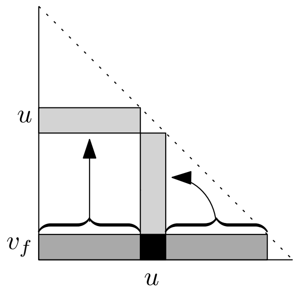


Fig. 2. Update if vertex with local id u is deleted

For the relabeling step, we first delete all edges incident to v_f or w_f in the shuffle and then reinsert them using the new edge ids. Hence, the overall expected time complexity of the deletion of vertex v is $O\big((\deg v + \deg v_f + \deg w_f) \cdot \log n\big)$, whereas for vertex insertion we get $O\big(\deg v \cdot \log n\big)$.

Cluster Dynamics. Cluster dynamics is independent of vertex and edge dynamics in the sense that in each time step, additionally to the vertex or edge operation performed, a cluster operation can take place. The probability of a cluster operation is determined by the input parameter p_ω and the probability that this cluster operation merges two clusters is determined by the parameter p_μ. With probability $1 - p_\mu$, one of the clusters is split by assigning each of its vertices to one of the new clusters with uniform probability.

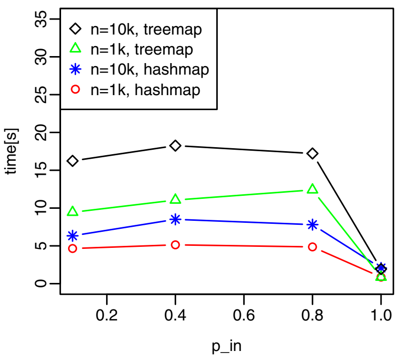
To calculate new values for p_{in}, an obvious possibility is to just use the old value(s). For a cluster split, this means that the new clusters inherit p_{in} from the old cluster, whereas for a cluster merge, the new cluster is assigned the average of the intracluster edge probabilities of the participating clusters. This process leads to increasingly uniform p_{in} values over the course of time. For this reason, depending on a user parameter, new p_{in} values are generated randomly according to a Gaussian distribution estimated from the initially given list of intracluster probabilities. A more detailed description can be found in our technical report [26].

One of the main motivations for using $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ graphs for the evaluation of clustering algorithms is the knowledge of a ground truth clustering the result of the algorithm can be compared to. However, for cluster dynamics, it can easily be imagined that immediately after a split or merge operation, the clustering algorithm has no chance to detect the current ground truth clustering, as the change is not yet reflected in the edge structure. For this reason, the generator keeps track of an additional *reference clustering* that follows the current ground-truth clustering with some delay, roughly speaking, as soon as the involved subgraph becomes similar enough to the ground truth's expectation. A detailed description of the behavior of the reference clustering can be found in our technical report [26]. To prevent the interleaving of concurrent cluster events, as long as the change in the ground-truth is not propagated to the reference clustering, the participating clusters are not available for further cluster operations. If, for this reason, in some time step a cluster event is triggered but no available clusters are found, the cluster event does not take place.

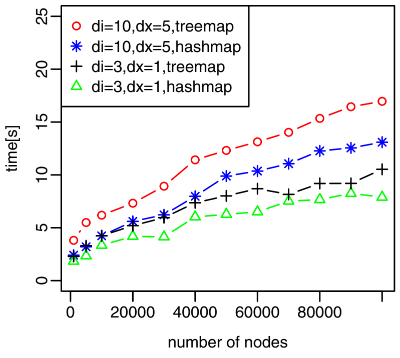
To keep the data structures up to date, we delete the involved old shuffle(s) and create one or two new shuffles from scratch, depending on the kind of cluster operation. Furthermore, the additional array associated with the pseudocluster that stores all intercluster edges has to be updated. The reason for this is that for a split operation, new intercluster edges between the two parts arise, whereas cluster merges turn some intercluster edges into intracluster edges. If pointers are used to link the occurrences of an edge in different data structures, all these operations take $O\big((n(C) + m(C)) \cdot \log n\big)$ time, where C is either the cluster that is about to be split or the new cluster after a merge. Removing the old cluster(s) and inserting the new cluster(s) in the cluster trees takes logarithmic time. Hence, this does not increase the (asymptotic) running time.

4 Experiments

We give a brief impression of the absolute running times of different sample configurations. All values are averaged over 15 runs and do not include the time to write the graph to hard disc. The first experiments in Figure 3a evaluate the running times for one million steps of the generator, while only the intracluster density varies. The planted partition contains 15 clusters and the cluster size distribution is unskewed ($\beta = 1$). The number of vertices as well as the intercluster edge probability $p_{\text{out}} = 0.1$ are constant. As the number of edges is in $\Theta(n^2)$, the time to generate the initial graph sometimes dominates the time needed for edge dynamics and is therefore not included in the plot. Similarly, as the expected degree of each vertex is in $\Theta(n)$, we didn't include vertex and cluster dynamics and set the corresponding probabilities to 0. As expected, the running time is almost independent of p_{in}. The low running times for the experiments with $p_{\text{in}} = 1$ can be explained by the fact that intracluster edges are never deleted or inserted and all dynamics involve intercluster vertex pairs. To obtain logarithmic worst case running time for edge operations, the map `replace` used for the virtual Fisher-Yates shuffles can be stored in a binary search tree. For comparison, we



(a) Edge dynamics, constant number of vertices, variable p_{in}



(b) Full dynamics, constant expected intracluster (intercluster) degree `di` (`dx`), variable n

Fig. 3. Running times for some sample configurations

repeated the experiments with hash maps instead of these trees. It can be seen that for graphs of high density, hash maps yield better practical running times[3].

Figure 3b illustrates the running time for less artificial parameter settings. Here, the number of clusters equals $\sqrt{n}$ and the size distribution is skewed ($\beta = 0.5$). The probability of a vertex event instead of an edge event is set to 0.1 and in half of the cases a vertex is added (deleted). The probability of a cluster event is 0.01 and in half of the cases a cluster is split (two clusters are merged). The expected vertex degree is constant, which yields very sparse graphs. To give a more realistic impression of the total running time, we included the time to generate the initial graph, followed by 100 000 updates. As above, the running times obtained by using hash maps are better than for the tree based variant.

In summary, the experiments show that hash maps yield better practical performance and that dynamics can be added to the planted partition model without causing much overhead.

Implementation Notes. We conducted all experiments on a Dual-Core AMD Opteron(tm) Processor clocked at 2.6 GHz, using Java version 1.6.0_22. The machine has 32GB of RAM and 2 × 1 MB of L2 cache. The implementation uses no external libraries. As hash map, we used `java.util.HashMap`, whereas the tree-based implementation uses `java.util.TreeMap`, which is based on a red-black tree.

5 Conclusion and Outlook

We proposed a dynamic generalization of the planted partition model that can be used to evaluate dynamic graph clustering algorithms, with the additional

[3] Note that entries in the hash map are not distributed evenly over all possible indices, which is why we don't have expected constant time for all parameter settings.

benefit of a known reference clustering. Furthermore, we described how large dynamic random graphs according to this model can be efficiently generated and showed the practicability of this approach on selected example configurations. In order to make this model more realistic, modifications similar to the static model are conceivable. Possible changes in the random model include vertex movements, less uniform degree distribution, higher clustering coefficient as well as the generalization to hierarchical reference clusterings.

References

1. Fortunato, S.: Community detection in graphs. Physics Reports 486(3-5), 75–174 (2010)
2. Schaeffer, S.E.: Graph Clustering. Computer Science Review 1(1), 27–64 (2007)
3. Bollobás, B.: Random Graphs. Cambridge University Press (2001)
4. Gilbert, H.: Random Graphs. The Annals of Mathematical Statistics 30(4), 1141–1144 (1959)
5. Condon, A., Karp, R.M.: Algorithms for Graph Partitioning on the Planted Partition Model. Randoms Structures and Algorithms 18(2), 116–140 (2001)
6. Brandes, U., Gaertler, M., Wagner, D.: Experiments on Graph Clustering Algorithms. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 568–579. Springer, Heidelberg (2003)
7. Gaertler, M., Görke, R., Wagner, D.: Significance-Driven Graph Clustering. In: Kao, M.-Y., Li, X.-Y. (eds.) AAIM 2007. LNCS, vol. 4508, pp. 11–26. Springer, Heidelberg (2007)
8. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proceedings of the National Academy of Science of the United States of America 99(12), 7821–7826 (2002)
9. Watts, D.J.: Small worlds: The dynamics of networks between order and randomness. Princeton University Press (1999)
10. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature 393(6684), 440–442 (1998)
11. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science 286, 509–512 (1999)
12. Leskovec, J., Kleinberg, J.M., Faloutsos, C.: Graphs Over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In: Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 177–187. ACM Press (2005)
13. Vázquez, A.: Growing network with local rules: Preferential attachment, clustering hierarchy, and degree correlations. Physical Review E 67, 056104 (2003)
14. Bagrow, J.: Evaluating local community methods in networks. Journal of Statistical Mechanics: Theory and Experiment, P05001 (2008), doi:10.1088/1742-5468/2008/05/P05001
15. Lancichinetti, A., Fortunato, S.: Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. Physical Review E 80(1), 016118 (2009)
16. Fan, Y., Li, M., Zhang, P., Wu, J., Di, Z.: Accuracy and precision of methods for community identification in weighted networks. Physica A 377(1), 363–372 (2007)
17. Guimerà, R., Sales-Pardo, M., Amaral, L.A.N.: Module identification in bipartite and directed networks. Physical Review E 76, 036102 (2007)

18. Zhou, H.: Network landscape from a Brownian particle's perspective. Physical Review E 67, 041908 (2003)
19. Sawardecker, E.N., Sales-Pardo, M., Amaral, L.A.N.: Detection of node group membership in networks with group overlap. The European Physical Journal B 67, 277–284 (2009)
20. Aldecoa, R., Marín, I.: Closed benchmarks for network community structure characterization. Physical Review E 85, 026109 (2012)
21. Brandes, U., Mader, M.: A Quantitative Comparison of Stress-Minimization Approaches for Offline Dynamic Graph Drawing. In: van Kreveld, M., Speckmann, B. (eds.) GD 2011. LNCS, vol. 7034, pp. 99–110. Springer, Heidelberg (2011)
22. Robins, G., Pattison, P., Kalish, Y., Lusher, D.: An introduction to exponential random graph (p*) models for social networks. Social Networks 29(2), 173–191 (2007)
23. Snijders, T.A.: The Statistical Evaluation of Social Network Dynamics. Sociological Methodology 31(1), 361–395 (2001)
24. Clementi, A.E.F., Macci, C., Monti, A., Pasquale, F., Silvestri, R.: Flooding time in edge-Markovian dynamic graphs. SIAM Journal on Discrete Mathematics 24(4), 1694–1712 (2010)
25. Baumann, H., Crescenzi, P., Fraigniaud, P.: Parsimonious flooding in dynamic graphs. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, pp. 260–269. ACM Press (2009)
26. Görke, R., Staudt, C.: A Generator for Dynamic Clustered Random Graphs. Technical report, Informatik, Uni Karlsruhe, TR 2009-7 (2009)
27. Görke, R.: An Algorithmic Walk from Static to Dynamic Graph Clustering. PhD thesis, Fakultät für Informatik (February 2010)
28. Görke, R., Maillard, P., Staudt, C., Wagner, D.: Modularity-Driven Clustering of Dynamic Graphs. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 436–448. Springer, Heidelberg (2010)
29. Görke, R., Kluge, R., Schumm, A., Staudt, C., Wagner, D.: An Efficient Generator for Clustered Dynamic Random Networks. Technical report, Karlsruhe Reports in Informatics 2012, 17 (2012)
30. Behrends, E.: Introduction to Markov Chains With Special Emphasis on Rapid Mixing. Friedrick Vieweg & Son (October 2002)
31. Batagelj, V., Brandes, U.: Efficient Generation of Large Random Networks. Physical Review E 036113 (2005)
32. Fisher, R.A., Yates, F.: Statistical Tables for Biological, Agricultural and Medical Research. Oliver and Boyd, London (1948)
33. Fan, C.T., Muller, M.E., Rezucha, I.: Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital-Computers. Journal of the American Statistical Association 57(298), 387–402 (1962)

Slow Down and Sleep for Profit in Online Deadline Scheduling*

Peter Kling, Andreas Cord-Landwehr, and Frederik Mallmann-Trenn

Heinz Nixdorf Institute and Computer Science Department, University of Paderborn
{andreas.cord-landwehr,peter.kling}@uni-paderborn.de,
xarph@mail.uni-paderborn.de

Abstract. We present and study a new model for energy-aware and profit-oriented scheduling on a single processor. The processor features dynamic speed scaling as well as suspension to a sleep mode. Jobs arrive over time, are preemptable, and have different sizes, values, and deadlines. On the arrival of a new job, the scheduler may either accept or reject the job. Accepted jobs need a certain energy investment to be finished in time, while rejected jobs cause costs equal to their values. Here, power consumption at speed s is given by $P(s) = s^{\alpha} + \beta$ and the energy investment is power integrated over time. Additionally, the scheduler may decide to suspend the processor to a sleep mode in which no energy is consumed, though awaking entails fixed transition costs γ. The objective is to minimize the total value of rejected jobs plus the total energy.

Our model combines aspects from advanced energy conservation techniques (namely speed scaling and sleep states) and profit-oriented scheduling models. We show that *rejection-oblivious* schedulers (whose rejection decisions are not based on former decisions) have – in contrast to the model without sleep states – an unbounded competitive ratio w.r.t. the processor parameters α and β. It turns out that the worst-case performance of such schedulers depends linearly on the jobs' value densities (the ratio between a job's value and its work). We give an algorithm whose competitiveness nearly matches this lower bound. If the maximum value density is not too large, the competitiveness becomes $\alpha^{\alpha} + 2e\alpha$. Also, we show that it suffices to restrict the value density of low-value jobs only. Using a technique from [13] we transfer our results to processors with a fixed maximum speed.

1 Introduction

Over the last decade, energy usage of data centers and computers in general has become a major concern. There are various reasons for this development: the ubiquity of technical systems, the rise of mobile computing, as well as a growing ecological awareness. Also from an economical viewpoint, energy usage can no longer be ignored. Energy costs for both the actual computation and the

* This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901) and by the Graduate School on Applied Network Science (GSANS).

cooling have become *the* decisive cost factor in today's data centers (see, e.g., Barroso and Hölzle [10]). In combination with improvements on the technical level, algorithmic research has great potential to reduce energy consumption. Albers [2] gives a good insight on the role of algorithms to fully exploit the energy-saving mechanisms of modern systems. Two of the most prominent techniques for power saving are *dynamic speed scaling* and *power-down*. The former allows a system to save energy by adapting the processor's speed to the current system load, while the latter can be used to transition into a sleep mode to conserve energy. There is an extensive body of literature on both techniques (see below). From an algorithmic viewpoint, the most challenging aspect in the design of scheduling strategies is to handle the lack of knowledge about the future: should we use a high speed to free resources in anticipation of new jobs or enter sleep mode in the hope that no new jobs arrive in the near future?

Given that profitability is a driving force for most modern systems and that energy consumption has gained such a high significance, it seems natural to take this relation explicitly into account. Pruhs and Stein [17] consider a scheduling model that does so by introducing job values. Their scheduler controls energy usage via speed scaling and is allowed to reject jobs if their values seem too low compared to their foreseeable energy requirements. The objective is to maximize the profit, which is modeled as the total value of finished jobs minus the invested energy. Our work is based on a result by Chan et al. [13]. We enhance their model by combining speed scaling and power-down mechanisms for energy management, which not only introduces non-trivial difficulties to overcome in the analysis, but proves to be inherently more complex compared to the original model insofar that classical algorithms can become arbitrarily bad.

History and Related Work. There is much literature concerning energy-aware scheduling strategies both in practical and theoretical contexts. A recent survey by Albers [1] gives a good and compact overview on the state of the art in the dynamic speed scaling setting, also in combination with power-down mechanisms. In the following, we focus on theoretical results concerning scheduling on a single processor for jobs with deadlines. Theoretical work in this area has been initiated by Yao et al. [18]. They considered scheduling of jobs having different sizes and deadlines on a single variable-speed processor. When running at speed s, its power consumption is $P(s) = s^\alpha$ for some constant $\alpha \geq 2$. Yao et al. derived a polynomial time optimal offline algorithm as well as two online algorithms known as *optimal available* (OA) and *average rate* (AVR). Up to now, OA remains one of the most important algorithms in this area, as it is used as a basic building block by many strategies (including the strategy we present in this paper). Using an elegant amortized potential function argument, Bansal et al. [7] were able to show that OA's competitive factor is exactly α^α. Moreover, the authors stated a new algorithm, named BKP, which achieves a competitive ratio of essentially $2e^{\alpha+1}$. This improves upon OA for large α. The best known lower bound for deterministic algorithms is ${}^{e^{\alpha-1}}/_{\alpha}$ due to Bansal et al. [5]. They also presented an algorithm (qOA) that is particularly well-suited for low powers of α. An interesting and realistic model extension is the restriction of the maximum processor speed. In such a setting, a scheduler may not always be able to finish all jobs by their deadlines. Chan et al. [12] were the first to consider the combination of classical speed scaling with such a maximum speed. They gave an algorithm that is $\alpha^\alpha + \alpha^2 4^\alpha$-competitive on energy and 14-competitive on throughput. Bansal et al. [6] improved this to a 4-competitive algorithm concerning the throughput while maintaining a constant competitive ratio with respect to the energy. Note that no algorithm – even if ignoring the energy consumption – can be better than 4-competitive for throughput (see [11]).

Power-down mechanisms were studied by Baptiste [8]. He considered a fixed-speed processor needing a certain amount of energy to stay awake, but which may switch into a sleep state to save energy. Returning from sleep needs energy γ. For jobs of unit size, he gave a polynomial time optimal offline algorithm, which was later extended to jobs of arbitrary size [9]. The first work to combine both dynamic speed scaling and sleep states in the classical YAO-model is due to Irani et al. [16]. They achieved a 2-approximation for arbitrary convex power functions. For the online setting and power function $P(s) = s^\alpha + \beta$ a competitive factor of $4^{\alpha-1}\alpha^\alpha + 2^{\alpha-1} + 2$ was reached. Han et al. [15] improved upon this in two respects: they lowered the competitive factor to $\alpha^\alpha + 2$ and transferred the result to scenarios limiting the maximum speed. Only recently, Albers and Antoniadis [3] proved that the optimization problem is NP-hard and gave lower bounds for several algorithm classes. Moreover, they improved the approximation factor for general convex power functions to $4/3$. The papers most closely related to ours are due to Pruhs and Stein [17] and Chan et al. [13]. Both considered the dynamic speed scaling model of Yao et al. However, they extended the idea of energy-minimal schedules to a profit-oriented objective. In the simplest case, jobs have values (or priorities) and the scheduler is no longer required to finish all jobs. Instead, it can decide to reject jobs whose values do not justify the foreseeable energy investment necessary to complete them. The objective is to maximize profit [17] or, similarly, minimize the loss [13]. As argued by the authors, the latter model has the benefit of being a direct generalization of the classical model of Yao et al. [18]. For maximizing the profit, Pruhs and Stein [17] showed that, in order to achieve a bounded competitive factor, resource augmentation is necessary and gave a scalable online algorithm. For minimizing the loss, Chan et al. [13] gave a $\alpha^\alpha + 2e\alpha$-competitive algorithm and transferred the result to the case of a bounded maximum speed.

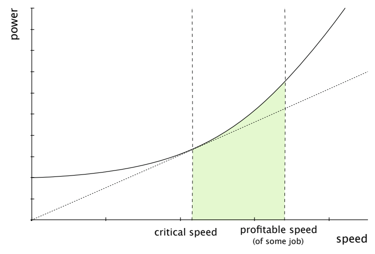
Our Contribution. We present the first model that not only takes into account two of the most prominent energy conservation techniques (namely, speed scaling and power-down) but couples the energy minimization objective with the idea of profitability. It combines aspects from both [16] and [13]. From [16] we inherit one of the most realistic processor models considered in this area: A single variable-speed processor with power function $P(s) = s^\alpha + \beta$ and a sleep state. Thus, even at speed zero the system is charged a certain amount β of energy, but it can suspend to sleep such that no energy is consumed. Waking up causes transition cost of γ. The job model stems from [13]: Jobs arrive in an online fashion, are preemptable, and have a deadline, size, and value. The scheduler can reject jobs (e.g., if their values do not justify the presumed energy investment). Its objective is to minimize the total energy investment plus the total value of rejected jobs.

A major insight of ours is that the maximum value density $\delta_{\max}$ (i.e., the ratio between a job's value and its work) is a parameter that is inherently connected to the necessary and sufficient competitive ratio achievable for our online scheduling problem. We present an online algorithm that combines ideas from [13] and [15] and analyze its competitive ratio with respect to $\delta_{\max}$. This yields an upper bound of $\alpha^\alpha + 2e\alpha + \delta_{\max}\frac{s_{\mathrm{cr}}}{P(s_{\mathrm{cr}})}$.[1] If the value density of low-valued jobs is not too large or job values are at least γ, the competitive ratio becomes $\alpha^\alpha + 2e\alpha$. Moreover, we show that one cannot do much better: any *rejection-oblivious* strategy has a competitive ratio of at least $\delta_{\max}\frac{s_{\mathrm{cr}}}{P(s_{\mathrm{cr}})}$. Here, rejection-oblivious means that rejection decisions are based on the *current* system state and job properties only. This lower bound is in stark contrast to the setting without sleep states, where a rejection-oblivious O(1)-competitive algorithm exists [13]. Using the definition of a job's penalty ratio (due to Chan et al. [13]), we extend our results to processors with a bounded maximum speed.

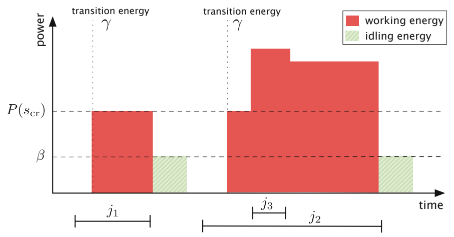
2 Model and Preliminaries

We are given a speed-scalable processor that can be set to any speed $s \in [0, \infty)$. When running at speed s its power consumption is $P_{\alpha,\beta}(s) = s^\alpha + \beta$ with $\alpha \geq 2$ and $\beta \geq 0$. If $s(t)$ denotes the processor speed at time t, the total power consumption is $\int_0^\infty P_{\alpha,\beta}(s(t))\,\mathrm{d}t$. We can suspend the processor into a sleep state to save energy. In this state, it cannot process any jobs and has a power consumption of zero. Though entering the sleep state is for free, waking up needs a fixed *transition energy* $\gamma \geq 0$. Over time, n jobs $J = \{1, 2, \ldots, n\}$ are released. Each job j appears at its release time r_j and has a deadline d_j, a (non-negative) value v_j, and requires a certain amount w_j of work. The processor can process at most one job at a time. Preemption is allowed, i.e., jobs may be paused at any time and continued later on. If I denotes the period of time (not necessarily an interval) when j is scheduled, the amount of work processed is $\int_I s(t)\,\mathrm{d}t$. A job is finished if $\int_I s(t)\,\mathrm{d}t \geq w_j$. Jobs not finished until their deadline cause a cost equal to their value. We call such jobs *rejected*. A schedule S specifies for any time t the processor's state (asleep or awake), the currently processed job (if the processor is awake), and sets the speed $s(t)$. W.l.o.g. we assume $s(t) = 0$ when no job is being processed. Initially, the processor is assumed to be asleep. Whenever it is neither sleeping nor working we say it is *idle*. A schedule's cost is the invested energy (for awaking from sleep, idling, and working on jobs) plus the loss due to rejected jobs. Let m denote the number of sleep intervals, l the total length of idle intervals, and $\mathcal{I}_{\mathrm{work}}$ the collection of all working intervals (i.e., times when $s(t) > 0$). Then, the schedule's *sleeping energy* is $E^S_{\mathrm{sleep}} := (m-1)\gamma$, its *idling*

[1] The expression $\frac{s_{\mathrm{cr}}}{P(s_{\mathrm{cr}})}$ depends only on α and β, see Section 2.



(a) Our algorithm tries to use job speeds that essentially stay in the shaded interval.



(b) A sample schedule and the involved energy types.

Fig. 1.

energy is $E^S_{\mathrm{idle}} := l\beta$, and its *working energy* is $E^S_{\mathrm{work}} := \int_{\mathcal{I}_{\mathrm{work}}} P_{\alpha,\beta}(s(t))\,\mathrm{d}t$. We use V^S_{rej} to denote the total value of rejected jobs. Now, the cost of schedule S is

$$\mathrm{cost}(S) := E^S_{\mathrm{sleep}} + E^S_{\mathrm{idle}} + E^S_{\mathrm{work}} + V^S_{\mathrm{rej}}. \tag{1}$$

We seek online strategies yielding a provably good schedule. More formally, we measure the quality of online strategies by their competitive factor: For an online algorithm A and a problem instance I let $A(I)$ denote the resulting schedule and $O(I)$ an optimal schedule for I. Then, A is said to be c-competitive if $\sup_I \frac{\mathrm{cost}(A(I))}{\mathrm{cost}(O(I))} \leq c$.

We define the *system energy* E^S_{sys} of a schedule to be the energy needed to hold the system awake (whilst idling and working). That is, if S is awake for a total of x time units, $E^S_{\mathrm{sys}} = x\beta$. Note that $E^S_{\mathrm{sys}} \leq E^S_{\mathrm{idle}} + E^S_{\mathrm{work}}$. The *critical speed* of the power function is defined as $s_{\mathrm{cr}} := \arg\min_{s\geq 0} {}^{P_{\alpha,\beta}(s)}/_{s}$ (cf. also [15, 16]). If job j is processed at constant speed s its energy usage is $w_j \cdot {}^{P_{\alpha,\beta}(s)}/_{s}$. Thus, assuming that j is the only job in the system and ignoring its deadline, s_{cr} is the energy-optimal speed to process j. One can easily check that $s^{\alpha}_{\mathrm{cr}} = \frac{\beta}{\alpha-1}$. Given a job j, let $\delta_j := {}^{v_j}/_{w_j}$ denote the job's *value density*. Following [13] and [17], we define the *profitable speed* $s_{j,\mathrm{p}}$ of job j to be the maximum speed for which its processing may be profitable. More formally, $s_{j,\mathrm{p}} := \max\{s \geq 0 \mid w_j \cdot {}^{P_{\alpha,0}(s)}/_{s} \leq v_j\}$. Note that the definition is with respect to $P_{\alpha,0}$, i.e., it ignores the system energy. The profitable speed can be more explicitly characterized by $s^{\alpha-1}_{j,\mathrm{p}} = \delta_j$. It is easy to see that a schedule that processes j at average speed faster than $s_{j,\mathrm{p}}$ cannot be optimal: rejecting j and idling during the former execution phase would be more profitable. See Figure 1 for an illustration of these notions.

Optimal Available and Structural Properties. One of the first online algorithms for dynamic speed scaling was Optimal Available (OA) due to [18]. As it is an essential building block not only of our but many algorithms for speed scaling, we give a short recap on its idea (see [7] for a thorough discussion and analysis). At any time, OA computes the optimal offline schedule assuming that no further jobs arrive. This optimal offline schedule is computed as follows: Let the density of an interval I be defined as $w(I)/|I|$. Here, $w(I)$ denotes the total work of jobs j with $[r_j, d_j) \subseteq I$ and $|I|$ the length of I. Now, whenever a job arrives OA computes so-called *critical intervals* by iteratively choosing an interval of maximum density. Jobs are then scheduled at a speed equal to the density of the corresponding critical interval using the earliest deadline first policy. Let us summarize several structural facts known about the OA schedule.

Fact 1. *Let S and S' denote the OA schedules just before and after j's arrival. We use $S(j)$ and $S'(j)$ to denote j's speed in the corresponding schedule.*

(a) The speed function of S (and S') is a non-increasing staircase function.
(b) The minimal speed of S' during $[r_j, d_j)$ is at least $S'(j)$.
(c) Let I be an arbitrary period of time during $[r_j, d_j)$ (not necessarily an interval). Moreover, let W denote the total amount of work scheduled by S and W' the one scheduled by S' during I. Then the inequality $W \leq W' \leq W + w_j$ holds.
(d) The speed of any $j' \neq j$ can only increase due to j's arrival: $S'(j') \geq S(j')$.

3 Lower Bound for Rejection-Oblivious Algorithms

This section considers a class of simple, deterministic online algorithms that we call *rejection-oblivious*. When a job arrives, a rejection-oblivious algorithm decides whether to accept or reject the job. This decision is based solely on the processor's current state (sleeping, idling, working), its current workload, and the job's properties. Especially it does not take former decisions into account. An example for such an algorithm is $PS(c)$ in [13]. For a suitable parameter c, it is $\alpha^\alpha + 2e\alpha$-competitive in a model without sleep state. In this section we show that in our model (i.e., with a sleep state) no rejection-oblivious algorithm can be competitive. More exactly, the competitiveness of any such algorithm can become arbitrarily large. We identify the jobs' value density as a crucial parameter for the competitiveness of these algorithms.

Theorem 1. *The competitiveness of any rejection-oblivious algorithm A is unbounded. More exactly, for any A there is a problem instance I with competitive factor $\geq \delta_{max} \frac{s_{cr}}{P_{\alpha,\beta}(s_{cr})}$. Here, δ_{max} is the maximum value density of jobs from I.*

Proof. For A to be competitive, there must be some $x \in \mathbb{R}$ such that, while A is asleep, all jobs of value at most x are rejected (independent of their work and deadlines). Otherwise, we can define a sequence of n identical jobs $1, 2, \ldots, n$ of arbitrary small value ϵ. W.l.o.g., we release them such that A goes to sleep during $[d_{j-1}, r_j)$ (otherwise A consumes an infinite amount of energy). Thus, A's cost is at least $n\gamma$. If instead considering schedule S that rejects all jobs, we have $\operatorname{cost}(S) = n\epsilon$. For $\epsilon \to 0$ we see that A's competitive ratio is unbounded.

So, let $x \in \mathbb{R}$ be such that A rejects any job of value at most x whilst asleep. Consider n jobs of identical value x and work w. For each job, the deadline is set such that $w = s_{\text{cr}}(d_j - r_j)$. The jobs are released in immediate succession, i.e., $r_j = d_{j-1}$. Algorithm A rejects all jobs, incurring cost nx. Let S denote the schedule that accepts all jobs and processes them at speed s_{cr}. The cost of S is given by $\operatorname{cost}(S) = \gamma + nw\frac{P_{\alpha,\beta}(s_{\text{cr}})}{s_{\text{cr}}}$. Thus, A's competitive ratio is at least

$$\frac{nx}{\gamma + nw\frac{P_{\alpha,\beta}(s_{\text{cr}})}{s_{\text{cr}}}} = \delta_{\max}\frac{1}{\frac{\gamma}{nw} + \frac{P_{\alpha,\beta}(s_{\text{cr}})}{s_{\text{cr}}}} \overset{n\to\infty}{\longrightarrow} \delta_{\max}\frac{s_{\text{cr}}}{P_{\alpha,\beta}(s_{\text{cr}})}. \qquad \square$$

4 Algorithm and Analysis

In the following, we use A to refer to both our algorithm and the schedule it produces; which is meant should be clear from the context. As most algorithms in this area (see, e.g., [4, 5, 13, 15, 16]), A relies heavily on the good structural properties of OA and its wide applicability to variants of the original energy-oriented scheduling model of Yao et al. [18]. It essentially consists of two components, the *rejection policy* and the *scheduling policy*. The rejection policy decides which jobs to accept or reject, while the scheduling policy ensures that all accepted jobs are finished until their deadline. Our rejection policy is an extension of the one used by the algorithm PS in [13]. It ensures that we process only jobs that have a reasonable high value (value > planned energy investment) and that we do not awake from sleep for very cheap jobs. The scheduling policy controls the speed, the job assignment, and the current mode of the processor. It is a straightforward adaption of the algorithm used in [15]. However, its analysis proves to be more involved because we have to take into account its interaction with the rejection policy and that the job sets scheduled by the optimal algorithm and A may be quite different.

The following description assumes a continuous recomputation of the current OA schedule. A pseudocode version of the algorithm can be found in the full version of the paper [14]. It is straightforward to implement A such that the planned schedule is recomputed only when new jobs arrive.

Scheduling Policy. All accepted jobs are scheduled according to the earliest deadline first rule. At any time, the processor speed is computed based on the OA schedule. Use OA^t to denote the schedule produced by OA if given the remaining (accepted) work at time t and the power function $P_{\alpha,0}$. Let ρ_t denote the speed planned by OA^t at time t. A puts the processor either in *working*, *idling*, or *sleeping* mode. During working mode the processor speed is set to $\max(\rho_t, s_{\text{cr}})$ until there is no more remaining work. Then, speed is set to zero and the processor starts idling. When idling or sleeping, we switch to the working mode only when ρ_t becomes larger than s_{cr}. When the amount of energy spent in the current idle interval equals the transition energy γ (i.e., after time $\gamma/P_{\alpha,\beta}(0)$) the processor is suspended to sleep.

Rejection Policy. Let c_1 and c_2 be parameters to be determined later. Consider the arrival of a new job j at time r_j. Reject it immediately if $\delta_j < s_{\text{cr}}^{\alpha-1}/\alpha c_2^{\alpha-1}$.

Otherwise, define the *current idle cost* $x \in [0, \gamma]$ depending on the processor's state as follows: (i) zero if it is working, (ii) the length of the current idle interval times β if it is idle, and (iii) γ if it is asleep. If $v_j < c_1 x$, the job is rejected. Otherwise, compute the job's speed s_{OA} which would be assigned by OA^{r_j} if it were accepted. Reject the job if $s_{\mathrm{OA}} > c_2 s_{j,\mathrm{p}}$, accept otherwise.

4.1 Bounding the Different Portions of the Cost

In the following, let O denote an optimal schedule. Remember that $\mathrm{cost}(A) = E^A_{\mathrm{sleep}} + E^A_{\mathrm{idle}} + E^A_{\mathrm{work}} + V^A_{\mathrm{rej}}$. We bound each of the three terms $E^A_{\mathrm{sleep}} + E^A_{\mathrm{idle}}$, E^A_{work}, and V^A_{rej} separately in Lemma 1, Lemma 2, and Lemma 3, respectively. Eventually, Section 4.2 combines these bounds and yields our main result: a nearly tight competitive factor depending on the maximum value density of the problem instance.

Lemma 1 (Sleep and Idle Energy). $E^A_{sleep} + E^A_{idle} \leq 6E^O_{sleep} + 2E^O_{sys} + \frac{4}{c_1} V^O_{rej}$

Proof. Let us first consider E^A_{idle}. Partition the set of idle intervals under schedule A into three disjoint subsets $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$ as follows:

– $\mathcal{I}_1$ contains idle intervals not intersecting any sleep interval of O. By definition, the total length of idle intervals from $\mathcal{I}_1$ is bounded by the time O is awake. Thus, the total cost of $\mathcal{I}_1$ is at most E^O_{sys}.
– For each sleep interval I of O, $\mathcal{I}_2$ contains any idle interval X that is not the the last idle interval having a nonempty intersection with I and that is completely contained within I (note that the former requirement is redundant if the last intersecting idle interval is not completely contained in I). Consider any $X \in \mathcal{I}_2$ intersecting I and let j denote the first job processed by A after X. It is easy to see that we must have $[r_j, d_j) \subseteq I$. Thus, O has rejected j. But since A accepted j, we must have $v_j \geq c_1 |X| \beta$. This implies that the total cost of $\mathcal{I}_2$ cannot exceed V^O_{rej}/c_1.
– $\mathcal{I}_3$ contains all remaining idle intervals. By definition, the first sleep interval of O can intersect at most one such idle interval, while the remaining sleep intervals of O can be intersected by at most two such idle intervals. Thus, if m denotes the number of sleep intervals under schedule O, we get $|\mathcal{I}_3| \leq 2m - 1$. Our sleeping strategy ensures that the cost of each single idle interval is at most γ. Using this and the definition of sleeping energy, the total cost of $\mathcal{I}_3$ is upper bounded by $(2m-1)\gamma = 2E^O_{\mathrm{sleep}} + \gamma$.

Together, we get $E^A_{\mathrm{idle}} \leq E^O_{\mathrm{sys}} + V^O_{\mathrm{rej}}/c_1 + 2E^O_{\mathrm{sleep}} + \gamma$. Moreover, without loss of generality we can bound γ by $V^O_{\mathrm{rej}}/c_1 + E^O_{\mathrm{sleep}}$: if not both A and O reject all incoming jobs (in which case A would be optimal), O will either accept at least one job and thus wake up ($\gamma \leq E^O_{\mathrm{sleep}}$) or reject the first job A accepts ($\gamma \leq V^O_{\mathrm{rej}}/c_1$). This yields $E^A_{\mathrm{idle}} \leq E^O_{\mathrm{sys}} + 2V^O_{\mathrm{rej}}/c_1 + 3E^O_{\mathrm{sleep}}$. For E^A_{sleep}, note that any but the first of A's sleep intervals is preceded by an idle interval of length $\gamma/P_{\alpha,\beta}(0)$. Each such idle interval has cost γ, so we get $E^A_{\mathrm{sleep}} \leq E^A_{\mathrm{idle}}$. The lemma's statement follows by combining the bounds for E^A_{idle} and E^A_{sleep}. □

Lemma 2 (Working Energy). $E^A_{work} \leq \alpha^\alpha E^O_{work} + c_2^{\alpha-1}\alpha^2 V^O_{rej}$

The proof of Lemma 2 is based on the standard amortized local competitiveness argument, first used by Bansal et al. [7]. Although technically quite similar to the typical argumentation, our proof must carefully consider the more complicated rejection policy (compared to [13]), while simultaneously handle the different processor states. For space reasons, the details concerning the necessary adaptions reside in the full version of the paper [14]. However, for completeness' sake, we outline the basic construction and the general proof idea.

Given a schedule S, let $E^S_{\text{work}}(t)$ denote the working energy spent until time t and $V^S_{\text{rej}}(t)$ the discarded value until time t. We show that at any time $t \in \mathbb{R}_{\geq 0}$ the *amortized energy inequality*

$$E^A_{\text{work}}(t) + \Phi(t) \leq \alpha^\alpha E^O_{\text{work}}(t) + c_2^{\alpha-1}\alpha^2 V^O_{\text{rej}}(t) \tag{2}$$

holds. Here, Φ is a potential function to be defined in a suitable way. It is constructed such that the following conditions hold:

(i) *Boundary Condition:* At the beginning and end we have $\Phi(t) = 0$.
(ii) *Running Condition:* At any time t when no job arrives we have

$$\frac{\mathrm{d}E^A_{\text{work}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} \leq \alpha^\alpha \frac{\mathrm{d}E^O_{\text{work}}(t)}{\mathrm{d}t} + c_2^{\alpha-1}\alpha^2 \frac{\mathrm{d}V^O_{\text{rej}}(t)}{\mathrm{d}t}. \tag{3}$$

(iii) *Arrival Condition:* At any time t when a job arrives we have

$$\Delta E^A_{\text{work}}(t) + \Delta\Phi(t) \leq \alpha^\alpha \Delta E^O_{\text{work}}(t) + c_2^{\alpha-1}\alpha^2 \Delta V^O_{\text{rej}}(t). \tag{4}$$

The Δ-terms denote the corresponding change caused by the job arrival.

Once these are proven, amortized energy inequality follows by induction: It obviously holds for $t = 0$, and Conditions (ii) and (iii) ensure that it is never violated. Applying Condition (i) yields Lemma 2. The crucial part is to define a suitable potential function Φ. Our analysis combines aspects from both [13] and [15]. Different rejection decisions of our algorithm A and the optimal algorithm O require us to handle possibly different job sets in the analysis, while the sleep management calls for a careful handling of the processor's current state.

Construction of Φ. Consider an arbitrary time $t \in \mathbb{R}_{\geq 0}$. Let $w^A_t(t_1, t_2)$ denote the remaining work at time t accepted by schedule A with deadline in $(t_1, t_2]$. We call the expression $\frac{w^A_t(t_1,t_2)}{t_2 - t_1}$ the *density* of the interval $(t_1, t_2]$. Next, we define *critical intervals* $(\tau_{i-1}, \tau_i]$. For this purpose, set $\tau_0 := t$ and define τ_i iteratively to be the maximum time that maximizes the density $\rho_i := \frac{w^A_t(\tau_{i-1},\tau_i)}{\tau_i - \tau_{i-1}}$ of the interval $(\tau_{i-1}, \tau_i]$. We end at the first index l with $\rho_l \leq s_{\text{cr}}$ and set $\tau_l = \infty$ and $\rho_l = s_{\text{cr}}$. Note that $\rho_1 > \rho_2 > \ldots > \rho_l = s_{\text{cr}}$. Now, for a schedule S let $w^S_t(i)$ denote the remaining work at time t with deadline in the i-th critical interval $(\tau_{i-1}, \tau_i]$ accepted by schedule S. The potential function is defined as $\Phi(t) := \alpha \sum_{i=1}^{l} \rho_i^{\alpha-1} \left(w^A_t(i) - \alpha w^O_t(i)\right)$. It quantifies how far A is ahead or behind in terms of energy. The densities ρ_i essentially correspond to OA's speed levels, but are adjusted to A's usage of OA. Note that whenever A is in working mode its speed equals $\rho_1 \geq s_{\text{cr}}$.

It remains to prove the boundary, running, and arrival conditions. The boundary condition is trivially true as both A and O have no remaining work at the beginning and end. Details concerning the running and arrival conditions can be found in the full version of the paper [14].

Bounding the Rejected Value. In the following we bound the total value V^A_{rej} of jobs rejected by A. The general idea is similar to the one by Chan et al. [13]. However, in contrast to the simpler model without sleep states, we must handle small-valued jobs of high density explicitly (cf. Section 3). Moreover, the sleeping policy introduces an additional difficulty: our algorithm does not preserve all structural properties of an OA schedule (cf. Fact 1). This prohibits a direct mapping between the energy consumption of algorithm A and of the intermediate OA schedules during a *fixed time interval*, as used in the corresponding proof in [13]. Indeed, the actual energy used by A during a fixed time interval may decrease compared to the energy planned by the intermediate OA schedule, as A may decide to raise the speed to s_{cr} at certain points in the schedule. Thus, to bound the value of a job rejected by A but processed by the optimal algorithm for a relatively long time, we have to consider the energy usage for the workload OA planned for that time (instead of the actual energy usage for the workload A processed during that time, which might be quite different).

Lemma 3 (Rejected Value). *Let δ_{max} be the maximum value density of jobs of value less than $c_1\gamma$ and consider an arbitrary parameter $b \geq {}^1/_{c_2}$. Then, A's rejected value is at most*

$$V^A_{rej} \leq \max\left(\delta_{max}\frac{s_{cr}}{P_{\alpha,\beta}(s_{cr})}, b^{\alpha-1}\right) E^O_{work} + \frac{b^{\alpha-1}}{(c_2 b-1)^\alpha} E^A_{work} + V^O_{rej}.$$

Proof. Partition the jobs rejected by A into two disjoint subsets J_1 (jobs rejected by both A and O) and J_2 (jobs rejected by A only). The total value of jobs in J_1 is at most V^O_{rej}. Thus, it suffices to show that the total value of J_2 is bounded by

$$\max\left(\delta_{\max}\tfrac{s_{\text{cr}}}{P_{\alpha,\beta}(s_{\text{cr}})}, b^{\alpha-1}\right) E^O_{\text{work}} + \tfrac{b^{\alpha-1}}{(c_2 b-1)^\alpha} E^A_{\text{work}}.$$

To this end, let $j \in J_2$. Remember that, because of the convexity of the power function, O can be assumed to process j at a constant speed s_O. Otherwise processing j at its average speed could only improve the schedule. Let us distinguish three cases, depending on the reason for which A rejected j:

Case 1: j got rejected because of $\delta_j < \frac{s_{\text{cr}}^{\alpha-1}}{\alpha c_2^{\alpha-1}}$.

Let $E^O_{\text{work}}(j)$ denote the working energy invested by O into job j. Using the rejection condition we can compute

$$E^O_{\text{work}}(j) = \tfrac{P_{\alpha,\beta}(s_O)}{s_O} w_j \geq \tfrac{P_{\alpha,\beta}(s_{\text{cr}})}{s_{\text{cr}}} w_j \geq s_{\text{cr}}^{\alpha-1} w_j > \alpha c_2^{\alpha-1} v_j.$$

Together with $b \geq {}^1/_{c_2}$ we get $v_j < b^{\alpha-1} E^O_{\text{work}}(j)$.

Case 2: j got rejected because of $v_j < c_1 x$

As in the algorithm description, let $x \in [0, \gamma]$ denote the current idle cost at time r_j. Since j's value is less than $c_1 x \leq c_1 \gamma$, we have $\delta_j \leq \delta_{\max}$. We get

$$E^O_{\text{work}}(j) = \frac{P_{\alpha,\beta}(s_O)}{s_O} w_j = \frac{P_{\alpha,\beta}(s_O)}{s_O} \frac{v_j}{\delta_j} \geq \frac{P_{\alpha,\beta}(s_{\text{cr}})}{s_{\text{cr}}} \frac{v_j}{\delta_{\max}},$$

which eventually yields $v_j \leq \delta_{\max} \frac{s_{\text{cr}}}{P_{\alpha,\beta}(s_{\text{cr}})} E^O_{\text{work}}(j)$.

Case 3: j got rejected because of $s_{\text{OA}} > c_2 s_{j,\text{p}}$

Here, s_{OA} denotes the speed OA^{r_j} would assign to j if it were accepted. We use $\text{OA}^{r_j}_{-}$ to refer to the OA schedule at time r_j without j. Let $b_j := {}^{s_{j,\text{p}}}/_{s_O}$. We bound v_j in different ways, depending on b_j. If b_j is small (i.e., $b_j \leq b$) we use

$$E^O_{\text{work}}(j) \geq \frac{P_{\alpha,0}(s_O)}{s_O} w_j = \frac{P_{\alpha,0}({}^{s_{j,\text{p}}}/_{b_j})}{{}^{s_{j,\text{p}}}/_{b_j}} w_j = \frac{s_{j,\text{p}}^{\alpha-1}}{b_j^{\alpha-1}} w_j = \frac{v_j}{b_j^{\alpha-1}}.$$

That is, we have $v_j \leq b_j^{\alpha-1} E^O_{\text{work}}(j)$. Otherwise, if b_j is relatively large, v_j is bounded by E^A_{work}. Let I denote the period of time when O processes j at constant speed s_O and let W denote the work processed by $\text{OA}^{r_j}_{-}$ during this time. Since $I \subseteq [r_j, d_j)$, Fact 1(b) implies that OA^{r_j}'s speed during I is at least $s_{\text{OA}} > c_2 s_{j,\text{p}}$. Thus, the total amount of work processed by OA^{r_j} during I is larger than $c_2 s_{j,\text{p}} |I|$. But then, by applying Fact 1(c), we see that W must be larger than $c_2 s_{j,\text{p}} |I| - w_j$. Now, W is a subset of the work processed by A. Moreover, Fact 1(d) and the definition of algorithm A ensure that the speeds used for this work in schedule A cannot be smaller than the ones used in $\text{OA}^{r_j}_{-}$. Especially, the average speed $s_\varnothing$ used for this work in schedule A is at least ${}^{W}/_{|I|}$ (the average speed used by $\text{OA}^{r_j}_{-}$ for this work). Let $E^A_{\text{work}}(W)$ denote the energy invested by schedule A into the work W. Then, by exploiting the convexity of the power function, we get

$$\begin{aligned} E^A_{\text{work}}(W) &\geq \frac{P_{\alpha,\beta}(s_\varnothing)}{s_\varnothing} W \geq \frac{P_{\alpha,0}(s_\varnothing)}{s_\varnothing} W = {s_\varnothing}^{\alpha-1} W \geq \frac{W^{\alpha-1}}{|I|^{\alpha-1}} W = |I| \frac{W^\alpha}{|I|^\alpha} \\ &> |I| (c_2 s_{j,\text{p}} - s_O)^\alpha = \frac{w_j}{s_O} s_O^\alpha (c_2 b_j - 1)^\alpha = \frac{(c_2 b_j - 1)^\alpha}{b_j^{\alpha-1}} v_j. \end{aligned}$$

That is, we have $v_j < \frac{b_j^{\alpha-1}}{(c_2 b_j - 1)^\alpha} E^A_{\text{work}}(W)$. Now, let us specify how to choose from these two bounds:

– If $b_j \leq b$, we apply the first bound: $v_j = b_j^{\alpha-1} E^O_{\text{work}}(j) \leq b^{\alpha-1} E^O_{\text{work}}(j)$.
– Otherwise we have $b_j > b \geq {}^1/_{c_2}$. Note that for $x > {}^1/_c$ the function $f(x) = \frac{x^{\alpha-1}}{(cx-1)^\alpha}$ decreases. Thus, we get $v_j < \frac{b^{\alpha-1}}{(c_2 b - 1)^\alpha} E^A_{\text{work}}(W)$.

By combining these cases we get

$$v_j \leq \max\left(\delta_{\max} \frac{s_{\text{cr}}}{P_{\alpha,\beta}(s_{\text{cr}})}, b^{\alpha-1}\right) E^O_{\text{work}}(j) + \frac{b^{\alpha-1}}{(c_2 b - 1)^\alpha} E^A_{\text{work}}(W).$$

Note that both energies referred to, $E^O_{\text{work}}(j)$ as well as $E^A_{\text{work}}(W)$, are mutually different for different jobs j. Thus, we can combine these inequalities for all jobs $j \in J_2$ to get the desired result. $\square$

4.2 Putting It All Together

The following theorem combines the results of Lemma 1, Lemma 2, and Lemma 3. Its proof can be found in the full version of the paper [14].

Theorem 2. *Let $\alpha \geq 2$ and let δ_{max} be the maximum value density of jobs of value less than $c_1\gamma$. Moreover, define $\eta := \max\bigl(\delta_{max}\frac{s_{cr}}{P_{\alpha,\beta}(s_{cr})}, b^{\alpha-1}\bigr)$ and $\mu := \frac{b^{\alpha-1}}{(c_2 b-1)^{\alpha}}$ for a parameter $b \geq 1/c_2$. Then, A's competitive factor is at most*

$$\max\left(c_2^{\alpha-1}\alpha^2, \alpha^{\alpha}\right)(1+\mu) + \max\left(2+\eta, 1+4/c_1\right).$$

By a careful choice of parameters we get a constant competitive ratio if restricting the value density of small-valued jobs accordingly. So, let $\alpha \geq 2$ and set $c_2 = \alpha^{\frac{\alpha-2}{\alpha-1}}$, $b = \frac{\alpha+1}{c_2}$, and $c_1 = \frac{4}{1+b^{\alpha-1}} \leq 1$. With Theorem 2 this yields

Corollary 1. *Algorithm A is $\alpha^{\alpha} + 2e\alpha + \delta_{max}\frac{s_{cr}}{P_{\alpha,\beta}(s_{cr})}$-competitive.*

Corollary 2. *Algorithm A is $\alpha^{\alpha} + 2e\alpha$-competitive if we restrict it to instances of maximum value density $\delta_{max} := b^{\alpha-1}\frac{P_{\alpha,\beta}(s_{cr})}{s_{cr}}$. This competitive factor still holds if the restriction is only applied to jobs of value less than $\frac{4}{1+b^{\alpha-1}}\gamma$.*

Corollary 3. *If only considering instances for which the job values are at least $\frac{8}{2+3\alpha}\gamma \leq \gamma$, A's competitive factor is at most $\alpha^{\alpha} + 2e\alpha$.*

Note that the bound from Corollary 1 is nearly tight with respect to $\delta_{\max}$ and the lower bound from Theorem 1.

5 The Speed-Bounded Case

As stated earlier, our model can be considered as a generalization of [13]. It adds sleep states, leading to several structural difficulties which we solved in the previous section. A further, natural generalization to the model is to cap the speed at some maximum speed T. Algorithms based on OA often lend themselves to such bounded speed models. In many cases, a canonical adaptation – possibly mixed with a more involved job selection rule – leads to an algorithm for the speed bounded case with similar properties (see, e.g., [6, 12, 13, 15]). A notable property of the profit-oriented scheduling model of [13] is that limiting the maximum speed leads to a non-constant competitive factor. Instead, it becomes highly dependent on a job's penalty ratio defined as $\Gamma_j := s_{j,\mathrm{P}}/T$. They derive a lower bound of $\Omega\bigl(\max(e^{\alpha-1}/\alpha, \Gamma^{\alpha-2+1/\alpha})\bigr)$ where $\Gamma = \max \Gamma_j$. Since our model generalizes their model, this bound transfers immediately to our setting (for the case $\beta = \gamma = 0$). On the positive side we can adapt our algorithm, similar to [13], by additionally rejecting a job if its speed planned by OA is larger than T (cf. rejection condition in algorithm description, Section 4). Our main theorem from Section 4 becomes

Theorem 3. *Let $\alpha \geq 2$ and let δ_{max} be the maximum value density of jobs of value less than $c_1\gamma$. Moreover, define $\eta := \max\bigl(\delta_{max}\frac{s_{cr}}{P_{\alpha,\beta}(s_{cr})}, \Gamma^{\alpha-1}b^{\alpha-1}\bigr)$ and $\mu := \Gamma^{\alpha-1}\frac{b^{\alpha-1}}{(b-1)^{\alpha}}$ for $b \geq 1$. Then, A's competitive factor is at most*

$$\alpha^{\alpha}\left(1+\mu\right)+\max\left(2+\eta, 1+{}^{4}\!/_{c_1}\right).$$

Proof (sketch). Note that the results from Lemmas 1 and 2 remain valid without any changes, as an additional rejection rule does not influence the corresponding proofs. The only lemma affected by the changed algorithm is Lemma 3. In its proof, we have to consider an additional rejection case, namely that job j got rejected because of $s_{\mathrm{OA}} > T = \frac{1}{\Gamma_j}s_{j,\mathrm{p}}$. This can be handled completely analogously to Case 3 in the proof, using the factor $\frac{1}{\Gamma_j}$ instead of c_2. We get the bounds $v_j \leq b_j^{\alpha-1}E_{\mathrm{work}}^{O}(j)$ and $v_j < b_j^{\alpha-1}/({}^{b_j}\!/_{\Gamma_j}-1)^{\alpha}E_{\mathrm{work}}^{A}(W)$. If $b_j \leq \Gamma_j b$ this yields $v_j \leq \Gamma_j^{\alpha-1}b^{\alpha-1}E_{\mathrm{work}}^{O}(j)$. Otherwise, if $b_j > \Gamma_j b$, we have $v_j < \Gamma_j^{\alpha-1}\frac{b^{\alpha-1}}{(b-1)^{\alpha}}E_{\mathrm{work}}^{A}(W)$. The remaining argumentation is the same as in the proof of Theorem 2. □

For $b = \alpha + 1$ and the interesting case $\Gamma > 1$ we get a competitive factor of $\alpha^{\alpha}(1+2\Gamma^{\alpha-1}) + \delta_{\max}\frac{s_{\mathrm{cr}}}{P_{\alpha,\beta}(s_{\mathrm{cr}})}$. For job values of at most γ it is $\alpha^{\alpha}(1+2\Gamma^{\alpha-1})$.

6 Conclusion and Outlook

We examined a new model that combines modern energy conservation techniques with profitability. Our results show an inherent connection between the necessary and sufficient competitive ratio of rejection-oblivious algorithms and the maximum value density. A natural question is how far this connection applies to other, more involved algorithm classes. Can we find better strategies if allowed to reject jobs even after we invested some energy, or if taking former rejection decisions into account? Such more involved rejection policies have proven useful in other models [15, 17], and we conjecture that they would do so in our setting. Other interesting directions include models for multiple processors as well as general power functions. Pruhs and Stein [17] modeled job values and deadlines in a more general way, which seems especially interesting for our profit-oriented model.

References

[1] Albers, S.: Algorithms for Dynamic Speed Scaling. In: Proc. of the 28th International Symp. On Theoretical Aspects of Computer Science (STACS), Schloss Dagstuhl, pp. 1–11 (2011)
[2] Albers, S.: Energy-Effcient Algorithms. Comm. of the ACM 53(5), 86–96 (2010)
[3] Albers, S., Antoniadis, A.: Race to Idle: New Algorithms for Speed Scaling with a Sleep State. In: Proceedings of the 23rd Symposium on Discrete Algorithms, SODA (2012)

[4] Albers, S., Antoniadis, A., Greiner, G.: On Multi-Processor Speed Scaling with Migration. In: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 279–288. ACM (2011)

[5] Bansal, N., Chan, H.-L., Pruhs, K., Katz, D.: Improved Bounds for Speed Scaling in Devices Obeying the Cube-Root Rule. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 144–155. Springer, Heidelberg (2009)

[6] Bansal, N., Chan, H.-L., Lam, T.-W., Lee, L.-K.: Scheduling for Speed Bounded Processors. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 409–420. Springer, Heidelberg (2008)

[7] Bansal, N., Kimbrel, T., Pruhs, K.: Speed Scaling to Manage Energy and Temperature. Journal of the ACM 54(1), 1–39 (2007)

[8] Baptiste, P.: Scheduling Unit Tasks to Minimize the Number of Idle Periods: A Polynomial Time Algorithm for Online Dynamic Power Management. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA), pp. 364–367. ACM (2006)

[9] Baptiste, P., Chrobak, M., Dürr, C.: Polynomial Time Algorithms for Minimum Energy Scheduling. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 136–150. Springer, Heidelberg (2007)

[10] Barroso, L.A., Hölzle, U.: The Case for Energy-Proportional Computing. Computer 40(12), 33–37 (2007)

[11] Baruah, S., Koren, G., Mishra, B., Raghunathan, A., Rosier, L., Shasha, D.: Online Scheduling in the Presence of Overload. In: Proc. of the 32nd Symp. on Foundations of Computer Science (FOCS), pp. 100–110 (1991)

[12] Chan, H.-L., Chan, W.-T., Lam, T.-W., Lee, L.-K., Mak, K.-S., Wong, P.W.H.: Energy Efficient Online Deadline Scheduling. In: Proceedings ofthe 18th Symposium on Discrete Algorithms (SODA), pp. 795–804. SIAM (2007)

[13] Chan, H.-L., Lam, T.-W., Li, R.: Tradeoff between Energy and Throughput for Online Deadline Scheduling. In: Jansen, K., Solis-Oba, R. (eds.) WAOA 2010. LNCS, vol. 6534, pp. 59–70. Springer, Heidelberg (2011)

[14] Cord-Landwehr, A., Kling, P., Mallmann-Trenn, F.: Slow Down & Sleep for Profit in Online Deadline Scheduling. arXiv:1209.2848 [cs.DS] (2012)

[15] Han, X., Lam, T.-W., Lee, L.-K., To, I.K.K., Wong, P.W.H.: Deadline Scheduling and Power Management for Speed Bounded Processors. Theoretical Computer Science 411(42), 3587–3600 (2010)

[16] Irani, S., Shukla, S., Gupta, R.: Algorithms for Power Savings. ACM Transactions on Algorithm 3(4) (2007)

[17] Pruhs, K., Stein, C.: How to Schedule When You Have to Buy Your Energy. In: Serna, M., Shaltiel, R., Jansen, K., Rolim, J. (eds.) APPROX 2010, LNCS, vol. 6302, pp. 352–365. Springer, Heidelberg (2010)

[18] Yao, F.F., Demers, A.J., Shenker, S.: A Scheduling Model for Reduced CPU Energy. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS), pp. 374–382 (1995)

FIFO Queueing Policies for Packets with Heterogeneous Processing

Kirill Kogan[1], Alejandro López-Ortiz[1], Sergey I. Nikolenko[2,3,⋆], Alexander V. Sirotkin[4,3,⋆], and Denis Tugaryov[3,⋆]

[1] School of Computer Science, University of Waterloo
{kkogan,alopez-o}@uwaterloo.ca
[2] Steklov Mathematical Institute, nab. r. Fontanka, 27, St. Petersburg, Russia
sergey@logic.pdmi.ras.ru
[3] St. Petersburg Academic University, ul. Khlopina, 8, korp. 3, St. Petersburg, Russia
[4] St. Petersburg Institute for Informatics and Automation of the RAS, 14 Line VO, 39, St. Petersburg, Russia
avs@iias.spb.su

Abstract. We consider the problem of managing a bounded size First-In-First-Out (FIFO) queue buffer, where each incoming unit-sized packet requires several rounds of processing before it can be transmitted out. Our objective is to maximize the total number of successfully transmitted packets. We consider both push-out (when the policy is permitted to drop already admitted packets) and non-push-out cases. In particular, we provide analytical guarantees for the throughput performance of our algorithms. We further conduct a comprehensive simulation study which experimentally validates the predicted theoretical behaviour.

Keywords: scheduling, buffer management, first-in-first-out queueing, switches, online algorithms, competitive analysis.

1 Introduction

This work is mostly motivated by buffer management problems within Network Processors (NPs) in a packet-switched network. Such NPs are responsible for complex packet processing tasks in modern high-speed routers, including, to name just a few, forwarding, classification, protocol conversion, and intrusion detection. Common NPs usually rely on multi-core architectures, where multiple cores perform various processing tasks required by the arriving traffic. Such architectures may be based on a pipeline of cores [34], a pool of identical cores [4,9,10], or a hybrid pool pipeline [12]. In response to operator demands, packet processing needs are becoming more heterogeneous, as NPs need to cope

⋆ Work of S.I. Nikolenko, A.V. Sirotkin, and D. Tugaryov was supported by the Russian Fund for Basic Research grant 12-01-00450-a, the Russian Presidential Grant Programme for Young Ph.D.'s, grant no. MK-6628.2012.1, for Leading Scientific Schools, grant no. NSh-3229.2012.1, and RFBR grants 11-01-12135-ofi-m-2011 and 11-01-00760-a.

with more complex tasks such as advanced VPN services, hierarchical classification for QoS, and many others. Unlike general purpose processors, modern NPs employ run-to-completion processing. Recent results in data path provisioning provide a possibility to have prior information about future required processing (for instance, this is possible in one of the modes of the OpenFlow protocol [24]). In this work, we consider a model that captures the characteristics of this architecture. The main concern in this setting is to maximize the throughput attainable by the NP, measured by the total number of packets successfully processed by the system.

In what follows, we adopt the terminology used to describe buffer management problems. We focus our attention on a general model where we are required to manage admission control and scheduling modules of a single bounded size queue that process packets in First-In-First-Out order. In this model, arriving traffic consists of unit-sized *packets*, and each packet has a *processing requirement* (in processor cycles). A packet is successfully *transmitted* once the scheduling module has scheduled the packet for processing for at least its required number of cycles. If a packet is dropped upon arrival or pushed out from the queue after being admitted due to admission control policy considerations (if push-out is allowed), then the packet is lost without gain to the algorithm's throughput.

1.1 Our Contributions

In this paper, we consider the problem of managing a FIFO queue buffer of size B, where each incoming unit-sized packet requires at most k rounds of processing before it can be transmitted out. Our objective is to maximize the total number of successfully transmitted packets. For online settings, we propose algorithms with provable performance guarantees. We consider both push-out (when the algorithm can drop a packet from the queue) and non-push-out cases. We show that the competitive ratio obtained by our algorithms depends on the maximum number of processing cycles required by a packet. However, none of our algorithms needs to know the maximum number of processing cycles in advance. We discuss the non-push-out case in Section 2 and show that the on-line greedy algorithm NPO is k-competitive, and that this bound is tight. For the push-out case, we consider a simple greedy algorithm PO that in the case of congestion pushes out the first packet with maximal required processing and the Lazy-Push-Out (LPO) algorithm. LPO works similar to PO in push-out decisions but changes the processing part: LPO does not transmit packets if there still exist packets in the queue with more than one required processing cycle (i.e., LPO waits until all packets have one cycle left and then sends them all out one by one). Intuitively, it seems that PO should outperform LPO since PO tends to empty its buffer faster but we demonstrate that these algorithms are not comparable in the worst case. Although we provide a lower bound of PO, the main result of this paper deals with the competitiveness of LPO. In particular, we demonstrate that LPO is at most $\left(\ln k + 3 + \frac{o(B)}{B}\right)$-competitive. In addition, we demonstrate several lower bounds on the competitiveness of both PO and LPO for different values of B and k. These results are presented in Section 3. The competitiveness result of LPO is interesting in itself but since "lazy" algorithms provide a well-defined accounting infrastructure we hope that a similar approach can be applied to other systems in similar settings. From an implementation point of view we can define a new on-line algorithm that will emulate the behaviour of LPO and will not delay the transmission of processed packets. In Section 4 we conduct a comprehensive simulation study to experimentally verify the performance of the proposed algorithms.

Due to space constraints, virtually no proofs appear in this paper; they can be found in the accompanying preprint [19].

1.2 Related Work

Keslassy et al. [14] were the first to consider buffer management and scheduling in the context of network processors with heterogeneous processing requirements for the arriving traffic. They study both SRPT (shortest remaining processing time) and FIFO (first-in-first-out) schedulers with recycles, in both push-out and non-push-out buffer management cases, where a packet is recycled after processing according to the priority policy (FIFO or SRPT). They showed competitive algorithms and worst-case lower bounds for such settings. Although they considered a different architecture (FIFO with recycles) than the one we consider in this paper, they provided only a lower bound for the push-out FIFO case, and it remains unknown if it can be attained.

Kogan et al. [20] considered priority-based buffer management and scheduling in both push-out and non-push-out settings for heterogeneous packet sizes. Specifically, they consider two priority queueing schemes: (i) Shortest Remaining Processing Time first (SRPT) and (ii) Longest Packet first (LP). They present competitive buffer management algorithms for these schemes and provide lower bounds on the performance of algorithms for such priority queues.

The work of Keslassy et al. [14] and Kogan et al. [20], as well as our current work, can be viewed as part of a larger research effort concentrated on studying competitive algorithms with buffer management for bounded buffers (see, e.g., a recent survey by Goldwasser [13] which provides an excellent overview of this field). This line of research, initiated in [18,22], has received tremendous attention in the past decade.

Various models have been proposed and studied, including, among others, QoS-oriented models where packets have weights [1,11,18,22] and models where packets have dependencies [15,23]. A related field that has received much attention in recent years focuses on various switch architectures and aims at designing competitive algorithms for such multi-queue scenarios; see, e.g., [3, 5, 6, 16, 17]. Some other works also provide experimental studies of these algorithms and further validate their performance [2].

There is a long history of OS scheduling for multithreaded processors which is relevant to our research. For instance, the SRPT algorithm has been studied extensively in such systems, and it is well known to be optimal with respect to the mean response [30]. Additional objectives, models, and algorithms have been studied extensively in this context [21,25,26]. For a comprehensive overview of competitive online scheduling for server systems, see a survey by Pruhs [28]. When comparing this body of research with our proposed framework, one should note that OS scheduling is mostly concerned with average response time, but we focus on estimation of the throughput. Furthermore, OS scheduling does not allow jobs to be dropped, which is an inherent aspect of our proposed model since we have a limited-size buffer.

The model considered in our work is also closely related to job-shop scheduling problems [8], most notably hybrid flow-shop scheduling [29] in scenarios where machines have bounded buffers and cannot drop or push out tasks.

1.3 Model Description

We consider a buffer with bounded capacity B that handles the arrival of a sequence of unit-sized packets. Each arriving packet p is branded with the number of required processing cycles $r(p) \in \{1, \ldots, k\}$. This number is known for every arriving packet; for a motivation of why such information may be available see [33]. Although the value of k will play a fundamental role in our analysis, we note that our algorithms need not know k in advance. In what follows, we adopt the terminology used in [20]. The queue performs two main tasks, namely *buffer management*, which handles admission control of newly arrived packets and push-out of currently stored packets, and *scheduling*, which decides which of the currently stored packets will be scheduled for processing. The scheduling will be determined by the FIFO order employed by the queue. Our framework assumes a multi-core environment, where we have C processors, and at most C packets may be chosen for processing at any given time. In the remainder of this paper we assume the system selects a single packet for processing at any given time (i.e., $C = 1$) in the theorems; however, variable C will resurface in the simulations (Section 4). This simple setting suffices to show both the difficulties of the model and our algorithmic scheme. We assume discrete slotted time, where each time slot t consists of three phases:

(i) *arrival*: new packets arrive, and the buffer management unit performs admission control and, possibly, push-out;
(ii) *assignment and processing*: a single packet is selected for processing by the scheduling module;
(iii) *transmission*: packets with zero required processing left are transmitted and leave the queue.

If a packet is dropped prior to being *transmitted* (i.e., while it still has a positive number of required processing cycles), it is lost. Note that a packet may be dropped either upon arrival or due to a push-out decision while it is stored in the buffer. A packet contributes one unit to the objective function only upon being successfully transmitted. The goal is to devise buffer management algorithms that maximize the overall throughput, i.e., the overall number of packets transmitted from the queue.

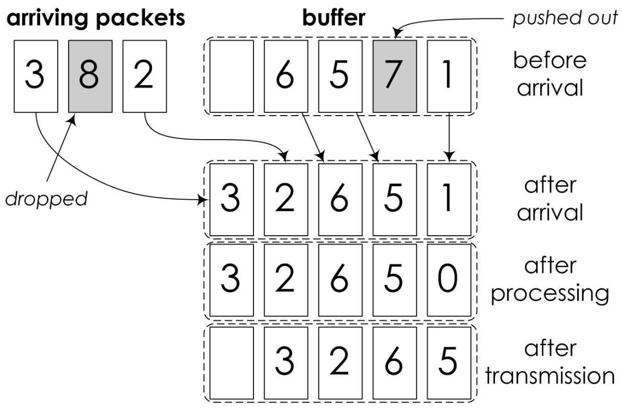


Fig. 1. Zoom in on a single time slot for a greedy push-out work-conserving algorithm

We define a *greedy* buffer management policy as a policy that accepts all arrivals if there is available buffer space in the queue. A policy is *work-conserving* if it always processes whenever it has admitted packets that require processing in the queue. We say that an arriving packet p *pushes out* a packet q that has already been accepted into the buffer iff q is dropped in order to free buffer space for p, and p is admitted to the buffer instead in FIFO order. A buffer management policy is called a *push-out* policy whenever it allows packets to push out currently stored packets. Figure 1 shows a sample time slot in our model (for greedy and push-out case). For an algorithm ALG and a time slot t, we denote the set of packets stored in ALG's buffer at time t by IB_t^{ALG}.

The number of *processing cycles* of a packet is key to our algorithms. Formally, for every time slot t and every packet p currently stored in the queue, its number of *residual processing cycles*, denoted $r_t(p)$, is defined to be the number of processing cycles it requires before it can be successfully transmitted.

Internet traffic is difficult to model and it does not seem to follow the traditional Poisson arrival model [27,32]. In this work we do not assume any specific traffic distribution and rather analyze our switch policies against arbitrary traffic using competitive analysis [7,31], which provides a uniform throughput guarantee for all traffic patterns. An online algorithm A is said to be *α-competitive* (for some $\alpha \geq 1$) if for any arrival sequence σ the number of packets successfully transmitted by A is at least $1/\alpha$ times the number of packets successfully transmitted by an optimal solution (denoted OPT) obtained by an offline clairvoyant algorithm. However, on the practical side we also provide simulations based on a Markov modulated Poisson process.

1.4 Proposed Algorithms

In this work, we study both push-out and non-push-out algorithms. The Non-Push-Out Algorithm (NPO) is a simple greedy work-conserving policy that accepts a packet if there is enough available buffer space. Already admitted packets

Algorithm 1. NPO(p): Buffer Management Policy

```
1: if buffer occupancy is less than $B$ then
2:     accept $p$
3: else
4:     drop $p$
5: end if
```

Algorithm 2. PO(p): Buffer Management Policy

```
1: if buffer occupancy is less than $B$ then
2:     accept $p$
3: else
4:     let $q$ be the first (from HOL) packet with maximal number of residual processing
5:     if $r_t(p) < r_t(q)$ then
6:         drop $q$ and accept $p$ according to FIFO order
7:     end if
8: end if
```

are processed in First-In-First-Out order (head of line packet is always selected for processing). If during arrivals NPO's buffer is full then any arriving packet is dropped even if it has less processing required than a packet already admitted to NPO's buffer (see Algorithm 1).

Next we introduce two push-out algorithms. The Push-Out Algorithm (PO) is also greedy and work-conserving, but now, when an arriving packet p requires less processing cycles than at least one packet in its buffer, PO pushes out the first packet with the maximal number of processing cycles in its buffer and accepts p according to FIFO order (see Algorithm 2). For processing, PO always selects the first packet in the queue. The second algorithm is a new Lazy-Push-Out algorithm LPO that mimics the behaviour of PO with two important differences: (i) LPO does not transmit a head of line packet with a single processing cycle if its buffer contains at least one packet with more than one residual processing cycle, until the buffer contains only packets with a single residual processing cycle; (ii) once all packets in LPO's buffer (say there are m packets there) have a single processing cycle remaining, LPO transmits them over the next m processing cycles; observe that during this time, if an arriving packet p requires less processing than the first packet q with maximal number of processing cycles in LPO's buffer, p pushes out q (similarly to PO).

Intuitively, LPO is a weakened version of PO since PO tends to empty its buffer faster. Simulations also support this view (see Section 4). However, Theorem 1 shows that LPO and PO are incomparable in the worst case (see the proof in [19]).

Theorem 1. *(1) There exists a sequence of inputs on which* PO *processes* $\geq \frac{3}{2}$ *times more packets than* LPO*. (2) There exists a sequence of inputs on which* LPO *processes* $\geq \frac{5}{4}$ *times more packets than* PO*.*

LPO is an online push-out algorithm that obeys the FIFO ordering model, so its competitiveness is an interesting result by itself. But we believe this type of algorithms to be a rather promising direction for further study since they provide a well-defined accounting infrastructure that can be used for system analysis in different settings. From an implementation point of view we can define a new on-line algorithm that will emulate the behaviour of LPO but will not delay the transmission of processed packets. Observe that such an algorithm is not greedy. Although we will briefly discuss the competitiveness of an NPO policy and lower bounds for PO, in what follows NPO and PO will be mostly used as a reference for the simulation study.

2 Competitiveness of the Non-push-Out Policy

The following theorem provides a tight bound on the worst-case performance of NPO; its proof is given in [19].

Theorem 2. *(1) For a sufficiently long arrival sequence, the competitiveness of* NPO *is at least k. (2) For a sufficiently long arrival sequence, the competitiveness of* NPO *is at most k.*

As demonstrated by the above results, the simplicity of non-push-out greedy policies does have its price. In the following sections we explore the benefits of introducing push-out policies and provide an analysis of their performance.

3 Competitiveness of Push-Out Policies

In this section, we show lower bounds on the competitive ratio of PO and LPO algorithms and prove an upper bound for LPO.

3.1 Lower Bounds

In this part we consider lower bounds on the competitive ratio of PO and LPO for different values of k and B. Proofs of Theorems 3 and 4 are given in [19]

Theorem 3. *The competitive ratio of both* LPO *and* PO *is at least* $2\left(1-\frac{1}{B}\right)$ *for* $k \geq B$*. The competitive ratio for* $k < B$ *is at least* $\frac{2k}{k+1}$ *for* PO *and at least* $\frac{2k-1}{k}$ *for* LPO*.*

For large k (of the order $k \approx B^n$, $n > 1$), logarithmic lower bounds follow.

Theorem 4. *The competitive ratio of* PO *(*LPO*) is at least* $\lfloor \log_B k \rfloor + 1 - O(\frac{1}{B})$*.*

3.2 Upper Bound on the Competitive Ratio of LPO

We already know that the performance of LPO and PO is incomparable in the worst case (see Theorem [1]), and it remains an interesting open problem to show an upper bound on the competitive ratio of PO. In this section we provide the first known upper bound of LPO. Specifically, we show the proof outline for the following theorem (full proof can be found in [19]).

Theorem 5. LPO *is at most* $\left(\ln k + 3 + \frac{o(B)}{B}\right)$*-competitive.*

Recall that LPO does not transmit any packet until all packets in the buffer have exactly one processing cycle left. The definition of LPO allows for a well-defined accounting infrastructure. In particular, it helps us define an *iteration* during which we will count the number of packets transmitted by the optimal algorithm and compare it to the contents of LPO's buffer. The first iteration begins with the first arrival. An iteration ends when all packets in the LPO buffer have a single processing pass left. Each subsequent iteration starts after the transmission of all LPO packets from the previous iteration.

We assume that OPT never pushes out packets and it is work-conserving; without loss of generality, every optimal algorithm can be assumed to have these properties since the input sequence is available for it a priori. Further, we enhance OPT with two additional properties: (1) at the start of each iteration, OPT flushes out all packets remaining in its buffer from the previous iteration (for free, with extra gain to its throughput); (2) let t be the first time when LPO's buffer is congested during an iteration; OPT flushes out all packets that currently reside in its buffer at time $t-1$ (again, for free, with extra gain to its throughput). Clearly, the enhanced version of OPT is no worse than the optimal algorithm since both properties provide additional advantages to OPT versus the original optimal algorithm. In what follows, we will compare LPO with this enhanced version of OPT for the purposes of an upper bound.

To avoid ambiguity for the reference time, t should be interpreted as the arrival time of a single packet. If more than one packet arrive at the same time slot, this notation is considered for every packet independently, in the sequence in which they arrive (although they might share the same actual time slot).

Lemma 1. *Consider an iteration I that begins at time t' and ends at time t. The following statements hold: (1) during I, the buffer occupancy of* LPO *is at least the buffer occupancy of* OPT*; (2) between two subsequent iterations I and I',* OPT *transmits at most* $|\mathrm{IB}_t^{\mathrm{LPO}}|$ *packets; (3) if during a time interval $[t', t'']$, $t' \leq t'' \leq t$, there is no congestion then during $[t', t'']$* OPT *transmits at most* $|\mathrm{IB}_{t''}^{\mathrm{LPO}}|$ *packets.*

We denote by M_t the maximal number of residual processing cycles among all packets in LPO's buffer at time t; by W_t, the total residual work for all packets in LPO's buffer at time t.

Lemma 2. *For every packet accepted by* OPT *at time t and processed by* OPT *during the interval $[t_s, t_e]$, $t \leq t_s \leq t_e$, if $|\mathrm{IB}_{t-1}^{\mathrm{LPO}}| = B$ then $W_{t_e} \leq W_{t-1} - M_t$.*

Let t be the time of the first congestion during an iteration I that has ended at time t'. Observe that by definition, at time t, OPT flushes out all packets that were still in its buffer at time $t-1$. We denote by $f(B,W)$ the maximal number of packets that OPT can process during $[t,t']$, where $W=W_{t-1}$. The next lemma is crucial for the proof; it shows that OPT cannot have more than logarithmic (in k) gain over LPO at any iteration.

Lemma 3. *For every $\epsilon > 0$, $f(B,W) \leq \frac{B-1}{1-\epsilon}\ln\frac{W}{B} + o(B\ln\frac{W}{B})$.*

Proof (of Theorem 5). Consider an iteration I that begins at time t' and ends at time t.

1. LPO*'s buffer is not congested during I.* In this case, by Lemma 1(3) OPT cannot transmit more than $|\mathrm{IB}_t^{\mathrm{LPO}}|$ packets during I.
2. *During I,* LPO*'s buffer is first congested at time t'', $t' \leq t'' \leq t$.* If during I OPT transmits less than B packets then we are done. By Lemma 1(3), during $[t',t'']$ OPT can transmit at most B packets. Moreover, at most B packets are left in OPT buffer at time $t''-1$. By Lemma 3, during $[t'',t]$ LPO transmits at most $(\ln k + \frac{o(B)}{B})B$ packets (because $W \leq kB$), so the total amount over a congested iteration is at most $(\ln k + 2 + \frac{o(B)}{B})B$ packets.

Therefore, during an iteration OPT transmits at most $(\ln k + 2 + o(1))|\mathrm{IB}_t^{\mathrm{LPO}}|$ packets. Moreover, by Lemma 1(2), between two subsequent iterations OPT can transmit at most $|\mathrm{IB}_t^{\mathrm{LPO}}|$ additional packets. Thus, LPO is at most $\ln k + 3 + \frac{o(B)}{B}$-competitive. □

The bound shown in Theorem 5 is asymptotic. To cover small values of B, we show a weaker bound ($\log_2 k$ instead of $\ln k$) on inputs where LPO never pushes out packets that are currently being processed.

The following theorem shows an upper bound for this family of inputs; it also provides motivation for a new algorithm that does not push out packets that are currently being processed. This restriction is practical (if a packet is being processed, perhaps this means that it has left the queue and gone on, e.g., to CPU cache), and the analysis of such an algorithm is an interesting problem that we leave open.

Theorem 6. *For every $B > 0$ and $k > 0$, if* LPO *never pushes out packets that are currently being processed then* LPO *is at most $\left(\log_2 k + 3 + \frac{B-1}{B}\right)$-competitive.*

4 Simulation Study

In this section, we consider the proposed policies (both push-out and non-push-out) for FIFO buffers and conduct a simulation study in order to further explore and validate their performance. Namely, we compare the performance of NPO, PO, and LPO in different settings. It was shown in [14] that a push-out algorithm that processes packets with less required processing first is optimal, so in what follows we denote it by OPT*. Clearly, OPT in the FIFO queueing model does not outperform OPT*.

Our traffic is generated from 100 independent sources, each generated by an on-off Markov modulated Poisson process (MMPP) which we use to simulate bursty traffic. The choice of parameters is governed by average arrival load, which is determined by the product of the average packet arrival rate and the average number of processing cycles required by packets. In our simulations, each source has average packet rate of $\frac{1}{21}\lambda_{\rm on}$, where $\lambda_{\rm on}$ is the parameter governing traffic generation while in the on state. Each source also has a fixed required processing value for every emitted packet; these values are distributed evenly across $\{1, \ldots, k\}$ (k being the maximum amount of processing required by any packet) We conducted our simulations for $500,000$ time slots, and allowed various parameters to vary in each set of simulations, in order to better understand the effect each parameter has on system performance and verify our analytic results.

By performing simulations for the maximal number of required passes k in the range $[1, 40]$ and the underlying source intensity $\lambda_{\rm on}$ in the range $[0.005, 0.25]$, we evaluate the performance of our algorithms in settings ranging from underload to extreme overload, validating their performance in various traffic scenarios. Fig. 2 shows simulation results. The vertical axis always represents the ratio between the algorithm's performance and OPT* performance arrival sequence (so the black line corresponding to OPT* is always horizontal at 1). Throughout our simulation study, the standard deviation never exceeded 0.05 (deviation bars are omitted for readability). For every choice of parameters, we conducted 500,000 rounds (time slots) of simulation, four sets of simulations in total.

Variable Intensity and Variable Maximum Number of Required Processing Cycles. Both the first and second set of simulations amount to testing different policies under gradually increasing processing requirements. The first and second row of graphs on Fig. 2 show that OPT* keeps outperforming LPO and NPO more and more as k grows; however, the difference in processing order between OPT* and PO does not matter much. NPO results show that non-push-out policies cope very badly with overload scenarios, as expected.

Variable Buffer Size. In this set of simulations we evaluated the performance of our algorithms for variable values of B in the range $[1, 40]$. Throughout our simulations we again assumed a single core ($C = 1$) and evaluated different values of k. The third row on Fig. 2 presents our results. Unsurprisingly, the performance of all algorithms significantly improves as the buffer size increases; the difference between OPT* and two other push-out algorithms visibly reduces, but, of course, it would take a huge buffer for NPO to catch up (one would need to virtually remove the possibility of congestion).

Variable Number of Cores. In this set of simulations we evaluated the performance of our algorithms for variable values of C in the range $[1, 10]$. The bottom row of Fig. 2 presents our results; the performance of all algorithms, naturally, improves drastically as the number of cores increases. There is an interesting phenomenon here: push-out capability becomes less important since buffers are

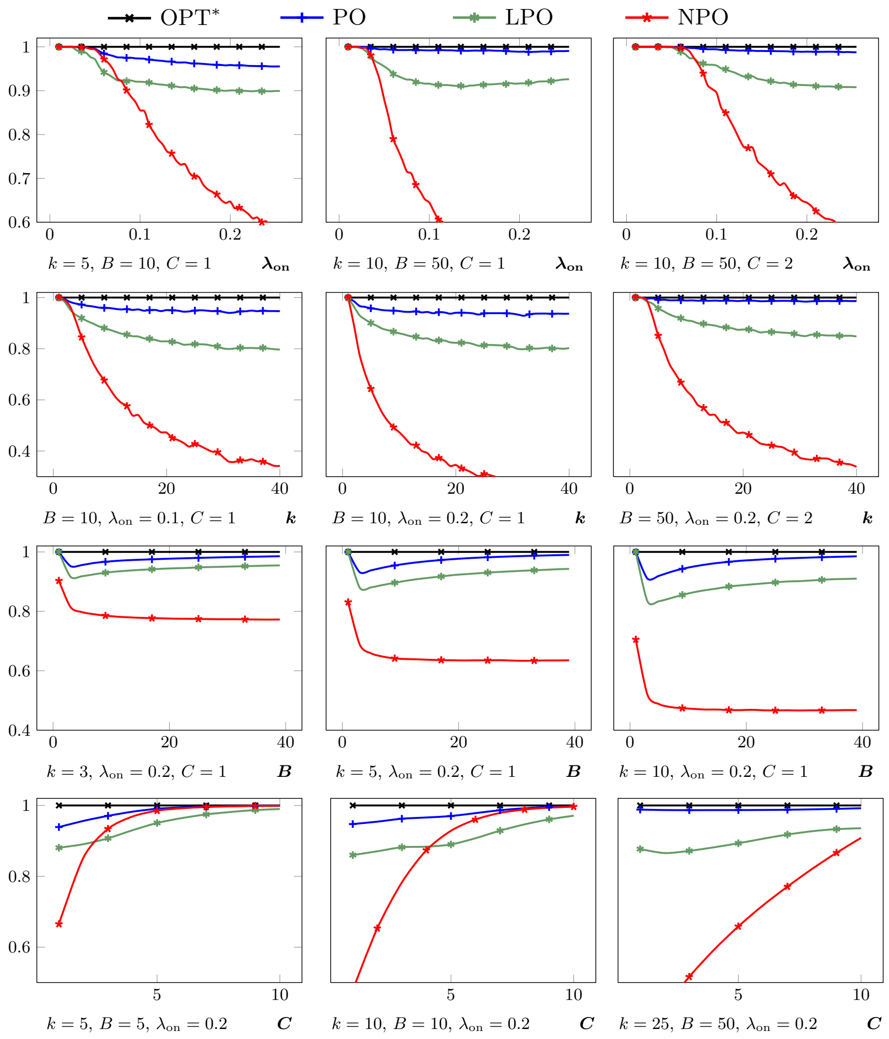


Fig. 2. Performance ratio of online algorithms versus optimal as a function of parameters: row 1, of λ_{on}; row 2, of k; row 3, of B; row 4, of C. The y-axis on all graphs shows the competitiveness vs. OPT*.

congested less often, but LPO keeps paying for its "laziness"; so as C grows, eventually NPO outperforms LPO. The increase in the number of cores essentially provides the network processor (NP) with a speedup proportional to the number of cores (assuming the average arrival rate remains constant).

5 Conclusion

Increasingly heterogeneous needs of NP traffic processing pose novel design challenges for NP architects. In this paper, we provide performance guarantees for NP buffer scheduling algorithms with FIFO queueing for packets with heterogeneous required processing. The objective is to maximize the number of transmitted packets under various settings such as push-out and non-push-out buffers. We validate our results by simulations. As future work, it will be interesting to show an upper bound for the PO algorithm and try to close the gaps between lower and upper bounds of the proposed on-line algorithms.

References

1. Aiello, W., Mansour, Y., Rajagopolan, S., Rosén, A.: Competitive queue policies for differentiated services. Journal of Algorithms 55(2), 113–141 (2005)
2. Albers, S., Jacobs, T.: An experimental study of new and known online packet buffering algorithms. Algorithmica 57(4), 725–746 (2010)
3. Albers, S., Schmidt, M.: On the performance of greedy algorithms in packet buffering. SIAM Journal on Computing 35(2), 278–304 (2005)
4. AMCC. nP7310 10 Gbps network processor, product brief (2010), `http://www.appliedmicro.com/MyAMCC/jsp/public/productDetail/product_detail.jsp?productID=nP7310`
5. Azar, Y., Litichevskey, A.: Maximizing throughput in multi-queue switches. Algorithmica 45(1), 69–90 (2006)
6. Azar, Y., Richter, Y.: An improved algorithm for CIOQ switches. ACM Transactions on Algorithms 2(2), 282–295 (2006)
7. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press (1998)
8. Brucker, P., Heitmann, S., Hurink, J., Nieberg, T.: Job-shop scheduling with limited capacity buffers. OR Spectrum 28(2), 151–176 (2006)
9. Cavium. OCTEON II CN68XX multi-core MIPS64 processors, product brief (2010), `http://www.caviumnetworks.com/OCTEON-II_CN68XX.html`
10. Cisco. The cisco QuantumFlow processor, product brief (2010), `http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_overview_c22-448936.html`
11. Englert, M., Westermann, M.: Lower and upper bounds on FIFO buffer management in QoS switches. Algorithmica 53(4), 523–548 (2009)
12. EZChip. NP-4 network processor, product brief (2010), `http://www.ezchip.com/p_np4.htm`
13. Goldwasser, M.: A survey of buffer management policies for packet switches. SIGACT News 41(1), 100–128 (2010)
14. Keslassy, I., Kogan, K., Scalosub, G., Segal, M.: Providing performance guarantees in multipass network processors. In: INFOCOM, pp. 3191–3199 (2011)

15. Kesselman, A., Patt-Shamir, B., Scalosub, G.: Competitive buffer management with packet dependencies. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, IPDPS (2009)
16. Kesselman, A., Kogan, K., Segal, M.: Improved Competitive Performance Bounds for CIOQ Switches. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 577–588. Springer, Heidelberg (2008)
17. Kesselman, A., Kogan, K., Segal, M.: Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing. Distributed Computing 23(3), 163–175 (2010)
18. Kesselman, A., Lotker, Z., Mansour, Y., Patt-Shamir, B., Schieber, B., Sviridenko, M.: Buffer overflow management in QoS switches. SIAM Journal on Computing 33(3), 563–583 (2004)
19. Kogan, K., López-Ortiz, A., Nikolenko, S.I., Sirotkin, A.V., Tugaryov, D.: FIFO queueing policies for packets with heterogeneous processing. arXiv:1204.5443 [cs.NI] (2012), http://arxiv.org/abs/1204.5443
20. Kogan, K., López-Ortiz, A., Scalosub, G., Segal, M.: Large profits or fast gains: A dilemma in maximizing throughput with applications to network processors (2012), http://arxiv.org/abs/1202.5755
21. Leonardi, S., Raz, D.: Approximating total flow time on parallel machines. In: STOC, pp. 110–119 (1997)
22. Mansour, Y., Patt-Shamir, B., Lapid, O.: Optimal smoothing schedules for real-time streams. Distributed Computing 17(1), 77–89 (2004)
23. Mansour, Y., Patt-Shamir, B., Rawitz, D.: Overflow management with multipart packets. In: INFOCOM, pp. 2606–2614 (2011)
24. McKeown, N., Parulkar, G., Shenker, S., Anderson, T., Peterson, L., Turner, J., Balakrishnan, H., Rexford, J.: OpenFlow switch specification (2011), http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf
25. Motwani, R., Phillips, S., Torng, E.: Non-clairvoyant scheduling. Theoretical Computer Science 130(1), 17–47 (1994)
26. Muthu Muthukrishnan, S., Rajaraman, R., Shaheen, A., Gehrke, J.E.: Online scheduling to minimize average stretch. SIAM Journal on Computing 34(2), 433–452 (2005)
27. Paxson, V., Floyd, S.: Wide area traffic: the failure of poisson modeling. IEEE/ACM Trans. Netw. 3(3), 226–244 (1995)
28. Pruhs, K.: Competitive online scheduling for server systems. SIGMETRICS Performance Evaluation Review 34(4), 52–58 (2007)
29. Ruiz, R., Vázquez-Rodrígue, J.A.: The hybrid flow shop scheduling problem. European Journal of Operational Research 205(1), 1–18 (2010)
30. Schrage, L.: A proof of the optimality of the shortest remaining processing time discipline. Operations Research 16, 687–690 (1968)
31. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Communications of the ACM 28(2), 202–208 (1985)
32. Veres, A., Boda, M.: The chaotic nature of TCP congestion control. In: INFOCOM, pp. 1715–1723 (2000)
33. Wolf, T., Pappu, P., Franklin, M.A.: Predictive scheduling of network processors. Computer Networks 41(5), 601–621 (2003)
34. Xelerated. X11 family of network processors, product brief (2010), http://www.xelerated.com/Uploads/Files/67.pdf

Author Index