PROCEEDINGS OF THE TENTH WORKSHOP ON ALGORITHM ENGINEERING AND EXPERIMENTS AND THE FIFTH WORKSHOP ON ANALYTIC ALGORITHMICS AND COMBINATORICS

SIAM PROCEEDINGS SERIES LIST

Computational Information Retrieval (2001), Michael Berry, editor

Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (2004), J. Ian Munro, editor

Applied Mathematics Entering the 21st Century: Invited Talks from the ICIAM 2003 Congress (2004), James M. Hill and Ross Moore, editors

Proceedings of the Fourth SIAM International Conference on Data Mining (2004), Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David Skillicorn, editors

Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (2005), Adam Buchsbaum, editor

Mathematics for Industry: Challenges and Frontiers. A Process View: Practice and Theory (2005), David R. Ferguson and Thomas J. Peters, editors

Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (2006), Cliff Stein, editor

Proceedings of the Sixth SIAM International Conference on Data Mining (2006), Joydeep Ghosh, Diane Lambert, David Skillicorn, and Jaideep Srivastava, editors

Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (2007), Hal Gabow, editor

Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithmics and Combinatorics (2007), David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick, editors

Proceedings of the Seventh SIAM International Conference on Data Mining (2007), Chid Apte, Bing Liu, Srinivasan Parthasarathy, and David Skillicorn, editors

Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (2008), Shang-Hua Teng, editor

Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments and the Fifth Workshop on Analytic Algorithmics and Combinatorics (2008), J. Ian Munro, Robert Sedgewick, Wojciech Szpankowski, and Dorothea Wagner, editors

PROCEEDINGS OF THE TENTH WORKSHOP ON ALGORITHM ENGINEERING AND EXPERIMENTS AND THE FIFTH WORKSHOP ON ANALYTIC ALGORITHMICS AND COMBINATORICS

Edited by J. Ian Munro, Robert Sedgewick,
Wojciech Szpankowski, and Dorothea Wagner

Society for Industrial and Applied Mathematics
Philadelphia

PROCEEDINGS OF THE TENTH WORKSHOP ON ALGORITHM ENGINEERING AND EXPERIMENTS AND THE FIFTH WORKSHOP ON ANALYTIC ALGORITHMICS AND COMBINATORICS

Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments, San Francisco, CA, January 19, 2008.

Proceedings of the Fifth Workshop on Analytic Algorithmics and Combinatorics, San Francisco, CA, January 19, 2008.

The Workshop on Algorithm Engineering and Experiments was supported by the ACM Special Interest Group on Algorithms and Computation Theory and the Society for Industrial and Applied Mathematics.

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2008923320
ISBN 978-0-898716-53-5

CONTENTS

CONTENTS

Workshop on Analytic Algorithmics and Combinatorics

ALENEX WORKSHOP PREFACE

The annual Workshop on Algorithm Engineering and Experiments (ALENEX) provides a forum for the presentation of original research in all aspects of algorithm engineering, including the implementation, tuning, and experimental evaluation of algorithms and data structures. ALENEX 2008, the tenth workshop in this series, was held in San Francisco, California on January 19, 2008. The workshop was sponsored by SIAM, the Society for Industrial and Applied Mathematics, and SIGACT, the ACM Special Interest Group on Algorithms and Computation Theory.

These proceedings contain 14 contributed papers presented at the workshop as well as the abstract of the invited talk by Rolf Möhring. The contributed papers were selected from a total of 40 submissions based on originality, technical contribution, and relevance. Considerable effort was devoted to the evaluation of the submissions with three reviews or more per paper. It is nonetheless expected that most of the papers in these proceedings will eventually appear in finished form in scientific journals.

The workshop took place in conjunction with the Fifth Workshop on Analytic Algorithmics and Combinatorics (ANALCO 2008), and papers from that workshop also appear in these proceedings. Both workshops are concerned with looking beyond the big-oh asymptotic analysis of algorithms to more precise measures of efficiency, albeit using very different approaches. The communities are distinct, but the size of the intersection is increasing as is the flow between the two sessions. We hope that others in the ALENEX community, not only those who attended the meeting, will find the ANALCO papers of interest.

We would like to express our gratitude to all the people who contributed to the success of the workshop. In particular, we would like thank the authors of submitted papers, the ALENEX Program Committee members, and the external reviewers. Special thanks go to Kirsten Wilden, for all of her valuable help in the many aspects of organizing this workshop, and to Sara Murphy, for coordinating the production of these proceedings.

J. Ian Munro and Dorothea Wagner

ALENEX 2008 External Reviewers

Reinhard Bauer
Michael Baur
Marc Benkert
Christian Blum
Ilaria Bordino
Ulrik Brandes
Jiangzhou Chen
Bojan Djordjevic
Frederic Dorn
John Eblen
Martin Ehmsen
Jeff Erickson
Arash Farzan
Mahmoud Fouz
Paolo Franciosa
Markus Geyer
Robert Görke
Meng He
Riko Jacob
Maleq Khan
Marcus Krug
Giuseppe Liotta
Hans van Maaren
Steffen Mecke
Damian Merrick
Matthias Mueller-Hannemann
Alantha Newman
Rajeev Raman
S.S. Ravi
Adi Rosen
Ignaz Rutter
Piotr Sankowski
Matthew Skala
Jan Vahrenhold
Anil Vullikanti
Thomas Wolle
Katharina Zweig

ANALCO WORKSHOP PREFACE

The aim of ANALCO is to provide a forum for original research in the analysis of algorithms and associated combinatorial structures. The papers study properties of fundamental combinatorial structures that arise in practical computational applications (such as trees, permutations, strings, tries, and graphs) and address the precise analysis of algorithms for processing such structures, including average-case analysis; analysis of moments, extrema, and distributions; and probabilistic analysis of randomized algorithms. Some of the papers present significant new information about classic algorithms; others present analyses of new algorithms that present unique analytic challenges, or address tools and techniques for the analysis of algorithms and combinatorial structures, both mathematical and computational.

The papers in these proceedings were presented in San Francisco on January 19, 2008, at the Fifth Workshop on Analytic Algorithmics and Combinatorics (ANALCO'08). We selected 9 papers out of a total of 20 submissions. An invited lecture by Don Knuth on "Some Puzzling Problems" was the highlight of the workshop.

The workshop took place on the same day as the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX'08). The papers from that workshop are also published in this volume. Since researchers in both fields are approaching the problem of learning detailed information about the performance of particular algorithms, we expect that interesting synergies will develop. People in the ANALCO community are encouraged to look over the ALENEX papers for problems where the analysis of algorithms might play a role; people in the ALENEX community are encouraged to look over these ANALCO papers for problems where experimentation might play a role.

Robert Sedgewick and Wojciech Szpankowski

Workshop on Algorithm Engineering and Experiments

Compressed Inverted Indexes for In-Memory Search Engines

Frederik Transier* Peter Sanders†

Abstract

We present the algorithmic core of a full text data base that allows fast Boolean queries, phrase queries, and document reporting using less space than the input text. The system uses a carefully choreographed combination of classical data compression techniques and inverted index based search data structures. It outperforms suffix array based techniques for all the above operations for real world (natural language) texts.

1 Introduction

Searching in large text data bases has become a key application of computers. Traditionally (a part of) the index data structure is held in main memory while the texts themselves are held on disks. However, this limits the speed of text access. Therefore, with ever increasing RAM capacities, there is now considerable interest in data bases that keep everything in main memory. For example, the TREX engine of SAP stores large quantities of small texts (like memos or product descriptions) and requires rapid access to all the data. Such a search engine consists of a cluster of multi-core machines, where each processing core is assigned an about equal share of the data base. Since RAM is hundreds of times more expensive than disks, good data compression is very important for in-memory data bases. In recent years, the algorithms community has developed sophisticated data structures based on suffix arrays that provide asymptotically efficient search using very little space that can even be less than the text itself.

This paper studies a different approach to compressed in-memory text search engines based on inverted indexes. We show that with careful data compression we can use very little space. The system consists of several interacting parts that support Boolean queries, phrase queries, and document reporting. It turns out that the parts interact in a nontrivial way. For example, the index data structure introduced in [1] turns out to allow better data compression as a side effect and can be used to simplify the positional index used for phrase queries. Both indexes together are used for a fast document reporting functionality that does not need to store the original text. Since most previous papers focus only on a subset of these aspects, we believe that our results are of some interest both from the application perspective and for algorithm engineering.

We therefore compare our framework with publicly available implementations based on suffix arrays. Somewhat unexpectedly, our system is more space efficient than compressed suffix arrays. Its query performance is up to 3–4 orders of magnitude faster for Boolean queries, about 20 times faster for phrase queries (the application domain for which suffix arrays are designed), and about five times faster for document reporting.[1]

We give a more concrete definition of the key terms in Section 2 and related work in Section 3. Section 4 with our inverted index data structures constitutes the main algorithmic part of the paper. We call it compressed inverted index. Its modular design consists of many ingredients each of which is given by a well defined software module whose implementations can be exchanged. In Section 5 we explain how to implement all required operations using compressed suffix arrays. Section 6 gives an experimental evaluation using real world benchmark data and Section 7 summarizes the results and outlines possible future work.

2 Preliminaries

We consider the problem of querying for a subset of a large collection of *normalized* text documents. A normalized text document is a set of words or terms obtained from an arbitrary text source by smoothing out all so-called term separators, i.e. spaces, commas and so on. Thereby we do not distinguish between upper and lower-case characters. For simplicity, we map each term and each document to an integer value. So sometimes we refer to documents or terms as *document identifiers (IDs)* or *term IDs* respectively.

*SAP AG (NW EIM TREX), 69190 Walldorf, Germany and Universität Karlsruhe (TH), 76128 Karlsruhe, Germany - transier@ira.uka.de

†Universität Karlsruhe (TH), 76128 Karlsruhe, Germany - sanders@ira.uka.de

[1] Note that we do not make any claims about the performance of suffix arrays for other applications such as in bioinformatics where the data has very different structure and where we also need different operations.

A few types of queries are of particular importance: A Boolean *AND* query on a document collection asks for all documents that contain all elements of a given list of terms. Analogously, a Boolean *OR* query searches for all documents that contain any of the terms in the list. And finally, a *phrase* query asks for documents that contain a given sequence of terms.

In a real-world search engine we are obviously not interested in a list of document IDs as the result set. Rather, we expect the document texts itself. So we consider the reporting of result documents as an important functionality of text search engines as well.

The most widely used data structure for text search engines is the *inverted index*. In an inverted index, there is a *inverted list* of document IDs for each term ID showing in which document a term appears. In this simplest version of an inverted index, it can just answer Boolean queries. For phrase queries additional positional information is needed, i.e. for each term in a document a list of positions where it occurs. We call indexes that contain such information *positional indexes*.

Of course, there are many alternatives to the inverted index in the literature. Some of them have similar but somewhat different fields of application. So *suffix arrays* are mainly used for substring queries. They store all suffixes of a given text in alphabetical order. In this way, a substring pattern can quickly be found using binary search. And since a while ago, there are also compressed versions of suffix arrays.

Just as well, compression can be applied to inverted indexes. There exists a couple of standard compression techniques suitable for inverted lists. A straight forward approach is *bit-compression*. For each value in the list, it uses the number of bits needed to code the largest one among them. Thus, further compression can be achived by trying to reduce the maximum entry of a list. For example, a sorted list can be transformed into a list of differences between every two consecutive values. These *delta* values are smaller or at least equal to their origins and thus, they can probably be stored using fewer bits. The big drawback of those *linear* encoding schemes is that a list have to be unpacked from the front for accessing any value in it.

There are also variable bit length encodings. A very popular one is the *Golomb* encoding from [2]. It is often used on delta sequences. Golomb coding has a tunable parameter. A value is coded in two parts. The result of a division by the parameter and the remainder. The quotient is coded in unary followed by the remainder in truncated binary coding.

3 Related Work

Data structures for text search engines have been studied extensively in the past. Zobel and Moffat [3] give a good overview. Across the relevant literature, the inverted index structure has been proved to be the method of choice for fast Boolean querying. As a consequence, a lot of work has been done in order to improve the basic data structure of a single inverted list with regard to compression and simple operations such as intersections or unions (q.v. [4, 1]).

Traditionally, search engines store at least the major parts of their inverted index structures on hard disks. They use either storage managements that are similar to those of Zobel et al. [5] or the persistent object stores of databases (cp. [6]). Due to the growing amount of random access memory in computer systems, recent work have considered to purge disk-based structures from index design. Strohman and Croft [7] show how *top-k* Boolean query evaluation can be significantly improved in terms of throughput by holding impact-sorted inverted index structures in main memory. Besides strict in-memory and disk-based solutions there are some hybrid approaches. Luk and Lam [8] propose a main memory storage allocation scheme suitable for inverted lists of smaller size using a linked list data structure. The scheme uses variable sized nodes and therefore well suited for inverted lists that are subject of frequent updates. Typically, such approaches are used in combination with larger *main* indexes to hide the bottleneck of updating search optimized index structures (cp. [9]).

Unbeaten for Boolean queries, there is a rub in querying for phrases within inverted indexes. Those queries are slow as the involved position lists have to be merged. Several investigations have been made for reducing this bottleneck. A simple approach of Gutwin et al. [10] is to index phrases instead of terms. So-called *nextword indexes* were proposed by Bahle et al. [11]. For each term, they store a list of all successors together with their corresponding positions. Furthermore, a combined index approach of the previous ones was proposed by Williams et al. [12].

Suffix trees and suffix arrays were originally designed for substring search. Hence they can also be used for efficient phrase querying. Due to their high space consumption, they were first unsuitable for large text indexing. However, by now, various authors have worked on the compression or succinct representations of suffix arrays. For an overview see Navarro and Mäkinen [13] or the Pizza&Chili Website[2]. According to González and Navarro [14], they need about 0.3 to 1.5 times of

[2] http://pizzachili.di.unipi.it/

the size of the indexed text — depending on the desired space-time trade-off. Moreover, they are *self-indexing*, i.e. they can reconstruct the indexed text efficiently. In comparison to that, an inverted index needs about 0.2 to 1.0 times of the text size ([15]) depending on compression, speed and their support of phrase searches. Unfortunately, inverted indexes can not reconstruct the text, so they require this amount of space in addition to the (compressed) text size.

In recent work, Ferragina and Fischer [16] investigated in building suffix arrays on terms. They need just about 0.8 times of the space of character based suffix arrays being twice as fast during the construction.

Puglisi et al. [17] compared compressed suffix arrays and inverted indexes as approaches for substring search. They indexed q-grams, i.e. strings of length q, instead of words to allow full substring searches rather than just simple term searches within their inverted index. As their result, they found out that inverted indexes are faster in reporting the location of substring patterns when the number of their occurrences is high. For rare patterns they noticed that suffix arrays outperform inverted indexes. Bast et al. have observed that suffix arrays are slower for prefix search [18] and for phrase search [19].

4 Compressed Inverted Index

The core of our text index is a document-grained inverted index. For each term ID we store either a pointer to an inverted list or — if the term occurs in one single document only — just a document ID. We distinguish between two types of inverted lists. The first type is used for all terms that occur in less than $K = 128$ documents. In our implementation we use the delta Golomb encoding in order to compress these *small* lists in a simple but compact way. For the remaining terms, we use an extension to the two-level lookup data structure of [1]. It is organized as follows. Let $1..U$ be the range of the N document IDs to be stored in a list. Then the list can be split into k buckets spanning the ranges $u_i = (i-1)\frac{U}{k}..i\frac{U}{k}$ with $i = 1..k$. According to [1], the parameter k is thereby chosen as $\lceil \frac{N}{B} \rceil$ with $B = 8$, so that the buckets can be indexed by the $\lceil \log k \rceil$ most significant bits of their contents. The top-level of the data structure stores a pointer to each bucket in the bottom-level. For accessing the positional index, we also need a *rank* information, i.e. $\sum_{j=0}^{i} n_j$ for each bucket i whereas n_j denotes the size of bucket j. An interesting side effect is that using the rank information, we can guess the average delta within each bucket i as the quotient of the range delta $\frac{U}{k}$ and the rank delta n_i. In this way, we can choose a suitable Golomb parameter for a *bucket-wise* delta encoding according to [15] as $b_i = \ln(2)\frac{U}{kn_i}$.[3] Obviously, the initial delta value of the i-th bucket is given by $i^{\lceil \log \frac{U}{k} \rceil}$. In contrast to the linear encoding of the buckets, all top-level values are required to be randomly accessible. The top-level is later used to truncate the considered data (cp. algorithm lookup in [1]) and hence, to reduce the querying time. We call this two-level data structure a *large* inverted list.

The distinction between single values, small and large lists offers two advantages. On the one hand, we can store lists smaller than K in a very compact way. On the other hand, this helps to avoid scanning huge lists.

LEMMA 4.1. *Assuming the classical Zipfean distribution of M terms over N documents, our index contains at least $M - \frac{N}{\ln(M)}$ single value entries.*

Proof. According to [20], the classical Zipfean law says that the rth most frequent term of a dictionary of M words occurs $\frac{N}{rH_M}$ times in a collection of N documents, where H_M is the harmonic number of M. Claiming an occurrence frequency of 1, we get $r_1 = \frac{N}{H_M}$ as the order of the first term that occurs just once. As the total number of terms in our index is M, we have $M - \frac{N}{H_M}$ unique terms. Due to [21] holds $H_n - \ln(n) - \gamma < \frac{1}{2n}$, where γ is the Euler-Mascheroni constant. Therefore, $M - \frac{N}{H_M} > M - N(\frac{1}{2M} + \ln(M) + \gamma)^{-1} > M - \frac{N}{\ln(M)}$.□

LEMMA 4.2. *Our document-grained index contains at least $(1 - \frac{1}{K})\frac{N}{\ln(M)}$ small lists.*

Proof. As in proof of Lemma 4.1, we can obtain the order of the first term that's occurrence frequency is equal to K using the Zipfean law, i.e. $r_K = \frac{N}{KH_M}$. By subtracting this rank from that one of the unique terms r_1, we get the number of terms that occur more than once but less or equal than K times. Thus, $r_1 - r_K = (1 - \frac{1}{K})\frac{N}{H_M}$ and due to [21] $r_1 - r_k > N(1 - \frac{1}{K})(\frac{1}{2M} + \ln(M) + \gamma)^{-1} > (1 - \frac{1}{K})\frac{N}{\ln(M)}$. □

As an extension of our document-grained inverted index we store the positional information of each term in a separate structure. Again, we distinguish the way of storing the data depending on the list lengths: A vector indexed by the term IDs provides the basis. For all terms that occur only once within the index, we store the single position itself. For the remaining terms we store pointers to lists of different types. We use simple linear encoded lists for terms that occur more often but in just one single document. Terms that appear in multiple documents just once in each, are stored in a

[3] In our experiments, this saves about 35 % space compared to using a global Golomb parameter.

random accessible list, i.e. using bit-compression. And finally, for all the others, we use *indexed lists*. An indexed list is a two-level data structure whose top-level points to buckets in the bottom level containing an arbitrary number of values each. In our case, a bucket contains a list of positions of the term within a certain document. It is indexed from the top-level by the rank of that document in the document-grained index. Figure 1 shows how to retrieve term positions from indexed lists for a given term v and a list of document ID-rank pairs D.

```
Function positions(v, D)
    O := ⟨⟩                                  // output
    foreach (d, r) ∈ D do                   // doc ID - rank pairs
        i := t^v[r]                         // start of position values
        e := t^v[r + 1]                     // end of position values
        while i < e do                      // traverse bucket
            O := O ∪ (d, b^v[i])            // add to result
            i++
    return O
```

Figure 1: High level pseudocode for retrieving positional information.

In a similar way, we have access to the positional information coded in the bit-compressed lists. We can consider them as lists that consist of a top-level only. However, instead of storing pointers to buckets therein, we store the single position values directly. Defining a function *positions* for these lists is trivial. And for the terms that occur in just one document it is even easier.

The advantages of the separation between document-grained and positional information are obvious: For cases in which we do not need the functionality of a positional index, we can easily switch off this part without any influence on the document-grained data structures. Furthermore, Boolean queries are very cache efficient as the inverted lists are free of other information than the document IDs itself. And even phrase queries are supposed to be fast. Due to the fact that we work in main memory we can jump to the positional information *actually* needed at little cost.

As usual for inverted index based text search engines we use a dictionary that maps normalized terms to term IDs and vice versa. Since this part of a search engine is not our main focus, we use a simple uncompressed dictionary. It stores the normalized terms alphabetically ordered and uses binary search for retrieving IDs for a given term. The inverse mapping is done via a separate vector that is indexed by the term IDs and contains pointers to the respective terms. Surely, our dictionary has not its strengths in mapping terms to term IDs very fast, but as it holds uncompressed terms, it has certainly no harmful influence on our reconstruction process in Section 4.2.

Memory management turned out to be crucial for obtaining *actual* space consumption close to what one would expect from summing the sizes of the many ingredients. Most data is stored in blocks of 512 Kbytes each. Lists that are greater than such a block get their own contiguous memory snippet. Within the blocks, we use word aligned allocation of different objects. We tried byte-alignment but did not find the space saving worth the computational overhead.

4.1 Querying the Compressed Inverted Index

Boolean Queries. Inverted indexes are by design well suited to answer Boolean queries. The *AND* query corresponds to an intersection of a set of inverted lists. This problem has been studied extensively by various authors in the past (cp. [22, 4, 1]). For our index we use a simple binary merge algorithm for all small lists and algorithm *lookup* from [1] for large lists.

Phrase Queries. The phrase search algorithm on the index can be divided into four steps. First, we sort the phrase terms according to their frequency, i.e. their inverted list lengths. Of course, we have to bear in mind the term positions within the phrase. Then, we intersect the document-grained inverted lists of the two least frequent terms — using algorithm *lookup* if applicable — keeping track of the current rank for each list. With this information, we can retrieve the corresponding position lists using function *positions* of Figure 1. As a result, we have two lists of pairs consisting of a document and the position within the document. In a third step, we combine the two lists using a simple binary merge algorithm that normalizes the positions according to the query phrase on the fly. Finally, the further terms have to be incorporated into the result set. In increasing order of frequency, we repeat the following for each term: First, we intersect a terms inverted list with the current result set. Then, we retrieve the positions. And in the end, we merge them with the previous result set.

4.2 Document Reporting.

The query result of the inverted index is a list of document IDs which are associated with pointers to the original documents. Traditionally, the documents are archived in a separate storage location, i.e. in files on disk or on the network. They were retrieved from there when the search engine returns the results. However, since we want to exploit

associated advantages of main memory, our space is scarce. So instead of storing the documents separately, we use the information about the indexed documents we have already stored. In fact, we know the term ID - document ID mappings from the inverted lists, as well as the position lists for these ID pairs. So we could restore the sequence of normalized terms of a document by traversing the inverted lists, gathering the term IDs and placing their dictionary items using algorithm *positions*. However, reconstructing a term sequence this way would take too much time. Instead, we store a *bag of words* for each document in addition to our existing data structures. A bag of words is a set of all the term IDs occurring in a document. We store them sorted in increasing order of IDs using delta Golomb encoding. Besides, we encode sequences of consecutive value IDs as a 0 followed by the size of the series. So we are able to build term sequences without traversing through all inverted lists, just by positioning all terms of the bag.

There is still a small step from the sequence of normalized terms towards the original document we have indexed before. For that reason, we store the changes made to the terms during the normalization process. In our dictionary, all terms are normalized to lower-case characters — the most frequent spelling of words in English texts. Any differences to this notation in the original document are recorded in a list of *term escapes*. An item in that list consists of a pair of a position where the variation in the sequence occurs and an escape value. The escape value indicates how the term has to be modified. A value of 0 means that the first character has to be capitalized and a value of 1 means that the whole term has to be spelled with capital letters. Each greater escape value points to a replacement term in a separate *escape dictionary*.

It remains to incorporate the term separators into the string of adjusted terms. We store the arrangement of separators and terms of each document as a (Golomb-coded) list of values: A 0-value followed by a number x indicates that there are x terms separated by spaces. A 1-value indicates a term (Recall that the normalized text already tells us *which* term). A value of 2 or larger encodes the ID of a separator. We use a simple binary merge algorithm to reconstuct a document from its term sequence and its separator arrangement.

5 Document Retrieval on Compressed Suffix Arrays

In order to qualify our phrase querying performance, we have implemented document retrieval algorithms on compressed suffix arrays. Of course, suffix arrays know neither terms nor term IDs, so we do not need a dictionary here. However, as we expect the same query results, we perform the same normalization process to obtain normalized documents, i.e. normalized document terms separated by spaces. We concatenate all documents separated by a special end of document character into single text from which we build the suffix array.

As the suffix array knows global character positions only, we need to store all document starting positions in a separate data structure. Again, we use a *lookup* list as described in section 4, in which the rank information correspond to the document IDs.

5.1 Querying Suffix Arrays

Phrase Queries. Phrase querying on suffix arrays is straight forward. Here, a phrase search is equal to a substring search of the normalized query phrase. Due to the special separator characters, it is impossible to get a result phrase that overlaps document bounds. As the result, the suffix array returns positions where the phrase occurrences start. They have to be remapped to document IDs using the lookup list described above.

Boolean Queries. For Boolean queries, we first locate the occurrences for each query term by a substring search in the suffix array. Afterwards, we map the result list of the least frequently occurring term to a list of document IDs. Then, we insert the list in a binary search tree (`std::map`) and incorporate the results of the next frequently occurring term by performing a tree lookup for each of its items. We put all hits in a second tree and swap them as soon as we have checked all of the items. This process is repeated until all term IDs are processed.

We also tried two other approaches for Boolean queries. The first one was to build a hash table from the shortest document ID list using `stdext::hash_map` and to check there for the IDs of the remaining lists. The second one was to sort all lists and to intersect them like the inverted lists. However, these two alternatives could not compete with the previous one. Anyway, the major parts of the querying time are spent during suffix array operations. We will see this in the experimental section.

5.2 Document reporting. Document reporting on suffix arrays is simple. They support this operation innately. However, the document bounds required for reporting have to be retrieved outside the suffix array in our lookup data structure.

Of course, as we have indexed the normalized text into the suffix array, it returns the normalized text as well. In order to get the original input document, we

have to take a similar process as we used while retrieving original documents from our inverted index. We did not implement this, so we will compare just the reconstruction of the normalized text in our experimental section.

6 Experiments

Table 1: Index properties (50 000 documents of WT2g)

	CII	CSA
dictionary	23.9	-
document index	32.3	-
positional index	126.3	-
bag of words	23.7	-
suffix array	-	230.8
doc bound list	-	0.1
sum [MB]	**206.1**	**230.9**
text delta	108.7	-
	314.8	-
input size (norm.)	412.8 (360.5)	- (360.5)
compression	0.76 (0.57)	- (0.64)
indexing time [min]	**5.6 (5.1)**	**- (9.3)**
peak mem usage [GB]	0.7	3.2

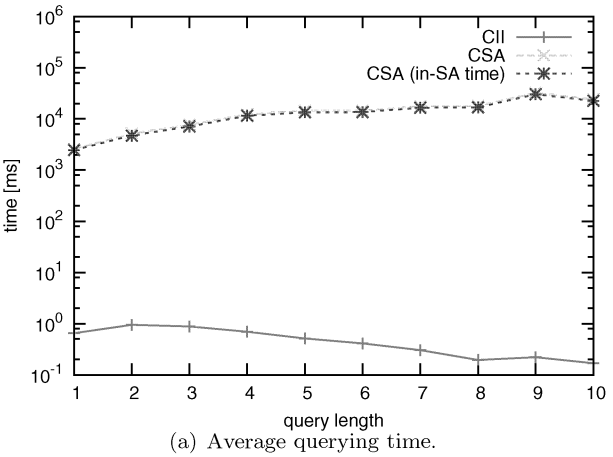
We have implemented all algorithms using C++. We used a framework wich allows us to switch easily between different approaches while preserving the environment for our measurements. Most tuning parameters in our implementation were set heuristically using back-of-the-envelope calculations and occasional experiments. We have not spent much effort in systematic tuning or extensive evaluation of tradeoffs since good space-time compromises are usually easy to obtain that are not very sensitive with respect to the values chosen.

The experiments were done on one core of an Intel Core 2 Duo E6600 processor clocked at 2.4 GHz with 4 GB main memory and 2 × 2048 Kbyte L2 cache, running OpenSuSE 10.2 (kernel 2.6.18). The program was compiled by the gnu C++ compiler 4.1.2 using optimization level -O3. Timing was done using PAPI 3.5.0[4].

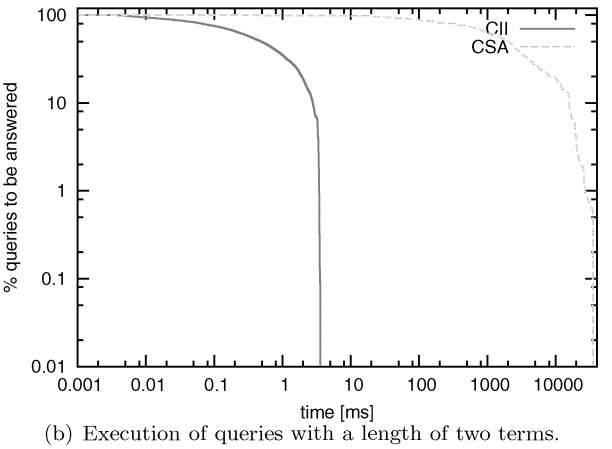
For our experiments we used the WT2g[5] text corpus, which is a standard benchmark for web search. Although WT2g is by now considered a 'small' input, it is right size for the amount of data assigned to *one processing core* of an *in-memory* text search engine based on a cluster of low cost machines. Using more than 1–2 GByte of data on one core would mean investing much more money into RAM than into processing power.

[4] http://icl.cs.utk.edu/papi/

[5] http://ir.dcs.gla.ac.uk/test_collections/access_to_data.html



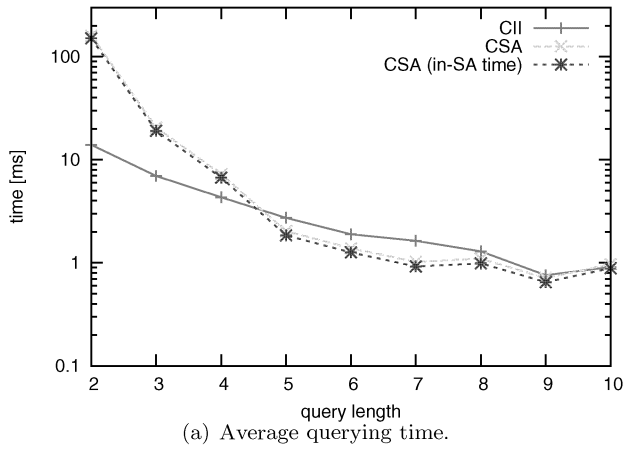
(a) Average querying time.



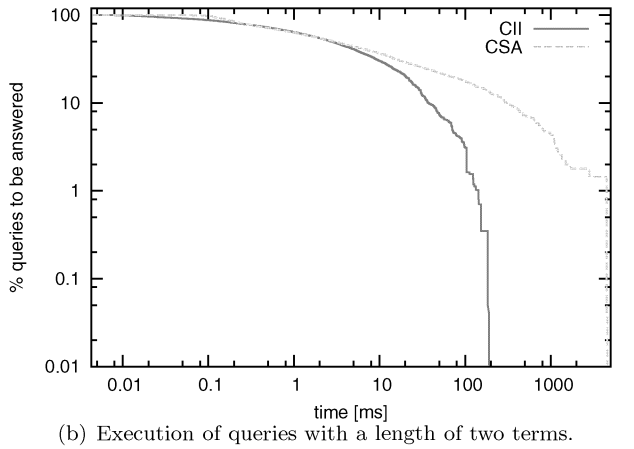
(b) Execution of queries with a length of two terms.

Figure 2: *AND* queries on the first 50 000 documents of WT2g.

As a representative of the compressed suffix indexes, we used the compressed suffix array (CSA) implementation by Sadakane available from the Pizza&Chili website. Compressed suffix arrays showed the best performance among the highly space efficient implementations on our system. For a comparision of further Pizza&Chili indexes based on a WT2g subset see Appendix A. We built the suffix array via the API using default parameters. The peak memory usage for indexing the first 50 000 documents of WT2g into a compressed suffix array was already at the limits of our physical main memory. So the following comparison will be based on this subset. Additionaly, we have conducted our experiments for the compressed inverted index on the complete WT2g corpus, as well as on WT2g.s which was already used in [1]. It is derived from the former by indexing each sentence of the plain normalized text as a single document. The results are shown in Appendix B.



(a) Average querying time.



(b) Execution of queries with a length of two terms.

Figure 3: *Phrase* queries on the first 50 000 documents of WT2g.

Table 1 contains the properties of the two different indexes. The size of the parts of the compressed inverted index (CII) that provide the same functionality as the CSA is about 10% smaller than the CSA index and about 43% smaller than the input text. In addition, 26% of the input size is required by the CII for reconstructing the original documents from their normalized versions. This still leaves a saving of more than 20% compared to the input text.

The indexing time of both methods is comparable but ours needs only about one fifth of the memory during construction. We still need about twice as much memory than the input text during compression. However, note that the text itself could be streamed from disk during construction. Furthermore, we can build the index for one core at a time (possibly in parallel), temporarily using the space of the other cores for the construction. Hence, this looks like an acceptable space consumption.

For evaluating the querying performance, we generated pseudo real-world queries by selecting random hits. From a randomly chosen document, we used between one and ten arbitrary terms to build an *AND* query. Similarly, we chose a random term from such a document as the start of a *phrase* query. The lengths of the generated phrase queries ranged between two and ten terms. We built 100 000 queries varying in their lengths according to the distribution given in [23], where it is reported that over 80% of real world queries consists of between one and three terms.

Our first experiment was to determine the *AND* querying performance of the two approaches. Figure 2(a) shows the average query time over the first 10 000 of our generated queries. As expected, the inverted index performs well over the entire range of query lengths. It benefits from a larger number of query terms since this makes it more likely that the query contains a rare term with a short inverted list. Since the running time of the intersection algorithm [1] is essentially linear in the smaller list, this decreases processing time. In contrast, the compressed suffix arrays produce unsorted lists of occurrences for all query terms that have to be generated using complicated algorithms which cause many cache faults. This takes the major part of the querying time (in-SA time). In comparison to that, the time required for mapping the positions to document IDs and merging the lists is negligible. The bottom line difference is huge. On the average, our data structure is more than 5 000 times faster than CSA. In Figure 2(b) we took a closer look at the 'difficult' queries with a length of two terms. The figure shows how many percent of the queries take longer than a given time. In a double logarithmic plot, both curves have a quite clear cutoff. However, the differences are again huge. While inverted indexes never took more than 3.6 ms, the CSA needs up to 35 s, almost 10 000 times longer. We can conclude that a search engine based on suffix array would probably need an additional inverted index at least for the Boolean queries.

Our next experiment was to investigate the *phrase* querying performance — a home match for suffix arrays. In Figure 3(a) we see the average time required for the 100 000 queries of our pseudo real-world query set. For the most frequent practical case of two terms, the inverted index is nevertheless more than 20 times faster. Suffix arrays are slightly faster for the rare and easy phrases with more than four terms. The distribution of the query times in Figure 3(b) indicates that the largest observed query times are a factor 24 apart. CSA needs nearly 5s for some of the phrases. The reason is that some phrases occur very frequently, and unpacking all of them can be very expensive.

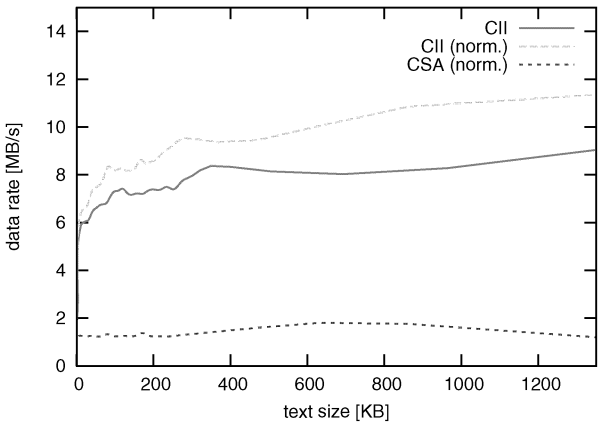


Figure 4: Document reporting speed. The curves are smoothed using gnuplot option `smooth bezier` in order to dampen fluctuations due to individual characteristics of documents.

Finally, Figure 4 shows the throughput for document reporting as a function of document size. Our scheme allows 6–8 MByte per second of text output. Retrieving data from disk (assuming an access latency of 5ms) would be faster beginning at around 32 KByte.[6] This suggests a hybrid storage scheme keeping longer files on disk. But note that in many applications, the texts are much shorter. For CSA, the bandwidth is about five times smaller.

In the appendix we also show results for the full wt2g data set which is about 5× larger. The average query time goes up proportionally to about 5ms for two term AND queries and about 70ms for two term phrase queries. Large query times are now 50ms for AND queries and 1.1 s for phrase queries. All these numbers are satisfactory for a text search engine, although it would be good to reduce the expensive phrase queries by indexing also some pairs and triples of word. The interesting question is how much space this would cost.

7 Conclusions and Future Work

A carefully engineered in-memory search engine based on inverted indexes and positional indexes allows fast queries using considerably less memory than the input size. We believe that we have not yet reached the end of what can be achieved with our approach: The bags of words are not optimally encodeable with Golomb codes and we could use more adaptive schemes. We could also compress the dictionary. With respect to query performance it seems most interesting to add a carefully selected set of phrases to the index in order to speed up the most expensive phrase queries. So far, space consumption and performance of index construction has not been our focus of attention. We will attack these issues together with the question of fast updates. The latter will be implemented using a small index for recent updates together with a batched update of the static data structures that should run in the background without replicating too much data. It would also be interesting to try suffix arrays that use the document IDs as their alphabet. Unfortunately, the current implementations seem to be tied to an 8 bit alphabet.

[6] On the first glance, it looks like parallel disks would be a way to mitigate disk access cost. However, with the advent of multicore processors, this option has become quite unattractive – one disk now costs more than a processing core so that even rack servers now tend to have less than one disk per processing core. Using blade servers, the ratio of cores to disks is even larger.

Acknowledgements We would like to thank Franz Färber and Holger Bast for valuable discussions.

References

[1] Sanders, P., Transier, F.: Intersection in integer inverted indices. In: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments ALENEX, SIAM (2007) 71–83

[2] Golomb, S.: Run-length encodings. IEEE Transactions on Information Theory **12** (1966) 399–401

[3] Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys **38** (2006)

[4] Culpepper, J.S., Moffat, A.: Compact set representation for information retrieval. In: SPIRE. Lecture Notes in Computer Science, Springer (2007)

[5] Zobel, J., Moffat, A., Sacks-Davis, R.: Storage management for files of dynamic records. In: Proceedings of the 4th Australian Database Conference, Brisbane, Australia, World Scientific (1993) 26–38

[6] Brown, E.W., Callan, J.P., Croft, W.B., Moss, J.E.B.: Supporting full-text information retrieval with a persistent object store. In: Proceedings of the Fourth Internationl Conference on Extending Database Technology EDBT'94. (1994) 365–378

[7] Strohman, T., Croft, W.B.: Efficient document retrieval in main memory. In: Proceedings of the 30th annual international conference on Research and development in information retrieval SIGIR '07, New York, NY, USA, ACM Press (2007) 175–182

[8] Luk, R.W.P., Lam, W.: Efficient in-memory extensible inverted file. Information Systems **32** (2007) 733–754

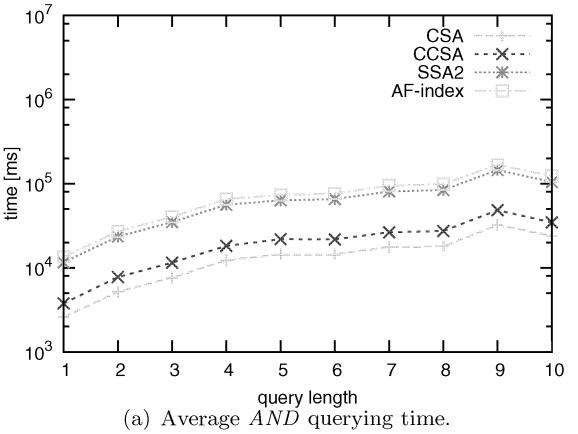
[9] Cutting, D., Pedersen, J.: Optimization for dynamic inverted index maintenance. In: Proceedings of the 13th annual international conference on Research and development in information retrieval SIGIR '90, New York, NY, USA, ACM Press (1990) 405–411

[10] Gutwin, C., Paynter, G., Witten, I., Nevill-Manning, C., Frank, E.: Improving browsing in digital libraries

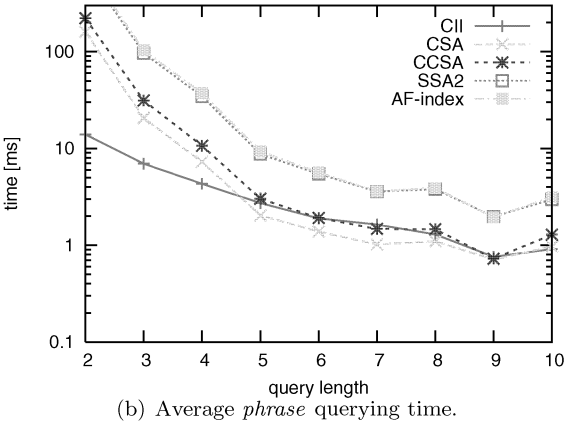
with keyphrase indexes. Decision Support Systems **27** (1999) 81–104
[11] Bahle, D., Williams, H.E., Zobel, J.: Efficient phrase querying with an auxiliary index. In: Proceedings of the 25th annual international conference on Research and development in information retrieval SIGIR '02, New York, NY, USA, ACM Press (2002) 215–221
[12] Williams, H.E., Zobel, J., Bahle, D.: Fast phrase querying with combined indexes. ACM Transactions on Information Systems **22** (2004) 573–594
[13] Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys **39** (2007)
[14] González, R., Navarro, G.: Compressed text indexes with fast locate. In: Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching CPM'07. Lecture Notes in Computer Science, Springer (2007) 216–227
[15] Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann (1999)
[16] Ferragina, P., Fischer, J.: Suffix arrays on words. In: Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching CPM'07. Lecture Notes in Computer Science, Springer (2007) 328–339
[17] Puglisi, S.J., Smyth, W.F., Turpin, A.: Inverted files versus suffix arrays for locating patterns in primary memory. In: SPIRE. Volume 4209 of Lecture Notes in Computer Science., Springer (2006) 122–133
[18] Bast, H., Mortensen, C.W., Weber, I.: Output-sensitive autocompletion search. JIR (2007) to appear, special issue on SPIRE 2006.
[19] H. Bast et al.: Seminar: Searching with suffix arrays. http://search.mpi-inf.mpg.de/wiki/IrSeminarWs06 (2007)
[20] Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, New York (1999)
[21] Young, R.M.: Euler's constant. The Mathematical Gazette **75** (1991) 187–190
[22] Barbay, J., López-Ortiz, A., Lu, T.: Faster adaptive set intersections for text searching. In Àlvarez, C., Serna, M.J., eds.: Experimental Algorithms, 5th International Workshop, WEA. Volume 4007 of LNCS., Springer (2006) 146–157
[23] Jansen, B.J., Spink, A., Bateman, J., Saracevic, T.: Real life information retrieval: a study of user queries on the web. SIGIR Forum **32** (1998) 5–17

A Comparision of different Pizza&Chili index implementations on the WT2g subset

We have tried all implementations available on the Pizza&Chili site. However, some implementations crashed and others had so long indexing times (e.g., due to swapping) that we had to abort them. Table 2 summarizes the results of the other indices. Since CSA gives the best performance, we use it as our reference in the main paper.



(a) Average *AND* querying time.



(b) Average *phrase* querying time.

B Further experimental results for CII

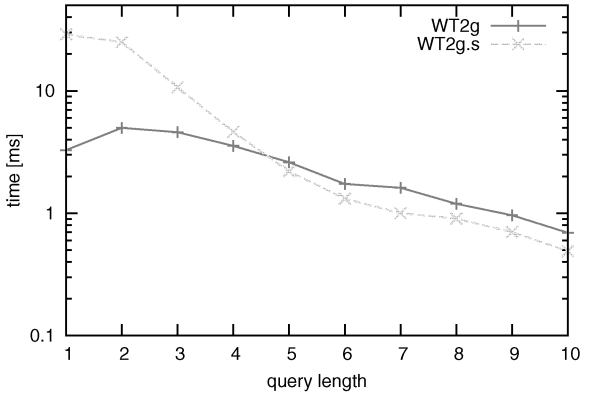
	WT2g	WT2g.s
# of documents	247 491	10 749 669
dictionary	85.9	55.3
document index	163.6	380.6
positional index	663.9	654.0
bag of words	139.4	410.2
text delta [MB]	557.6	-
sum [MB]	**1610.4**	**1500.2**
input size [MB]	2118.8	1487.0
compression	0.76	1.0[7]
indexing time [min]	**81.7**	**35.6**
peak mem usage [GB]	3.2	3.5

Table 3: Index properties

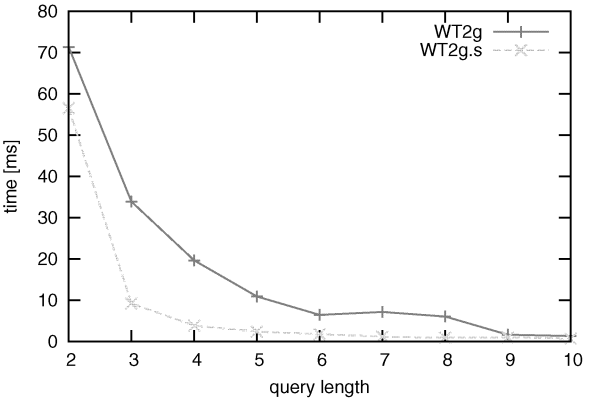
[7] Here, an average document length of 25.6 seems to be the limit of the compression potential of the bag-of-words approach. Obviously, the shorter the document lengths are, the more values have to be stored in the bags while indexing equal contents.

	CSA	CCSA	SSA2	AF-index
suffix array	230.8	500.8	302.5	279.2
doc bound list	0.1	0.1	0.1	0.1
sum [MB]	**230.9**	**500.9**	**302.6**	**279.3**
compression	0.64	1.39	0.84	0.77
indexing time [min]	**9.3**	**11.5**	**8.7**	**23.0**
peak mem usage [GB]	3.2	3.1	2.1	3.1

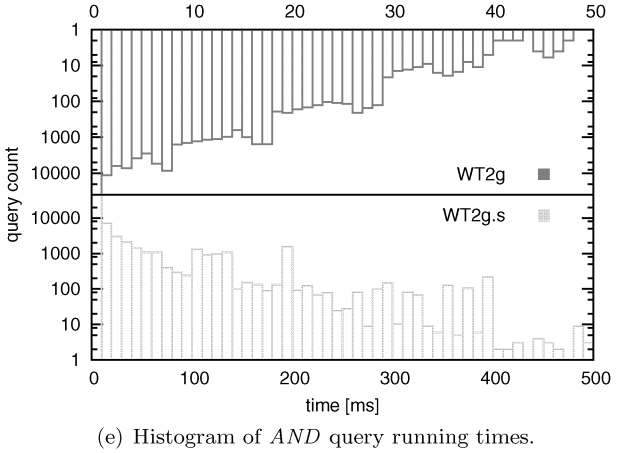
Table 2: Index properties of Pizza&Chili indexes.



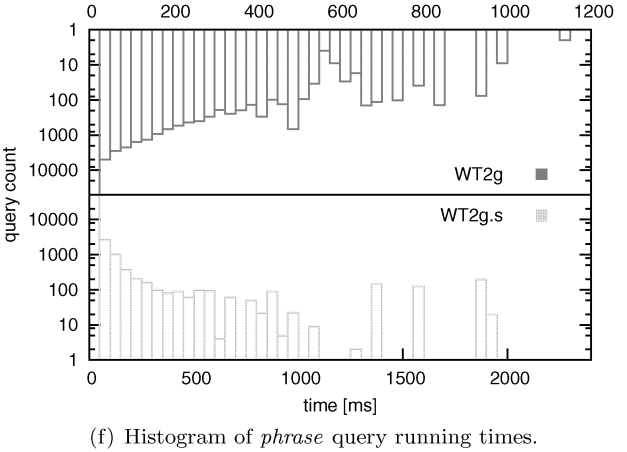
(c) Average *AND* querying time: AND queries for small query lenghts are slower on WT2g.s, as there are more documents and hence more results. Larger queries benefit from larger inverted lists, as in these cases more lookup lists are used.



(d) Average *phrase* querying time: Phrase queries are faster on WT2g.s because the position lists are shorter for small documents.



(e) Histogram of *AND* query running times.



(f) Histogram of *phrase* query running times.

Figure 5: Operations on CII

SHARC: Fast and Robust Unidirectional Routing *

Reinhard Bauer † Daniel Delling†

Abstract

During the last years, impressive speed-up techniques for DIJKSTRA's algorithm have been developed. Unfortunately, the most advanced techniques use *bidirectional* search which makes it hard to use them in scenarios where a backward search is prohibited. Even worse, such scenarios are widely spread, e.g., timetable-information systems or *time-dependent* networks.

In this work, we present a *unidirectional* speed-up technique which competes with bidirectional approaches. Moreover, we show how to exploit the advantage of unidirectional routing for fast exact queries in timetable information systems and for fast approximative queries in time-dependent scenarios. By running experiments on several inputs other than road networks, we show that our approach is very robust to the input.

1 Introduction

Computing shortest paths in graphs is used in many real-world applications like route planning in road networks, timetable information for railways, or scheduling for airplanes. In general, DIJKSTRA's algorithm [10] finds a shortest path between a given source s and target t. Unfortunately, the algorithm is far too slow to be used on huge datasets. Thus, several speed-up techniques have been developed (see [33, 29] for an overview) yielding faster query times for typical instances, e.g., road or railway networks. Due to the availability of huge road networks, recent research on shortest paths speed-up techniques solely concentrated on those networks [9]. The fastest known techniques [5, 1] were developed for road networks and use specific properties of those networks in order to gain their enormous speed-ups.

However, these techniques perform a bidirectional query or at least need to know the exact target node of a query. In general, these hierarchical techniques step up a hierarchy—built during preprocessing—starting both from source and target and perform a fast query on a very small graph. Unfortunately, in certain scenarios a backward search is prohibited, e.g. in timetable information systems and time-dependent graphs the time of arrival is unknown. One option would be to guess the arrival time and then to adjust the arrival time after forward and backward search have met. Another option is to develop a fast *unidirectional* algorithm.

In this work, we introduce SHARC-Routing, a fast and robust approach for *unidirectional* routing in large networks. The central idea of SHARC (Shortcuts + Arc-Flags) is the adaptation of techniques developed for Highway Hierarchies [28] to Arc-Flags [21, 22, 23, 18]. In general, SHARC-Routing iteratively constructs a contraction-based hierarchy during preprocessing and automatically sets *arc-flags* for edges removed during contraction. More precisely, arc-flags are set in such a way that a unidirectional query considers these removed *component*-edges only at the beginning and the end of a query. As a result, we are able to route very efficiently in scenarios where other techniques fail due to their bidirectional nature. By using approximative arc-flags we are able to route very efficiently in *time-dependent* networks, increasing performance by one order of magnitude over previous time-dependent approaches. Furthermore, SHARC allows to perform very fast queries—*without* updating the preprocessing—in scenarios where metrics are changed frequently, e.g. different speed profiles for fast and slow cars. In case a user needs even faster query times, our approach can also be used as a bidirectional algorithm that outperforms the most prominent techniques (see Figure 1 for an example on a typical search space of uni- and bidirectional SHARC). Only Transit-Node Routing is faster than this variant of SHARC, but SHARC needs considerably less space. A side-effect of SHARC is that preprocessing takes much less time than for pure Arc-Flags.

Related Work. To our best knowledge, three approaches exist that iteratively contract and prune the graph during preprocessing. This idea was introduced in [27]. First, the graph is contracted and afterwards partial trees are built in order to determine highway edges. Non-highway edges are removed from the graph. The contraction was significantly enhanced in [28] reducing preprocessing and query times drastically. The RE algorithm, introduced in [14, 15], also uses the contraction from [28] but pruning is based on reach values

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

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

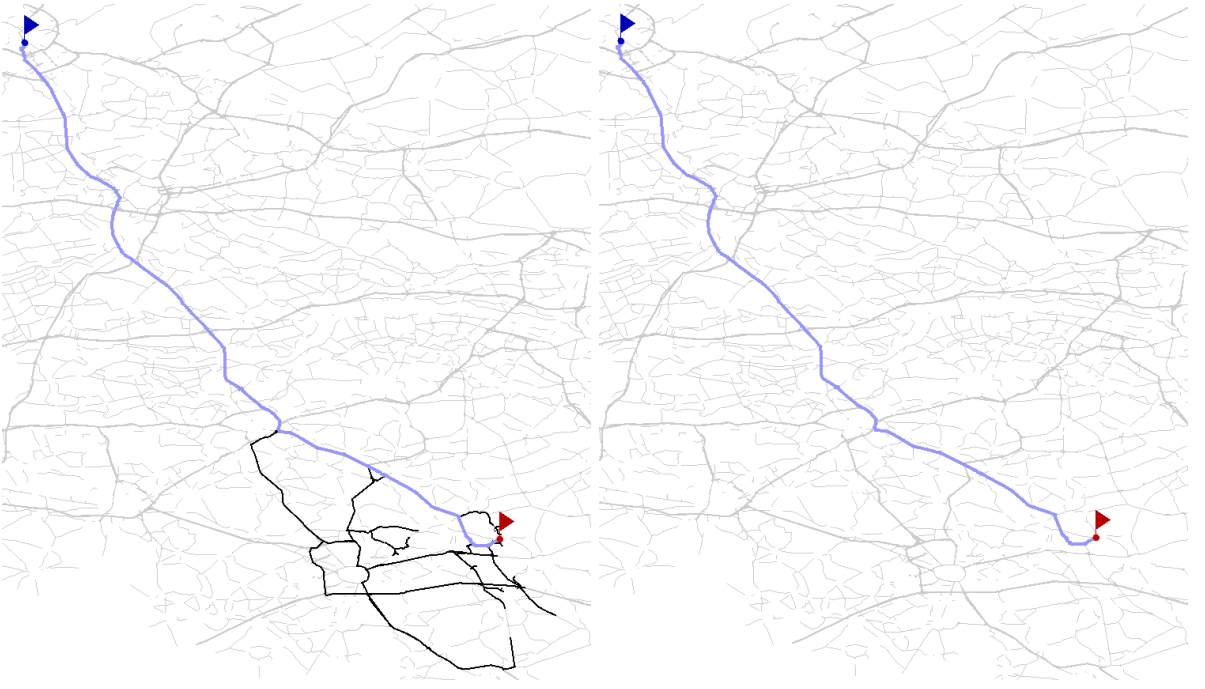


Figure 1: Search space of a typical uni-(left) and bidirectional(right) SHARC-query. The source of the query is the upper flag, the target the lower one. Relaxed edges are drawn in black. The shortest path is drawn thicker. Note that the bidirectional query *only* relaxes shortest-path edges.

for edges. A technique relying on contraction as well is Highway-Node Routing [31], which combines several ideas from other speed-up techniques. All those techniques build a hierarchy during the preprocessing and the query exploits this hierarchy. Moreover, these techniques gain their impressive speed-ups from using a bidirectional query, which—among other problems—makes it hard to use them in time-dependent graphs. Up to now, solely pure ALT [13] has been proven to work in such graphs [7]. Moreover, REAL [14, 15]—a combination of RE and ALT—can be used in a unidirectional sense but still, the exact target node has to be known for ALT, which is unknown in timetable information systems (cf. [26] for details).

Similar to Arc-Flags [21, 22, 23, 18], Geometric Containers [34] attaches a label to each edge indicating whether this edge is important for the current query. However, Geometric Containers has a worse performance than Arc-Flags and preprocessing is based on computing a full shortest path tree from every node within the graph. For more details on *classic* Arc-Flags, see Section 2.

Overview. This paper is organized as follows. Section 2 introduces basic definitions and reviews the classic Arc-Flag approach. Preprocessing and the query algorithm of our SHARC approach are presented in Section 3, while Section 4 shows how SHARC can be used in time-dependent scenarios. Our experimental study on real-world and synthetic datasets is located in Section 5 showing the excellent performance of SHARC on various instances. Our work is concluded by a summary and possible future work in Section 6.

2 Preliminaries

Throughout the whole work we restrict ourselves to simple, directed graphs $G = (V, E)$ with positive length function $len : E \to \mathbb{R}^+$. The reverse graph $\overline{G} = (V, \overline{E})$ is the graph obtained from G by substituting each $(u, v) \in E$ by (v, u). Given a set of edges H, $source(H)$ / $target(H)$ denotes the set of all source / target nodes of edges in H. With $\deg_{in}(v)$ / $\deg_{out}(v)$ we denote the number of edges whose target / source node is v. The 2-core of an undirected graph is the maximal node induced subgraph of minimum node degree 2. The 2-core of a directed graph is the 2-core of the corresponding simple, unweighted, undirected graph. A tree on a graph for which exactly the root lies in the 2-core is called an *attached tree*.

A *partition* of V is a family $\mathcal{C} = \{C_0, C_1, \ldots, C_k\}$ of sets $C_i \subseteq V$ such that each node $v \in V$ is contained in exactly one set C_i. An element of a partition is

called a *cell*. A *multilevel partition* of V is a family of partitions $\{\mathcal{C}^0, \mathcal{C}^1, \ldots, \mathcal{C}^l\}$ such that for each $i < l$ and each $C_n^i \in \mathcal{C}^i$ a cell $C_m^{i+1} \in \mathcal{C}^{i+1}$ exists with $C_n^i \subseteq C_m^{i+1}$. In that case the cell C_m^{i+1} is called the *supercell* of C_n^i. The supercell of a level-l cell is V. The *boundary nodes* B_C of a cell C are all nodes $u \in C$ for which at least one node $v \in V \setminus C$ exists such that $(v, u) \in E$ or $(u, v) \in E$. The distance according to len between two nodes u and v we denote by $d(u, v)$.

Classic Arc-Flags. The classic Arc-Flag approach, introduced in [21, 22], first computes a partition $\mathcal{C}$ of the graph and then attaches a *label* to each edge e. A label contains, for each cell $C_i \in \mathcal{C}$, a flag $AF_{C_i}(e)$ which is `true` iff a shortest path to a node in C_i starts with e. A modified DIJKSTRA then only considers those edges for which the flag of the target node's cell is `true`. The big advantage of this approach is its easy query algorithm. Furthermore an Arc-Flags DIJKSTRA often is optimal in the sense that it *only* visits those edges that are on the shortest path. However, preprocessing is very extensive, either regarding preprocessing time or memory consumption. The original approach grows a full shortest path tree from each boundary node yielding preprocessing times of several weeks for instances like the Western European road network. Recently, a new *centralized* approach has been introduced [17]. It grows a centralized tree from each cell keeping the distances to *all* boundary nodes of this cell in memory. This approach allows to preprocess the Western European road network within one day but for the price of high memory consumption during preprocessing.

Note that $AF_{C_i}(e)$ is `true` for almost all edges $e \in C_i$ (we call this flags the *own-cell*-flag). Due to these own-cell-flags an Arc-Flags DIJKSTRA yields no speed-up for queries within the same cell. Even worse, using a unidirectional query, more and more edges become important when approaching the target cell (the *coning effect*) and finally, all edges are considered as soon as the search enters the target cell. While the coning effect can be weakened by a bidirectional query, the former also holds for such queries. Thus, a two-level approach is introduced in [23] which weakens these drawbacks as cells become quite small on the lower level. It is obvious that this approach can be extended to a multi-level approach.

3 Static SHARC

In this section, we explain SHARC-Routing in *static* scenarios, i.e., the graph remains untouched between two queries. In general, the SHARC query is a standard multi-level Arc-Flags DIJKSTRA, while the preprocessing incorporates ideas from hierarchical approaches.

3.1 Preprocessing of SHARC is similar to Highway Hierarchies and REAL. During the *initialization* phase, we extract the 2-core of the graph and perform a multi-level partition of G according to an input parameter P. The number of levels L is an input parameter as well. Then, an *iterative* process starts. At each step i we first *contract* the graph by *bypassing* unimportant nodes and set the arc-flags *automatically* for each removed edge. On the contracted graph we compute the arc-flags of level i by growing a *partial* centralized shortest-path tree from each cell C_j^i. At the end of each step we *prune* the input by detecting those edges that already have their final arc-flags assigned. In the *finalization* phase, we assemble the output-graph, refine arc-flags of edges removed during contraction and finally reattach the 1-shell nodes removed at the beginning. Figure 2 shows a scheme of the SHARC-preprocessing. In the following we explain each phase separately. We hereby restrict ourselves to arc-flags for the unidirectional variant of SHARC. However, the extension to computing bidirectional arc-flags is straight-forward.

3.1.1 1-Shell Nodes. First of all, we extract the 2-core of the graph as we can directly assign correct arc-flags to attached trees that are fully contained in a cell: Each edge targeting the core gets all flags assigned `true` while those directing away from the core only get their own-cell flag set `true`. By removing 1-shell nodes *before* computing the partition we ensure the "fully contained" property by assigning all nodes in an attached tree to the cell of its root. After the last step of our preprocessing we simply reattach the nodes and edges of the 1-shell to the output graph.

3.1.2 Multi-Level Partition. As shown in [23], the classic Arc-Flag method heavily depends on the partition used. The same holds for SHARC. In order to achieve good speed-ups, several requirements have to be fulfilled: cells should be connected, the size of cells should be balanced, and the number of boundary nodes has to be low. In this work, we use a locally optimized partition obtained from SCOTCH [25]. For details, see Section 5. The number of levels L and the number of cells per level are tuning-parameters.

3.1.3 Contraction. The graph is contracted by iteratively *bypassing* nodes until no node is *bypassable* any more. To bypass a node n we first remove n, its incoming edges I and its outgoing edges O from the graph. Then, for each $u \in source(I)$ and for each $v \in target(I) \setminus \{u\}$ we introduce a new edge of the length $len(u, n) + len(n, v)$. If there already is an edge

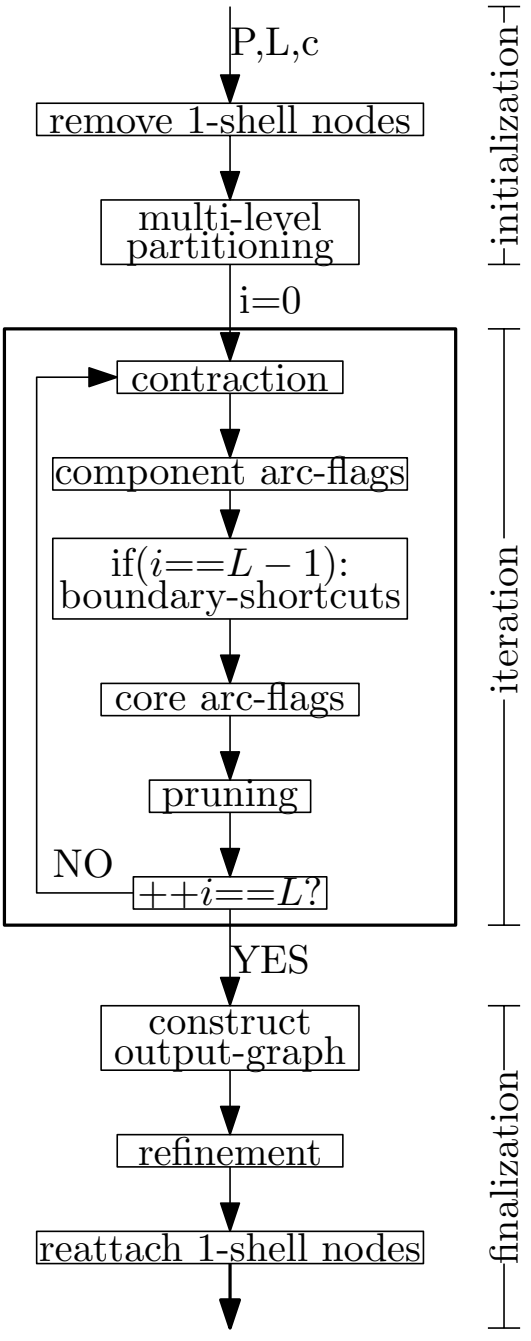


Figure 2: Schematic representation of the preprocessing. Input parameters are the partition parameters P, the number of levels L, and the contraction parameter c. During initialization, we remove the 1-shell nodes and partition the graph. Afterwards, an iterative process starts which contracts the graph, sets arc-flags, and prunes the graph. Moreover, during the last iteration step, boundary shortcuts are added to the graph. During the finalization, we construct the output-graph, refine arc-flags and reattach the 1-shell nodes to the graph.

connecting u and v in the graph, we only keep the one with smaller length. We call the number of edges of the path that a shortcut represents on the graph at the beginning of the current iteration step the *hop number* of the shortcut. To check whether a node is bypassable we first determine the number $\#shortcut$ of *new* edges that would be inserted into the graph if n is bypassed, i.e., existing edges connecting nodes in $source(I)$ with nodes in $target(O)$ do not contribute to $\#shortcut$. Then we say a node is bypassable iff the *bypass criterion* $\#shortcut \leq c \cdot (\deg_{in}(n) + \deg_{out}(n))$ is fulfilled, where c is a tunable *contraction parameter*.

A node being bypassed influences the degree of their neighbors and thus, their bypassability. Therefore, the order in which nodes are bypassed changes the resulting contracted graph. We use a heap to determine the next bypassable node. The key of a node n within the heap is $h \cdot \#shortcut/(\deg_{in}(n) + \deg_{out}(n))$ where h is the hop number of the hop-maximal shortcut that would be added if n was bypassed, smaller keys have higher priority. To keep the length of shortcuts limited we do not bypass a node if that results in adding a shortcut with hop number greater than 10. We say that the nodes that have been bypassed belong to the *component*, while the remaining nodes are called *core-nodes*. In order to guarantee correctness, we use *cell-aware* contraction, i.e., a node n is never marked bypassable if any of its neighboring nodes is *not* in the same cell as n.

Our contraction routine mainly follows the ideas introduced in [28]. The idea to control the order, in which the nodes are bypassed using a heap is due to [14]. In addition, we slightly altered the bypassing criterion, leading to significantely better results, e.g. on the road network of Western Europe, our routine bypasses twice the number of nodes with the same contraction parameter. The main difference to [28] is that we do not count existing edges for determining #shortcut. Finally, the idea to bound the hop number of a shortcut is due to [6].

3.1.4 Boundary-Shortcuts. During our study, we observed that—at least for long-range queries on road networks—a classic bidirected Arc-Flags Dijkstra often is optimal in the sense that it visits *only* the edges on the shortest path between two nodes. However, such shortest paths may become quite long in road networks. One advantage of SHARC over classic Arc-Flags is that the contraction routine reduces the number of hops of shortest paths in the network yielding smaller search spaces. In order to further reduce this hop number we enrich the graph by additional shortcuts. In general we could try any shortcuts as our preprocessing favors paths with less hops over those with more hops, and

thus, added shortcuts are used for long range queries. However, adding shortcuts crossing cell-borders can increase the number of boundary nodes, and hence, increase preprocessing time. Therefore, we use the following heuristic to determine good shortcuts: we add *boundary shortcuts* between some boundary nodes belonging to the same cell C at level $L-1$. In order to keep the number of added edges small we compute the betweenness [4] values c_B of the boundary nodes on the remaining core-graph. Each boundary node with a betweenness value higher than half the maximum gets $3 \cdot \sqrt{|B_C|}$ additional outgoing edges. The targets are those boundary nodes with highest $c_B \cdot h$ values, where h is the number of hops of the added shortcut.

3.1.5 Arc-Flags. Our query algorithm is executed on the original graph enhanced by shortcuts added during the contraction phase. Thus, we have to assign arc-flags to each edge we remove during the contraction phase. One option would be to set every flag to `true`. However, we can do better. First of all, we keep all arc-flags that already have been computed for lower levels. We set the arc-flags of the current and all higher levels depending on the source node s of the deleted edge. If s is a core node, we only set the own-cell flag to `true` (and others to `false`) because this edge can only be relevant for a query targeting a node in this cell. If s belongs to the component, all arc-flags are set to `true` as a query has to leave the component in order to reach a node outside this cell. Finally, shortcuts get their own-cell flag *fixed* to `false` as relaxing shortcuts when the target cell is reached yields no speed-up. See Figure 3 for an example. As a result, an Arc-Flags query only considers components at the beginning and the end of a query. Moreover, we reduce the search space.

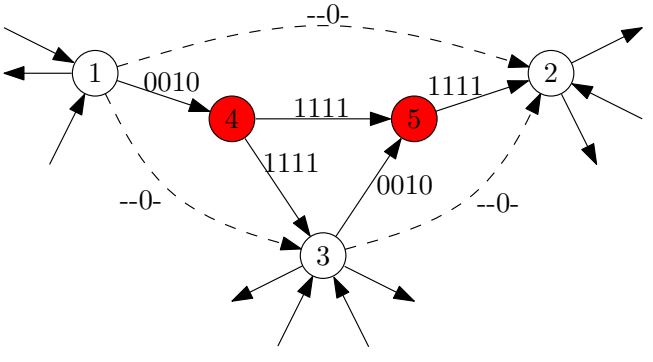


Figure 3: Example for assigning arc-flags during contraction for a partition having four cells. All nodes are in cell 3. The red nodes (4 and 5) are removed, the dashed shortcuts are added by the contraction. Arc-flags are indicated by a 1 for `true` and 0 for `false`. The edges directing into the component get *only* their own-cell flag set `true`. All edges in and out of the component get full flags. The added shortcuts get their own-cell flags fixed to `false`.

Assigning Arc-Flags to Core-Edges. After the contraction phase and assigning arc-flags to removed edges, we compute the arc-flags of the core-edges of the current level i. As described in [17], we grow, for each cell C, one centralized shortest path tree on the reverse graph starting from every boundary node $n \in B_C$ of C. We stop growing the tree as soon as all nodes of C's supercell have a distance to each $b \in B_C$ greater than the smallest key in the priority queue used by the centralized shortest path tree algorithm (see [17] for details). For any edge e that is in the supercell of C and that lies on a shortest path to at least one $b \in B_C$, we set $AF^i_C(e) = \texttt{true}$.

Note that the centralized approach sets arc-flags to `true` for *all* possible shortest paths between two nodes. In order to favor boundary shortcuts, we extend the centralized approach by introducing a second matrix that stores the number of hops to every boundary node. With the help of this second matrix we are able to assign `true` arc-flags only to *hop-minimal* shortest paths. However, using a second matrix increases the high memory consumption of the centralized approach even further. Thus, we use this extension only during the last iteration step where the core is small.

3.1.6 Pruning. After computing arc-flags at the current level, we prune the input. We remove unimportant edges from the graph by running two steps. First, we identify *prunable* cells. A cell C is called prunable if all neighboring cells are assigned to the same supercell. Then we remove all edges from a prunable cell that have at most their own-cell bit set. For those edges no flag can be assigned `true` in higher levels as then at least one flag for the surrounding cells must have been set before.

3.1.7 Refinement of Arc-Flags. Our contraction routine described above sets all flags to `true` for almost all edges removed by our contraction routine. However, we can do better: we are able to *refine* arc-flags by *propagation* of arc-flags from higher to lower levels. Before explaining our propagation routine we need the notion of level. The level $l(u)$ of a node u is determined by the iteration step it is removed in from the graph. All nodes removed during iteration step i belong to level i. Those nodes which are part of the core-graph after the last iteration step belong to level L. In the following, we explain our propagation routine for a given node u.

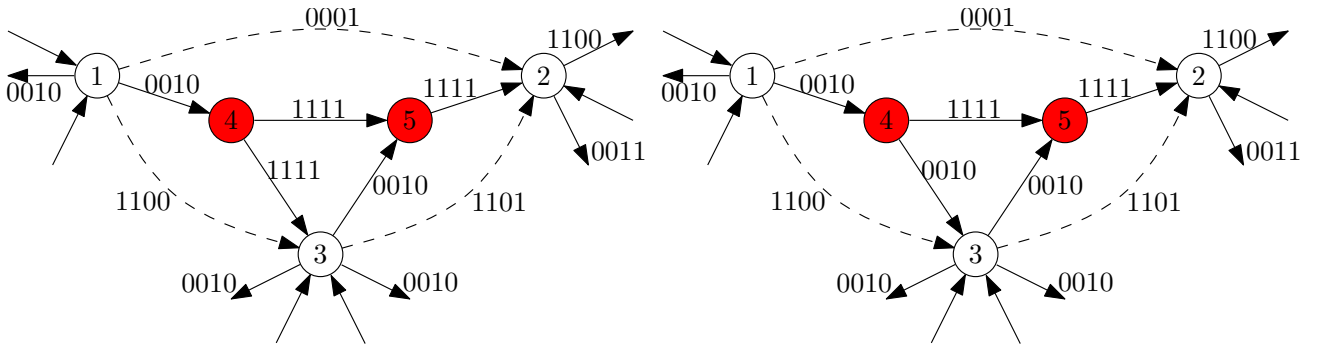


Figure 4: Example for refining the arc-flags of outgoing edges from node 4. The figure in the left shows the graph from Figure 3 after the last iteration step. The figure on the right shows the result of our refinement routine starting at node 4.

First, we build a partial shortest-path tree T starting at u, not relaxing edges that target nodes on a level smaller than $l(u)$. We stop the growth as soon as all nodes in the priority queue are *covered*. A node v is called covered as soon as a node between u and v—with respect to T—belongs to a level $> l(u)$. After the termination of the growth we remove all covered nodes from T resulting in a tree rooted at u and with leaves either in $l(u)$ or in a level higher than $l(u)$. Those leaves of the built tree belonging to a level higher than $l(u)$ we call *entry nodes* $\vec{N}(u)$ of u.

With this information we refine the arc-flags of all edges outgoing from u. First, we set all flags—except the own-cell flags—of all levels $\geq l(u)$ for all outgoing edges from u to `false`. Next, we assign entry nodes to outgoing edges from u. Starting at an entry node n_E we follow the predecessor in T until we finally end up in a node x whose predecessor is u. The edge (u, x) now inherits the flags from n_E. Every edge outgoing from n_E whose target t is *not* an entry node of u and *not* in a level $< l(u)$ propagates all `true` flags of all levels $\geq l(u)$ to (u, x).

In order to propagate flags from higher to lower levels we perform our propagation-routine in $L - 1$ refinement steps, starting at level $L-1$ and in descending order. Figure 4 gives an example. Note that during refinement step i we only refine arc-flags of edges outgoing from nodes belonging to level i.

3.1.8 Output Graph. The *output graph* of the preprocessing consists of the original graph enhanced by all shortcuts that are in the contracted graph at the end of at least one iteration step. Note that an edge (u, v) may be contained in no shortest path because a shorter path from u to v already exists. This especially holds for the shortcuts we added to the graph. As a consequence, such edges have no flag set `true` after the last step. Thus, we can remove all edges from the output graph with no flag set `true`. Furthermore the multi-level partition and the computed arc-flags are given.

3.2 Query. Basically, our query is a multi-level Arc-Flags Dijkstra adapted from the two-level Arc-Flags Dijkstra presented in [23]. The query is a modified Dijkstra that operates on the output graph. The modifications are as follows: When settling a node n, we compute the lowest level i on which n and the target node t are in the same supercell. When relaxing the edges outgoing from n, we consider only those edges having a set arc-flag on level i for the corresponding cell of t. It is proven that Arc-Flags performs correct queries. However, as our preprocessing is different, we have to prove Theorem 3.1.

Theorem 3.1. *The distances computed by SHARC are correct with respect to the original graph.*

The proof can be found in Appendix A. We want to point out that the SHARC query, compared to plain Dijkstra, only needs to additionally compute the common level of the current node and the target. Thus, our query is very efficient with a much smaller overhead compared to other hierarchical approaches. Note that SHARC uses shortcuts which have to be unpacked for determining the shortest path (if not only the distance is queried). However, we can directly use the methods from [6], as our contraction works similar to Highway Hierarchies.

Multi-Metric Query. In [3], we observed that the shortest path structure of a graph—as long as edge weights somehow correspond to travel times—hardly changes when we switch from one metric to another. Thus, one might expect that arc-flags are similar to each other for these metrics. We exploit this observation for our *multi-metric* variant of SHARC. During preprocess-

ing, we compute arc-flags for all metrics and at the end we store only *one* arc-flag per edge by setting a flag `true` as soon as the flag is `true` for at least one metric. An important precondition for multi-metric SHARC is that we use the same partition for each metric. Note that the structure of the core computed by our contraction routine is independent of the applied metric.

Optimizations. In order to improve both performance and space efficiency, we use three optimizations. Firstly, we increase *locality* by reordering nodes according to the level they have been removed at from the graph. As a consequence, the number of cache misses is reduced yielding lower query times. Secondly, we check before running a query, whether the target is in the 1-shell of the graph. If this check holds we do not relax edges that target 1-shell nodes whenever we settle a node being part of the 2-core. Finally, we store each different arc-flag only once in a separate array. We assign an additional pointer to each edge indicating the correct arc-flags. This yields a lower space overhead.

4 Time-Dependent SHARC

Up to this point, we have shown how preprocessing works in a *static* scenario. As our query is unidirectional it seems promising to use SHARC in a *time-dependent* scenario. The fastest known technique for such a scenario is ALT yielding only mild speed-ups of factor 3-5. In this section we present how to perform queries in time-dependent graphs with SHARC. In general, we assume that a time-dependent network $\overrightarrow{G} = (V, \overrightarrow{E})$ derives from an independent network $G = (V, E)$ by *increasing* edge weights at certain times of the day. For road networks these increases represent rush hours.

The idea is to compute *approximative* arc-flags in G and to use these flags for routing in $\overrightarrow{G}$. In order to compute approximative arc-flags, we relax our criterion for setting arc-flags. Recall that for exact flags, $AF_C((u,v))$ is set `true` if $d(u,b) + len(u,v) = d(v,b)$ holds for at least one $b \in B_C$. For γ-approximate flags (indicated by $\overline{AF}$), we set $\overline{AF}_C((u,v)) =$ `true` if equation $d(u,b) + len(u,v) \leq \gamma \cdot d(v,b)$ holds for at least one $b \in B_C$. Note that we *only* have to change this criterion in order to compute approximative arc-flags instead of exact ones by our preprocessing. However, we do *not* add boundary shortcuts as this relaxed criterion does not favor those shortcuts.

It is easy to see that there exists a trade-off between performance and quality. Low γ-values yield low query times but the error-rate may increase, while a large γ reduces the error rate of γ-SHARC but yields worse query performance, as much more edges are relaxed during the query than necessary.

5 Experiments

In this section, we present an extensive experimental evaluation of our SHARC-Routing approach. To this end, we evaluate the performance of SHARC in various scenarios and inputs. Our tests were executed on one core of an AMD Opteron 2218 running SUSE Linux 10.1. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.1, using optimization level 3.

Implementation Details. Our implementation is written in C++ using solely the STL. As priority queue we use a binary heap. Our graph is represented as forward star implementation. As described in [30], we have to store each edge twice if we want to iterate efficiently over incoming and outgoing edges. Thus, the authors propose to compress edges if target and length of incoming and outgoing edges are equal. However, SHARC allows an even simpler implementation. During preprocessing we only operate on the reverse graph and thus do *not* iterate over outgoing edges while during the query we *only* iterate over outgoing edges. As a consequence, we only have to store each edge once (for preprocessing at its target, for the query at its source). Thus, another advantage of our unidirectional SHARC approach is that we can reduce the memory consumption of the graph. Note that this does not hold for our bidirectional SHARC variant which needs considerably more space (cf. Tab. 1).

Multi-Level Partition. As already mentioned, the performance of SHARC highly depends on the partition of the graph. Up to now [2], we used METIS [20] for partitioning a given graph. However, in our experimental study we observed two downsides of METIS: On the one hand, cells are sometimes disconnected and the number of boundary nodes is quite high. Thus, we also tested PARTY [24] and SCOTCH [25] for partitioning. The former produces connected cells but for the price of an even higher number of boundary nodes. SCOTCH has the lowest number of boundary cells, but connectivity of cells cannot be guaranteed. Due to this low number of boundary nodes, we used SCOTCH and improve the obtained partitioning by adding smaller pieces of disconnected cells to neighbor cells. As a result, constructing and optimizing a partition can be done in less than 3 minutes for all inputs used.

Default Setting. Unless otherwise stated, we use a *unidirectional* variant of SHARC with a 3-level partition with 16 cells per supercell on level 0 and 1 and 96 cells on level 2. Moreover, we use a value of $c = 2.5$ as contraction parameter. When performing random s-t queries, the source s and target t are picked uniformly at random and results are based on 10 000 queries.

Table 1: Performance of SHARC and the most prominent speed-up techniques on the European and US road network with travel times. *Prepro* shows the computation time of the preprocessing in hours and minutes and the eventual *additional* bytes per node needed for the preprocessed data. For queries, the search space is given in the number of settled nodes, execution times are given in milliseconds. Note that other techniques have been evaluated on slightly different computers. The results for Highway Hierarchies and Highway-Node Routing derive from [30]. Results for Arc-Flags are based on 200 PARTY cells and are taken from [17].

	Europe				USA			
	Prepro		Query		Prepro		Query	
	[h:m]	[B/n]	#settled	[ms]	[h:m]	[B/n]	#settled	[ms]
SHARC	2:17	13	1 114	0.39	1:57	16	1 770	0.68
bidirectional SHARC	3:12	20	145	0.091	2:38	21	350	0.18
Highway Hierarchies	0:19	48	709	0.61	0:17	34	925	0.67
Highway-Node	0:15	8	1 017	0.88	0:16	8	760	0.50
REAL-(64,16)	2:21	32	679	1.10	2:01	43	540	1.05
Arc-Flags	17:08	19	2 369	1.60	10:10	10	8 180	4.30
Grid-based Transit-Node	–	–	–	–	20:00	21	NA	0.063
HH-based Transit-Node	2:44	251	NA	0.006	3:25	244	NA	0.005

5.1 Static Environment. We start our experimental evaluation with various tests for the *static* scenario. We hereby focus on road networks but also evaluate graphs derived from timetable information systems and synthetic datasets that have been evaluated in [2].

5.1.1 Road Networks. As inputs we use the largest strongly connected component of the road networks of Western Europe, provided by PTV AG for scientific use, and of the US which is taken from the DIMACS homepage [9]. The former graph has approximately 18 million nodes and 42.6 million edges and edge lengths correspond to travel times. The corresponding figures for the USA are 23.9 million and 58.3 million, respectively.

Random Queries. Tab. 1 reports the results of SHARC with default settings compared to the most prominent speed-up techniques. In addition, we report the results of a variant of SHARC which uses bidirectional search in connection with a 2-level partition (16 cells per supercell at level 0, 112 at level 1).

We observe excellent query times for SHARC in general. Interestingly, SHARC has a lower preprocessing time for the US than for Europe but for the price of worse query performance. On the one hand, this is due to the bigger size of the input yielding bigger cell sizes and on the other hand, the average hop number of shortest paths are bigger for the US than for Europe. However, the number of boundary nodes is smaller for the US yielding lower preprocessing effort. The bidirectional variant of SHARC has a more extensive preprocessing: both time and additional space increase, which is due to computing and storing forward and backward arc-flags. However, preprocessing does not take twice the time than for default SHARC as we use a 2-level setup for the bidirectional variant and preprocessing the third level for default SHARC is quite expensive (around 40% of the total preprocessing time). Comparing query performance, bidirectional SHARC is clearly superior to the unidirectional variant. This is due to the known disadvantages of uni-directional classic Arc-Flags: the coning effect and no arc-flag information as soon as the search enters the target cell (cf. Section 2 for details).

Comparing SHARC with other techniques, we observe that SHARC can compete with any other technique except HH-based Transit Node Routing, which requires much more space than SHARC. Stunningly, for Europe, SHARC settles more nodes than Highway Node Routing or REAL, but query times are smaller. This is due to the very low computational overhead of SHARC. Regarding preprocessing, SHARC uses less space than REAL or Highway Hierarchies. The computation time of the preprocessing is similar to REAL but longer than for Highway-Node Routing. The bidirectional variant uses more space and has longer preprocessing times, but the performance of the query is very good. The number of nodes settled is smaller than for any other technique and due to the low computational overhead query times are clearly lower than for Highway Hierarchies, Highway-Node Routing or REAL. Compared to the classic Arc-Flags, SHARC significantely reduces preprocessing time and query performance is better.

Local Queries. Figure 5 reports the query times of uni- and bidirectional SHARC with respect to the Dijkstra rank. For an s-t query, the Dijkstra rank of node v is the number of nodes inserted in the priority queue before v is reached. Thus, it is a kind of distance measure. As input we again use the European road network instance.

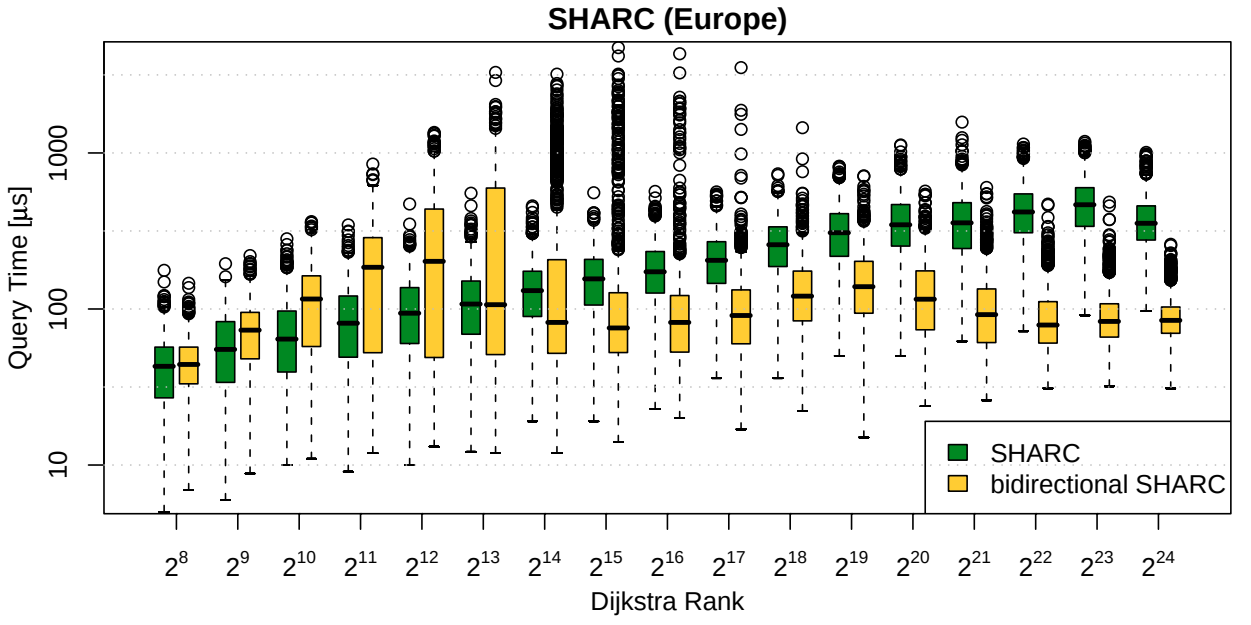


Figure 5: Comparison of uni- and bidirectional SHARC using the Dijkstra rank methodology [27]. The results are represented as box-and-whisker plot [32]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

Note that we use a logarithmic scale due to outliers. Unidirectional SHARC gets slower with increasing rank but the median stays below 0.6 ms while for bidirectional SHARC the median of the queries stays below 0.2 ms. However, for the latter, query times increase up to ranks of 2^{13} which is roughly the size of cells at the lowest level. Above this rank query times decrease and increase again till the size of cells at level 1 is reached. Interestingly, this effect deriving from the partition cannot be observed for the unidirectional variant. Comparing uni- and bidirectional SHARC we observe more outliers for the latter which is mainly due to less levels. Still, all outliers are below 3 ms.

Table 2: Performance of SHARC on different metrics using the European road instance. *Multi-metric* refers to the variant with one arc-flag and three edge weights (one weight per metric) per edge, while *single* refers to running SHARC on the applied metric.

		Prepro		Query	
profile	metric	[h:m]	[B/n]	#settled	[ms]
linear	single	2:17	13	1 114	0.39
	multi	6:51	16	1 392	0.51
slow car	single	1:56	14	1 146	0.41
	multi	6:51	16	1 372	0.50
fast car	single	2:24	13	1 063	0.37
	multi	6:51	16	1 348	0.49

Multi-Metric Queries. The original dataset of Western Europe contains 13 different road categories. By applying different speed profiles to the categories we obtain different metrics. Tab. 2 gives an overview of the performance of SHARC when applied to metrics representing typical average speeds of slow/fast cars. Moreover, we report results for the *linear* profile which is most often used in other publications and is obtained by assigning average speeds of 10, 20, ..., 130 to the 13 categories. Finally, results are given for multi-metric SHARC, which stores only *one* arc-flag for each edge.

As expected, SHARC performs very well on other metrics based on travel times. Stunningly, the loss in performance is only very little when storing only one arc-flag for all three metrics. However, the overhead increases due to storing more edge weights for shortcuts and the size of the arc-flags vector increases slightly. Due to the fact that we have to compute arc-flags for all metrics during preprocessing, the computational effort increases.

5.1.2 Timetable Information Networks. Unlike bidirectional approaches, SHARC-Routing can be used for timetable information. In general, two approaches exist to model timetable information as graphs: time-dependent and time-expanded networks (cf. [26] for details). In such networks timetable information can be obtained by running a shortest path query. However, in

Table 3: Performance of plain DIJKSTRA and SHARC on a local and long-distance time-expanded timetable networks, unit disk graphs (udg) with average degree 5 and 7, and grid graphs with 2 and 3 number of dimensions. Due to the smaller size of the input, we use a 2-level partition with 16,112 cells.

		PREPRO		QUERY	
graph	tech.	[h:m]	[B/n]	#sett	[ms]
rail	Dijkstra	0:00	0	1 299 830	406.2
local	SHARC	10:02	9	11 006	3.8
rail	Dijkstra	0:00	0	609 352	221.2
long	SHARC	3:29	15	7 519	2.2
udg	Dijkstra	0:00	0	487 818	257.3
deg.5	SHARC	0:01	16	568	0.3
udg	Dijkstra	0:00	0	521 874	330.1
deg.7	SHARC	0:10	42	1 835	1.0
grid	Dijkstra	0:00	0	125 675	36.7
2–dim	SHARC	0:32	60	1 089	0.4
grid	Dijkstra	0:00	0	125 398	78.6
3–dim	SHARC	1:02	97	5 839	1.9

both models a backward search is prohibited as the time of arrival is unknown in advance. Tab. 3 reports the results of SHARC on 2 time-expanded networks: The first represents the local traffic of Berlin/Brandenburg, has 2 599 953 nodes and 3 899 807 edges, the other graph depicts long distance connections of Europe (1 192 736 nodes, 1 789 088 edges). For comparison, we also report results for plain DIJKSTRA.

For time-expanded railway graphs we observe an increase in performance of factor 100 over plain DIJKSTRA but preprocessing is still quite high which is mainly due to the partition. The number of boundary nodes is very high yielding high preprocessing times. However, compared to other techniques (see [2]) SHARC (clearly) outperforms any other technique when applied to timetable information system.

5.1.3 Other inputs. In order to show the robustness of SHARC-Routing we also present results on synthetic data. On the one hand, 2- and 3-dimensional grids are evaluated. The number of nodes is set to 250 000, and thus, the number of edges is 1 and 1.5 million, respectively. Edge weights are picked uniformly at random from 1 to 1000. On the other hand, we evaluate random geometric graphs—so called unit disk graphs—which are widely used for experimental evaluations in the field of sensor networks (see e.g. [19]). Such graphs are obtained by arranging nodes uniformly at random on the plane and connecting nodes with a distance below a given threshold. By applying different threshold values we vary the density of the graph. In our setup, we use graphs with about 1 000 000 nodes and an average degree of 5 and 7, respectively. As metric, we use the distance between nodes according to their embedding. The results can be found in Tab. 3.

We observe that SHARC provides very good results for all inputs. For unit disk graphs, performance gets worse with increasing degree as the graph gets denser. The same holds for grid graphs when increasing the number of dimensions.

5.2 Time-Dependency. Our final testset is performed on a time-dependent variant of the European road network instance. We interpret the initial values as empty roads and add transit times according to rush hours. Due to the lack of data we increase *all* motorways by a factor of two and all national roads by a factor of 1.5 during rush hours. Our model is inspired by [11]. Our time-dependent implementation assigns 24 different weights to edges, each representing the edge weight at one hour of the day. Between two full hours, we interpolate the real edge weight *linearly*. An easy approach would be to store 24 edge weights separately. As this consumes a lot of memory, we reduce this overhead by storing factors for each hour between 5:00 and 22:00 of the day and the edge weight representing the empty road. Then we compute the travel time of the day by multiplying the initial edge weight with the factor (afterwards, we still have to interpolate). For each factor at the day, we store 7 bits resulting in 128 additional bits for each time-dependent edge. Note that we assume that roads are empty between 23:00 and 4:00.

Another problem for time-dependency is shortcutting time-dependent edges. We avoid this problem by *not* bypassing nodes which are incident to a time-dependent edge which has the advantage that the space-overhead for additional shortcuts stay small. Tab. 4 shows the performance of γ-SHARC for different approximation values. Like in the static scenario we use our default settings. For comparison, the values of time-dependent DIJKSTRA and ALT are also given. As we perform approximative SHARC-queries, we report three types of errors: By *error-rate* we denote the percentage of inaccurate queries. Besides the number of inaccurate queries it is also important to know the quality of a found path. Thus, we report the maximum and average *relative* error of all queries, computed by $1 - \mu_s/\mu_D$, where μ_s and μ_D depict the lengths of the paths found by SHARC and plain DIJKSTRA, respectively.

We observe that using γ values higher than 1.0 drastically reduces query performance. While error-rates are quite high for low γ values, the relative error is still quite low. Thus, the quality of the computed paths

Table 4: Performance of the time-dependent versions of DIJKSTRA, ALT, and SHARC on the Western European road network with time-dependent edge weights. For ALT, we use 16 *avoid* landmarks [16].

		ERROR			PREPRO		QUERY	
	γ	rate	rel. avg.	rel. max	[h:m]	[B/n]	#settled	[ms]
Dijkstra	-	0.0%	0.000%	0.00%	0:00	0	9 016 965	8 890.1
ALT	-	0.0%	0.000%	0.00%	0:16	128	2 763 861	2 270.7
SHARC	1.000	61.5%	0.242%	15.90%	2:51	13	9 804	3.8
	1.005	39.9%	0.096%	15.90%	2:53	13	113 993	61.2
	1.010	32.9%	0.046%	15.90%	2:51	13	221 074	131.3
	1.020	29.5%	0.024%	14.37%	2:50	13	285 971	182.7
	1.050	27.4%	0.013%	2.19%	2:51	13	312 593	210.9
	1.100	26.5%	0.009%	0.56%	2:52	12	321 501	220.8

is good, although in the worst-case the found path is 15.9% longer than the shortest. However, by increasing γ we are able to reduce the error-rate and the relative error significantely: The error-rate drops below 27%, the average error is below 0.01%, and in worst case the found path is only 0.56% longer than optimal. Generally speaking, SHARC routing allows a trade-off between quality and performance. Allowing moderate errors, we are able to perform queries 2 000 times faster than plain DIJKSTRA, while queries are still 40 times faster when allowing only very small errors.

Comparing SHARC (with $\gamma = 1.1$) and ALT, we observe that SHARC queries are one order of magnitude faster but for the price of correctness. In addition, the overhead is much smaller than for ALT. Note that we do not have to store time-dependent edge weights for shortcuts due to our weaker bypassing criterion. Summarizing, SHARC allows to perform fast queries in time-dependent networks with moderate error-rates and small average relative errors.

6 Conclusion

In this work, we introduced SHARC-Routing which combines several ideas from Highway Hierarchies, Arc-Flags, and the REAL-algorithm. More precisely, our approach can be interpreted as a unidirectional hierarchical approach: SHARC steps up the hierarchy at the beginning of the query, runs a strongly *goal-directed* query on the highest level and *automatically* steps down the hierarchy as soon as the search is approaching the target cell. As a result we are able to perform queries as fast as bidirectional approaches but SHARC can be used in scenarios where former techniques fail due to their bidirectional nature. Moreover, a bidirectional variant of SHARC clearly outperforms existing techniques except Transit Node Routing which needs much more space than SHARC.

Regarding future work, we are very optimistic that SHARC is very helpful when running multi-criteria queries due to the performance in multi-metric scenarios. In [15], an algorithm is introduced for computing exact reach values which is based on partitioning the graph. As our pruning rule would also hold for reach values, we are optimistic that we can compute *exact* reach values for our output graph with our SHARC preprocessing. For the time-dependent scenario one could think of other ways to determine good approximation values. Moreover, it would be interesting how to perform *correct* time-dependent SHARC queries.

SHARC-Routing itself also leaves room for improvement. The pruning rule could be enhanced in such a way that we can prune all cells. Moreover, it would be interesting to find better additional shortcuts, maybe by adapting the algorithms from [12] to approximate betweenness better. Another interesting question arising is whether we can further improve the contraction routine. And finally, finding partitions optimized for SHARC is an interesting question as well.

Summarizing, SHARC-Routing is a powerful, easy, fast and robust *unidirectional* technique for performing shortest-path queries in large networks.

Acknowledgments. We would like to thank Peter Sanders and Dominik Schultes for interesting discussions on contraction and arc-flags. We also thank Daniel Karch for implementing classic Arc-Flags. Finally, we thank Moritz Hilger for running a preliminary experiment with his new centralized approach.

References

[1] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

[2] R. Bauer, D. Delling, and D. Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. In C. Liebchen, R. K. Ahuja, and J. A. Mesa, editors, *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*. Schloss Dagstuhl, Germany, 2007.

[3] R. Bauer, D. Delling, and D. Wagner. Shortest-Path Indices: Establishing a Methodology for Shortest-Path Problems. Technical Report 2007-14, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2007.

[4] U. Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[5] D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-Performance Multi-Level Graphs. In Demetrescu et al. [9].

[6] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway Hierarchies Star. In Demetrescu et al. [9].

[7] D. Delling and D. Wagner. Landmark-Based Routing in Dynamic Graphs. In Demetrescu [8], pages 52–65.

[8] C. Demetrescu, editor. *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*. Springer, June 2007.

[9] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.

[10] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[11] I. C. Flinsenberg. *Route Planning Algorithms for Car Navigation*. PhD thesis, Technische Universiteit Eindhoven, 2004.

[12] R. Geisberger, P. Sanders, and D. Schultes. Better Approximation of Betweenness Centrality. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*. SIAM, 2008. to appear.

[13] A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.

[14] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, pages 129–143. SIAM, 2006.

[15] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Better Landmarks Within Reach. In Demetrescu [8], pages 38–51.

[16] A. V. Goldberg and R. F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

[17] M. Hilger. Accelerating Point-to-Point Shortest Path Computations in Large Scale Networks. Master's thesis, Technische Universität Berlin, 2007.

[18] M. Hilger, E. Köhler, R. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [9].

[19] F. Kuhn, R. Wattenhofer, and A. Zollinger. Worst-Case Optimal and Average-Case Efficient Geometric Ad-Hoc Routing. In *Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC'03)*, 2003.

[20] K. Lab. METIS - Family of Multilevel Partitioning Algorithms, 2007.

[21] U. Lauther. Slow Preprocessing of Graphs for Extremely Fast Shortest Path Calculations, 1997. Lecture at the Workshop on Computational Integer Programming at ZIB.

[22] U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. volume 22, pages 219–230. IfGI prints, 2004.

[23] R. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11:2.8, 2006.

[24] B. Monien and S. Schamberger. Graph Partitioning with the Party Library: Helpful-Sets in Practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE Computer Society, 2004.

[25] F. Pellegrini. SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007.

[26] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12:Article 2.4, 2007.

[27] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[28] P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.

[29] P. Sanders and D. Schultes. Engineering Fast Route Planning Algorithms. In Demetrescu [8], pages 23–36.

[30] P. Sanders and D. Schultes. Engineering Highway Hierarchies. submitted for publication, preliminary version at `http://algo2.iti.uka.de/schultes/hwy/`, 2007.

[31] D. Schultes and P. Sanders. Dynamic Highway-Node Routing. In Demetrescu [8], pages 66–79.

[32] R. D. Team. R: A Language and Environment for Statistical Computing, 2004.

[33] D. Wagner and T. Willhalm. Speed-Up Techniques

for Shortest-Path Computations. In *Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS'07)*, Lecture Notes in Computer Science, pages 23–36. Springer, February 2007.

[34] D. Wagner, T. Willhalm, and C. Zaroliagis. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics*, 10:1.3, 2005.

A Proof of Correctness

We here present a proof of correctness for SHARC-Routing. SHARC directly adapts the query from classic Arc-Flags, which is proved to be correct. Hence, we only have to show the correctness for all techniques that are used for SHARC-Routing but not for classic Arc-Flags.

The proof is logically split into two parts. First, we prove the correctness of the preprocessing without the refinement phase. Afterwards, we show that the refinement phase is correct as well.

A.1 Initialization and Main Phase. We denote by G_i the graph after iteration step i, $i = 1, \ldots, L-1$. By G_0 we denote the graph directly before iteration step 1 starts. The level $l(u)$ of a node u is defined to be the integer i such that u is contained in G_{i-1} but not in G_i. We further define the level of a node contained in G_{L-1} to be L.

The correctness of the multi-level arc-flag approach is known. The correctness of the handling of the 1-shell nodes is due to the fact that a shortest path starting from or ending at a 1-shell node u is either completely included in the attached tree T in which also u is contained, or has to leave or enter T via the corresponding core-node.

We want to stress that, when computing arc-flags, shortest paths do not have to be unique. We remember how SHARC handles that: In each level $l < L-1$ all shortest paths are considered, i.e., a shortest path directed acyclic graph is grown instead of a shortest paths tree and a flag for a cell C and an edge (u, v) is set `true`, if at least one shortest path to C containing (u, v) exists. In level $L-1$, all shortest paths are considered, that are hop minimal for given source and target, i.e., a flag for a cell C and an edge (u, v) is set `true`, if at least one shortest path to C containing (u, v) exists that is hop minimal among all shortest paths with same source and target.

We observe that the distances between two arbitrary nodes u and v are the same in the graph G_0 and $\bigcup_{k=0}^{i} G_k$ for any $i = 1, \ldots, L-1$.

Hence, to proof the correctness of unidirectional SHARC-Routing without the refinement phase and without 1-shell nodes we additionally have to proof the following lemma:

LEMMA A.1. *Given arbitrary nodes s and t in G_0, for which there is a path from s to t in G_0. At each step i of the SHARC-preprocessing there exists a shortest s-t-path $P = (v_1, \ldots, v_{j_1}; u_1, \ldots, u_{j_2}; w_1, \ldots, w_{j_3})$, $j_1, j_2, j_3 \in \mathbb{N}_0$, in $\bigcup_{k=0}^{i} G_k$, such that*

- *the nodes $v_1, \ldots, v_{j_1}$ and $w_1, \ldots, w_{j_3}$ have level of at most i,*
- *the nodes $u_1, \ldots, u_{j_2}$ have level of at least $i+1$*
- *u_{j_2} and t are in the same cell at level i*
- *for each edge e of P, the arc-flags assigned to e until step i allow the path P to t.*

We use the convention that $j_k = 0$, $k \in \{1, 2, 3\}$ means that the according subpath is void.

The lemma guarantees that, at each iteration step, arc-flags are set properly. The correctness of the bidirectional variant follows from the observation that a hop-minimal shortest path on a graph is also a hop-minimal shortest path on the reverse graph.

Proof. We show the claim by induction on the iteration steps. The claim holds trivially for $i = 0$. The inductive step works as follows: Assume the claim holds for step i. Given arbitrary nodes s and t, for which there is a path from s to t in G_0. We denote by $P = (v_1, \ldots, v_{j_1}; u_1, \ldots, u_{j_2}; w_1, \ldots, w_{j_3})$ the s-t-path according to the lemma for step i.

The iteration step $i+1$ consists of the contraction phase, the insertion of boundary shortcuts in case $i+1 = L-1$, the arc-flag computation and the pruning phase. We consider the phases one after another:

After the Contraction Phase. There exists a maximal path $(u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d})$ with $1 \leq \ell_1, \leq \ldots \leq \ell_d \leq k$ for which

- for each $f = 1, \ldots, d-1$ either $\ell_f + 1 = \ell_{f+1}$ or the subpaths $(u_{\ell_f}, u_{\ell_f+1}, \ldots u_{\ell_{f+1}})$ have been replaced by a shortcut,
- the nodes $u_1, \ldots, u_{\ell_1-1}$ have been deleted, if $\ell_1 \neq 1$ and
- the nodes $u_{\ell_d+1}, \ldots, u_k$ have been deleted, if $\ell_d \neq k$.

By the construction of the contraction routine we know

- $(u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d})$ is also a shortest path

- u_{ℓ_d} is in the same component as u_k in all levels greater than i (because of cell aware contraction)

- the deleted edges in $(u_1, \ldots, u_{\ell_1 - 1})$ either already have their arc-flags for the path P assigned. Then the arc-flags are correct because of the inductive hypothesis. Otherwise, We know that the nodes $u_1, \ldots, u_{\ell_1 - 1}$ are in the component. Hence, all arc-flags for all higher levels are assigned `true`.

- the deleted edges in $(u_{\ell_d+1}, \ldots, u_k)$ either already have their arc-flags for the path P assigned, then arc-flags are correct because of the inductive hypothesis. Otherwise, by cell-aware contraction we know that $u_{\ell_d+1}, \ldots, u_k$ are in the same component as t for all levels at least i. As the own-cell flag always is set `true` for deleted edges the path stays valid.

As distances do not change during preprocessing we know that, for arbitrary i, $0 \leq i \leq L-1$ a shortest path in G_i is also a shortest path in $\bigcup_{k=0}^{L-1} G_k$. Concluding, the path $\hat{P} = (v_1, \ldots, v_{j_1}, u_1, \ldots, u_{\ell_1 - 1}; u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d};\ u_{\ell_d+1}, \ldots, u_k, w_1, \ldots, w_{j_3})$ fullfills all claims of the lemma for iteration step $i+1$.

After Insertion of Boundary Shortcuts. Here, the claim holds trivially.

After Arc-Flags Computation. Here, the claim also holds trivially.

After Pruning. We consider the path $\hat{P}$ obtained from the contraction step. Let (u_{l_r}, u_{l_r+1}) be an edge of $\hat{P}$ deleted in the pruning step, for which u_{l_r} is not in the same cell as u_{l_d} at level $i+1$. As there exists a shortest path to u_{l_d} not only the own-cell flag of (u_{l_r}, u_{l_r+1}) is set, which is a contradiction to the assumption that (u_{l_r}, u_{l_r+1}) has been deleted in the pruning step.

Furthermore, let (u_{l_z}, u_{l_z+1}) be an edge of P deleted in the pruning step. Then, all edges on P after (u_{l_z}, u_{l_z+1}) are also deleted in that step. Summarizing, if no edge on $\hat{P}$ is deleted in the pruning step, then $\hat{P}$ fullfills all claims of the lemma for iteration step $i+1$. Otherwise, the path $(v_1, \ldots, v_{j_1}, u_1, \ldots, u_{\ell_1 - 1}; u_{\ell_1}, u_{\ell_2}, \ldots; u_{l_k}, \ldots, u_{\ell_d}, u_{\ell_d+1}, \ldots, u_k, w_1, \ldots, w_{j_3})$ fullfills all claims of the lemma for iteration step $i+1$ where u_{l_k}, u_{l_k+1} is the first edge on P that has been deleted in the pruning step.

Summarizing, Lemma A.1 holds during all phases of all iteration steps of SHARC-preprocessing. So, the preprocessing algorithm (without the refinement phase) is correct. □

A.2 Refinement phase. Recall that the own-cell flag does not get altered by the refinement routine. Hence, we only have to consider flags for other cells. Assume we perform the propagation routine at a level l to a level l node s.

A path P from s to a node t in another cell on level $\geq l$ needs to contain a level $> l$ node that is in the same cell as u because of the cell-aware contraction. Moreover, with iterated application of Lemma A.1 we know that there must be an (arc-flag valid) shortest s-t-path P for which the sequence of the levels of the nodes first is monotonically ascending and then monotonically descending. In fact, to cross a border of the current cell at level l, at least two level $> l$ nodes are on P. We consider the first level $> l$ node u_1 on P. This must be an entry node of s. The node u_2 after u_1 on P is covered and therefore no entry node. Furthermore it is of level $> l$. Hence, the flags of the edge (u_1, u_2) are propagated to the first edge on P and the claim holds which proves that the refinement phase is correct. Together with Lemma A.1 and the correctness of the multi-level Arc-Flags query, SHARC-Routing is correct.

Obtaining Optimal k-Cardinality Trees Fast

Markus Chimani* Maria Kandyba*† Ivana Ljubić‡§ Petra Mutzel*

Abstract

Given an undirected graph $G = (V, E)$ with edge weights and a positive integer number k, the k-Cardinality Tree problem consists of finding a subtree T of G with exactly k edges and the minimum possible weight. Many algorithms have been proposed to solve this NP-hard problem, resulting in mainly heuristic and metaheuristic approaches.

In this paper we present an exact ILP-based algorithm using directed cuts. We mathematically compare the strength of our formulation to the previously known ILP formulations of this problem, and give an extensive study on the algorithm's practical performance compared to the state-of-the-art metaheuristics.

In contrast to the widespread assumption that such a problem cannot be efficiently tackled by exact algorithms for medium and large graphs (between 200 and 5000 nodes), our results show that our algorithm not only has the advantage of proving the optimality of the computed solution, but also often outperforms the metaheuristic approaches in terms of running time.

1 Introduction

We consider the *k-Cardinality Tree problem* (KCT): given an undirected graph $G = (V, E)$, an edge weight function $w : E \to \mathbb{R}$, and a positive integer number k, find a subgraph T of G which is a minimum weight tree with exactly k edges. This problem has been extensively studied in literature as it has various applications, e.g., in oil-field leasing, facility layout, open pit mining, matrix decomposition, quorum-cast routing, telecommunications, etc [9]. A large amount of research was devoted to the development of heuristic [5, 14] and, in particular, metaheuristic methods [4, 8, 11, 7, 25]. An often used argument for heuristic approaches is that exact methods for this NP-hard problem would require too much computation time and could only be applied to very small graphs [9, 10].

The problem also received a lot of attention in the approximation algorithm community [1, 3, 17, 18]: a central idea thereby is the primal-dual scheme, based on integer linear programs (ILPs), which was proposed by Goemans and Williamson [19] for the prize-collecting Steiner tree problem. An exact approach was presented by Fischetti et al. [15], by formulating an ILP based on general subtour elimination constraints (GSEC). This formulation was implemented by Ehrgott and Freitag [13] using a Branch-and-Cut approach. The resulting algorithm was only able to solve graphs with up to 30 nodes, which may be mainly due to the comparably weak computers in 1996.

In this paper we show that the traditional argument for metaheuristics over exact algorithms is deceptive on this and related problems. We propose a novel exact ILP-based algorithm which can indeed be used to solve all known benchmark instances of KCTLIB [6]—containing graphs of up to 5000 nodes—to provable optimality. Furthermore, our algorithm often, in particular on mostly all graphs with up to 1000 nodes, is faster than the state-of-the-art metaheuristic approaches, which can neither guarantee nor assess the quality of their solution.

To achieve these results, we present Branch-and-Cut algorithms for KCT and NKCT—the node-weighted variant of KCT. Therefore, we transform both KCT and NKCT into a similar directed and rooted problem called *k-Cardinality Arborescence problem* (KCA), and formulate an ILP for the latter, see Section 2. In the section thereafter, we provide polyhedral and algorithmic comparison to the known GSEC formulation. In Section 4, we describe the resulting Branch-and-Cut algorithm in order to deal with the exponential ILP size. We conclude the paper with the extensive experimental study in Section 5, where we compare our algorithm with the state-of-the-art metaheuristics for the KCT.

2 Directed Cut Approach

2.1 Transformation into the k-Cardinality Arborescence Problem. Let $D = (V_D, A_D)$ be a directed graph with a distinguished root vertex $r \in V_D$ and arc costs c_a for all arcs $a \in A_D$. The k-Cardinality Arborescence problem (KCA) consists of finding a weight minimum rooted tree T_D with k arcs

*Technical University of Dortmund; {markus.chimani, maria.kandyba, petra.mutzel}@cs.uni-dortmund.de

†Supported by the German Research Foundation (DFG) through the Collaborative Research Center "Computational Intelligence" (SFB 531)

‡University of Vienna; ivana.ljubic@univie.ac.at

§Supported by the Hertha-Firnberg Fellowship of the Austrian Science Foundation (FWF)

which is directed from the root outwards. More formally, T_D has to satisfy the following properties:

(P1) T_D contains exactly k arcs,

(P2) for all $v \in V(T_D) \setminus \{r\}$, there exists a directed path $r \to v$ in T_D, and

(P3) for all $v \in V(T_D) \setminus \{r\}$, v has in-degree 1 in T_D.

We transform any given KCT instance $(G = (V, E), w, k)$ into a corresponding KCA instance $(G_r, r, c, k+1)$ as follows: we replace each edge $\{i, j\}$ of G by two arcs (i, j) and (j, i), introduce an artificial root vertex r and connect r to every node in V. Hence we obtain a digraph $G_r = (V \cup \{r\}, A \cup A_r)$ with $A = \{(i, j), (j, i) \mid \{i, j\} \in E\}$ and $A_r = \{(r, j) \mid j \in V\}$. For each arc $a = (i, j)$ we define the cost function $c(a) := 0$ if $i = r$, and $c(a) := w(\{i, j\})$ otherwise.

To be able to interpret each feasible solution T_{G_r} of this resulting KCA instance as a solution of the original KCT instance, we impose an additional constraint

(P4) T_{G_r} contains only a single arc of A_r.

If this property is satisfied, it is easy to see that a feasible KCT solution with the same objective value can be obtained by removing r from T_{G_r} and interpreting the directed arcs as undirected edges.

2.2 The Node-weighted k-Cardinality Tree Problem. The Node-weighted k-Cardinality Tree problem (NKCT) is defined analogously to KCT but its weight function $w' : V \to \mathbb{R}$ uses the nodes as its basic set, instead of the edges (see, e.g., [10] for the list of references). We can also consider the general All-weighted k-Cardinality Tree problem (AKCT), where a weight-function w for the edges, and a weight-function w' for the nodes are given.

We can transform any NKCT and AKCT instance into a corresponding KCA instance using the ideas of [24]: the solution of KCA is a rooted, directed tree where each vertex (except for the unweighted root) has in-degree 1. Thereby, a one-to-one relationship between each selected arc and its target node allows us to precompute the node-weights into the arc-weights of KCA: for all $(i, j) \in A \cup A_r$ we have $c((i, j)) := w'(j)$ for NKCT, and $c((i, j)) := w(\{i, j\}) + w'(j)$ for AKCT.

2.3 ILP for the KCA. In the following let the graphs be defined as described in Section 2.1. To model KCA as an ILP, we introduce two sets of binary variables:

$$x_a, y_v \in \{0, 1\} \quad \forall a \in A \cup A_r, \forall v \in V$$

Thereby, the variables are 1, if the corresponding vertex or arc is in the solution and 0 otherwise.

Let $S \subseteq V$. The sets $E(S)$ and $A(S)$ are the edges and arcs of the subgraphs of G and G_r, respectively, induced by S. Furthermore, we denote by $\delta^+(S) = \{(i, j) \in A \cup A_r \mid i \in S, j \in V \setminus S\}$ and $\delta^-(S) = \{(i, j) \in A \cup A_r \mid i \in V \setminus S, j \in S\}$ the outgoing and ingoing edges of a set S, respectively. We can give the following ILP formulation, using $x(B) := \sum_{b \in B} x_b$, with $B \subseteq A$, as a shorthand:

$$\text{(2.1)} \qquad \text{DCut}: \qquad \min \sum_{a \in A} c(a) \cdot x_a$$

$$\text{(2.2)} \quad x(\delta^-(S)) \geq y_v \qquad \forall S \subseteq V \setminus \{r\}, \forall v \in S$$

$$\text{(2.3)} \quad x(\delta^-(v)) = y_v \qquad \forall v \in V$$

$$\text{(2.4)} \quad x(A) = k$$

$$\text{(2.5)} \quad x(\delta^+(r)) = 1$$

$$\text{(2.6)} \quad x_a, y_v \in \{0, 1\} \qquad \forall a \in A \cup A_r, \forall v \in V$$

The *dcut-constraints* (2.2) ensure property (P2) via directed cuts, while property (P3) is ensured by the in-degree constraints (2.3). Constraint (2.4) ensures the k-cardinality requirement (P1) and property (P4) is modeled by (2.5).

Lemma 2.1. *By replacing all in-degree constraints* (2.3) *by a single node-cardinality constraint*

$$\text{(2.7)} \qquad y(V) = k + 1,$$

we obtain an equivalent ILP and an equivalent LP-relaxation.

Proof. The node-cardinality constraint can be generated directly from (2.3) and (2.4), (2.5). Vice versa, we can generate (2.3) from (2.7), using the dcut-constraints (2.2). □

Although the formulation using (2.7) requires less constraints, the ILP using in-degree constraints has certain advantages in practice, see Section 4.

3 Polyhedral Comparison

In [15], Fischetti et al. give an ILP formulation for the undirected KCT problem based on general subtour elimination constraints (Gsec). We reformulate this approach and show that both Gsec and DCut are equivalent from the polyhedral point of view.

In order to distinguish between undirected edges and directed arcs we introduce the binary variables $z_e \in \{0, 1\}$ for every edge $e \in E$, which are 1 if $e \in T$ and 0 otherwise. For representing the selection of the

nodes we use the y-variables as in the previous section. The constraints (3.9) are called the *gsec-constraints*.

$$\text{(3.8)} \qquad \textsc{Gsec}: \qquad \min \quad \sum_{e\in E} c(e)\cdot z_e$$

$$\text{(3.9)} \quad z(E(S)) \le y(S\setminus\{t\}) \quad \forall S\subseteq V, |S|\ge 2, \forall t\in S$$

$$\text{(3.10)} \qquad z(E) = k$$

$$\text{(3.11)} \qquad y(V) = k+1$$

$$\text{(3.12)} \qquad z_e, y_v \in \{0,1\} \qquad \forall e\in E, \forall v\in V$$

Let $\mathcal{P}_D$ and $\mathcal{P}_G$ be the polyhedra corresponding to the DCut and Gsec LP-relaxations, respectively. I.e.,

$$\mathcal{P}_D := \{ \quad (x,y)\in\mathbb{R}^{|A\cup A_r|+|V|} \mid 0\le x_e, y_v \le 1 \text{ and } (x,y) \text{ satisfies (2.2)–(2.5)} \quad \}$$

$$\mathcal{P}_G := \{ \quad (z,y)\in\mathbb{R}^{|E|+|V|} \mid 0\le z_e, y_v \le 1 \text{ and } (z,y) \text{ satisfies (3.9)–(3.11)} \quad \}$$

Theorem 3.1. *The* Gsec *and the* DCut *formulations have equally strong LP-relaxations, i.e.,*

$$\mathcal{P}_G = \mathrm{proj}_z(\mathcal{P}_D),$$

whereby $\mathrm{proj}_z(\mathcal{P}_D)$ *is the projection of* $\mathcal{P}_D$ *onto the* (z,y) *variable space with* $z_{\{i,j\}} = x_{(i,j)} + x_{(j,i)}$ *for all* $\{i,j\}\in E$.

Proof. We prove equality by showing mutual inclusion:

- $\mathrm{proj}_z(\mathcal{P}_D)\subseteq\mathcal{P}_G$**:** Any $(\bar z,\bar y)\in\mathrm{proj}_z(\mathcal{P}_D)$ satisfies (3.10) by definition, and (3.11) by (2.3) and Lemma 2.1. Let $\bar x$ be the vector from which we projected the vector $\bar z$, and consider some $S\subseteq V$ with $|S|\ge 2$ and some vertex $t\in S$. We show that $(\bar z,\bar y)$ also satisfies the corresponding gsec-constraint (3.9):

$$\bar z(E(S)) = \bar x(A(S)) = \textstyle\sum_{v\in S}\bar x(\delta^-(v)) - \bar x(\delta^-(S)) \overset{(2.3)}{=} \bar y(S) - \bar x(\delta^-(S)) \overset{(2.2)}{\le} \bar y(S) - \bar y_t.$$

- $\mathcal{P}_G\subseteq\mathrm{proj}_z(\mathcal{P}_D)$**:** Consider any $(\bar z,\bar y)\in\mathcal{P}_G$ and a set

$$X := \{ \quad x\in\mathbb{R}^{|A\cup A_r|}_{\ge 0} \mid x \text{ satisfies (2.5) and } x_{ij}+x_{ji} = \bar z_{\{ij\}}\ \forall (i,j)\in A \quad \}.$$

  Every such projective vector $\bar x\in X$ clearly satisfies (2.4). In order to generate the dcut-inequalities (2.2) for the corresponding $(\bar x,\bar y)$, it is sufficient to show that we can always find an $\hat x\in X$, which together with $\bar y$ satisfies the indegree-constraints (2.3). Since then, for any $S\subseteq V$ and $t\in S$:

$$\hat x(\delta^-(S)) = \textstyle\sum_{v\in S}\hat x(\delta^-(v)) - \hat x(A(S)) \overset{(2.3)}{=} \bar y(S) - \bar z(E(S)) \overset{(3.9)}{\ge} \bar y_t.$$

  We show the existence of such an $\hat x$ using a proof technique similar to [20, proof of Claim 2], where it was used for the Steiner tree problem.

  An $\hat x\in X$ satisfying (2.3) can be interpreted as the set of feasible flows in a bipartite transportation network (N,L), with $N := (E\cup\{r\})\cup V$. For each undirected edge $e=(u,w)\in E$ in G, our network contains exactly two outgoing arcs $(e,u),(e,w)\in L$. Furthermore, L contains all arcs of A_r. For all nodes $e\in E$ in N we define a supply $s(e) := \bar z_e$; for the root r we set $s(r) := 1$. For all nodes $v\in V$ in N we define a demand $d(v) := \bar y_v$.

  Finding a feasible flow for this network can be viewed as a capacitated transportation problem on a complete bipartite network with capacities either zero (if the corresponding edge does not exist in L) or infinity. Note that in our network the sum of all supplies is equal to the sum of all demands, due to (3.10) and (3.11). Hence, each feasible flow in such a network will lead to a feasible $\hat x\in X$. Such a flow exists if and only if for every set $M\subseteq N$ with $\delta^+_{(N,L)}(M)=\emptyset$ the condition

$$\text{(3.13)} \qquad s(M)\le d(M)$$

  is satisfied, whereby $s(M)$ and $d(M)$ are the total supply and the total demand in M, respectively, cf. [16, 20]. In order to show that this condition holds for (N,L), we distinguish between two cases; let $U := E\cap M$:

  $r\in M$**:** Since r has an outgoing arc for every $v\in V$ and $\delta^+_{(N,L)}(M)=\emptyset$, we have $V\subset M$. Condition (3.13) is satisfied, since $s(r)=1$ and therefore:

$$s(M) = s(r) + \bar z(U) \le s(r) + \bar z(E) = \bar z(E) + 1 \overset{(3.10),(3.11)}{=} \bar y(V) = d(M).$$

  $r\notin M$**:** Let $S := V\cap M$. We then have $U\subseteq E(S)$. If $|S|\le 1$ we have $U=\emptyset$ and therefore (3.13) is automatically satisfied. For $|S|\ge 2$, the condition is also satisfied, since for every $t\in S$ we have:

$$s(M) = \bar z(U) \le \bar z(E(S)) \overset{(3.9)}{\le} \bar y(S) - \bar y_t \le \bar y(S) = d(M). \qquad \square$$

3.1 Other approaches.

3.1.1 Multi-Commodity Flow. One can formulate a multi-commodity-flow based ILP for KCA (MCF) as it was done for the prize-collecting Steiner tree problem (PCST) [22], and augment it with cardinality inequalities. Analogously to the proof in [22], which shows the equivalence of DCUT and MCF for PCST, we can obtain:

LEMMA 3.1. *The LP-relaxation of* MCF *for KCA is equivalent to* GSEC *and* DCUT.

Nonetheless, we know from similar problems [12, 23] that directed-cut based approaches are usually more efficient than multi-commodity flows in practice.

3.1.2 Undirected Cuts for Approximation Algorithms. In [17], Garg presents an approximation algorithm for KCT, using an ILP for lower bounds (GUCUT). It is based on undirected cuts and has to be solved $|V|$ times, once for all possible choices of a root node r.

LEMMA 3.2. DCUT *is stronger than* GUCUT.

Proof. Clearly, each feasible point in $\mathcal{P}_D$ is feasible in the LP-relaxation of GUCUT using the projection proj_z. On the other hand, using a traditional argument, assume a complete graph on 3 nodes is given, where each vertex variable is set to 1, and each edge variable is set to 0.5. This solution is feasible for the LP-relaxation of GUCUT, but infeasible for DCUT. □

4 Branch-and-Cut Algorithm

Based on our DCUT formulation, we developed and implemented a Branch-and-Cut algorithm. For a general description of the Branch-and-Cut scheme see, e.g., [27]: Such algorithms start with solving an *LP relaxation*, i.e., the ILP without the integrality properties, only considering a certain subset of all constraints. Given the *fractional solution* of this partial LP, we perform a *separation routine*, i.e., identify constraints of the full constraint set which the current solution violates. We then add these constraints to our current LP and reiterate these steps. If at some point we cannot find any violated constraints, we have to resort to *branching*, i.e., we generate two disjoint subproblems, e.g., by fixing a variable to 0 or 1. By using the LP relaxation as a lower bound, and some heuristic solution as an upper bound, we can prune irrelevant subproblems.

In [13], a Branch-and-Cut algorithm based on the GSEC formulation has been developed. Note that the dcut-constraints are sparser than the gsec-constraints, which in general often leads to a faster optimization in practice. This conjecture was experimentally confirmed, e.g., for the similar prize-collecting Steiner tree problem [23], where a directed-cut based formulation was compared to a GSEC formulation. The former was both faster in overall running time and required less iterations, by an order of 1–2 magnitudes. Hence we can expect our DCUT approach to have advantages over GSEC in practice. In Section 4.2 we will discuss the formal differences in the performances between the DCUT and the GSEC separation algorithms.

4.1 Initialization. Our algorithm starts with the constraints (2.3), (2.4), and (2.5). We prefer the indegree constrains (2.3) over the node-cardinality constraint (2.7), as they strengthen the initial LP and we do not require to separate dcut-constraints with $|S| = 1$ later.

For the same reason, we add the orientation-constraints

$$x_{ij} + x_{ji} \leq y_i \quad \forall i \in V, \forall \{i,j\} \in E \tag{4.14}$$

to our initial ILP. Intuitively, these constraints ensure a unique orientation for each edge, and require for each selected arc that both incident nodes are selected as well. These constraints do not actually strengthen the DCUT formulation as they represent the gsec-constraints for all two-element sets $S = \{i,j\} \subset V$. From the proof of Theorem 3.1, we know that these inequalities can be generated with the help of (2.3) and (2.2). Nonetheless, as experimentally shown in [22] for PCST and is also confirmed by our own experiments, the addition of (4.14) speeds up the algorithm tremendously, as they do not have to be separated explicitly by the Branch-and-Cut algorithm.

We also tried *asymmetry constraints* [22] to reduce the search space by excluding symmetric solutions:

$$x_{rj} \leq 1 - y_i \quad \forall i,j \in V, i < j. \tag{4.15}$$

They assure that for each KCA solution, the vertex adjacent to the root is the one with the smallest possible index. Anyhow, we will see in our experiments that the quadratic number of these constraints becomes a hindrance for large graphs and/or small k in practice.

4.2 Separation. The dcut-constraints (2.2) can be separated in polynomial time via the traditional maximum-flow separation scheme: we compute the maximum-flow from r to each $v \in V$ using the edge values of the current solution as capacities. If the flow is less than y_v, we extract one or more of the induced minimum (r,v)-cuts and add the corresponding constraints to our model. In order to obtain more cuts

with a single separation step we also use nested- and back-cuts [21, 23]. Indeed, using these additional cuts significantly speeds up the computation.

Recall that in a general separation procedure we search for the most violated inequality of the current LP-relaxation. In order to find the most violated inequality of the DCut formulation, or to show that no such exists, we construct the flow network only once and perform at most $|V|$ maximum-flow calculations on it. This is a main reason why the DCut formulation performs better than Gsec in practice: a single separation step for Gsec requires $2|V| - 2$ maximum-flow calculations, as already shown by Fischetti et al. [15]. Furthermore, the corresponding flow network is not static over all those calculations, but has to be adapted prior to each call of the maximum-flow algorithm.

Our test sets, as described in Section 5, also contain grid graphs. In such graphs, it is easy to detect and enumerate all 4-cycles by embedding the grids into the plane and traversing all faces except for the single large one. Note that due to our transformation, all 4-cycles are bidirected. Let $\mathcal{C}_4$ be the set of all bidirected 4-cycles; a cycle $C \in \mathcal{C}_4$ then consists of 8 arcs and $V[C]$ gives the vertices on C. We use a separation routine for gsec-constraints on these cycles:

$$\sum_{a \in C} x_a \leq \sum_{i \in V[C] \setminus \{v\}} y_i \quad \forall C \in \mathcal{C}_4, \forall v \in V[C]. \tag{4.16}$$

4.3 Upper Bounds and Proving Optimality. In the last decade, several heuristics and metaheuristics have been developed for KCT. See, e.g., [4, 5, 8, 11] for an extensive comparison. Traditional Branch-and-Cut algorithms allow to use such algorithms as primal heuristics, giving upper bounds which the Branch-and-Cut algorithm can use for bounding purposes when branching. The use of such heuristics is two-fold: (a) they can be used as start-heuristics, giving a good initial upper bound before starting the actual Branch-and-Cut algorithm, and (b) they can be run multiple times during the exact algorithm, using the current fractional solutions as an additional input, or *hint*, in order to generate new and tighter upper bounds on the fly.

Let h be a primal bound obtained by such a heuristic. Mathematically, we can add this bound to our LP as

$$\sum_{a \in A} c(a) \cdot x_a \leq h - \Delta.$$

Thereby, $\Delta := \min\{c(a) - c(b) \mid c(a) > c(b),\ a, b \in A\}$ denotes the minimal difference between any two cost values. If the resulting ILP is found to be infeasible, we have a proof that h was optimal, i.e., the heuristic solution was optimal.

As our experiments reveal, our algorithm is already very successful without the use of any such heuristic. Hence we compared our heuristic-less Branch-and-Cut algorithm (DC^-) with one using a *perfect heuristic*: a (hypothetical) algorithm that requires no running time and gives the optimal solution. We can simulate such a perfect heuristic by using the optimal solution obtained by a prior run of DC^-. We can then measure how long the algorithm takes to discover the infeasibility of the ILP. We call this algorithm variant DC^+. If the runtime performance of DC^- and DC^+ are similar, we can conclude that using any heuristic for bounding is not necessary.

5 Experimental results

We implemented our algorithm in C++ using CPLEX 9.0 and LEDA 5.0.1. The experiments were performed on 2.4 GHz AMD Opteron with 2GB RAM per process. We tested our algorithm on all instances of the KCTLIB [6] which consists of the following benchmark sets:

(BX) The set by Blesa and Xhafa [2] contains 35 4-regular graphs with 25–1000 nodes. The value of k is fixed to 20. The results of [8] have already shown that these instances are easy, which was confirmed by our experiments: our algorithm needed on average 1.47 seconds per instance to solve them to optimality, the median was 0.09 seconds.

(BB) The set by Blesa and Blum [8] is divided into four subsets of dense, sparse, grid and 4-regular graphs, respectively, with different sizes of up to 2500 nodes. Each instance has to be solved for different values of k, specified in the benchmark set: these are k_{rel} of $n = |V|$, for $k_{\text{rel}} = \{10\%, \ldots, 90\%\}$[1], and additionally $k = 2$ and $k = n - 2$. Note that the latter two settings are rather insignificant for our analysis, as they can be solved optimally via trivial algorithms in quadratic time.

The most successful known metaheuristics for (BB) are the hybrid evolutionary algorithm (HyEA) [4] and the ant colony optimization algorithm (ACO) [11].

(UBM) The set by Urošević et al. [25] consists of large 20-regular graphs with 500–5000 nodes which were originally generated randomly. The values for k are defined as for (BB) by using $k_{\text{rel}} = \{10\%, \ldots, 50\%\}$. In [25] a variable neighborhood decomposition search (VNDS) was presented, which is still the best known metaheuristic for this benchmark set.

[1] For the grid instances, the values k_{rel} differ slightly.

# of nodes	500	1000	1500	2000	3000	4000	5000
avg. time in sec.	7.5	48.3	107.4	310.7	1972	5549	15372.2
avg. gap of BKS	1.5%	0.1%	0.1%	0.2%	0.2%	0.3%	0.3%

Table 1: Average running times and average gap to the BKS provided in [5, 7, 25] for (UBM).

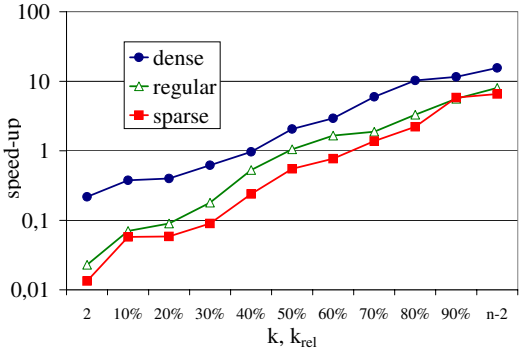


Figure 1: Speed-up factors for dense, regular and sparse graphs with $|V| < 2000$ obtained when asymmetry constraints (4.15) are included in the initial LP

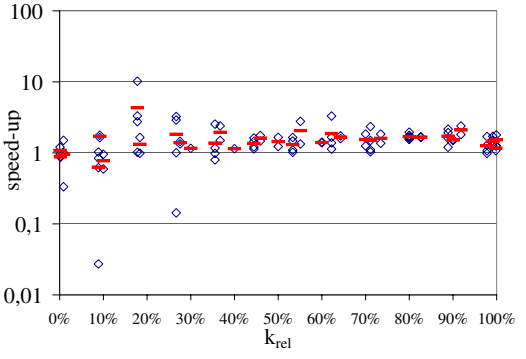


Figure 2: Speed-up factors for the grid instances of (BB) when gsec-constraints (4.16) are separated. For each instance and k_{rel} value there is a diamond-shaped datapoint; the short horizontal bars denote the average speed-up per k_{rel}.

Our computational experiments on (UBM) show that all instances with up to 3000 nodes can be solved to optimality within two hours. We are also able to solve the graphs with 4000 and 5000 nodes to optimality, although only about 50% of them in less than two hours. Note that for these large instances the VNDS metaheuristic of [25] is faster than our algorithm, however they thereby could not reach optimal solutions. Table 1 gives the average running times and the differences between the optimal solutions and the previously best known solutions (BKS).

In the following we will concentrate on the more common and diversified benchmark set (BB), and compare our results to those of HyEA and ACO. Unless specified otherwise, we always report on the DC$^-$ algorithm, i.e., the Branch-and-Cut algorithm without using any heuristic for upper bounds.

Algorithmic Behaviour. Figure 1 illustrates the effectiveness of the asymmetry constraints (4.15) depending on increasing relative cardinality k_{rel}. Therefore we measured the speed-up by the quotient $\frac{t_\emptyset}{t_{\text{asy}}}$, whereby t_{asy} and $t_\emptyset$ denote the running time with and without using (4.15), respectively. The constraints allow a speed-up by more than an order of magnitude for sparse, dense and regular graphs, but only for large cardinality $k > \frac{n}{2}$. Our experiments show that for smaller k, a variable x_{ri}, for some $i \in V$, is quickly set to 1 and stays at this value until the final result. In these cases the constraints cannot help and only slow down the algorithm. Interestingly, the constraints were never profitable for the grid instances. For graphs with more than 2000 nodes using (4.15) is not possible due to memory restrictions, as the $\mathcal{O}(|V|^2)$ many asymmetry constraints are too much to handle. Hence, we ommitted these graphs in our figure.

We also report on the experiments with the special gsec-constraints (4.16) within the separation routine for the grid graphs. The clear advantage of these constraints is shown in Figure 2, which shows the obtained speed-up factor $\frac{t_\emptyset}{t_{\text{gsec}}}$ by the use of these constraints.

Based on these results we choose to include the assymmetry constraints for all non-grid instances with less then 2000 nodes and $k > \frac{n}{2}$, in all the remaining experiments. For the grid instances we always separate the gsec-constraints (4.16).

In Table 2, we show that the computation time is not only dependent on the graph size, but also on the density of the graph. Generally, we leave table cells empty if there is no problem instance with according properties.

As described in Section 4.3, we also investigate the influence of primal heuristics on our Branch-and-Cut algorithm. For the tested instances with 1000 nodes the comparison of the running times of DC$^+$ and DC$^-$ is shown in Figure 3. In general, our experiments show

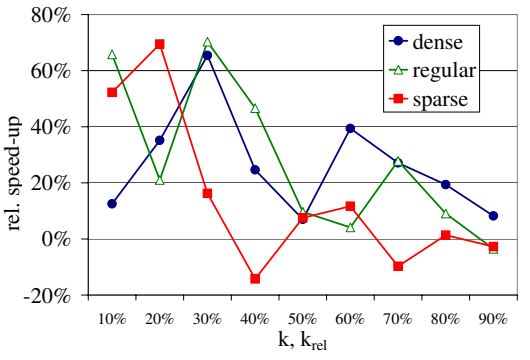


Figure 3: Relative speed-up $\frac{(t_{\mathrm{DC}^-} - t_{\mathrm{DC}^+})}{t_{\mathrm{DC}^-}}$ (in percent) of DC^+ compared to DC^- for the instances with 1000 nodes.

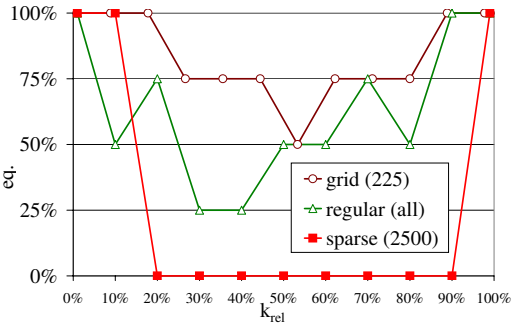


Figure 4: Dependancy of the BKS quality on k_{rel}, for selected instances. The vertical axis gives the percentage of the tested instances for which the BKS provided in [6] are optimal.

avg. deg	set	500 nodes	1000 nodes
2.5	(BB)	1	8.1
4	(BB)	0.9	15.7
10	(BB)	2.6	25
20	(UBM)	7.5	48.4
36.3	(BB)	10.7	—

Table 2: Average CPU time (in seconds) over k_{rel} values of 10%, 20%, ..., 50%, sorted by the average degree of the graphs.

that DC^+ is only 10–30% percent faster than DC^- on average, even for the large graphs. Hence, we can conclude that a bounding heuristic is not crucial for the success of our algorithm.

Runtime Comparison. Table 3 summarizes the average and median computation times of our algorithm, sorted by size and categorized according to the special properties of the underlying graphs. We can observe that performance does not differ significantly between the sparse, regular and dense graphs, but that the grid instances are more difficult and require more computational power. This was also noticed in [9, 10, 14].

The behaviour of DC^- also has a clear dependency on k, see Figures 5(a), 5(c) and 5(d): for the sparse, dense and regular instances the running time increases with increasing k. In contrast to this, solving the grid instances (cf. Figure 5(b)) is more difficult for the relatively small k-values.

The original experiments for HyEA and ACO were performed on an Intel Pentium IV, 3.06 GHz with 1GB RAM and a Pentium IV 2.4 GHz with 512MB RAM, respectively. Using the well-known SPEC performance evaluation [26], we computed scaling factors of both machines to our computer: for the running time comparison we divided the times given in [4] and [11] by 1.5 and 2, respectively. Anyhow, note that these factors are elaborate guesses and are only meant to help the reader to better evaluate the relative performance.

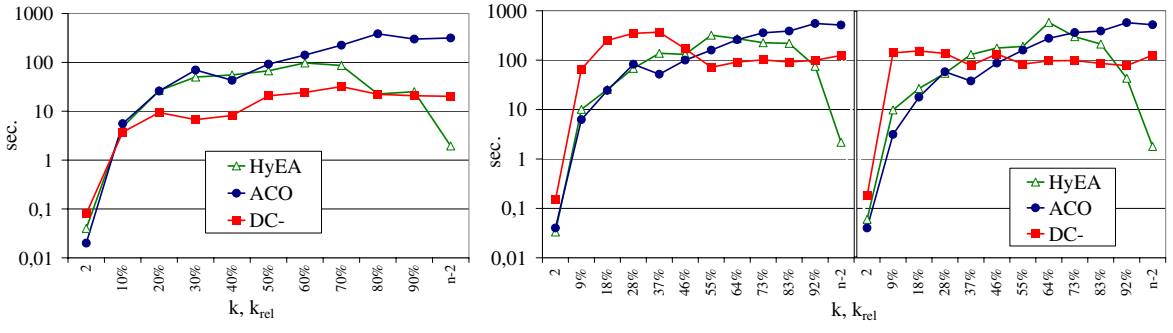
Table 3 additionally gives the average factor of $\frac{t_{\mathrm{HyEA}}}{t_{\mathrm{DC}^-}}$, i.e., the running time of our algorithm compared to (scaled) running time of HyEA. Analogously, Figure 5 shows the CPU time in (scaled) seconds of HyEA, ACO and our algorithm.

We observe that our DC^- algorithm performs better than the best metaheuristics in particular for the medium values of k, i.e., $40-70\%$ of $|V|$, on all instances with up to 1089 nodes, except for the very dense graph `le450_15a.g` with 450 nodes and 8168 edges, where HyEA was slightly faster. Interestingly, the gap between the heuristic and the optimal solution tended to be larger especially for medium values of k (cf. next paragraph and Figure 4 for details).

Solution Quality. For each instance of the sets (BX) and (BB) we compared the previously best known solutions, see [6], with the optimal solution obtained by our algorithm, in order to assess their quality. Most of the BKS were found by HyEA, followed by ACO. Note that these solutions where obtained by taking the best solutions over 20 independent runs per instance. In Table 4 we show the number of instances for which we proved that BKS was in fact not optimal, and give the corresponding average gap $\mathrm{gap}_{\mathrm{bks}} := \frac{\mathrm{BKS}-\mathrm{OPT}}{\mathrm{OPT}}$ (in percent), where OPT denotes the optimal objective value obtained by DC^- and BKS denotes the best known solution obtained by either ACO or HyEA. Analogously, we give the average gaps $\mathrm{gap}_{\mathrm{avg}} := \frac{\mathrm{AVG}-\mathrm{OPT}}{\mathrm{OPT}}$ (in percent), AVG denotes the average solution obtained by a metaheuristic. We observe that—concerning the solution quality—metaheuristics work quite well on instances with up to 1000 nodes and relatively small k.

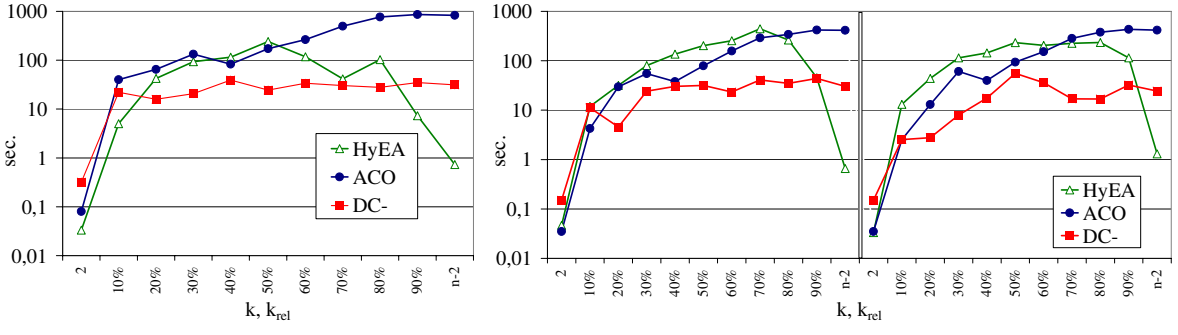
# nodes	500		1000–1089		2500	
group	**avg/med**	$\frac{t_{\text{HyEA}}}{t_{\text{DC}-}}$	**avg/med**	$\frac{t_{\text{HyEA}}}{t_{\text{DC}-}}$	**avg/med**	$\frac{t_{\text{HyEA}}}{t_{\text{DC}-}}$
sparse	1.7/2.0	2.2	15.2/20.2	2.6	923.2/391.5	0.1
regular	1.7/1.5	3.1	22.2/21.4	5.7	—	—
dense	7.5/7.9	2.2	25.5/27.9	2.7	—	—
grid	11.7/1.2	0.1	124.8/98.7	1.1	3704.1/2800.1	0.1

Table 3: Average/median CPU time (in seconds) and the average speed-up factor of DC$^-$ to HyEA for the instance set (BB). Cells are left empty if there exists no instance matching the given criteria.



(a) sparse (1000 nodes, 1250 edges)

(b) grid 33x33 (1089 nodes, 2112 edges)



(c) dense (1000 nodes, 1250 edges)

(d) 4-regular (1000 nodes, 2000 edges)

Figure 5: Running times of DC$^-$, HyEA, and ACO (in seconds) for instances of (BB) with ∼1000 nodes, depending on k. The figures for the grid and regular instances show the times for two different instances of the same type, respectively.

instance	($\|V\|$,$\|E\|$)	eq.	$\mathbf{gap}_{\text{bks}}$	$\mathbf{gap}_{\text{avg}}$ **ACO**	$\mathbf{gap}_{\text{avg}}$ **HyEA**
regular g400-4-1.g	(400,800)	10/11	0.09	0.07	0.04
regular g400-4-5.g	(400,800)	8/11	0.19	0.31	0.35
regular g1000-4-1.g	(1000,2000)	7/11	0.07	0.65	0.12
regular g1000-4-5.g	(1000,2000)	3/11	0.08	0.45	0.35
sparse steinc5.g	(500,625)	11/11	–	0.97	0.06
sparse steind5.g	(1000,1250)	11/11	–	0.48	0.11
sparse steine5.g	(2500,3125)	3/11	0.13	n/a	0.23
dense le450a.g	(450,8168)	11/11	–	n/a	0.04
dense steinc15.g	(500,2500)	11/11	–	0.36	0.02
dense steind15.g	(1000,5000)	10/11	0.22	0.38	0.04
grid 15x15-1	(225,400)	13/13	–	1.27	0.18
grid 15x15-2	(225,400)	13/13	–	2.04	0.12
grid 45x5-1	(225,400)	4/13	0.54	n/a	1.22
grid 45x5-2	(225,400)	10/13	0.08	n/a	0.13
grid 33x33-1	(1089,2112)	3/12	0.31	1.70	0.57
grid 33x33-2	(1089,2112)	3/12	0.39	2.48	0.49
grid 50x50-1	(2500,4900)	2/11	0.95	n/a	1.27
grid 50x50-2	(2500,4900)	2/11	0.55	n/a	0.82

Table 4: Quality of previously best known solutions (BKS) provided in [6] for selected instances. "**eq.**" denotes the number of instances for which the BKS was optimal. For the other instances where BKS was not optimal, we give the average relative gap ($\mathbf{gap}_{\text{bks}}$) between OPT and BKS. For all instances we also give the average relative gap ($\mathbf{gap}_{\text{avg}}$) between the average solution of the metaheuristic and OPT. All gaps are given in percent. Cells marked as "n/a" cannot be computed as the necessary data for ACO is not not available.

In particular, for $k = 2$ and $k = n - 2$ they always found an optimal solution.

Acknowledgements. We would like to thank Christian Blum and Dragan Urošević for kindly providing us with test instances and sharing their experience on this topic.

References

[1] S. Arora and G. Karakostas. A $(2 + \epsilon)$-approximation algorithm for the k-MST problem. In *Proc. of the eleventh annual ACM-SIAM symposium on Discrete algorithms (SODA'00)*, pages 754–759, 2000.

[2] M. J. Blesa and F. Xhafa. A C++ Implementation of Tabu Search for k-Cardinality Tree Problem based on Generic Programming and Component Reuse. In c/o tranSIT GmbH, editor, *Net.ObjectDsays 2000 Tagungsband*, pages 648–652, Erfurt, Germany, 2000. Net.ObjectDays-Forum.

[3] A. Blum, R. Ravi, and S. Vempala. A constant-factor approximation algorithm for the k-MST problem. In *ACM Symposium on Theory of Computing*, pages 442–448, 1996.

[4] C. Blum. A new hybrid evolutionary algorithm for the huge k-cardinality tree problem. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'06)*, pages 515–522, New York, NY, USA, 2006. ACM Press.

[5] C. Blum. Revisiting dynamic programming for finding optimal subtrees in trees. *European Journal of Operational Research*, 177(1):102–115, 2007.

[6] C. Blum and M. Blesa. KCTLIB – a library for the edge-weighted k-cardinality tree problem. `http://iridia.ulb.ac.be/ cblum/kctlib/`.

[7] C. Blum and M. Blesa. Combining ant colony optimization with dynamic programming for solving the k-cardinality tree problem. In *Proc. of Computational Intelligence and Bioinspired Systems, 8th International Work-Conference on Artificial Neural Networks, (IWANN'05)*, volume 3512 of *Lecture Notes in Computer Science*, Barcelona, Spain, 2005. Springer.

[8] C. Blum and M. J. Blesa. New metaheuristic approaches for the edge-weighted k-cardinality tree problem. *Computers & OR*, 32:1355–1377, 2005.

[9] C. Blum and M. Ehrgott. Local search algorithms for the k-cardinality tree problem. *Discrete Applied Mathematics*, 128(2–3):511–540, 2003.

[10] J. Brimberg, D. Urošević, and N. Mladenović. Variable neighborhood search for the vertex weighted k-cardinality tree problem. *European Journal of Operational Research*, 171(1):74–84, 2006.

[11] T. N. Bui and G. Sundarraj. Ant system for the k-cardinality tree problem. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'04)*, volume 3102 of *Lecture Notes in Computer Science*, pages 36–47, Seattle, WA, USA, 2004. Springer.

[12] M. Chimani, M. Kandyba, and P. Mutzel. A new ILP formulation for 2-root-connected prize-collecting

Steiner networks. In *Proc. of 15th Annual European Symposium on Algorithms (ESA'07)*, volume 4698 of *Lecture Notes in Computer Science*, pages 681–692, Eilat, Israel, 2007. Springer.

[13] M. Ehrgott and J. Freitag. K_TREE/K_SUBGRAPH: a program package for minimal weighted k-cardinality tree subgraph problem. *European Journal of Operational Research*, 1(93):214–225, 1996.

[14] M. Ehrgott, J. Freitag, H.W. Hamacher, and F. Maffioli. Heuristics for the k-cardinality tree and subgraph problem. *Asia Pacific J. Oper. Res.*, 14(1):87–114, 1997.

[15] M. Fischetti, W. Hamacher, K. Jornsten, and F. Maffioli. Weighted k-cardinality trees: Complexity and polyhedral structure. *Networks*, 24:11–21, 1994.

[16] D. Gale. A theorem on flows in networks. *Pacific J. Math*, 7:1073–1082, 1957.

[17] N. Garg. A 3-approximation for the minimum tree spanning k vertices. In *Proc. of the 37th Annual Symposium on Foundations of Computer Science (FOCS'96)*, pages 302–309, Washington, DC, USA, 1996. IEEE Computer Society.

[18] N. Garg. Saving an epsilon: a 2-approximation for the k-MST problem in graphs. In *Proc. of the thirty-seventh annual ACM symposium on Theory of computing (STOC'05)*, pages 396–402. ACM Press, 2005.

[19] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.

[20] M.X. Goemans and Y. Myung. A catalog of Steiner tree formulations. *Networks*, 23:19–28, 1993.

[21] T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[22] I. Ljubić. *Exact and Memetic Algorithms for Two Network Design Problems.* PhD thesis, Technische Universität Wien, 2004.

[23] I. Ljubić, R. Weiskircher, U. Pferschy, G. Klau, P. Mutzel, and M. Fischetti. An algorithmic framework for the exact solution of the prize-collecting Steiner tree problem. *Mathematical Programming, Series B*, 105(2–3):427–449, 2006.

[24] A. Segev. The node-weighted Steiner tree problem. *Networks*, 17(1):1–17, 1987.

[25] D. Urošević, J. Brimberg, and N. Mladenović. Variable neighborhood decomposition search for the edge weighted k-cardinality tree problem. *Computers & OR*, 31(8):1205–1213, 2004.

[26] Standard performance evaluation corporation. `http://www.spec.org/`.

[27] L. A. Wolsey. *Integer Programming.* Wiley-Interscience, 1998.

Implementing Partial Persistence in Object-Oriented Languages

Frédéric Pluquet[†], Stefan Langerman[†], Antoine Marot[†], and Roel Wuyts[⋆]

[†]Université Libre de Bruxelles
{fpluquet,stefan.langerman,amarot}@ulb.ac.be

[⋆]Imec and KULeuven
roel.wuyts@imec.be

Abstract

A *partially persistent data structure* is a data structure which preserves previous versions of itself when it is modified. General theoretical schemes are known (e.g. the *fat node method*) for making any data structure partially persistent. To our knowledge however no general implementation of these theoretical methods exists to date. This paper evaluates different methods to achieve this goal and presents the first working implementation of partial persistence in the object-oriented language Java. Our approach is *transparent*, i.e., it allows any existing data structures to become persistent without changing its implementation where all previous solutions require an extensive modification of the code by hand. This transparent property is important in view of the large number of algorithmic results that rely on persistence. Our implementation uses aspect-oriented programming, a modularization technique which allows us to instrument the existing code with the needed hooks for the persistence implementation. The implementation is then validated by running benchmarks to analyze both the cost of persistence and of the aspect oriented approach. We also illustrate its applicability by implementing a random binary search tree and making it persistent, and then using the resulting structure to implement a point location data structure in just a few lines.

1 Introduction

In the algorithm literature, a gap exists between textual descriptions of algorithms in scientific articles and their implementation in a programming language. Algorithms are described in english text or (pseudo-)code and expressed using three kinds of operations: *basic operations* (such as basic arithmetic, assignment or simple control-flow), *new operations* (introduced by the paper) and *external operations* (referencing other research). When implementing the algorithm these operations need to be implemented efficiently and easily in a concrete language. This is easy enough for the basic operations that are supported directly in almost any programming language in use today. New operations are typically described in detail by the author of the proposed algorithm, and therefore can be implemented with some more work. But problems can arise with the external operations because they potentially hide very complex implementations not explained in the same article (referencing previous articles), thereby posing a real problem for developers.

An example of a well-understood yet complex algorithm frequently encountered is that of *persistence*. A regular data structure is *ephemeral*, i.e., only the last state of the data structure is stored and previous values are lost. On the other hand, a *persistent* data structure keeps old values when an update operation is performed. Several flavors of persistence were defined by Driscoll, Sarnak, Sleator and Tarjan[15]. A structure is *partially persistent* if previous versions remain accessible for queries but not for updates. A *fully persistent* structure offers accesses to its previous versions for queries and updates, where each update operation on a version of the data structure creates a new branch from this version for the new version.

Different methods presented by Driscoll et al.[15] can be used to make ephemeral structures persistent with only a constant factor slowdown. However, their most efficient techniques can only be applied in a pointer model, i.e., when data structures are only composed of a network of records of bounded size and in-degree. Their less efficient *fat node* method can be applied in the RAM model to obtain partially persistent data structures with

a $O(\log m)$[1] slowdown in speed (where m is the number of updates on the structure). Using the fact that version numbers are integers between 0 and m, one can use Y-Fast trees of Willard [33] combined with the dynamic perfect hashing scheme of Dietzfelbinger et al.[13] to obtain partially persistent arrays (or persistence in the RAM model) with a slowdown of $O(\log \log m)$ in speed. This result was extended to full persistence by Dietz[12].

Note that all the above results are theoretical, and to our knowledge, no general and fully *transparent* persistent system has been implemented to this day. By *transparent* (or *non intrusive*), we mean that no modification must be done to the code implementing an ephemeral structure to transform it into a persistent one. However, a quick review of the literature reveals that over 20 papers use persistence as an external operation [5, 34, 17, 18, 1, 20, 3, 6, 21, 10, 25, 22, 2, 32, 23, 9, 16, 8, 7, 4, 11], notified by the simple sentence "Make this structure persistent" or "The time and space bounds can be reduced if persistent structures are used". Given that no implementation of a mechanism is available to make a structure persistent, implementing either of these more advanced results is very difficult and time-consuming.

Two partial solutions were previously proposed to introduce the Driscoll et al. persistence as an external operation. The first one is the Zhiqing Liu persistent runtime system [24]. The entire system is persistent and uses a persistent stack and persistent heap to save changes. The granularity of changes to be recorded can be tuned to manage the quantity of recorded data. This solution is not flexible enough to change a subset of classes to persistent ones. However, in scientific articles, it is common that only a subset of all used structures for algorithms must be made persistent. The second previously existing solution is the Allen Parrish et al. persistent template class [27]. A template class `Per` is provided by the author. The author admits that the solution suffers from some problems (e.g. because of references in C++) and it is not transparent for the initial program since all variable declarations must be modified by hand.

On the other side all previous practical attempts to save previous states in a general and transparent way lack some of the main advantages of Driscoll et al. efficient persistence: some papers[28, 29] propose techniques to trace a program, events are logged, but full snapshots of previous versions are not readily accessible. Caffeine[19] on the other hand stores previous states as prolog facts for fast future queries, but the snapshots are taken by brute force, as a copy of the entire set of objects to trace.

This paper shows how any ephemeral data structures in an object-oriented language can become partially persistent (i.e., each state of any object can be saved and accessed efficiently in time and space) without modifying the ephemeral program, in a simple, transparent and fine-grained way. To obtain these results we developed a variant of the fat node method proposed in [15] to save previous states in the objects themselves, and tested several data structures optimized for this task. We use aspect-oriented programming to transparently include a mechanism for detecting state changes.

Any paper that uses persistence as an external operation therefore becomes much easier to implement. We show this by implementing a treap, a random binary search tree [31] and making it partially persistent in one line. We use persistent treaps to implement the planar point location solution from [14] in just a few lines of code.

The rest of this paper is organized as follows. Section 2 introduces the fat node method of Driscoll et al. and discusses some improvements. Section 3 explains how we implement this theoretical method in an object-oriented language. Section 4 shows how to use our system to create a data structure for planar point location using persistent search trees in a few lines of code. Benchmark results of our implementation are described in Section 5.

2 The Fat Node Method

2.1 Fat Node Method in the RAM Model. The *fat node method* as proposed by Driscoll et al.[15] is used to transform an ephemeral structure into a partially persistent structure in which changes of fields occurring in a node are saved in the node itself without erasing old values of fields. Although the fat node method was originally described only for data structures in the pointer model of computation, we will discuss it in the more general RAM model of computation to which it easily generalizes.

In the RAM model of computation, each memory unit has an address and instructions can be used to either read or store a value at some address. Each memory unit is ephemeral by nature, i.e., when an instruction is used to store a new value at some address, the previous value is lost forever.

A data structure is composed of a set of memory units containing the data, along with routines (lists of instructions in the RAM model) that are used to perform *operations* (queries or updates) on the data. In order to transform such a data structure into a partially persistent one, we first need to maintain a global version

[1] Throughout this article, we write $\lg 0 = \lg 1 = 1$ and $\lg x = \log_2 x$ for $x \geq 2$.

counter which is incremented every time an operation is performed on the data structure (note that several basic instructions could occur when performing an operation). We then simulate the RAM model by maintaining for every memory unit an auxiliary structure which records the values stored at that address after every operation where it is modified, along with the *timestamp* (value of the version counter) for the time at which that value was stored.

Whenever the original structure performs a store on an address, if the current version counter is present in the auxiliary structure, the corresponding value is updated. Otherwise, a new entry is added in the auxiliary structure, with the new value and the current timestamp. Whenever an instruction wants to read a value from an address at time t, the auxiliary structure is searched to find the value whose timestamp is the largest among those less than t. This way, persistent query operations can be performed at any desired time in the past. A data structure to maintain the auxiliary data structure in $O(1)$ time per update and $O(\log n)$ time per search can be developed using standard data structure techniques. The specific data structure we use and optimize for those operations is described in the next subsection.

2.2 An Efficient Structure for States. As described above, the auxiliary structure for each memory address must allow to add a new value with a timestamp greater than all previously stored timestamps, to update the value associated with the most recent timestamp, and to search for the value whose timestamp is the largest among those smaller than a given t. Of course any dictionary data structure that implements predecessor queries would do (e.g., any balanced tree, skip-list, etc.), but since this will be the most heavily used structure after applying the persistence transformation, special care has to be taken to make the structure as efficient as possible while keeping the memory overhead within reasonable bounds.

The simple structure we describe stores an extensible array where new elements can be appended at the end in $O(1)$ time (assuming constant time memory allocation), and where the number of pointers to follow and the number of comparisons to be performed during a search are both bounded by $\lg m + 2$ in the worst case where m is the number of elements in the array. The space used is $O(m)$.

The structure is composed of a linked list of $\lfloor \lg m \rfloor + 1$ arrays of exponentially decreasing sizes $2^{\lfloor \lg m \rfloor}, 2^{\lfloor \lg m \rfloor - 1}, \ldots, 1$. Each array stores (value, version number) pairs in decreasing order of version number, and all arrays are completely filled except maybe

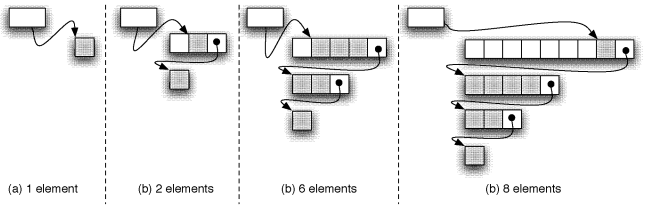


Figure 1: Structure to save versions of a field.

the frontmost and largest array. The frontmost array maintains the position of its element with the largest version number.

When storing the first value in an empty structure during initialization, an array of size one is created and the value and version number are stored (Fig.2.2 a). When a change occurs a second version is generated, an array of size 2 is created and linked to first array; the new version is stored along with its version number at the end of this array (Fig.2.2 b). Further changes fill the frontmost array from back to front until the array is full. When the array is full and the next update occurs, a new array is created whose size is twice that of the previous array, and is linked to that previous array, and the next version is stored in the last position of the new array (see Fig.2.2 c and d). The last version of the field is stored in the frontmost list and its value can be retrieved or updated in $O(1)$ time. The insertion of a new version costs $O(1)$ time as well.

When searching for the entry whose timestamp is the largest among those smaller than t, the list is followed until the array containing the sought element is found. This can be done by comparing t to the first element of each array until the last array is reached or an element larger than t is found. If $1 \leq k \leq \lfloor \lg m \rfloor$ pointers are followed, then k comparisons have been made and the array found is of size $2^{\lfloor \lg m \rfloor - k + 1}$. That array is then binary searched to find the desired element, using $\lfloor \lg m \rfloor - k + 2$ further comparisons.

2.3 Snapshots. Initially we had planned to make persistent objects record all versions of each field, regardless of the specific task at hand, but when implementing this solution we realized that different applications could need different granularities of versioning information, and that it is usually not necessary to save all states all the time. Suppose for example that we want to implement a balanced tree in the persistent language. If the application only requires to go back to previous consistent states of the tree, the interme-

diate state changes during the balancing operations do not need to be stored. On the other hand, if we want to use persistence to debug an operation in the same tree, it could be useful to store all steps. We therefore need a way for the user to indicate the consistent states in a system or, in other words, to define the granularity of the persistence.

The solution we decided to implement allows the user to explicitly indicate when the states have to be remembered by taking a *snapshot*, which can be done anytime. The result of a snapshot is a picture of the complete system at the time when it is created. To the user it is an object that can be used to access any data at the moment the snapshot was taken. In our previous example, the user is interested in seeing consistent states of a tree, would only take snapshots before or after performing operations on the tree (add, delete, ...), while a debugger application would take snapshots after any change to any tree object.

In practice, when the user takes a snapshot, the implementation will return an object containing the global version number, and will increment the global version number. Then, whenever the value of a field f is changed, if the version number of the last saved state of f is equal to the global version number, the value of the last state can be forgotten and replaced by the new value. Otherwise, a new state is created with the global version number as version number.

The *global view* mechanism is used to browse past states. At each read of an attribute of a persistent object the system checks if the global view is activated or not. If not (the system is at now), the original read is performed. Otherwise the system looks for, in the states structure associated to this attribute, the last value before or at version number joined to the global view. Using a persistent object in the past is completely transparent for the user: he chooses a previously taken snapshot and manipulates objects as he could in the present. Because we implement partial persistence, a change in past is not permitted (only querying the structure is allowed).

3 Implementation

3.1 Possible Choices. The easiest way to use persistence could be to select a language that already persistent. For instance, in a functional language all data structures are intrinsically persistent [26]. However few algorithms are developed in a functional model and their analysis is often difficult. Because most algorithms are described using imperative models of computation we restrict our research to this paradigm.

Unfortunately, to our knowledge, no usual imperative language (e.g. C++, Java, ...) allows to implement directly the persistence in a transparent way: an instrumentation of all accesses to given variables must be performed, without modifying original code. So we must interfere in the compilation process. We see several possible methods to insert a persistence mechanism in a transparent way for imperative languages: before the compilation (pre-compilation), after the compilation (change the bytecode), during the compilation and add more reflection to languages.

Before the compilation Adding transparent persistence via a pre-compilation would offer the advantage of being able to add the same pre-compiler to any implementation of the language. However, it requires to parse the code and extract necessary information to perform a transformation of the code. These operations are non-trivial.

After the compilation The manipulation can be also performed on the bytecode generated by a compiler. In that case, any language that can be transformed into the same bytecode could be rendered persistent. However, manipulating bytecode is also a complex task and it is not clear how one would instruct the persistence system on which parts of the code should be rendered persistent.

During the compilation More generally, one could interact at any level of the compilation (lexical analysis, parsing, semantic analysis, generation of code and optimization of generated code). The compiler must provide enough flexibility to accept this kind of interaction (either directly in the source code if it is open-source or via a plugin mechanism).

Adding more reflection Some solutions exist that add more reflection and introspection at a high abstraction level to instrument accesses to variables in existing languages. Aspect-Oriented Programming (AOP) has been developed in this way. Having that type of mechanisms at hand greatly simplifies the transparent implementation of persistence.

To develop our solution we selected the widespread object-oriented imperative programming language Java with the AspectJ module for the AOP functionalities. Note that our solution is easily adaptable to any language supporting the aspect paradigm (e.g. C++,C#, Python, Smalltalk), a Smalltalk implementation following similar principles was developed in parallel to the Java system described here.

3.2 Aspect-Oriented Programming. Aspect-oriented programming (AOP) is a modularization mechanism that allows a program to be split between

(functional) base code, and so called cross-cutting behavior that needs to be applied throughout the base code.

Take for example an application that implements a number of data structures (vectors, balanced trees, ...). For helping with debugging, the developers want to keep a log file that shows whenever elements are deleted from these data structures. A good solution to implement this behavior using a non-AOP language would be to implement a logging facility, and to change the delete functionality in the data structure implementations to call this logging facility. An alternative would be to call the logging facility in the code that uses the data structures. In both cases however, the logging code that is only there for the purpose of debugging is added to the base program (either in the data structures themselves or in the code that uses the data structures).

Using aspect-oriented programming, the data structures and the client code are written without taking the logging code into account. The logging code is implemented in its own module (an *aspect*), that contains the logging facility itself as well as expressions that indicate where this logging facility needs to be called. The base program and this logging code are then composed by a so-called *weaver*, that produces the final program that does logging. An aspect implements the behavior that needs to be called, and specifies when the behavior needs to be called. In our example, we could decide to call the *log* functionality as last statement in the implementation of any delete procedure in any of our data structures (which corresponds to the first manual solution). We could also decide to execute the *log* functionality after every call to a delete procedure, corresponding to the second solution.

An aspect language hands a developer a number of points (*join points*) in the execution of the program where code can be called (the *advice code*), and a language to use them. Such language typically supports quantifiers and wildcard expressions that make it easy to specify global criteria. In our example, the second approach needs to express *'After any call to a method named delete, call the following piece of code: ...'*. Exactly what join points are offered depends on the aspect language. Typically code can be executed before, after or around the execution of behavior (calling a function, constructing an object, etc.). Aspect languages also offer support to add elements to existing code (e.g., methods, fields, interfaces, if AOP extends OOP).

3.3 Java Specific Implementation Details. In order to implement persistence we must map each field of the object to an instance of our states structure containing the different values of this field. Fields in Java are statically typed, that is, their type can not be modified during the execution of a program. Thus a states structure can not be stored directly in place of those fields. Furthermore AspectJ only provides the names of the fields accessed. Because of this, we have to create a dictionary in each object for mapping the field name to the structure storing its states (named `fieldsAndStates` in previous codes). When a field in a persistent object is accessed, a lookup in the corresponding dictionary is performed.

3.4 Transparent Persistence with AspectJ. Our implementation uses the aspect-oriented AspectJ system to add persistence to existing Java programs without having to change these programs.

In order to use AspectJ to make classes persistent, the developer writes an *aspect declaration*. For example, the following AspectJ code makes all classes in a package `treap` persistent (note the wildcard expression `treap.*`):

```
declare parents: (treap.*) implements PObject;
```

Note that other criteria could be used, such as explicitly enumerating classes or selecting a number of classes based on their name. Technically, the aspect declaration updates the existing class to add our `PObject` interface to it. AspectJ will install all necessary wrappers to classes implementing the `PObject` interface, adds an instance variable in these classes, initializes them, and finally extends classes with some methods to access old states of fields of instances. Note that this solution is transparent. The existing structure is made persistent with the aspect declaration, which is not part of the ephemeral implementation. The rest of the aspect is used to manage the states:

- Adding a new variable to contain a dictionary in each persistent object:

```
public FieldsAndStates PObject.fieldsAndStates
               = new FieldsWithStates();
```

- Declaration of the pointcuts of setters and getters of persistent objects, but not in the `aspects` package:

```
// declaration of pointcut setters with 1 arg
pointcut setters(PObject t):
     // all updates of PObject implementors
     set(* PObject+.*)
     // not in 'aspects' package
     && ! within(aspects.*)
     // put the target in the variable t
     && target(t);
```

```
// same for all read operations
pointcut getters(PObject t): get(* PObject+.*)
     && ! within(aspects.*) && target(t);
```

- Definition of the advice code after each update of a field of a persistent object (we ask to save the new value for the set field):

```
after(Object newValue, PObject t) :
        setters(t)  && args(newValue) {
  t.fieldsAndStates.addStateWithValueFor(
     // the field name:
     thisJoinPoint.getSignature().getName(),
     // the new value stored in field:
     newValue); }
```

- Definition of the advice code around each read on a field of a persistent object:

```
Object around(PObject t) : getters(t) {
    if(!Snapshot.globalViewActivated())
         return proceed(t); // original read
    // retrieve the states of the field
    OrderedStates states = t.getStatesFor(
      thisJoinPoint.getSignature().getName());
    // search the good version of field
    // in respect to current snapshot VN
    return Snapshot.valueOfStates(states);}
```

Note that, because to AspectJ limitations, the arrays can not be made persistent in this way: AspectJ does not offer a mechanism to instrument accesses to the elements of an array. However such a feature is available in the Smalltalk implementation, the reflection mechanism being more powerful.

4 Planar Point Location and Treaps

Planar point location is a classical problem in computational geometry: given a subdivision of the plane into polygonal regions (delimited by n segments), construct a data structure such that given a point, the region containing it can be reported quickly.

Dobkin and Lipton[14] proposed a solution consisting in subdividing the plane into vertical slabs determined by vertical lines positioned at each vertex. Within each slab, there exists a total order between line segments determined by the order in which any vertical line in the slab intersects them. Each segment is associated to the polygon just above it, and a balanced binary search tree storing the segments is constructed for each slab.

When a point is queried, its x-coordinate is used to determine which slab contains it in $O(\log n)$ time, and the binary search tree of the corresponding slab is used to locate the region containing the point, also in $O(\log n)$ time. Unfortunately, the worst-case space requirement for this structure is $\Theta(n^2)$. To solve this problem, Sarnak and Tarjan[30] use persistence in order to reduce the space to $O(n)$. A vertical line sweeps the plane from $x = -\infty$ to $x = +\infty$, maintaining at every point the vertical order of the segment in a balanced binary search tree. The tree is modified every time the line sweeps over a point, but all previous versions of the tree are kept, effectively constructing Dobkin and Lipton's structure while using a space proportional to the number of structural changes in the tree.

In order to illustrate how transparent persistence can simplify the implementation of complex data structures, we implemented a *random treap*[31], a randomized binary search tree. The system then transforms automatically this structure into a partially persistent structure via the persistence aspect.

The following code is placed in a class storing a set of points. Each point stores its incoming and outgoing segments. In the construction of the point location data structure, each point of the set is swept by the sweepline, its outgoing segments are added in the treap, the incoming are removed and a snapshot is taken and stored in the info associated to the point.

```
private void constructRTreap(){
  rtreap = new RandomTreap();
  Iterator it = points.iterator();
  while(it.hasNext()){
     Point point = (Point)it.next();
     LinkedInfosPoint info = point.getInfo();
     Iterator segmentsIt =
        info.incomingSegmentsIterator();
     while(segmentsIt.hasNext()){
       rtreap.delete((Segment)segmentsIt.next());}
     segmentsIt = info.outgoingSegmentsIterator();
     while(segmentsIt.hasNext()){
       rtreap.put((Segment)segmentsIt.next());}
     info.setSnapshot(Snapshot.takeSnapshot());}}
```

In the location step of a point p, the slab containing p is determined. The user asks to the system to see the structure through the snapshot associated to the left point of the slab. The treap can then be used normally to locate the point.

```
public Segment locatePoint(Point p){
  Point thePoint = getLastPointBefore(p);
  LinkedInfosPoint assoc =
     (LinkedInfosPoint) points.get(thePoint);
 Snapshot.globalViewOn(assoc.getSnapshot());
 return
    (Segment)rtreap.searchEqualsOrJustBefore(p);}
```

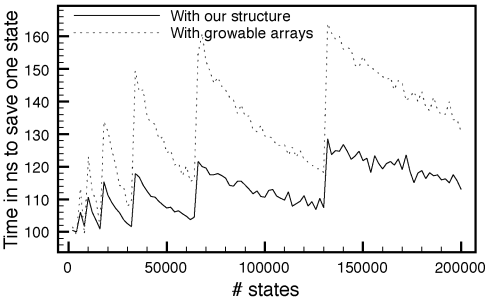


Figure 2: Number of insertions vs. average time per insertion

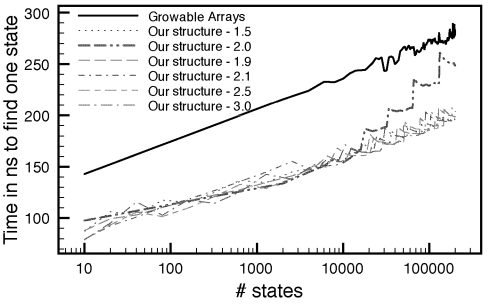


Figure 3: Number of elements in the structure vs. average time per search

5 Tests, Experiences and Performance

All tests were made on a Dual 2 GHz PowerPC G5 with 2 Go DDR of memory, using the NetBeans IDE 5.5 with version 1.5.0_06 of Java and the AspectJ Development Environment (AJDE) version 1.5.0[2]. The following parameters were used: `-Xms1024m` and `-Xmx1024m` (the size of stack is exactly of 1 Go) and `-Xnoclassgc` (no automatic garbage collector). We disable the garbage collector to avoid parasite behavior during the performance tests. A manual garbage collection is performed before each test to clean the stack.

All experiments follow the same structure. For n objects, we perform some operation (insert, search, ...) 10^6 times, accumulate the total time t (using `System.nanoTime()`) and we finally calculate the average time per operation ($t/(10^6 n)$). Thus we estimate the average time (in nanoseconds) per operation.

Java is a dynamic language and has many features to improve its performance (Just In Time compilation, Hotspot dynamic compilation, ...)[3]. As we will see, this will make it challenging to interpret our tests.

5.1 States Structure. The first test on states structure measures the insertion time. We create an empty instance of states structure and add n states in it (see Fig. 2). We also show the time taken by a growable array to perform the same operation. A growable array begins with an array of one element. At each insertion, if the array is full, a double sized array is created, the full array is copied into the new one using `System.arraycopy(...)` and the element is inserted in the first free place in the array. This technique is similar to the implementation of the Vector class from the standard Java libraries, but tailored to our needs.

For each structure the time per insertion seems constant, with peaks at each new allocation (in Java, an initialization is performed during each array allocation causing a linear allocation time). This operation is amortized by next insertions until the next allocation. The growable array is 1.2 time slower than states structure.

We next measure search time. We perform a search for each saved state in a structure containing n states and compute the average time (see Fig. 3). Results are shown using different growing factor. All curves are roughly logarithmic, as expected, but an intriguing phenomenon occurs: the performance becomes significantly worse when using a factor 2. Subsequent analysis revealed that dereferencing a Java array whose size is a power of two takes much more time than for most other array sizes ($\pm$200ns vs $\pm$20ns).

As in the previous test we also performed the same test with growable arrays. The time to find an element is also logarithmic. Our structure was always more efficient.

5.2 Persistence Aspect. The Java Just In Time(JIT) compiler is a real challenge for algorithm analysis: a read of a variable takes 40ns when the compiler is enabled. In the same conditions two reads take 45ns as total time. The sum of individual times is thus not equal to the time of combined operations. On the other hand this property is respected without enabling the compiler. Therefore we chose to disable the compiler, in order to collect more coherent data.

We now analyze the performance of our implementation of persistence in Java. A given number of changes is performed on an attribute of an object. We separate the time for each step of the persistence of an update operation (see Fig. 4):

Original Java It is the time to perform one change in the native Java program;

[2] http://www.netbeans.org, http://java.sun.com, http://aspectj-netbeans.sourceforge.net

[3] http://www-128.ibm.com/developerworks/library/j-jtp12214/

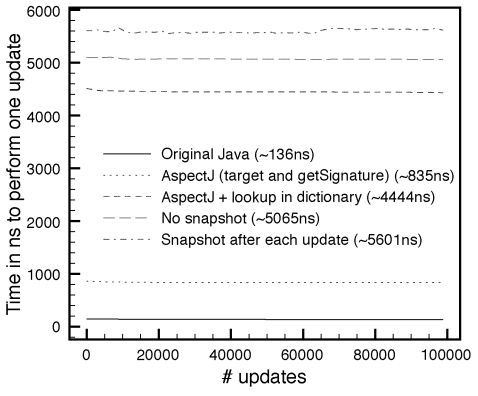


Figure 4: Number of updates vs. average time per update.

AspectJ (target and getSignature) The aspect adds some new code after each change. It takes extra time to retrieve the target object of the change and get the name of affected attribute (in signature). We measure an overhead of about 6;

AspectJ + lookup in dictionary As explained in Section 3.3 a dictionary is used to map the name of the target variable to the states data structure. The measured overhead is of about 32;

No Snapshot Adding the mechanism described in Subsection 3.4, without taking a snapshot (all changes update the value of the last state associated to the attribute). We measure an overhead of 37;

Snapshot after each update Same as the previous test but taking a snapshot after each change. The measured overhead is now 41.

Remark that the cost of AspectJ (to retrieve the affected attribute name and the target object) followed by the search in the dictionary induce an overload of 32 compared with the average time to perform a change on an attribute of a simple object in Java. Saving the state in the structure takes only between 600ns and 1200ns, i.e., only 4 to 9 times slower that the original code. If AspectJ were able to provide a mechanism to put the states directly in the attributes, much better results should be achievable.

In a second test (Fig. 5) we analyze the read a value that was just updated (only the read time is observed). Here the activation of the global view (Section 2.3) is important: if the global view v is activated we are looking for the value of an attribute in the last saved state before or at version number $v.versionNumber$. Otherwise the actual value of the attribute is returned (no lookup in dictionary is then performed). We decompose the operation:

Original Java The time of a read in the original Java. Does not differ much from an update ;

AspectJ (GV not activated) The global view is not activated. The aspect returns the actual value contained in the attribute. We measure an overhead of about 5.5;

AspectJ (GV activated: getSignature) The global view is activated, states of this attribute must be consulted. As a first step we report the search of the name of relevant attribute, using the signature of the read operation given by AspectJ. We measure an overhead of about 7.7;

AspectJ + lookup in dictionary After the previous operation the dictionary is consulted to retrieve the states data structure associated to the target variable. The measured overhead is of about 35;

No Snapshot The entire mechanism is activated, without taking a snapshot after the updates. We measure an overhead of 43;

Snapshot after each update, GV on first VN The same previous test but taking a snapshot after each change. The global view is activated and its version number is the first one of the system: at each read a search must be performed to find the first state in the associated states structure of the target attribute. The curve is logarithmic as expected.

The general observations made in our previous tests are confirmed here: the total performance is dominated by the three first phases.

Two important remarks can be made. Firstly a drawback of our implementation is that a lookup in dictionary must be done for each operation on an attribute (update or read via the global snapshot). The time of an update followed by read (with the global view activated) is so the sum of their individual time. We could not find a better method considering the features of Java and AspectJ. Secondly in order to interpret the large overhead of our system, the following must be taken into consideration :

The compiler is disabled With the compiler enabled the analysis can be done less precisely but we remark that the performance optimizations performed by the compiler reduce considerably this overhead ;

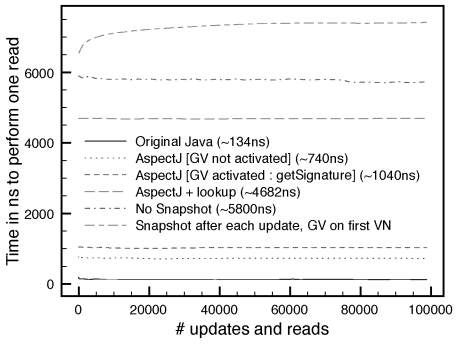


Figure 5: Number of updates+reads vs. average time per read.

Applications We will see in next section that in real applications, the persistent operations can be mixed with a large number of regular operations, making the overhead acceptable.

5.3 Persistent Treaps and Planar Point Location. We now test the performance of persistent treaps. Fig. 6 shows the average time per insertion in a treap vs. the number of elements in the treap. The same test is done on non persistent and persistent treaps (without snapshot and with snapshot after each insertion). The global view is not activated. An overhead of roughly 2 is observed for persistent ones. Remark that taking snapshot after each insertion does not increase the time by insertion considerably, due to the fact that the lookup in the dictionary takes more time that updating the last state or adding a new state in the states structure.

The second test (Fig. 7) gives the average time for searching in persistent and non persistent treaps. As a first result experiments indicate that search in a non persistent random treap takes time $O(\lg n)$. For persistent treaps several cases of the global view is considered: disabled (the overhead is about 3.6), global view on present (the last saved value in the states structure) and global view at middle of states (if there are k insertions with snapshots, the global view version number is the $(k/2)$th version number generated). The overhead of two last ones is about 25. Note that the theoretical expected search time is $O(\lg n * \lg \lg n)$: the expected number of states in a treap node is no more than the logarithm of the size of its subtree. However in our tests, the dictionary lookup dominates the running time, explaining the roughly logarithmic curve.

The explanation of these surprising low overheads (3.6 instead of 5.5 and 25 instead of 43) is the next one. When an insertion is performed in a persistent treap the operations are either non persistent ones (e.g.

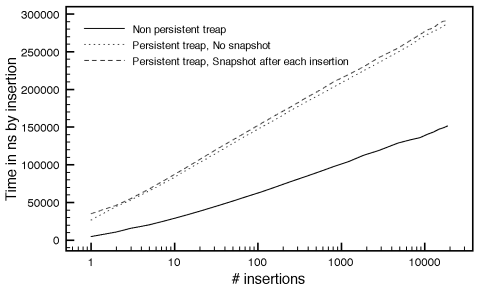


Figure 6: Insertion in non persistent and persistent treaps: number of insertions vs. average time per insertion.

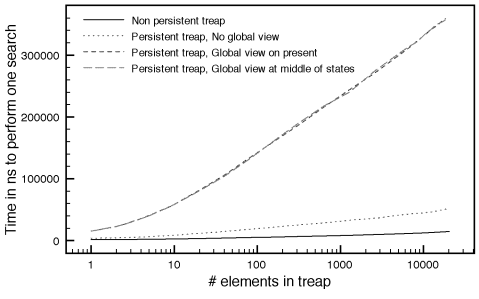


Figure 7: Search in non persistent and persistent treaps: number of elements in treap vs. average time per search.

comparisons or assignments of temporary variables) or a read in present (the global view is disabled) or a persistent update. Persistent operations are minority and do not increase too much the total time. The same observations can be applied to the search operations. So the high overheads observed during the aspect tests are lowered in the case of persistent treaps.

The test for the planar point location was realized as follow. For each n, number of given points in the plane, we generate random points and generate a Delaunay triangulation for these points. We run our implementation of the planar point location using as parameters the set of points and the segments generated. We measure the time t to locate n other random points in the plane. Fig. 8 shows the average time t/n to perform a search. As expected the curve is nearly logarithmic.

5.4 Size Tests. Now we analyze the space in memory of our implementation of the persistence in Java.

As first state we take a simple class composed by 1, 2, 3 or 4 `Integer` fields. The original size is 8 bytes + 16 bytes per field (4 for the pointer and 12 for the Integer object). Transforming this class to a persistent one the aspect adds a field containing a optimized `Hashtable`

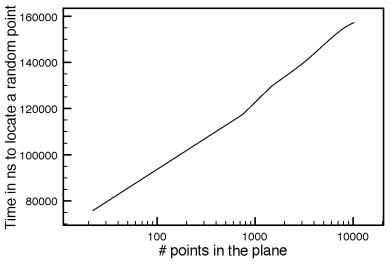


Figure 8: Planar Point Location: number of points in the plane vs. the time to locate a random point

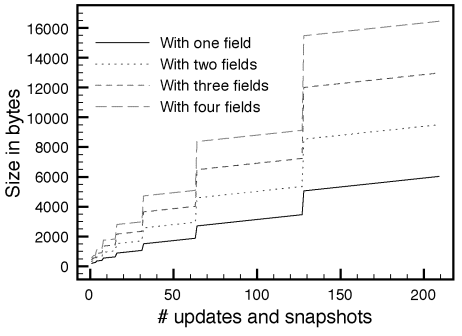


Figure 9: Sizes for object with 1, 2, 3 and 4 fields: number of update followed by snapshot vs. the size of the object

instance and some useful informations for AspectJ. The size grows to 50 + 140 bytes per field, an overhead about 8.

Fig. 9 shows the total sizes of objects with, respectively, 1, 2, 3 and 4 fields after updates (of all fields), each one followed by a snapshot. The total size grows linearly according vertical steps due to instantiation of a new array in states structure at each power of 2. The steps of the stair graphs are not horizontal because at each change a new state is created and added in the states structure.

Fig. 10 shows the sizes of ephemeral and persistent (no snapshot and snapshot after each insertion) treaps. When no snapshot is taken the observed average overhead is about 7.5. It grows to 9.5 with snapshots.

6 Conclusions

This paper presents a first fully transparent implementation of persistence in an object-oriented language. The performance of our implementation is far from optimal, partly due to the restriction of the language and of the overhead intrinsic to the aspect-oriented programming system used. There are several ways in which a more efficient implementation of persistence could be

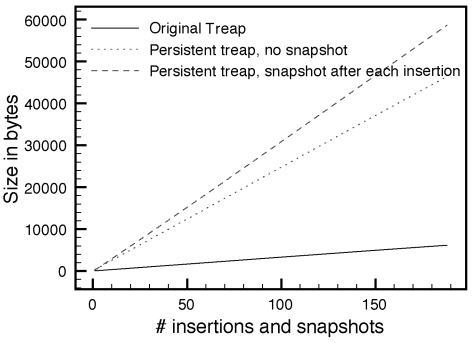


Figure 10: Size of non persistent and persistent treaps: number of insertions followed by snapshot vs. the size of the treap

designed, e.g., by writing precompilers to generate persistent code or by directly modifying the virtual machine. Nevertheless the approach presented here has the advantage of being easy to implement in any language that supports the aspect paradigm (C++, C#, Java, JavaScript, PHP, Python, Smalltalk and many others). A Smalltalk version is moreover currently developed in the Squeak environment.

Several interesting theoretical questions emerge from our work: is it possible to implement persistence in a way that would exploit the structure of the data structure, i.e., if the structure is indeed composed of nodes of low indegree, could the implementation be automatically faster? How would we implement garbage collecting on saved states when a snapshot is deleted?

Acknowledgements

The authors thank Luc Devroye, Raymond Kalimunda and Pat Morin for helpful discussions.

References

[1] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 531–540, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[2] P. K. Agarwal. Ray shooting and other applications of spanning trees with low stabbing number. *SIAM J. Comput.*, 21(3):540–570, 1992.

[3] P. K. Agarwal, S. Har-Peled, M. Sharir, and Y. Wang. Hausdorff distance under translation for points and balls. In *SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry*, pages 282–291, New York, NY, USA, 2003. ACM Press.

[4] B. Aronov, P. Bose, E. D. Demaine, J. Gudmundsson, J. Iacono, S. Langerman, and M. H. M. Smid. Data structures for halfplane proximity queries and incremental voronoi diagrams. In *Proc. of the 7th Latin American Symposium on Theoretical Informatics (LATIN'06)*, pages 80–92, Valdivia, Chile, 2006.

[5] F. Aurenhammer and O. Schwarzkopf. A simple online randomized incremental algorithm for computing higher order voronoi diagrams. In *SCG '91: Proceedings of the seventh annual symposium on Computational geometry*, pages 142–151, New York, NY, USA, 1991. ACM Press.

[6] M. Bern. Hidden surface removal for rectangles. In *SCG '88: Proceedings of the fourth annual symposium on Computational geometry*, pages 183–192, New York, NY, USA, 1988. ACM Press.

[7] M. Bern, D. Dobkin, D. Eppstein, and R. Grossman. Visibility with a moving point of view. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 107–117, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[8] P. Bose, M. van Kreveld, A. Maheshwari, P. Morin, and J. Morrison. Translating a regular grid over a point set. *Comput. Geom. Theory Appl.*, 25(1-2):21–34, 2003.

[9] S. Cabello, Y. Liu, A. Mantler, and J. Snoeyink. Testing homotopy for paths in the plane. In *SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry*, pages 160–169, New York, NY, USA, 2002. ACM Press.

[10] S.-W. Cheng and M.-P. Ng. Isomorphism testing and display of symmetries in dynamic trees. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 202–211, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[11] E. D. Demaine, J. Iacono, and S. Langerman. Retroactive data structures. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 281–290, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[12] P. F. Dietz. Fully persistent arrays (extended array). In *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, pages 67–74, London, UK, 1989. Springer-Verlag.

[13] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer Auf Der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[14] D. Dobkin and R. Lipton. Multidimensional searching problems. *SIAM Journal of Computing 5*, pages 181–186, 1976.

[15] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, pages 86–124, 1986.

[16] H. Edelsbrunner, J. Harer, A. Mascarenhas, and V. Pascucci. Time-varying reeb graphs for continuous space-time data. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 366–372, New York, NY, USA, 2004. ACM Press.

[17] D. Eppstein. Clustering for faster network simplex pivots. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 160–166, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[18] M. T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 523–533, New York, NY, USA, 1991. ACM Press.

[19] Y.-G. Guéhéneuc, R. Douence, and N. Jussien. No java without caffeine – a tool for dynamic analysis of java programs. In *Proceedings of ASE 2002 : 17th International IEEE Conference on Automated Software Engineering*, Edinburgh, UK, September 2002.

[20] P. Gupta, R. Janardan, and M. Smid. Efficient algorithms for generalized intersection searching on non-iso-oriented objects. In *SCG '94: Proceedings of the tenth annual symposium on Computational geometry*, pages 369–378, New York, NY, USA, 1994. ACM Press.

[21] J. Hershberger. Improved output-sensitive snap rounding. In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 357–366, New York, NY, USA, 2006. ACM Press.

[22] P. N. Klein. Multiple-source shortest paths in planar graphs. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 146–155, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[23] V. Koltun. Segment intersection searching problems in general settings. In *SCG '01: Proceedings of the seventeenth annual symposium on Computational geometry*, pages 197–206, New York, NY, USA, 2001. ACM Press.

[24] Z. Liu. A persistent runtime system using persistent data structures. In *SAC '96: Proceedings of the 1996 ACM symposium on Applied Computing*, pages 429–436, New York, NY, USA, 1996. ACM Press.

[25] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality-tests in polylogarithmic time. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 213–222, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[26] C. Okasaki. *Purely functional data structures.* Cambridge University Press, New York, NY, USA, 1998.

[27] A. Parrish, B. Dixon, D. Cordes, S. Vrbsky, and J. Lusth. Implementing persistent data structures using c++. *Softw. Pract. Exper.*, 28(15):1559–1579, 1998.

[28] S. P. Reiss and M. Renieris. Generating Java trace data. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 71–77. ACM Press, 2000.

[29] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 221–230, Toronto,

Ontario, Canada, 2001. IEEE.

[30] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[31] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[32] J. Turek, J. L. Wolf, K. R. Pattipati, and P. S. Yu. Scheduling parallelizable tasks: putting it all on the shelf. In *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 225–236, New York, NY, USA, 1992. ACM Press.

[33] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

[34] D. M. Yellin. Algorithms for subset testing and finding maximal sets. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 386–392, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

Comparing Online Learning Algorithms to Stochastic Approaches for the Multi-Period Newsvendor Problem

Shawn O'Neil Amitabh Chaudhary

Abstract

The multi-period newsvendor problem describes the dilemma of a newspaper salesman—how many papers should he purchase each day to resell, when he doesn't know the demand? We develop approaches for this well known problem based on two machine learning algorithms: Weighted Majority of Warmuth and Littlestone, and Follow the Perturbed Leader of Kalai and Vempala. With some modified analysis, it isn't hard to show theoretical bounds for our modified versions of these algorithms. More importantly, we test the algorithms in a variety of simulated conditions, and compare the results to those given by traditional stochastic approaches which assume more information about the demands than is typically known. Our tests indicate that such online learning algorithms can perform well in comparison to stochastic approaches, even when the stochastic approaches are given perfect information.

1 Introduction

On each morning of some sequence of days, a newspaper salesman needs to decide how many newspapers to order at a cost of c per paper, so that he can resell them for an income of r per paper. Unfortunately, every day it is unknown how many papers d will be demanded. If too many are ordered, some profits are lost on unused stock. If too few are ordered, some profits are lost due to unmet demand. The actual profit seen by a vendor who orders x items on a day with demand d is given by $r\min\{d, x\} - xc$. The papers ordered for a single day are of course only useful for that day; leftover papers cannot be sold in any later period.

This model describes a wide variety of products in industry. Fashion items and the trends they rely on are typically short lived, inducing many manufacturers to introduce new product lines every season[16]. Consumer electronics also have a short selling season due to their continuously evolving nature; cellular phones can have a lifecycle as short as six months[2]. Some vaccines such as those for influenza are only useful for a single season[6].

For many such products, due to required minimum manufacturing or processing times, the vendor must finalize his order before any demand is seen. Further, because properties of the products themselves can vary markedly between selling periods, so too can the demand seen each period. This *demand uncertainty* is the most challenging hallmark of the newsvendor model.

A common approach taken to resolve the demand uncertainty issue is using a stochastic model for the demands; assuming, for example that for each period the demand is drawn independently from some known distribution. In using such an approach, the goal is then to choose an order amount which maximizes expected profit (see, e.g., [11]). However, such approaches are commonly inadequate, as the quality of the final result depends heavily on the quality of the assumptions made about the distribution. Given the strong uncertainty inherent in many newsvendor items, such quality is usually low. (See [21] for a lengthier discussion on the shortcomings of this approach.)

Alternate approaches to the newsvendor problem are more "adversarial" in nature. In these models, very little is assumed about the nature of the demands, and worst-case analysis is used. Typically, only a lower bound m and upper bound M on the range of possible demand values are assumed. One solution in this area develops a strategy to minimize the *maximum regret*:

$$\max_{\text{demand values}} (\mathsf{OPT} - \mathsf{ALG}),$$

where OPT denotes the profit of the *offline optimal* algorithm which knows the demand values, and ALG is the profit of the strategy used (see [17, 21, 22]).

Another method used to evaluate and design online algorithms for such problems is competitive ratio, where the goal is to minimize the ratio $\mathsf{OPT}/\mathsf{ALG}$ in the worst case. However, one can show, using Yao's technique, a lower bound of $\Omega(M/(mk))$ for this ratio in the single period case when $r = kc$. This bound is tight, as a simple balancing algorithm can guarantee profits of this form.

Similarly restrictive results can be found for the worst case approach to regret with respect to OPT seen above. The single period *minimax regret* solution results in a maximum regret of $c(M-m)(r-c)/r$[21], which implies that for t periods of a newsvendor game it is possible to suffer a regret of $tc(M-m)(r-c)/r$, even for the best possible deterministic algorithm.

For these reasons, we turn away from evaluating the performance of algorithms in terms of the *dynamic* offline optimal, and consider a more realistic target: the *static* offline optimal, which we denote here by STOPT. STOPT is a weaker version of OPT which makes an optimal decision based on perfect knowledge of the demands, but is required to choose one single order quantity to use for all periods.

Comparing the performance of algorithms with the performance of STOPT has practical significance, because any bounds for an algorithm with respect to STOPT also hold with respect to an algorithm which makes decisions based on stationary stochastic assumptions. Much of the inventory theory literature deals with algorithms of this type[15, 11].

We look at adaptations of two Expert Advice algorithms: Weighted Majority, developed by Littlestone and Warmuth[14], and Follow the Perturbed Leader, developed by Kalai and Vempala[12].

In the expert advice problem, the algorithm designer is given access to n experts, each of whom make a prediction for each period, and suffer some cost for incorrect predictions. The goal is to design an algorithm that makes its own predictions based on the experts' advice, and yet does not suffer much more cost than the best performing expert in hindsight.

In our setting, we use naive experts which make fixed predictions in the range $[m, M]$, and the cost they suffer in each period is the regret (difference in profit) from the dynamic offline OPT. Adapting the Weighted Majority algorithm to the non linear profit function of the newsvendor problem requires some careful attention if one wants to show theoretical performance bounds, whereas Follow the Perturbed Leader is a more straightforward implementation. Details of the algorithms' operation and theoretical performance bounds in this setting can be found in the appendices.

2 Goals of This Paper

In Section 4, we'll give overviews of the operation of three algorithms, two based on Weighted Majority variants which we call WMN and WMNS, and one based on Follow the Perturbed Leader which we call FPL. Each of these algorithms takes parameters which are chosen by the experimenter as input, which affect their operation and the performance bounds they achieve.

The primary interest of this paper, then, is to empirically evaluate the performance of these algorithms and compare the results to those generated by STOPT as well as more traditional stochastic approaches. Each of the stochastic solutions takes as input the assumptions made by the experimenter about the mean and standard deviation of the input distribution.

Further, the specifics of the problem instance itself may lead to interesting observations about all of the solutions specified. For instance, we know that the relationship of r and c can make a large difference on the performance of the minimax regret solution; does this ratio also affect the performance of other approaches we are going to test? Do certain types of input distributions favor one approach over the other?

Given such a large number of possible experimental variables, we are forced to select those which we believe will be most interesting, and design experiments using simulated data which are most likely to highlight the advantages and deficiencies of the different approaches.

3 Related Work

The Newsvendor Problem The origins of the newsvendor problem can be traced as far back as Edgeworth's 1888 paper[10] in which the author considers how much money a bank should keep in reserve to satisfy customer withdrawal demands, with high probability. If the demand distribution and the first two moments are assumed known (normal, log-normal, and Poisson are common), then it can be shown that the expected profit is maximized at x, where $\phi(x) = (r-c)/r$ and $\phi(\cdot)$ is the cumulative probability density function for the distribution. Gallego's lecture notes[11] as well as the book by Porteus[15] have useful overviews. When only the mean and standard deviation are known, Scarf's results[18] give the optimal stocking quantity which maximizes the expected profit assuming the worst case distribution with those two moments (a *maxi-min* approach). In some situations this solution prescribes ordering no items at all.

Among worst-case analyses, one of the earliest uses of the minimax regret criterion for *decision making under uncertainty* was introduced by Savage[17]. Applying the techniques to the newsvendor problem, Vairaktarakis describes adversarial solutions for several performance criteria in the setting of multiple item types per period and a budget constraint[21]. Bertsimas and Thiele give solutions for several variants of the newsvendor problem which optimize the order quantity based on historical data[3]. The solutions discussed take into account risk preferences by "trimming," or ignoring, historical data which leads to overly optimistic predictions.

Learning from Experts Weighted Majority is a very adaptable machine learning algorithm developed by Littlestone and Warmuth[14]. There are several versions of the weighted majority algorithm, including discrete, continuous, and randomized. Each consults the predictions of experts, and seeks to minimize the regret (in terms of prediction mistakes) with respect to the best

expert in the pool.

Weighted Majority and variations thereof have been applied to a wide variety of areas including online portfolio selection[8, 7] and robust option pricing[9]. Other variants include the WINNOW algorithm also developed by Littlestone[13], which has been applied to such areas as predicting user actions on the world wide web[1].

Follow the Perturbed leader is a general algorithm for online decision making which is also applicable to the learning from experts problem. It's creators, Kalai and Vempala[12], apply the algorithm to such problems as online shortest paths[20] and the tree update problem[19].

4 Algorithms

For these experiments, we impliment the following algorithms as described:

STOPT This approach is given perfect information about the demand sequence, and chooses the single order quantity to use for all periods which maximizes the overall profit (and thus also minimizes the total regret). As Bertsimas and Thiele discuss[3], the static offline optimal choice is the $\lceil t - t(c/r) \rceil^{th}$ order statistic of the demand sequence.

NORMAL This stochastic solution assumes the demands will be drawn from a known normal distribution, and maximizes the expected profit. This approach prescribes ordering the amount $\mu + \sigma\phi^{-1}\big((r-c)/r\big)$, where $\phi^{-1}(\cdot)$ is the inverse of the standard normal cumulative distribution function[11].

SCARF This stochastic solution is described in Scarf's original paper[18] as well as in [11]. The solution maximizes the expected profit for the worst case distribution (a *maximin* approach in the stochastic sense) with first and second moments μ and σ. The order quantity is prescribed to be $\mu + \frac{\sigma}{2}\Big(\sqrt{(r-c)/c} - \sqrt{c/(r-c)}\Big)$ if $c(1+\sigma^2/\mu^2) < r$, and 0 otherwise.

MINIMAX This is the *minimax regret* approach mentioned in Section 1. Described by [21], the algorithm orders the quantity $\big(M(r-c)+mc\big)/r$ for every period, which minimizes the maximum possible regret from the optimal for each period. As such, it also minimizes the maximum possible regret for the whole sequence.

The solution works by balancing the regret suffered by the two worst case possibilities: the demand being m or M. Because of this, its order never changes (as long as the range $[m, M]$ doesn't change), and is very pessimistic in nature.

WMN We develop this algorithm (Weighted Majority Newsvendor) as an adaptation of the Weighted Majority algorithm of Littlestone and Warmuth[14]. The algorithm takes two parameters: n, the number of "experts" to consult, and $\beta \in (0, 1]$, the weight adjustment parameter. Essentially, we divide up the range $[m, M]$ into n buckets, and have expert i predict the minimax regret order quantity for the i^{th} bucket. Buckets and experts are set up so that each bucket/expert pair has the same minimax regret.

As per the standard operation of Weighted Majority, each expert is given an initial weight of 1. After each round, we decrease each expert i's weight by some factor F, where F depends on β and the regret that expert would have suffered on the demand seen using its prediction. If an expert is often wrong, its weight will be decreased faster than others. This punishment happens faster overall with smaller β's.

The amount ordered by WMN in a given period is the weighted average of all experts. The intuition is that wherever the static optimal choice is, it must fall in one of the n buckets, and thus one of our experts will be close to this static optimal choice. Further, because experts' weights are decreased according to how poorly they do, the algorithm is able to learn where the static optimal choice is after a few periods, and even adapt to changing inputs over time.

Adapting the analysis of Weighted Majority to the non linear newsvendor profit function requires special care to ensure bounds similar to that of Weighted Majority can still be given. In Appendix A, we give a detailed description of WMN and a proof of the following theorem:

THEOREM 4.1. *The total regret experienced by* WMN *for a t period newsvendor game with per item cost c, per item revenue r, and all demands within $[m, M]$ satisfies*

$$\begin{aligned}\mathsf{WMN}_{TotalRegret} &\leq \frac{\mathbb{C}\ln(n)}{1-\beta} + \frac{\ln\left(\frac{1}{\beta}\right) c(M-m)(r-c)t}{nr(1-\beta)} \\ &\quad + \frac{\ln\left(\frac{1}{\beta}\right)\mathsf{STOPT}_{TotalRegret}}{1-\beta}\end{aligned}$$

where $\mathbb{C} = \max\{(M-m)(r-c), (M-m)c\}$ is the maximum possible single period regret, n is the number of buckets used by WMN*, and β is the update parameter used.*

WMNS WMNS, for Weighted Majority Newsvendor Shifting, is based on the "shifting target" version of the standard Weighted Majority algorithm. Here, if the input sequence can be decomposed into subsequences such

that for each subsequence a particular expert does very well, then WMNS will do nearly as well for that subsequence. WMNS needs no information about how many shifts there will be, or when they will be. For example, if for the first third of the sequence all demands are near m, WMNS will initially adjust the weights of the experts so that it is ordering near m as well. If the sequence shifts so that demands are then drawn from near M, WMNS will adjust the weights quickly (quicker than WMN) so that the order quantities will match.

This ability comes from WMNS's use of a weight limiting factor $\delta \in (0, 1]$, so that no expert's weight will be less than δ times the average weight. When a new expert starts doing significantly better, the old best expert's weight is decreased to below the new expert's weight more rapidly, as the new expert's weight is guaranteed not to be too low in relation.

THEOREM 4.2. *The total regret experienced by* WMNS *for a t period newsvendor game with per item cost c, per item revenue r, and all demands within $[m, M]$ satisfies*

$$\begin{aligned}&\mathsf{WMNS}_{TotalRegret}\\ &\leq \frac{k\mathbb{C}\ln\left(\frac{n}{\beta\delta}\right)}{(1-\beta)(1-\delta)} + \frac{\ln\left(\frac{1}{\beta}\right)c(M-m)(r-c)t}{nr(1-\beta)(1-\delta)}\\ &\quad + \frac{\ln\left(\frac{1}{\beta}\right)\mathsf{SSTOPT}_{TotalRegret}}{(1-\beta)(1-\delta)}\end{aligned}$$

where $\mathbb{C} = \max\{(M-m)(r-c), (M-m)c\}$ is the maximum possible single period regret, n is the number of buckets used by WMNS*, β is the update parameter used, and δ is the weight limiting parameter used.* SSTOPT *is allowed to use a static optimal choice for k subsequences (i.e., is allowed to change order values $k-1$ times, see below).*

Details of WMNS's operation and proof of the above bounds are given in Appendix B.

SSTOPT This "optimal," which makes its decisions based on the entire sequence, is a slightly stronger version of STOPT, which is allowed to change its order quantity exactly $k-1$ times during the sequence.

FPL Similar to WMN, FPL is a randomized algorithm based upon the Follow the Perturbed Leader approach developed by Kalai and Vempala[12]. As a general algorithm it is well suited to making decisions a number of times, when one wants to minimize the total cost in relation to the best single decision for all periods. Here, decisions will be of the form "use expert i's prediction," where the experts again predict minimax values in buckets which divide the $[m, M]$ range. FPL as we use it takes two parameters, n, for the number of experts/buckets, and ϵ, which affects the final cost bound in relation to the best static decision.

THEOREM 4.3. *The total regret experienced by* FPL *for a t period newsvendor game with per item cost c, per item revenue r, and all demands within $[m, M]$ satisfies*

$$\begin{aligned}&E[\mathsf{FPL}_{TotalRegret}]\\ &\leq \frac{4\mathbb{C}(1+\ln(n))}{\epsilon} + \frac{(1+\epsilon)c(M-m)(r-c)t}{nr}\\ &\quad + (1+\epsilon)\mathsf{STOPT}_{TotalRegret}\end{aligned}$$

where $\mathbb{C} = \max\{(M-m)(r-c), (M-m)c\}$ is the maximum possible single period regret, n is the number of buckets used by FPL*, and ϵ is the randomness parameter used.*

Details of the algorithm and proof of the bounds it gives in our application appear in Appendix C.

5 Experiments

In order to evaluate the online learning algorithms for the newsvendor problem, we run them on simulated demand sequences comparing the total regret suffered by each approach to the regret suffered by the stochastic algorithms SCARF and NORMAL, as well as MINIMAX and STOPT.

Unless otherwise noted, all experiments consist of 100 demand newsvendor sequences, and each data point represents the average of 100 such trials. Thus, data points in the following figures typically represent the average total regret of various approaches on newsvendor sequences of length 100. Also, due to space limitations, we won't experiment with the affect of the upper and lower demand bounds $[m, M]$; we'll instead fix these bounds to $[10, 100]$ for all tests. Whenever a normal distribution is used, we restrict it to this range by resampling if a demand falls outside the range, and we further restrict all demands to be integers.

5.1 Algorithm Parameters

5.1.1 β, ϵ, and μ For this first batch of tests, we investigate the performance of our three machine learning approaches while varying some of the parameters they accept as input. WMN and WMNS use β as a weight adjustment parameter: the smaller β is, the quicker expert weights are adjusted downward. WMNS also uses a "weight limiting" parameter δ, which we hold constant at 0.3 for these tests.

FPL uses the parameter ϵ, which affects the amount of "randomness" used in deciding which expert to

follow. Smaller ϵ values lead to more randomness being used. Even though the bounds discussed for FPL are only valid for $\epsilon \in (0, 1]$, the algorithm is still operable for larger values, so we test $\epsilon \in (0, 5]$. While we test the effects of varying β and ϵ, we hold the number of experts, n, at 32.

For Figures 1, 2, and 3, the distribution is normal with mean demand of 25 and standard deviation 15. Note that because the distribution is bounded to $[10, 100]$ via resampling, the actual mean of the distribution used is about 29.3. The per item cost c is held at 1, and the per item profit r is 4.

Figure 1 plots the average total regret of WMN, WMNS, and FPL as we vary β and ϵ. We also show the average total regret of STOPT as a baseline for comparison. One thing to notice in this figure is that while WMNS is adapted to be useful in situations where the distribution makes drastic changes over time, it does very nearly as well as WMN in this case.

On the other hand, even though FPL suffers a respectably low amount of regret, it's performance is only comparable to the other two approaches when rather large ϵ's are used which aren't valid for theoretical analysis.

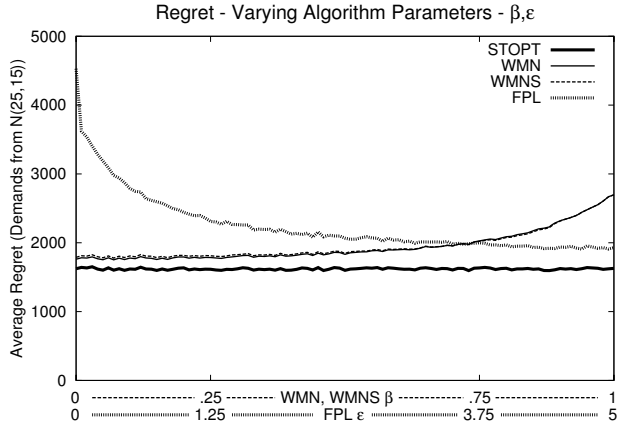


Figure 1: Average regret suffered on a 100 period newsvendor sequence, varying algorithm parameters. Even though ϵ must be less than or equal to 1 for FPL's theoretical bounds to hold, we see that in this situation it performs well with larger values also.

Figure 2 shows the regret of NORMAL and SCARF on the same test, varying the mean assumed about the demand distribution. Both approaches assume the correct standard deviation of 15. As this figure shows, the consequences of assuming incorrect information can be quite drastic for such stochastic algorithms.

In fact, it is interesting to look at the range of μ values used by NORMAL for which it suffers less regret than WMN. When WMN uses a β of 0.5, a rather naive choice, the average regret suffered is about 1856. NORMAL suffers less regret than this only when it assumes $\mu \in [21.7, 37]$, or within about 7.6 units on either side of the actual mean.

In Figure 2 we also plot the rather large regret suffered by the pessimistic worst case algorithm MINIMAX.

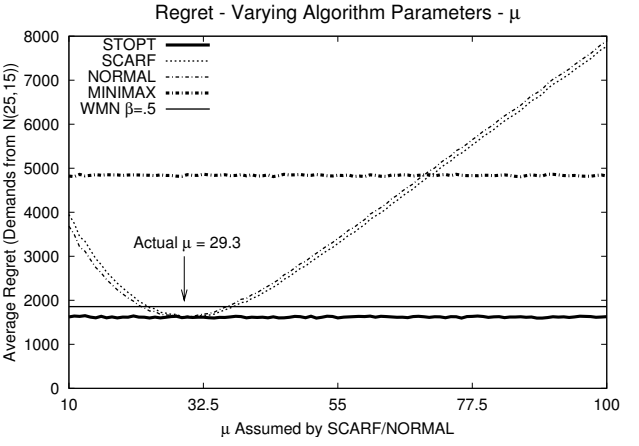


Figure 2: Average regret suffered on a 100 period newsvendor sequence, varying the mean assumed by NORMAL and SCARF. This plot shows the consequences to the stochastic approaches of assuming incorrect information. For comparison, we also plot WMN's regret of 1856 when WMN uses a $\beta = 0.5$.

In Figure 3 we indicate what the theoretical bounds for WMN, WMNS, and FPL would be in the previous experiment. That is, given the values used for r,c,m,M,t, as well as the parameters used by the algorithms, we use the actual regret suffered by STOPT to compute the worst case regret for the algorithms no matter the input sequence. We see that the theoretical bounds are much higher than the actual empirical performance seen in figure 1, by as much as an order of magnitude.

As is reflected in Figure 3, the theoretical bounds given in Theorems 4.1 and 4.2 increase without bound as β is reduced to 0, because of the $\ln(1/\beta)$ term.

Because of this, we begin to notice the trade off between minimizing the theoretical bounds and getting good performance in actual simulation. (Later, in Figures 9 and 10, we'll see the same phenomenon.) Similar to MINIMAX, the three experts algorithms give theoretical worst case bounds (though in relation to STOPT, rather than OPT), and as such have a pessimistic nature to them as well. Using a β which decreases weights rapidly will quickly find the correct amount to order, however may be more susceptible to

poor performance with very adversarial sequences.

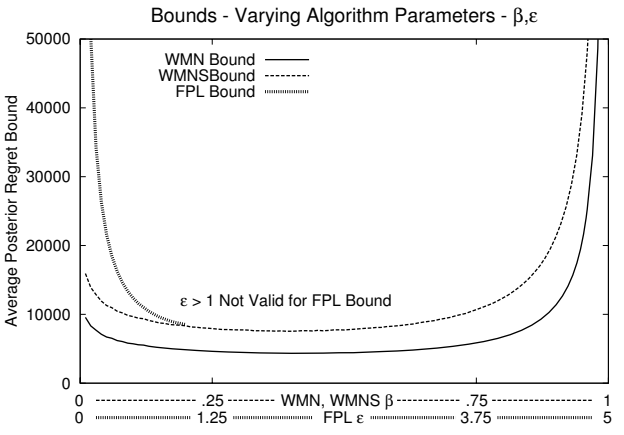


Figure 3: Average worst case theoretical bounds computed using algorithm parameters, problem parameters, and the actual regret suffered by STOPT. Comparing to Figure 1, we see a trade off between low worst case bounds and better empirical performance.

5.1.2 Number of Experts: n Having looked at the effects of varying β and ϵ, we now turn our attention to the other main parameter of WMN, WMNS, and FPL: the number of experts/buckets used. Intuitively, using a larger value for n means that we are more likely to have an expert close to STOPT's order value. On the other hand, all of the theoretical bounds grow as n becomes very large.

For Figures 4 and 5 we run the same test as section 5.1.1, with all demands drawn from N(25,15) bounded to $[10, 100]$. Here, WMN uses $\beta = 0.5$, WMNS uses $\beta = 0.5, \delta = 0.3$, and FPL uses $\epsilon = 0.75$.

Figure 4 plots the average total regret of the three algorithms varying the number of experts used from 1 to 100. Like the last test, WMN and WMNS perform remarkably similar, and FPL performs somewhat worse. (Had we used a larger ϵ, this difference would probably not be as striking.) In this plot, it appears that above a certain point, around 5 or 10, increasing the number of experts is ineffective. One possible reason for this is that because WMN and WMNS use the weighted average of experts, it is possible for them to settle upon an order quantity between two experts, making the number of experts somewhat less important. This cannot be the case for FPL, however, as FPL always goes with a single expert's choice.

Figure 5 shows the computed theoretical bounds given by varying the number of experts for this test. Again, we see that a modest number of experts appears

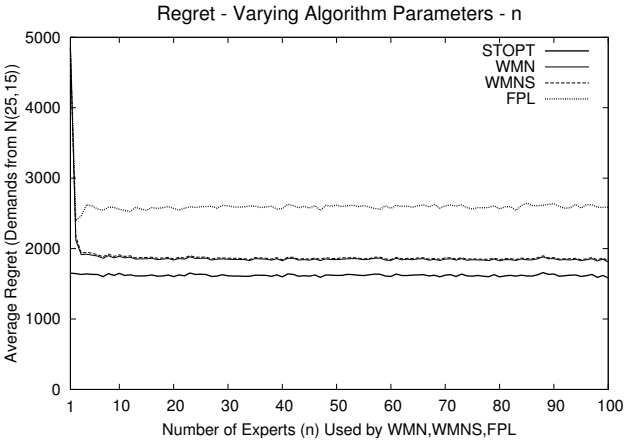


Figure 4: Average regret suffered while holding β, δ, and ϵ constant and varying the number of experts/buckets used n. While small values of n result in poor performance, past a certain point increasing the number doesn't help.

to be best, and increasing beyond this point has no benefit. Though it is difficult to see, there is a slight upcurve for WMN and WMNS toward the right side of the graph; theoretically, there will always be a minimizing value of n given a value for STOPT's regret.

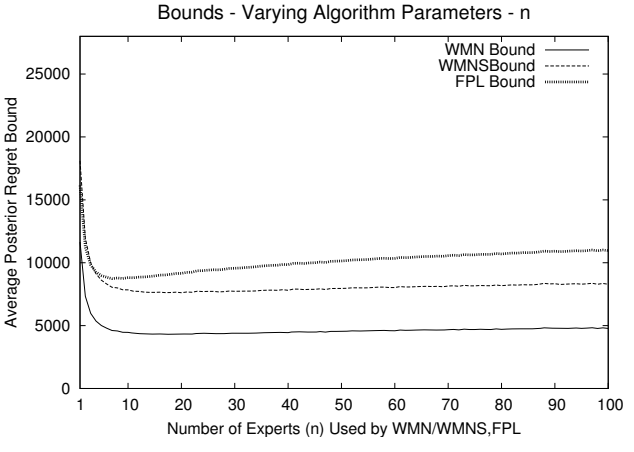


Figure 5: Theoretical bounds while varying the number of experts used, computed using the actual average reget suffered by STOPT.

5.2 Problem Parameters Now we turn our attention to how the various approaches perform under different problem conditions. For all of the tests in this section, WMN uses a $\beta = 0.5$, WMNS uses $\beta = 0.5, \delta = 0.3$, and FPL uses $\epsilon = 0.75$. Then number of experts/buckets, n, used by all three is 32. Given the

results so far, these seem to be typical naive choices which one might use in practice if no information about the problem is given.

In contrast, for all the tests in this section, we give SCARF and NORMAL the actual mean and standard deviation of the sequence to be used. Giving such perfect information about the input distribution represents a best case for these; comparing with typical naive implementations of the experts algorithms should give some insight as to their real-world applicability.

5.2.1 Per Item Profit: r Since we know that the worst case regret that can be suffered by MINIMAX depends on r and c, it will be interesting to look at a situation where we hold c constant to 1, and vary the per item profit r.

As in Section 5.1.1, all demands are drawn from the bounded normal N(25,15). Figure 6 shows the average regret for the various algorithms as we increase the value of r from 1 to 10. Notice that when $r = 1$, the correct order quantity is 0, as no net profit is possible in this situation. STOPT, MINIMAX, and the stochastic approaches all take this into account, and as such suffer no regret. The experts algorithms on the other hand aren't given information about r and c, and must adjust their operation over time as they normally do.

Overall, as r increases with respect to c, the cost of poor decisions is amplified in comparison with OPT (which is how regret is measured). Thus, we see that all regret curves increase as r increases, with NORMAL and SCARF tracking STOPT most closely, followed by WMN and WMNS, whose plots nearly overlap.

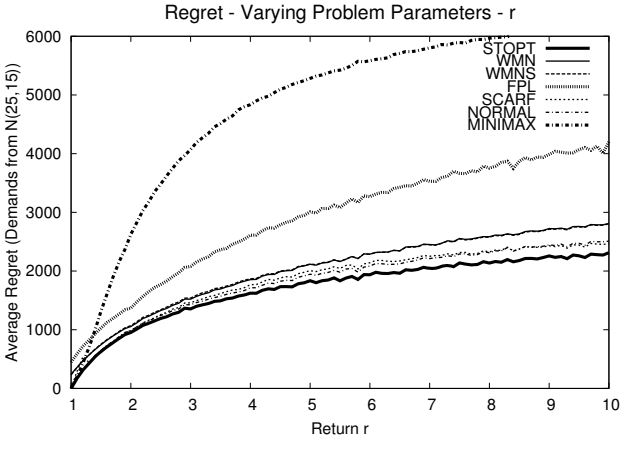


Figure 6: Average regret on sequences with demands drawn from N(25,15) bounded to $[10, 100]$, varying r and holding c at 1. SCARF and NORMAL are given perfect information, while WMN, WMNS, and FPL use relatively naive operating parameters.

5.2.2 Distribution: m,M Mix Perhaps the most important consideration for a multi-period newsvendor algorithm is how well it deals with the inherent demand uncertainty. We've already looked at the effects of various algorithm parameters, but there we used a fairly "tame" distribution for demand values based on the normal. Here, we'll look at a somewhat more difficult distribution: all 100 demand values will either be the minimum value m or the maximum value M. (Recall that these are 10 and 100, respectively.) Further, the sequences will be randomized so that where each type occurs is unknown. We still fix r to be 4 and c to be 1.

Figure 7 plots the regret of the WMN,WMNS, and FPL as we vary the number of minimum value m's which appear in the sequence. Thus, at 0 on the left half of the graph, all demands in the sequence are M's. In the middle at 50, each sequence is a random mixture of 50 m's and 50 M's.

In this figure we see a performance difference between WMN and WMNS, though this difference is still fairly small. All three algorithms do fairly well despite the large variance in demand values, tracking STOPT's regret in a somewhat linear fashion throughout the mix range.

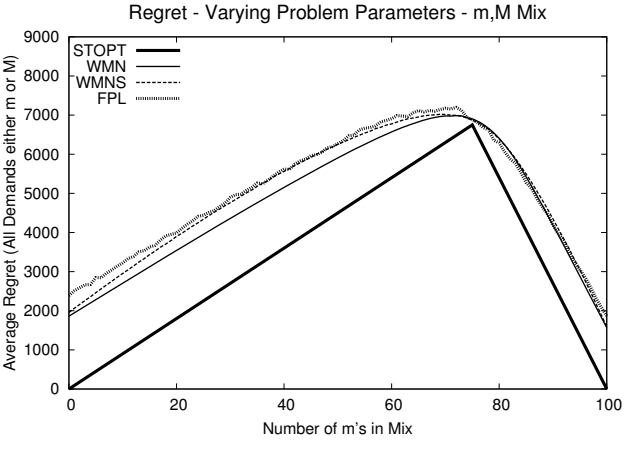


Figure 7: Average regret for m,M mix. Here, all demands in the 100 period sequence are either $m = 10$ or $M = 100$, and we vary how many of the demands are m's. The order the demands are presented in is randomized.

Figure 8, which plots the regret of the stochastic approaches and MINIMAX, shows several interesting characteristics. There are a few points in the curve where SCARF and NORMAL perform as well as STOPT, but for much of the range they suffer a significant amount of regret in spite of the fact that they are working with perfect information about the actual mean

and standard deviation. In comparison, the other algorithms perform similarly, if not better in some areas, given no a-priori information about the demand sequence.

MINIMAX manages the same regret for the entire range because whether a period demand is m or M, MINIMAX is designed to suffer the same regret. STOPT experiences the most regret when there are $\lceil t - t(c/r) \rceil = 75$ minimum demands and 25 maximum demands. Note than in this case, STOPT's perfect knowledge of the demand sequence doesn't help it fare any better than MINIMAX, which operates completely blind. (Both of these features can also be shown algebraically, for any values of r and c.)

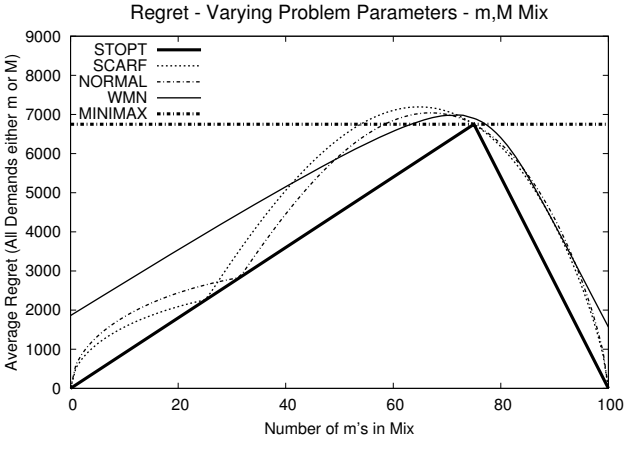


Figure 8: Average regret for m,M mix for the stochastic approaches as well as MINIMAX. Despite SCARF and NORMAL being given perfect information about the mean and standard deviation, they still suffer regret comparable to the other algorithms. (WMN's regret using $\beta = 0.5$ is also shown for comparison.)

5.2.3 Distribution: Shifting Normals, Algorithm Parameters Revisited Finally, for this last set of tests, we explore if WMNS can perform better than WMN when the input is characterized by dramatic "shifts" in the demand sequence. Because WMNS employs a weight limiting factor δ, it is theoretically able to adjust the relative weights of the experts more quickly, and thus change decisions more rapidly.

Here, our sequence length is now 400 periods. The first 100 demands are drawn from N(25,15), the second 100 are drawn from N(75,15), the third 100 are again from N(25,15), and the last 100 demands are drawn from N(75,15). As usual, all demands are bounded to $[10, 100]$, so the true means of each subsequence are about 29.3 and 73.4, respectively.

Figure 9 plots WMN's regret when WMN uses $\beta = 0.5$, FPL's regret when $\epsilon = 0.75$, and WMNS's regret varying the δ used from 0 to .99. WMNS also used a constant β of 0.5. As we can see, up to a point increasing δ leads to less regret, such that WMNS can outperform STOPT and achieve regret closer to that of SSTOPT. If δ is too high, however, we see an increase in regret, as fairly little weight adjustment is happening at all, limiting WMNS's learning ability.

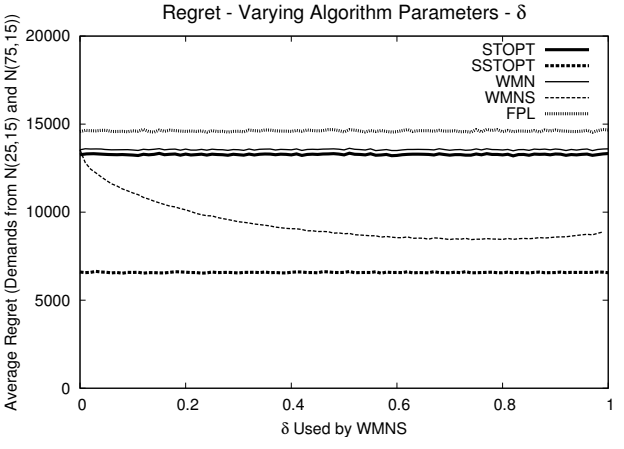


Figure 9: Regret of WMNS varying δ, the weight limiting parameter. For this plot, demand sequences were 400 periods long, with the first and third sets of 100 demands being drawn from N(25,15), and the second and fourth being drawn from N(75,15). SCARF and NORMAL, not shown in this plot, do approximately as well as STOPT given perfect information.

The increase in performance, however, comes at a steep price in terms of the theoretical regret bound, shown in Figure 10. bound for WMNS is computed from the actual average regret of SSTOPT which used the single best static order value on each of the four subsequences. This increase in the computed bound happens primarily because of the $(1-\delta)$ term in the denominator of the bound (in Theorem 4.2), as $\text{SSTOPT}_{TotalRegret}$ will generally be a fairly large number.

6 Conclusion

Looking at all of the figures and discussion in aggregate, we see that overall WMN and WMNS perform comparably to the traditional stochastic approaches SCARF and NORMAL, even when those approaches are given perfect information about the demand distribution. When the stochastic methods assume incorrect information, they suffer as expected.

Though the bounds given for FPL are comparable to the bounds for WMN, the actual performance for

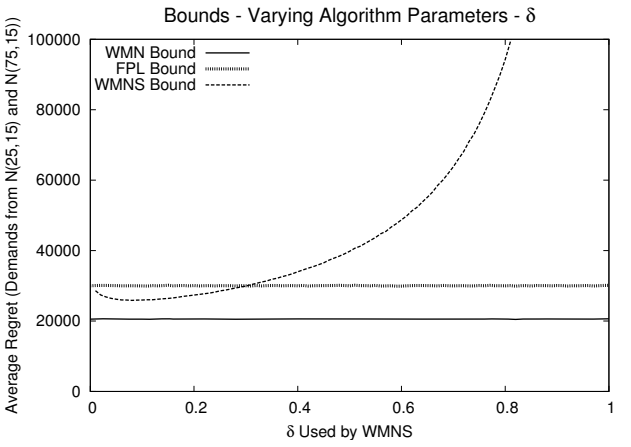


Figure 10: Regret bound of WMNS varying δ. The bound was computed in terms of SSTOPT's regret, where SSTOPT was allowed to use the static optimal decision for each 100 period subsequence.

this problem wasn't as good, except for in the more difficult $[m, M]$ distribution mix scenario. Nevertheless, all algorithms significantly outperformed the bounds given for all tests that we ran.

We believe part of the reason for this is that in the general experts problem, it is possible to have a single expert perform well in a period while all other experts simultaneously suffer maximum regret. In the newsvendor setting, this is not possible, as the optimal order for a single period suffers no regret, and the regret suffered increases linearly as one looks at order quantities on either side. Designing an algorithm which exploits this fact will be the focus of future research.

Philosophically speaking, designing approaches which successfully balance the competing goals of good worst case performance and acceptable average case performance is one of the most interesting and challenging areas of online algorithms research. Sometimes, it seems to be necessary to further restrict the input criteria to achieve good average case results. Other times, simple extensions to an algorithm can improve average case results without sacrificing worst case performance, such as the THREAT algorithm discussed in [4].

Of course, a solution isn't worth much if no one uses it. Brown and Tang surveyed 250 MBA students and 6 professional buyers, supplying them with simple newsvendor problems[5]. Very few of the subjects used the classical newsvendor solution as prescribed by NORMAL, though the approach was known to almost all. One possible explanation given is that the classical solution doesn't take into account risk preferences—buyers may be more comfortable underestimating demand to have a stronger guarantee on a particular profit rather than shoot for a higher profit with less certainty.

References

[1] R. Armstrong, D. Freitag, T. Joachims, and T. Mitchell. Webwatcher: A learning apprentice for the world wide web. In *1995 AAAI Spring Symposium on Information Gathering from Heterogeneous Distributed Environments*, 1995.

[2] E. Barnes, J. Dai, S. Deng, D. Down, M. Goh, H. C. Lau, and M. Sharafali. Electronics manufacturing service industry. The Logistics Institute–Asia Pacific, Georgia Tech and The National University of Singapore, Singapore, 2000.

[3] D. Bertsimas and A. Thiele. A data driven approach to newsvendor problems. Technical report, Massechusetts Institute of Technology, Cambridge, MA, 2005.

[4] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[5] A. O. Brown and C. S. Tang. The single-period inventory problem: A new perspective. *Decisions, Operations, and Technology Management*, 2000.

[6] S.E. Chick, H. Mamani, and D. Simchi-Levi. Supply chain coordination and the influenza vaccination. In *Manufacturing and Service Operations Management*. Institute for Operations Research and the Management Sciences, 2006.

[7] T. M. Cover and E. Ordentlich. On-line portfolio selection. In *Proceedings of the ninth annual conference on Computational Learning Theory*, pages 310–313, 1996.

[8] T. M. Cover and E. Ordentlich. Universal portfolios with side information. *IEEE Transactions on Information Theory*, 42(2), 1996.

[9] P. Demarzo, I. Kremer, and Y. Mansour. Online trading algorithms and robust option pricing. In *Proceedings of the thirty-eighth annual AMC Symposium on Theory of Computing*, pages 477–486, 2006.

[10] F. Y. Edgeworth. The mathematical theory of banking. *Journal of the Royal Statistical Society*, 1888.

[11] G. Gallego. Ieor 4000: Production management lecture notes. Available as `http://www.columbia.edu/~gmg2/4000/pdf/lect_07.pdf`.

[12] A. Kalai and S. Vempala. Efficient algorithms for online decision problems. *J. Comput. Syst. Sci.*, 71(3):291–307, 2005.

[13] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, pages 285–318, 1988.

[14] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Information and Computation*, pages 212–261, 1994.

[15] E. L. Porteus. *Foundations of Stochastic Inventory Theory*. Stanford University Press, Stanford, CA, 2002.

[16] A. Raman and M. Fisher. Reducing the cost of demand uncertainty through accurate response to early sales. *Operations Research*, 44(4):87–99, January 1996.

[17] L. J. Savage. The theory of statistical decisions. *Journal of the American Statistical Association*, 46:55–67, 1951.

[18] H. E. Scarf. A min-max solution of an inventory problem. In *Stanford University Press*, 1958.

[19] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[20] Eiji Takimoto and Manfred K. Warmuth. Path kernels and multiplicative updates. *J. Mach. Learn. Res.*, 4:773–818, 2003.

[21] C. L. Vairaktarakis. Robust multi-item newsboy models with a budget constraint. *International Journal of Production Economics*, pages 213–226, 2000.

[22] G. Yu. Robust economic order quantity models. *European Journal of Operations Research*, 100(3):482–493, 1997.

A WMN Operation and Bounds

Definitions In this section, we suppose we are going to play t periods of the newsvendor problem, with item cost c and item revenue r. We have access to n experts, each of which makes a prediction of the demand each period. (In Section A.1, we'll discuss how WMN actually chooses experts' predictions and give the final bounds.) In period j, each expert i predicts a demand of $x_i^{(j)}$ and the true demand is revealed to be $d^{(j)}$ at the end of the period. The experts are allowed to change their predictions any way they wish between periods; the only restriction is that $x_i^{(j)}$ and $d^{(j)}$ are within the interval $[m, M]$ for all i and j. WMN (Weighted Majority Newsvendor) will aggregate the predictions of the experts, and order an amount $\gamma^{(j)}$ for the j^{th} period.

Clearly, the optimal choice for period j would be $d^{(j)}$, so the true dynamic offline optimal's profit for period j, which we'll denote as $\mathsf{OPT}^{(j)}$, is $d^{(j)}(r-c)$. WMN's profit is $\mathsf{WMN}^{(j)} = \min\{d^{(j)}, \gamma^{(j)}\}r - \gamma^{(j)}c$, and the profit each expert i would have made is $\mathsf{EX}_i^{(j)} = \min\{d^{(j)}, x_i^{(j)}\}r - x_i^{(j)}c$.

Algorithm WMN Algorithm WMN operates as follows: each expert i is assigned an initial weight $w_i^{(1)} = 1$. Also, a weight adjustment parameter $\beta \in (0, 1]$ is chosen. In each period j, WMN orders an amount $\gamma^{(j)}$ which is the weighted average of the predictions of the experts: $\gamma^{(j)} = \sum_{i=1}^n w_i^{(j)} x_i^{(j)} / \sum_{i=1}^n w_i^{(j)}$.

In every period, after $d^{(j)}$ is revealed, we update each expert i's weight by some factor F: $w_i^{(j+1)} = w_i^{(j)} F$, where F satisfies

$$\beta^{f(d^{(j)}, x_i^{(j)})} \leq F \leq 1 - (1-\beta) f(d^{(j)}, x_i^{(j)}) \ .$$

We use $f(d^{(j)}, x_i^{(j)}) = \big(d^{(j)}(r-c) - \min\{d^{(j)}, x_i^{(j)}\}r + x_i^{(j)}c\big)/\mathbb{C}$, where $\mathbb{C} = \max\{(M-m)(r-c), (M-m)c\}$ is the maximum possible regret any prediction can suffer. Choosing $\mathbb{C}$ in this fashion guarantees that $0 \leq f(d^{(j)}, x_i^{(j)}) \leq 1$ for any valid $d^{(j)}$ and $x_i^{(j)}$, which allows us to ensure that such an update factor F exists[14]. Intuitively, $f(d^{(j)}, x_i^{(j)})$ gives a sense of the regret expert i would have suffered. In practice, we use the upper bound on F as the update factor.

Analysis For clarity, we define $s^{(j)} = \sum_{i=1}^n w_i^{(j)}$ to be the total sum of weights over the experts in period j. To prove bounds on the total regret of WMN, we begin as in [14] by showing a bound on $\ln(s^{(t+1)}/s^{(1)})$. ($s^{(t+1)}$ is the sum of weights at the end of the game, $s^{(1)}$ is the sum of weights at the start.) Because of the upper bound on each update factor F, we have that $s^{(j+1)}$ is less than or equal to:

$$\begin{aligned}
&\sum_{i=1}^n w_i^{(j)} \left[1 - (1-\beta) f(d^{(j)}, x_i^{(j)})\right] \\
&\quad = s^{(j)} - (1-\beta) \sum_{i=1}^n w_i^{(j)} f(d^{(j)}, x_i^{(j)}) \\
&\quad = s^{(j)} - (1-\beta) \left[\sum_{i=1}^n \frac{w_i^{(j)} d^{(j)}(r-c)}{\mathbb{C}}\right. \\
&\qquad \left. - \sum_{i=1}^n \frac{w_i^{(j)} \min\{d^{(j)}, x_i^{(j)}\} r}{\mathbb{C}} + \sum_{i=1}^n \frac{w_i^{(j)} x_i^{(j)} c}{\mathbb{C}}\right] \\
&\quad = s^{(j)} - (1-\beta) \left[\frac{s^{(j)} d^{(j)}(r-c)}{\mathbb{C}}\right. \\
&\qquad \left. - \frac{r}{\mathbb{C}} \sum_{i=1}^n \min\{w_i^{(j)} d^{(j)}, w_i^{(j)} x_i^{(j)}\} + \frac{\gamma^{(j)} s^{(j)} c}{\mathbb{C}}\right] .
\end{aligned}$$

We arrive at the last line by the definition of $s^{(j)}$ and $\gamma^{(j)}$. (Also, it must be noted that $d^{(j)}, x_i^{(j)}, w_i^{(j)} \geq 0$.) Now, by virtue of the fact that the summation over a minimum is less than or equal to the minimum of two

summations, the above is less then or equal to:

$$\begin{aligned}
& s^{(j)} - (1-\beta)\Bigg[\frac{s^{(j)}d^{(j)}(r-c)}{\mathbb{C}} \\
& \quad - \frac{r}{\mathbb{C}}\min\left\{\sum_{i=1}^{n} w_i^{(j)} d^{(j)}, \sum_{i=1}^{n} w_i^{(j)} x_i^{(j)}\right\} + \frac{\gamma^{(j)} s^{(j)} c}{\mathbb{C}}\Bigg] \\
&= s^{(j)} - s^{(j)}(1-\beta)\frac{1}{\mathbb{C}}\Bigg[d^{(j)}(r-c) \\
& \quad - \min\{d^{(j)}, \gamma^{(j)}\}r + \gamma^{(j)}c\Bigg] \\
&= s^{(j)}\left[1 - (1-\beta)f(d^{(j)}, \gamma^{(j)})\right] \ .
\end{aligned}$$

So, over the entire sequence,

$$\begin{aligned}
s^{(t+1)} &\leq s^{(1)} \prod_{j=1}^{t}\left[1 - (1-\beta)f(\gamma^{(j)}, d^{(j)})\right] \ , \\
\ln(s^{(t+1)}/s^{(1)}) &\leq \sum_{j=1}^{t} \ln\left[1 - (1-\beta)f(d^{(j)}, \gamma^{(j)})\right] \\
&\leq \sum_{j=1}^{t} -(1-\beta)f(d^{(j)}, \gamma^{(j)}) \ .
\end{aligned}$$

Going a step further, we have the beginnings of a bound on the total regret of WMN:

$$\sum_{j=1}^{t} f(d^{(j)}, \gamma^{(j)}) \leq \frac{\ln(s^{(1)}/s^{(t+1)})}{1-\beta} \ .$$

Now we'll bound $s^{(t+1)}$. We let $m_i = \sum_{j=1}^{t} f(d^{(j)}, x_i^{(j)})$ be the total "adjusted regret" for expert i. By the lower bound on our update factor F, and the fact that $w_i^{(1)} = 1$ for all i:

$$\begin{aligned}
s^{(t+1)} &\geq \sum_{i=1}^{n} w_i^{(1)} \beta^{m_i} \\
&\geq \beta^{m_i}, \ \forall i \ , \\
\ln(s^{(1)}/s^{(t+1)}) &\geq \ln(n) - m_i \ln(\beta), \ \forall i \ .
\end{aligned}$$

Combining this with the above, we finally get, for any expert i,

$$\begin{aligned}
& \sum_{j=1}^{t} f(d^{(j)}, \gamma^{(j)}) \\
& \quad \leq \frac{\ln(n) + \ln\left(\frac{1}{\beta}\right)\sum_{j=1}^{t} f(d^{(j)}, x_i^{(j)})}{1-\beta}, \\
& \sum_{j=1}^{t}\left(\mathsf{OPT}^{(j)} - \mathsf{WMN}^{(j)}\right) \\
& \quad \leq \frac{\mathbb{C}\ln(n)}{1-\beta} + \frac{\ln\left(\frac{1}{\beta}\right)\sum_{j=1}^{t}\left(\mathsf{OPT}^{(j)} - \mathsf{EX}_i^{(j)}\right)}{1-\beta} \ .
\end{aligned}$$

Thus, we have our total regret for WMN bounded in terms of the total regret for our best performing expert (since the bound holds for all experts).

A.1 Placing Experts in Buckets So far, we have a bound on the regret of WMN in terms of the regret of the best expert. Now, we're interested in deriving a similar bound in terms of the regret of the offline static optimal, STOPT.

The approach we take is to let each of our n experts consistently predict a unique demand for all t newsvendor periods. We divide the overall range $[m, M]$ into n "buckets," such that each bucket has the same minimax regret should the demand fall in that bucket. There are $n+1$ bucket endpoints, $\{q_0, q_1, \ldots, q_n\}$. As Vairaktarakis shows[21], for a given bucket i (with endpoints q_{i-1} and q_i) the minimax regret order quantity is $\big(q_i(r-c) + cq_{i-1}\big)/r$, which results in a maximum regret of $\big(c(q_i - q_{i-1})(r-c)\big)/r$ when the demand is at either endpoint.

To achieve our "many buckets, same regret" goal, we simply need to choose the endpoints according to:

$$q_i = \frac{i(M-m)}{n} + m \ .$$

We then let expert i consistently predict the optimal order quantity for the i^{th} bucket:

$$x_i^{(j)} = \frac{q_i(r-c) + cq_{i-1}}{r} = \frac{(ir-c)(M-m)}{rn} + m, \ \forall j \ .$$

Claim A.1. *For a t-period newsvendor game, there exists an expert i such that the difference in i's profit and any given static offline algorithm is at most*

$$\frac{c(M-m)(r-c)t}{rn} \ .$$

Proof. Suppose the static offline algorithm chooses a value which lies in the i^{th} bucket. The expert who

minimizes his difference in profit is the i^{th} expert, since regret increases as demand moves further from the expert's prediction, and each expert has the same regret at his bucket boundaries. For a single period, the true demand could fall in one of three places: below the bucket, in the bucket, or above the bucket.

If the demand d falls below the bucket ($d < q_{i-1}$), the maximum difference in profit occurs if the static algorithm has chosen the lowest point in the bucket at q_{i-1}. The difference in profit is then

$$dr - q_{i-1}c - (dr - x_i^{(j)}c) = \frac{c(M-m)(r-c)}{nr} .$$

If the demand falls in the i^{th} bucket, we know from above that the maximal difference in profit (which is now equivalent to regret within this bucket, since the static algorithm can now predict the demand exactly) is the same thing. Similarly, if the demand falls above the i^{th} bucket, the worst case is when the static algorithm is at the top of the bucket at q_i, and the difference in profit can again be shown to be the same.

All three cases give identical worst case profit difference. Summing over all t periods, we have the claim.

Since the claim holds for any static offline algorithm, it also holds for the static offline optimal algorithm, STOPT. Using the notation from Section A, the claim implies that there exists an expert i such that

$$(1.1)\quad \sum_{j=1}^{t}\left(\mathsf{STOPT}^{(j)} - \mathsf{EX}_i^{(j)}\right) \leq \frac{c(M-m)(r-c)t}{nr} .$$

Using the substitution

$$\sum_{j=1}^{t}(\mathsf{OPT}^{(j)} - \mathsf{EX}_i^{(j)}) = \sum_{j=1}^{t}(\mathsf{OPT}^{(j)} - \mathsf{STOPT}^{(j)}) + \sum_{j=1}^{t}(\mathsf{STOPT}^{(j)} - \mathsf{EX}_i^{(j)}) ,$$

in the bound shown at the end of Section A, and the bound implied by Equation 1.1, we have the following theorem:

THEOREM A.1. *The total regret experienced by* WMN *for a t period newsvendor game with per item cost c, per item revenue r, and all demands within $[m, M]$ satisfies*

$$\begin{aligned}\mathsf{WMN}_{TotalRegret} &= \sum_{j=1}^{t}(\mathsf{OPT}^{(j)} - \mathsf{WMN}^{(j)}) \\ &\leq \frac{\mathbb{C}\ln(n)}{1-\beta} + \frac{\ln\left(\frac{1}{\beta}\right)c(M-m)(r-c)t}{nr(1-\beta)} \\ &\quad + \frac{\ln\left(\frac{1}{\beta}\right)\sum_{j=1}^{t}(\mathsf{OPT}^{(j)} - \mathsf{STOPT}^{(j)})}{1-\beta}\end{aligned}$$

where $\mathbb{C} = \max\{(M-m)(r-c), (M-m)c\}$ is the maximum possible single period regret, n is the number of experts used by WMN*, and β is the update parameter used.*

The first term, which depends on the maximum possible single period regret, is independent of the number of periods. This gives the following corollary:

COROLLARY A.1. *The average per period regret of* WMN *approaches*

$$\begin{aligned}&\frac{\ln\left(\frac{1}{\beta}\right)c(M-m)(r-c)}{nr(1-\beta)} \\ &\quad + \frac{\ln\left(\frac{1}{\beta}\right)\sum_{j=1}^{t}(\mathsf{OPT}^{(j)} - \mathsf{STOPT}^{(j)})}{t(1-\beta)}\end{aligned}$$

as the number of periods becomes arbitrarily large.

B WMNS Operation and Bounds

In this section, we'll look at an extension of the results and provide an algorithm which does well in the face of an input sequence which can be decomposed into subsequences, where for each subsequence a different expert does well. This algorithm is analogous to the "shifting target" version of Littlestone and Warmuth's Weighted Majority algorithm. Using the same method of selecting experts as in Section A.1, we show a bound on the regret of the algorithm in terms of the "semi-static offline optimal" algorithm SSTOPT. SSTOPT is a version of STOPT described above which is allowed to change its choice $k-1$ times during the sequence.

B.1 Doing as Well as the Best Expert on any Subsequence

Definitions We again consider playing t periods of a newsvendor game, with all demands drawn from $[m, M]$, per item cost c, and per item revenue r. The algorithm described here, WMNS (Weighted Majority Newsvendor Shifting), will however operate slightly differently compared to WMN. These differences will

allow us to partition the sequence of periods into subsequences, and show that WMNS will perform as well as the best expert *for each subsequence*, despite the fact that nothing is known about when the subsequences start or end or how many there are.

Aside from the definitions mentioned in Section A, we need to define two subsets of the n experts: $\mathcal{UPD}$, those which are "updatable", and $!\mathcal{UPD}$, those which are not. As we will see, WMNS uses a weight limiting factor δ. For any period j, $\mathcal{UPD}$ is defined as those experts whose weights satisfy $w_i^{(j)} > (\delta \sum_{i=1}^n w_i^{(j)})/n$. $!\mathcal{UPD}$ contains all other experts.

Algorithm WMNS WMNS operates as WMN, with a couple of notable differences. First, a weight limiting parameter $\delta \in (0,1]$ is chosen in addition to the weight update parameter $\beta \in (0,1]$. Initially all experts' weights are set to 1. In each period j, WMNS orders the amount $\gamma^{(j)}$ which is the weighted average of the predictions of the *updatable* experts: $\gamma^{(j)} = \sum_{i\in\mathcal{UPD}} w_i^{(j)} x_i^{(j)} / \sum_{i\in\mathcal{UPD}} w_i^{(j)}$.

In every period, after the actual demand $d^{(j)}$ is revealed, we update *only the experts in* $\mathcal{UPD}$ by the same factor F described in Section A: $w_i^{(j+1)} = w_i^{(j)} F$ where

$$\beta^{f(d^{(j)},x_i^{(j)})} \le F \le 1-(1-\beta)f(d^{(j)},x_i^{(j)}) \ .$$

Again, we use $f(d^{(j)},x_i^{(j)}) = \big(d^{(j)}(r-c) - \min\{d^{(j)},x_i^{(j)}\}r + x_i^{(j)}c\big)/\mathbb{C}$, where $\mathbb{C} = \max\{(M-m)(r-c),(M-m)c\}$.

Analysis We start by showing that for any subsequence, WMNS performs nearly as well as the best expert for that subsequence. This means that if an expert does quite well for some subsequence, and then for another (later) subsequence another expert does quite well, WMNS will track the change quickly.

We let $s^{(j)} = \sum_{i=1}^n w_i^{(j)}$ be the total sum of weights of all experts in period j, $s_{\mathcal{UPD}}^{(j)} = \sum_{i\in\mathcal{UPD}} w_i^{(j)}$ be the sum of weights of updatable experts, and $s_{!\mathcal{UPD}}^{(j)} = \sum_{i\in!\mathcal{UPD}} w_i^{(j)}$ be the sum of weights of not updatable experts. We define *init* to be the index of the first period of the subsequence, and fin to be the index of the last period of the subsequence.

We are interested in finding a bound for $\ln(s^{(fin+1)}/s^{(init)})$. First we note that, by the operation of WMNS, for any period j and any expert i, $w_i^{(j)} \ge \beta\delta s^{(j)}/n$. This is also true for the first period, because $1 \ge \beta\delta s^{(1)}/n = \beta\delta$.

By the bound on the update factor F and the mechanism of WMNS, we have that $s^{(j+1)}$ is less than or equal to:

$$\begin{aligned}
&\sum_{i\in\mathcal{UPD}} w_i^{(j)}\Big[1-(1-\beta)f(d^{(j)},x_i^{(j)})\Big] + \sum_{i\in!\mathcal{UPD}} w_i^{(j)} \\
&= s^{(j)} - (1-\beta)\sum_{i\in\mathcal{UPD}} w_i^{(j)} f(d^{(j)},x_i^{(j)}) \\
&= s^{(j)} - (1-\beta)\Bigg[\sum_{i\in\mathcal{UPD}} \frac{w_i^{(j)} d^{(j)}(r-c)}{\mathbb{C}} \\
&\quad - \sum_{i\in\mathcal{UPD}} \frac{w_i^{(j)}\min\{d^{(j)},x_i^{(j)}\}r}{\mathbb{C}} + \sum_{i\in\mathcal{UPD}} \frac{w_i^{(j)}x_i^{(j)}c}{\mathbb{C}}\Bigg] \\
&\le s^{(j)} - (1-\beta)\frac{1}{\mathbb{C}}\Bigg[s_{\mathcal{UPD}}^{(j)} d^{(j)}(r-c) \\
&\quad - \min\{s_{\mathcal{UPD}}^{(j)}\gamma^{(j)}, s_{\mathcal{UPD}}^{(j)} d^{(j)}\}r + s_{\mathcal{UPD}}^{(j)}\gamma^{(j)}c\Bigg] \ .
\end{aligned}$$

We arrive at the last line by the definition of $s_{\mathcal{UPD}}^{(j)}$ and $\gamma^{(j)}$, as well as moving the summation inside of the min expression as in Section A. Next we need a lower bound for $s_{\mathcal{UPD}}^{(j)}$:

$$\begin{aligned}
s_{\mathcal{UPD}}^{(j)} &= s^{(j)} - \sum_{i\in!\mathcal{UPD}} w_i^{(j)} \\
&\ge s^{(j)} - \sum_{i\in!\mathcal{UPD}} \delta s^{(j)}/n \\
&\ge s^{(j)}(1-\delta) \ .
\end{aligned}$$

So, we have that

$$\begin{aligned}
s^{(j+1)} &\le s^{(j)} - (1-\beta)s_{\mathcal{UPD}}^{(j)} f(d^{(j)},\gamma^{(j)}) \\
&\le s^{(j)}\Big[1-(1-\beta)(1-\delta)f(d^{(j)},\gamma^{(j)})\Big] \ .
\end{aligned}$$

Over all periods in this subsequence,

$$s^{(fin+1)} \le s^{(init)} \prod_{j=init}^{fin}\Big[1-(1-\beta)(1-\delta)f(d^{(j)},\gamma^{(j)})\Big] \ ,$$

$$\ln\left(\frac{s^{(fin+1)}}{s^{(init)}}\right) \le \sum_{j=init}^{fin} -(1-\beta)(1-\delta)f(d^{(j)},\gamma^{(j)}) \ ,$$

$$\sum_{j=init}^{fin} f(d^{(j)},\gamma^{(j)}) \le \frac{\ln(s^{(init)}/s^{(fin+1)})}{(1-\beta)(1-\delta)} \ .$$

In any period j, because WMNS doesn't update weights below $\beta\delta s^{(j)}/n$, we know that $w_i^{(init)} > \beta\delta s^{(init)}/n$. If we let $m_i = \sum_{j=init}^{fin} f(d^{(j)},x_i^{(j)})$, we have by the lower bound on the update factor F:

$$\begin{aligned}
s^{(fin+1)} \ge w_i^{(fin+1)} &\ge w_i^{(init)}\beta^{m_i}, \ \forall i \\
&\ge \frac{\beta\delta s^{(init)}}{n}\beta^{m_i}, \ \forall i
\end{aligned}$$

Consequently, for all experts i:

$$\sum_{j=init}^{fin} f(d^{(j)}, \gamma^{(j)}) \leq \frac{\ln\left(\frac{s^{(init)}}{s^{(fin+1)}}\right)}{(1-\beta)(1-\delta)} \leq \frac{\ln\left(\frac{s^{(init)}}{\beta\delta s^{(init)}\beta^{m_i}/n}\right)}{(1-\beta)(1-\delta)} = \frac{\ln\left(\frac{n}{\beta\delta}\right) + m_i \ln\left(\frac{1}{\beta}\right)}{(1-\beta)(1-\delta)} .$$

By substitution and rearrangement similar to that in Section A, we arrive at the following theorem:

THEOREM B.1. *For any subsequence of newsvendor periods indexed from* init *to* fin *and any expert* i, WMNS*'s regret satisfies*

$$\mathsf{WMNS}_{SubseqRegret} = \sum_{j=init}^{fin} (\mathsf{OPT}^{(j)} - \mathsf{WMNS}^{(j)}) \leq \frac{\mathbb{C}\ln\left(\frac{n}{\beta\delta}\right)}{(1-\beta)(1-\delta)} + \frac{\ln\left(\frac{1}{\beta}\right)\sum_{j=init}^{fin}(\mathsf{OPT}^{(j)} - \mathsf{EX}_i^{(j)})}{(1-\beta)(1-\delta)}$$

B.2 Doing as Well as SSTOPT SSTOPT, the "Semi-Static Offline Optimal" algorithm is a slightly stronger version of STOPT, which is allowed to change its order choice $k-1$ times for the whole t period newsvendor game. Consider subsequence l, $(1 \leq l \leq k)$, which is the subsequence where SSTOPT is using it's l^{th} choice. For this subsequence, SSTOPT acts as a static offline optimal for periods from $initl$ to $finl$, the beginning and ending indices of l. We define $t_l = finl + 1 - initl$; the number of periods in subsequence l. (Thus, $\sum_{l=1}^{k} t_l = t$.) In essense, we are now considering k individual newsvendor games against different static optimal algorithms.

By defining experts according to the same construction of Section A.1, we can show that for any subsequence l,

$$\sum_{j=initl}^{finl} (\mathsf{OPT}^{(j)} - \mathsf{WMNS}^{(j)}) \leq \frac{\mathbb{C}\ln\left(\frac{n}{\beta\delta}\right)}{(1-\beta)(1-\delta)} + \frac{\ln\left(\frac{1}{\beta}\right)c(M-m)(r-c)t_l}{nr(1-\beta)(1-\delta)} + \frac{\ln\left(\frac{1}{\beta}\right)\sum_{j=initl}^{finl}(\mathsf{OPT}^{(j)} - \mathsf{SSTOPT}^{(j)})}{(1-\beta)(1-\delta)} .$$

Summing over all k subsequences (again, WMNS requires no knowledge of how long subsequences are, or even how many there are), we ultimately reach the following theorem:

THEOREM B.2. *The total regret experienced by* WMNS *for a t period newsvendor game with per item cost c, per item revenue r, and all demands within $[m, M]$ satisfies*

$$\mathsf{WMNS}_{TotalRegret} = \sum_{j=1}^{t}(\mathsf{OPT}^{(j)} - \mathsf{WMNS}^{(j)}) \leq \frac{k\mathbb{C}\ln\left(\frac{n}{\beta\delta}\right)}{(1-\beta)(1-\delta)} + \frac{\ln\left(\frac{1}{\beta}\right)c(M-m)(r-c)t}{nr(1-\beta)(1-\delta)} + \frac{\ln\left(\frac{1}{\beta}\right)\sum_{j=1}^{t}(\mathsf{OPT}^{(j)} - \mathsf{SSTOPT}^{(j)})}{(1-\beta)(1-\delta)}$$

where $\mathbb{C} = \max\{(M-m)(r-c), (M-m)c\}$ is the maximum possible single period regret, n is the number of experts used by WMNS*, β is the update parameter used, and δ is the weight limiting parameter used.*

When $k = t$, then SSTOPT is equivalent to OPT, though in this case the bound becomes useless because of the $k\mathbb{C}$ factor in the first term. When k is constant, however, we can note the following corollary:

COROLLARY B.1. *When the number of changes k* SSTOPT *is allowed is constant, the average per period regret of* WMNS *approaches*

$$\frac{\ln\left(\frac{1}{\beta}\right)c(M-m)(r-c)}{nr(1-\beta)(1-\delta)} + \frac{\ln\left(\frac{1}{\beta}\right)\sum_{j=1}^{t}(\mathsf{OPT}^{(j)} - \mathsf{SSTOPT}^{(j)})}{t(1-\beta)(1-\delta)}$$

as the number of periods becomes arbitrarily large.

C FPL Operation and Bounds

FPL, for Follow the Perturbed Leader, was developed by Kalai and Vempala in [12]. (In this paper, they give two versions of the algorithm, FPL and FPL *. We use the latter for our problem.) In the experts setting, the algorithm keeps track of the total regret suffered by each expert. When a decision needs to be made, a random cost is added to each expert's sum regret so far, and FPL chooses the expert with the lowest overall regret.

Definitions FPL takes as input a "randomness" parameter ϵ. For the bounds given to hold, ϵ must be in the range $(0, 1]$, however the algorithm will still operate with larger values.

Two other values are used by FPL and the bounds given in [12], A and D. A is defined as the maximum of the sum of all experts' regret for a single period, and D is the maximum "diameter difference" between

two decisions. (Because FPL is applicable to general decision making settings, these values have a more precise meaning which we won't go into.) However, in Section 2 of [12], the authors note that for the experts problem, D is 1, and A is the maximum regret of a single expert for one period. In our case, this is then $A = \mathbb{C} = \max\{(M-m)(r-c), (M-m)c\}$.

Algorithm FPL For each period, let s_i be the total regret suffered by expert i so far. For each expert i, choose a perturbation factor p_i from the exponential distribution with rate $\epsilon/2A$. The best perturbed expert so far then is $\text{argmax}_i\{s_i + p_i\}$. Use the prediction of this expert.

Note that this algorithm is a specific, limited version of the general algorithm FPL *.

Bounds of FPL Kalai and Vempala give the following theorem, which bounds the regret of FPL in terms of the regret of the best performing expert:

THEOREM C.1. *(Due to Kalai and Vempala.) The expected regret of* FPL *satisfies*

$$E[FPL_{TotalRegret}] \leq (1+\epsilon)Min_{TotalRegret} + \frac{4AD(1+\ln(n))}{\epsilon}$$

Where $Min_{TotalRegret}$ is the regret of the best performing expert.

If we place experts in n buckets accoring to Section A.1, we know that the best expert won't suffer more than

$$\frac{c(M-m)(r-c)t}{nr}$$

extra regret from the true static optimal on any t period newsvendor sequence. Using the notation of Appendix A, we can now give a theorem similar to Theorem A.1:

THEOREM C.2. *The total regret experienced by* FPL *for a t period newsvendor game with per item cost c, per item revenue r, and all demands within $[m, M]$ satisfies*

$$E\left[\text{FPL}TotalRegret\right] \leq \frac{4\mathbb{C}(1+\ln(n))}{\epsilon} + \frac{(1+\epsilon)c(M-m)(r-c)t}{nr} + (1+\epsilon)\sum_{j=1}^{t}(\text{OPT}^{(j)} - \text{STOPT}^{(j)})$$

where $\mathbb{C} = \max\{(M-m)(r-c), (M-m)c\}$ is the maximum possible single period regret, n is the number of experts used by FPL*, and ϵ is the randomness parameter used.*

$\text{OPT}^{(j)}$ and $\text{STOPT}^{(j)}$ are the profits of OPT *and* STOPT*, respectively, in period j.*

D Newsvendor Extensions

One frequently discussed extension to the classic newsvendor problem considers, in addition, per item overstock costs c_o and per item understock costs c_u. Understock costs can be used to express customer ill will due to unmet demand, or perhaps in the vaccine ordering setting to express costs to the economy due to unvaccinated portions of the workforce. Overstock costs may represent extra storage costs or disposal costs for outdated products such as unwanted consumer electronics. Vairaktarakis[21] and Bertsimas and Thiele[3] also discusses these extensions.

The profit function for a prediction x in a period is given by $\min\{d, x\}r - xc - \max\{(d-x)c_u, (x-d)c_o\}$. Because of the negative max term in the expression, we can use this model and the bounds will follow through a proof similar to that of Section A using the trick of moving the summation inside the max expression. Of course, we'll also need to adjust $\mathbb{C}$ and $f(d^{(j)}, x_i^{(j)})$ accordingly. Using the same bucket endpoints as in Section A.1 and letting expert i consistently choose the minimax regret order quantity

$$x_i^{(j)} = \frac{q_i(r-c+c_u) + q_{i-1}(c+c_o)}{r+c_o+c_u} = \frac{i(M-m)}{n} - \frac{(M-m)(c+c_o)}{n(r+c_o+c_u)} + m, \ \forall j,$$

we can get the following bound for WMN:

$$\text{WMN}_{TotalRegret} \leq \frac{\mathbb{C}_2 \ln(n)}{1-\beta} + \frac{\ln\left(\frac{1}{\beta}\right)(M-m)(c+c_o)(r-c+c_u)t}{n(1-\beta)(r+c_o+c_u)} + \frac{\ln\left(\frac{1}{\beta}\right)\sum_{j=1}^{t}(\text{OPT}^{(j)} - \text{STOPT}^{(j)})}{1-\beta}$$

where $\mathbb{C}_2 = \max\{(M-m)(r-c+c_u), (M-m)(c+c_o)\}$. Similar bounds can be had for WMNS and FPL.

Another commonly discussed extension considers per item salvage profit s (usually $s < c$), wherein unused items can be sold for a guaranteed smaller profit. In this version, the profit function for a prediction x is $\min\{d, x\}r - xc + \max\{0, (x-d)s\}$. Here, the max term is positive, so the expression won't follow through a proof technique similar to the one used for WMN and WMNS. This indicates that, unfortunately, salvage profits are incompatible with these approaches, though a bound can still be had for FPL.

Routing in Graphs with Applications to Material Flow Problems

Rolf H. Möhring*

Material flow problems are complex logistic optimization problems. We want to utilize the available logistic network in such a way that the load is minimized or the throughput is maximized. This lecture deals with these optimization problems from the viewpoint of network flow theory and reports on two industrial applications: (1) contolling material flow with automated guided vehicles in a container terminal (cooperation with HHLA), and (2) timetabling in public transport (cooperation with Deutsche Bahn and Berlin Public Transport). The key ingredient for (1) is a very fast real-time algorithm which avoids collisions, deadlocks, and other conflicts already at route computation, while for (2) it is the use of integer programs based on special bases of the cycle space of the routing graph.

References

[1] E. Gawrilow, E. Köhler, R. H. Möhring, and B. Stenzel, *Dynamic routing of automated guided vehicles in real-time*, Tech. Rep. 039-2007, TU Berlin, 2007, http://www.math.tu-berlin.de/coga/publications/techreports/2007/Report-039-2007.html

[2] C. Liebchen and R. H. Möhring, *The modeling power of the periodic event scheduling problem: Railway timetables - and beyond*, in Algorithmic Methods for Railway Optimization, F. Geraets, L. Kroon, A. Schöbel, D. Wagner, and C. D. Zaroliagis, eds., vol. 4359 of Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, 2007, pp. 3–40.

*Institut für Mathematik, Technische Universität Berlin, Berlin, Germany. Research supported by DFG Research Center Matheon

How much Geometry it takes to Reconstruct a 2-Manifold in $\mathbb{R}^3$

Daniel Dumitriu[†] Stefan Funke[‡] Martin Kutz[†] Nikola Milosavljevic[§]

Abstract

Known algorithms for reconstructing a 2-manifold from a point sample in $\mathbb{R}^3$ are naturally based on decisions/predicates that take the geometry of the point sample into account. Facing the always present problem of round-off errors that easily compromise the exactness of those predicate decisions, an exact and robust implementation of these algorithms is far from being trivial and typically requires the employment of advanced datatypes for exact arithmetic as provided by libraries like CORE, LEDA or GMP. In this paper we present a new reconstruction algorithm, one of whose main novelties is to throw away geometry information early on in the reconstruction process and to mainly operate combinatorially on a graph structure. As such it is less susceptible to robustness problems due to round-off errors and also benefits from not requiring expensive exact arithmetic by faster running times. A more theoretical view on our algorithm including correctness proofs under suitable sampling conditions can be found in a companion paper [3].

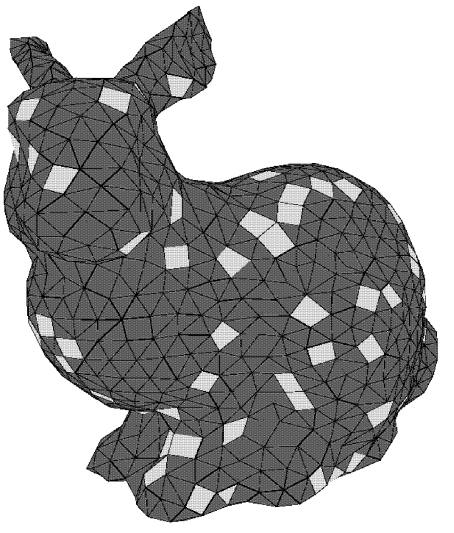


Figure 1: Output of our algorithm for the Stanford Bunny. Due to the conservative adjacency creation, some faces (light) are non-triangular, which can easily be triangulated later on, though.

1 Introduction

Robust Geometric Computation Geometric algorithms use geometric predicates in their conditionals (e.g. does a point r lie to the left or right of an oriented line through p and q). A common strategy for the exact implementation of geometric algorithms is to evaluate all geometric predicates exactly. While floating-point filters have proved to be quite efficient both in theory and practice to speed-up the exact evaluation of predicates, they tend to become less efficient for more complicated (i.e. higher degree) predicates like the *insphere predicate*[1] which is the basis for all algorithms based on the Delaunay tetrahedralization in $\mathbb{R}^3$.

The evaluation of a geometric predicate amounts to the computation of the sign of an arithmetic expression. Round-off errors during floating-point computation might easily lead to reporting a wrong sign of such an arithmetic expression, which most of the time has disastrous consequences ([7]). A floating-point filter evaluates the expression using floating-point arithmetic and also computes an error bound to determine whether the floating point computation is reliable. If the error bound does not suffice to prove reliability, the expression is re-evaluated using exact arithmetic. The quality of the error bounds typically deteriorates, though, with the complexity of the predicate expression and hence, more predicate decisions require the fallback of expensive exact arithmetic computation.

A different approach to the robustness problem is to design the algorithm to be able to cope with round-off errors right from the beginning. The difficulty of designing robust algorithms is illustrated in [4, 8, 9]. In [9] Sugihara et al. develop an algorithm for computing the Voronoi diagram of a set of points whose termination is guaranteed by certain *purely combinatorial* graph properties; geometric predicate evaluations are only used to steer the execution of the algorithm in a certain direction. In particular, their algorithm would also terminate if the outcome of all geometric predicates was completely randomized.

In this paper we present an algorithm for recon-

[†]Max-Planck-Institut für Informatik, Saarbrücken, Germany, dumitriu@mpi-inf.mpg.de

[‡]Ernst-Moritz-Arndt-Universität Greifswald, Germany, stefan.funke@uni-greifswald.de

[§]Stanford University, Stanford, U.S.A, nikolam@stanford.edu

[1]The *insphere predicate* for points p, q, r, s, t decides whether point t lies inside, on or outside the sphere defined by p, q, r, s.

structing a 2-manifold in $\mathbb{R}^3$ in the spirit of the work by Sugihara et al. The main idea is that only at the beginning of the algorithm we require the exact evaluation of a few *simple* (i.e. low-degree) predicates and then essentially work combinatorially without evaluating any geometric predicates. Nevertheless we show (both experimentally as well as theoretically in a companion paper [3]) that the output of our algorithm faithfully resembles the original 2-manifold. Most importantly, our algorithm produces this output *without* any evaluation of a complicated geometric test like the insphere predicate in $\mathbb{R}^3$, which is the basis of all related Voronoi-based reconstruction algorithms.

Related Work The problem of reconstructing a surface Γ in $\mathbb{R}^3$ from a finite point sample V has attracted a lot of attention both in the computer graphics community as well as in the computational geometry community. While in the former the emphasis is mostly on algorithms that work 'well in practice', the latter has focused on algorithms that come with a theoretical guarantee: if the point sample V satisfies a certain *sampling condition*, the output of the respective algorithm is guaranteed to be 'close' to the original surface.

In [1], Amenta and Bern proposed a framework for rigorously analyzing algorithms reconstructing smooth closed surfaces. They define for every point $p \in \Gamma$ on the surface the *local feature size* $\mathrm{lfs}(p)$ as the distance of p to the *medial axis*[2] of Γ. A set of points $V \subset \Gamma$ is called a ε-sample of Γ if $\forall p \in \Gamma\ \exists s \in V : |sp| \leq \varepsilon \cdot \mathrm{lfs}(p)$. Many algorithms have been proposed that determine a collection of Delaunay triangles which form a piecewise linear surface that is topologically equivalent to the original surface and converges to the latter both point-wise as well as in terms of the surface normals as the sampling density goes to infinity ($\varepsilon \to 0$). Common to almost all those algorithms is the fact that they require the computation of a Voronoi diagram/Delaunay triangulation of a point set in $\mathbb{R}^3$ which incurs both an inherent $\Omega(n^2)$ running time as well as the need for the exact evaluation of the *insphere predicate*.

In [5] Funke and Milosavljevic present an algorithm for computing *virtual coordinates* for the nodes of a wireless sensor network which are themselves unaware of their location. Their approach crucially depends on a subroutine to identify a provably planar subgraph of a communication graph that is a quasi-Unit-Disk graph. A similar subroutine will also be used in our surface reconstruction algorithm presented in this paper.

[2] The medial axis of Γ is defined as the set of points which have at least two closest points on Γ.

Our Contribution We propose a new graph-based algorithm for reconstructing a 2-manifold in $\mathbb{R}^3$. Our algorithm fundamentally differs from previous approaches in two respects: first, it mainly operates combinatorially on a graph structure, which is derived from the original geometry; secondly, the created adjacencies/edges are "conservative" in a sense that two samples are only connected if in all reasonable reconstructions those two samples are adjacent. Interestingly we can show in the theoretical companion paper [3], though, that conservative edge creation only leads to small, constant-size faces in the respective reconstruction, hence the topology of the original surface is faithfully captured. While the theoretical analysis requires an absurdly high sampling density – like most of the above mentioned algorithms do – this paper shows that our algorithm is actually very practical for real-world datasets. Due to the local nature of computation in our algorithm there is also potential for use e.g. in parallel computing or external memory scenarios.

2 Our Algorithm

The main idea of our algorithm is first to derive a graph $G(V)$ which captures mutual proximity information of the samples in V and then decide on adjacencies between (some of the) samples of V only based on the connectivity structure of $G(V)$. To keep the presentation simple, we assume that V is a *uniform*[3] ε-sample from a closed smooth 2-manifold Γ in $\mathbb{R}^3$. In practice, sample sets are indeed often close to uniform and even small non-uniformities in the sample set never prevented our algorithm to work in practice. In theory, a preprocessing stage can enforce *local uniformity* which is sufficient to prove correctness of our algorithm. See [3] for more details. The high-level view of our algorithm is as follows:

1. Construct a local neighborhood graph $G(V)$ by creating an edge from every point v to all other points v' with $|vv'| \leq O(\varepsilon)$.
2. Compute a subsample S of V as a maximal k-hop stable set in $G(V)$
3. Construct the *graph Voronoi diagram* for V with respect to the subsample S
4. Identify adjacencies between elements in S by inspection of the graph Voronoi diagram
5. Output faces of the reconstruction as minimal cycles in the graph induced by the adjacencies between elements in S

In step one we create a unit-disk graph of the point set connecting two samples iff their distance is less

[3] V is a uniform ε-sample if for any $p \in \Gamma\ \exists s \in V$ with $|ps| \leq \varepsilon$.

than some constant times ε. In step two we compute a maximal subsample $S \subset V$ with the property that there are no two $s_1, s_2 \in S$ closer than k hops in the graph $G(V)$. Such a maximal (not maximum!) k-hop stable set can easily computed in a greedy fashion. In step three, we essentially compute a discrete analogue of the geometric Voronoi diagram but using $G(V)$ as a discrete approximation of the space between the subsample vertices in S. That is, we assign each node $v \in V$ its closest (in terms of hop-distance) node in S (breaking ties according to node IDs). This partitions V into tiles consisting of nodes which have the same closest $s \in S$ which we also call *landmark* or *site*. Two elements $s_1, s_2 \in S$ are then declared adjacent in step four, iff the respective tiles are touching each other by a sufficient amount, i.e. the number of nodes in one of the tiles with direct neighbors in the other tile is above some threshold. Finally, in step five, we collect the adjacencies to actually create faces of the reconstruction of our algorithm. Observe that only Step 1. involves the geometry of the sample set V, all other steps can be implemented fully combinatorially. Also, the only geometric predicate required in step one is the comparison of coordinates and distances between input points – very low degree predicates that can be evaluated exactly very efficiently using known floating-point filter techniques. We want to emphasize two things:

1. Our algorithm does not compute a reconstruction of V with respect to the original surface Γ (involving all $v \in V$) but only of a subsample $S \subset V$. For sufficiently dense sample sets, this reconstruction of S with respect to Γ still captures the topology of Γ. There is also a generic (and rather simple) way of incorporating the remaining points of V into the reconstruction (requiring some higher-degree geometric predicates, though), see [6]. In practice, with the presence of scanning devices producing millions or even billions of point samples, the fact that we only compute a reconstruction of a subsample is not a real concern.

2. The reconstruction computed by our algorithm does not only contain triangular faces, but also larger faces (though of size less than 5 in practice and of constant size in theory). If required, these non-triangular faces can be triangulated easily (requiring only low-degree geometric predicates).

In [3] we provide a theoretical justification for the correctness of our algorithm under the ε-sampling condition proposed in [1]. The core components of the correctness proof are:

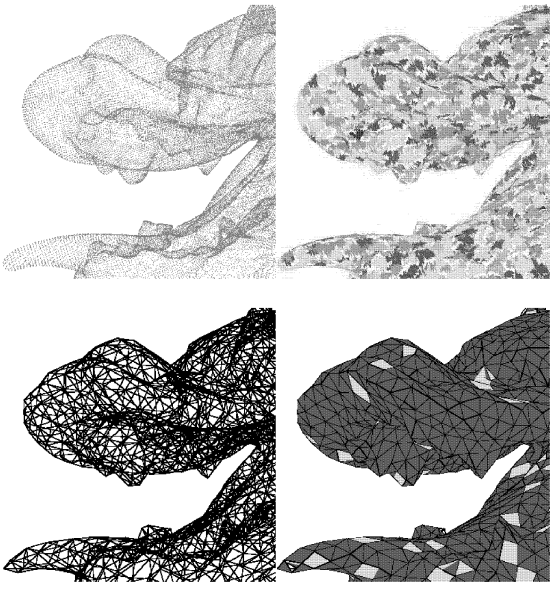


Figure 2: The Dragon, head close-up: point cloud, graph Voronoi diagram, identified adjacencies between subsamples and discovered faces (left to right, top to bottom)

- We show that the local neighborhood graph corresponds locally to a quasi-unit-disk graph for a set of points in the plane.
- The identified adjacencies locally form a planar graph.
- The faces of this graph have bounded size.

What does this mean? The graph that we constructed on the subsample of points S is a *mesh* that is locally planar and covers the whole 2-manifold. The mesh has the nice property that all its *cells* (aka faces) have constant size (number of bounding vertices). Therefore its connectivity structure faithfully reflects the topology of the underlying 2-manifold.

THEOREM 2.1. *([3]) The adjacencies created between samples in S form a graph which faithfully reflects the topology of the underlying 2-manifold.*

As can be read from Table 1, the running times are heavily dominated by the construction of the local neighborhood graph. We expect considerable improvements by an order of magnitude by performing batched nearest neighbor queries (and not asking for the κ nearest neighbors separately for each sample).

3 Implementation

We have implemented our algorithm in `C++`, using `Qt4` and `OpenGL` for rendering and the graphical user

Model	Points	Open	Read	Octr	Ngb	Mis	Vor	Cdm	Cyc	Total	CoCone
Bunny	35947	0.83	0.39	0.14	3.42	0.45	0.29	0.22	0.05	5.79	29
Bone	177907	0.83	1.62	0.97	20.45	2.77	1.70	1.29	0.62	30.25	137
Hand	327323	0.83	3.08	2.13	36.87	4.77	2.81	2.31	0.86	53.66	1216
Dragon	435545	0.83	4.79	2.97	49.03	6.44	4.01	3.59	1.33	72.99	761
Buddha	543524	0.83	6.10	4.10	63.18	8.30	5.11	4.20	1.89	93.71	876
Blade	882954	0.83	6.95	6.23	104.32	14.31	8.45	7.09	4.74	152.92	>1800

Table 1: Time spent in different stages (seconds)

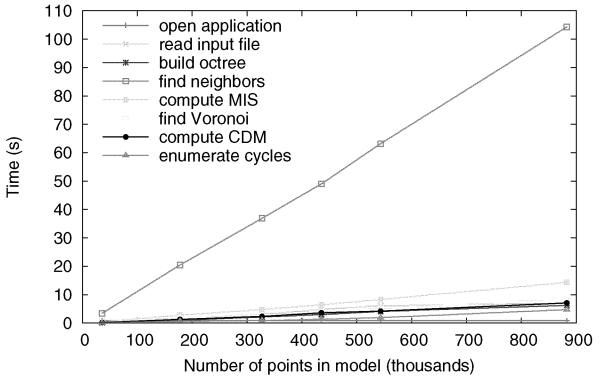


Figure 3: Plot of the time spent in different stages of the algorithm for different input sizes

interface.

For our implementation we make the simplifying assumption that the set of input data points V is a *locally uniform* sampling, which is typically true for sample data acquired by laser scanning (this means in particular that apart from a lower bound on the density of the sample there is also an upper bound, see the companion paper for a more precise definition). In that way we can determine the local neighborhood graph $G(V)$ (first step of our algorithm) by connecting a sample to its κ nearest neighbors ($\kappa = 15$ worked well in our experiments).

Also making use of the assumption that V is a locally uniform sample, we can declare two samples $s_1, s_2 \in S$ adjacent as follows: s_1 and s_2 are adjacent if the number of edges linking a sample v_1 (belonging to the graph Voronoi cell of s_1) to a sample v_2 (belonging to s_2's cell) is above a certain threshold a ($a = 7$ worked well in our experiments). Figure 2 illustrates the main steps of our algorithm using a close-up of the *Dragon* model.

3.1 Implementation Details We provide here supplementary information about our design choices.

Input. The input represents the points in a point cloud P and comes in a simplified `PLY` format, a simple object description designed as a convenient format for working with polygonal models. In our case there is no special header, since we are only interested in the points themselves. That is why each line in the file corresponds to a sample point and is a simple list of its three coordinates. We simply report but skip any line not conforming to this format.

Spatial representation. In order to allow for efficient positional queries, the points in V are stored in an octree. An octree is a structure suited for storing spatial data, which takes into consideration the relative distribution of points, backed up by an 8-ary tree. Each node in the tree represents a cube. The tree root corresponds to the bounding box of the point cloud and stores all the points. This cube is split in two along each axis, yielding eight smaller cubes.

A threshold o (called octree *granularity*) is imposed on the number of points that can lie within any node/cube. When the threshold is exceeded, the cube is split into eight smaller cubes of equal volume. They provide a refinement of the larger cube and this is recorded by making them the children of the tree node corresponding to the big cube. The process of splitting continues recursively and stops when all the tree leaves contain less than o points.

κ-nearest neighbors. The octree structure is particularly useful when looking for the k-nearest neighbors of a given point.

First, we perform a traversal of the tree starting at the root and aiming at ending in the leaf that contains the query point. The decisions about which direction to follow at each step in this traversal are made based on the geometric coordinates of the query point; three simple comparisons with the middle coordinates of the current node's bounding box along each axis are enough to decide which child to move to. As we go down in the traversal, we insert all siblings of nodes on this path into a priority queue based on the Euclidean distance from the query point to the circumsphere of the current node's corresponding cube. The priority queue's first element will contain the node with the *smallest* distance.

After finishing the traversal, a *top-κ*-like algorithm

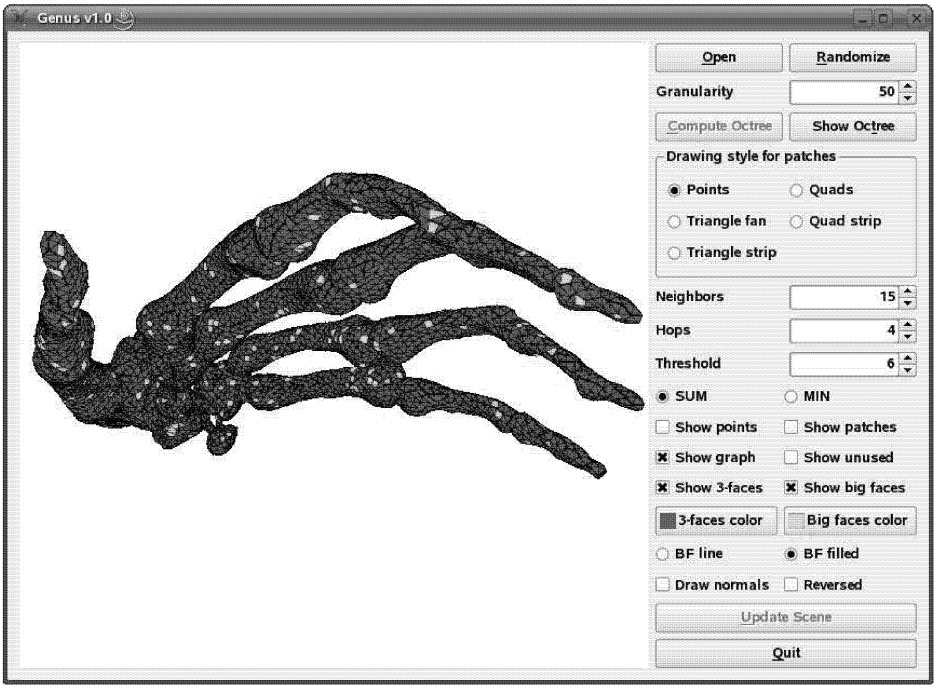


Figure 4: Main window of our application

is used. The first node in the priority queue is extracted and if it is a leaf, a test is performed for the points stored in the node to check whether they are eligible for the set of κ-nearest neighbors. For internal nodes, their eight children are inserted back into the queue. The algorithm ends when the distance d_{max} to the furthest nearest-neighbor cannot be improved, i.e. when the distance to the circumsphere of the node currently popped out of the queue is larger than d_{max}.

Neighborhood graph. We compute an (undirected) neighborhood graph $G_N = (V_N, E_N)$ by taking as vertices all the initial sample points V. For each of them we compute its κ-nearest neighbors as shown above, and we create an edge between points and each of its nearest neighbors discovered in this way.

Independent set. In the next step, a maximal independent set S in G_N is computed, such that any two vertices in S are at least k hops away.

To this end, we use a straightforward greedy algorithm: in the beginning, all vertices of G_N are marked as eligible. We choose an eligible vertex v and perform a breadth-first search to discover all vertices reachable in less than k hops from v, which are then marked as ineligible. This process continues until there are no more eligible vertices left.

Graph Voronoi diagram. The vertices in S are used as landmark nodes (sites) for a graph Voronoi diagram in the neighborhood graph G_N. Since the distance function for this diagram is simply the number of hops in the graph (i.e. all edges have unit weight), we use a parallel breadth-first search.

The processing queue is initialized with all the landmarks, then when a new vertex v is first seen, it is assigned to the vertex u from which it was reached. If u had already been assigned to a landmark ℓ (i.e. it is not a landmark itself), we assign v also to ℓ. In this way in the end we know for each vertex its "dominating" landmark, and we can easily determine the graph Voronoi cells.

Combinatorial Delaunay map. We create cell adjacencies in a conservative manner, leading to a so called combinatorial Delaunay map $CDM(S) = (V_{CDM(S)}, E_{CDM(S)})$. Let S_1 and S_2 be two graph Voronoi cells; the number $b_{S_1S_2}$ of points in S_1 with at least one (1-hop) neighbor in S_2 is calculated. If $b_{S_1S_2} + b_{S_2S_1}$ is greater than some threshold a, the points s_1 and s_2, corresponding to Voronoi cells S_1 and S_2 respectively, are added to $V_{CDM(S)}$, and the edge (s_1, s_2) is added to $E_{CDM(S)}$.

Face enumeration. The adjacencies on S are further used for face detection. Faces correspond to elementary cycles in this graph, with the supplementary constraint that any edge is allowed to appear in at most two cycles. Since we are interested in finding faces aka small cycles, we need an algorithm that enumerates the graph cycles in increasing order of cycle length. To this end, we implemented a simple enumeration algorithm

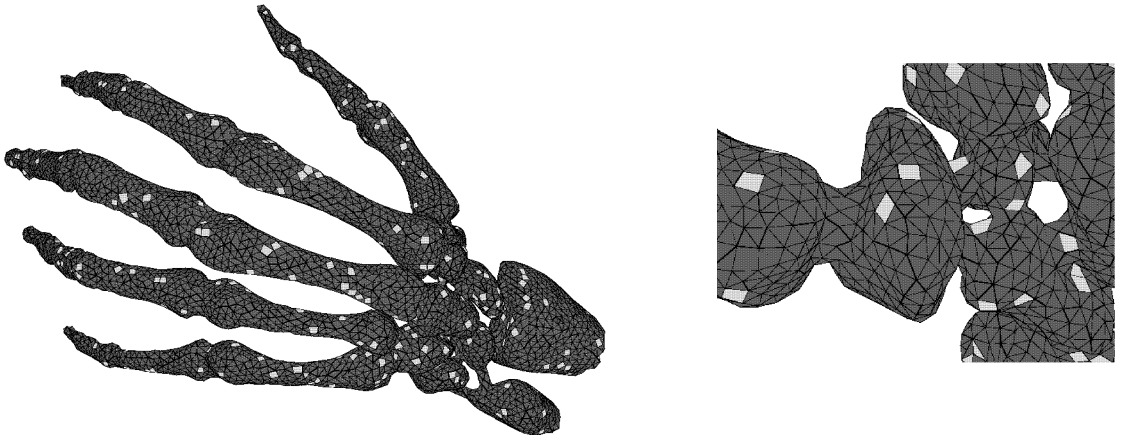


Figure 5: Skeleton hand: full view (left), metacarpi (right)

which first outputs cycles of length 3, then 4, and so forth, always only considering those edges which have not already been used in two cycles before.

4 Experimental Evaluation

4.1 Benchmarking on standard Data Sets Our reconstruction application was compiled with `g++ 3.3.5`, optimization `-O3`, and evaluation was made on a machine with a Pentium 4 processor at 3 GHz, 1 GB of RAM, running Debian Linux kernel version `2.6.13`. A screenshot of the main window of our implementation is given in Figure 4.

We benchmarked our implementation[4] on publicly available data sets obtained from laser scans of physical models, most of them from the *Large Geometric Models Archive* at Georgia Institute of Technology and the *3D Scanning Repository* at Stanford University.

In Table 1 you can see a detailed account of the running times spent in different parts of our implementation (*Octr* refers to the octree construction, *Ngb* to the construction of the local neighborhood graph; MIS, *Vor*, CDM denote respectively the three phases to construct S (the subsample), the tile partitioning and $CDM(S)$ (the adjacencies between the elements of the subsample), *Cyc* the routine to identify face cycles). We also ran[5] the CoCone algorithm [2] on the same datasets to give a rough comparison with other algorithms. We want to emphasize, though, that the Cocone algorithm does more than ours as it incorporates *all* sample points into the reconstruction (which would be done in a post-processing step of our algorithm which we expect to take only a fraction of the overall time). Figure 3 shows a plot of the same data. We never ran into any robustness problems with our algorithm, even when only employing floating-point arithmetic (the only real predicate that we require is the comparison of distances between points). The CoCone algorithm – as far as the authors reported to us – though, requires exact arithmetic for reliable operation due to its far more complex geometric tests like the insphere predicate.

In Figure 1 you can see the output of our algorithm for the dataset *Bunny*; the light color denotes non-triangular faces. Figure 6 shows the largest model tested (*Blade*), with almost 900,000 sample points. On the left, a full view of the model; at its right edge, we can observe the ragged structure, and on the lower part the reconstructed curved areas. Top right, the long sharp edge has been correctly discovered; we can also clearly recognize the concavity of the blade. Bottom right: close-up on the lower side of the model, where a screw hole pierces through the blade from one side to the other. In Figure 5 we see the model *Hand*. In the full view we observe that the phalanges are correctly individualized, as well as the wrist. On the right: close-up on the metacarpi: the holes in-between are visible; the algorithm detects the narrow spaces between the bones. Figure 7 shows one of the most difficult models (*Happy Buddha*) because of its curved features: the face, the hand-supported object, the necklace, and especially the folded outfit embellished with religious symbols. This causes sharp transitions, holes and concave regions. On top right, close-up on discovered openings in the cape. On bottom right, details of the bottom of the small round three-legged seat on which Buddha stands; its rims were correctly emphasized, as well as the decorative embossed rings just above the seat legs.

[4] The source code and some models are available online at `http://www.mpi-inf.mpg.de/~dumitriu/work/genus`

[5] We thank Tamal K. Dey for providing us with an executable of the CoCone code.

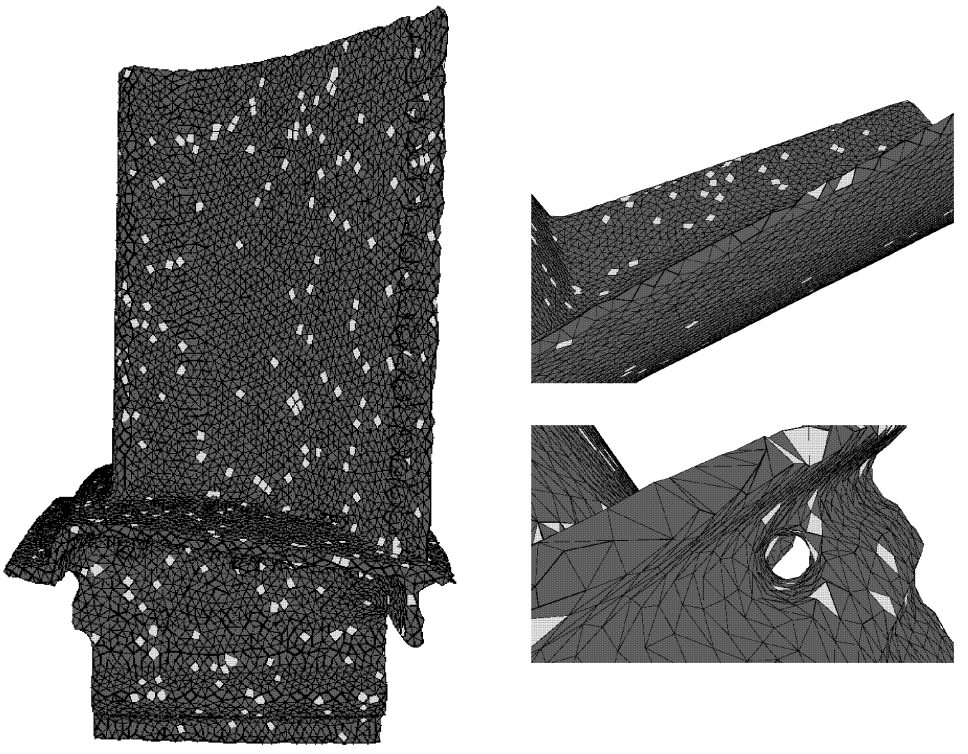


Figure 6: Turbine blade: full view (left), cutting edge and screw hole (right)

4.2 Parameter Variation We examine more closely the effect of varying parameters k and a on the experimental results obtained. In the following we use the model *Stanford Bunny*, fixing the other parameters at $o = 50$ (the octree is built with at most fifty original points stored in any leaf), and $\kappa = 15$ (for each input point, we find its fifteen nearest neighbors).

As a supplementary measure of the reconstruction quality, we also computed a value corresponding to the *genus* of the reconstruction if it forms a 2-manifold. The *genus* is a topologically invariant property, defined as the largest number of non-intersecting simple closed curves that can be drawn on the surface without separating it. Roughly speaking, it is the number of handles of an orientable 2-manifold.

The genus of a surface, also called the *geometric genus*, is related to the Euler's formula. In fact, Euler's formula is a particular version (for simply connected polyhedra, i.e. of genus 0) of the general relation

$$n - e + f = 2 - 2g \tag{4.1}$$

where n = number of vertices, e = number of edges, f = number of faces, and g = geometric genus value.

Solving out Equation 4.1 for g, we get

$$g = \frac{2 - n + e - f}{2} \tag{4.2}$$

We computed the genus value of our Bunny reconstruction according to the formula in Equation 4.2. A genus different than zero shows that something went wrong in the reconstruction (either the reconstruction does not form a 2-manifold or it does but exhibits handles). The results are shown in Table 2 and Figure 8.

Dependence on k To determine the dependence on k, we fixed the threshold a at 7 and varied k. For very small values of k, there is not enough "room" to identify reasonable adjacencies, hence the reconstruction does not patch up to a 2-manifold, leading to bad genus values. As we increase the parameter, things improve quickly (with a strange anomaly at $k = 7$ which we cannot explain at this time), of course at the cost of a much coarser subsampling.

Dependence on a This time we fixed k at 5 and varied a. A small value yields a denser combinatorial Delaunay map, with a higher percentage of triangles, but the genus number is extremely large (since there are too many adjacencies and the reconstruction does

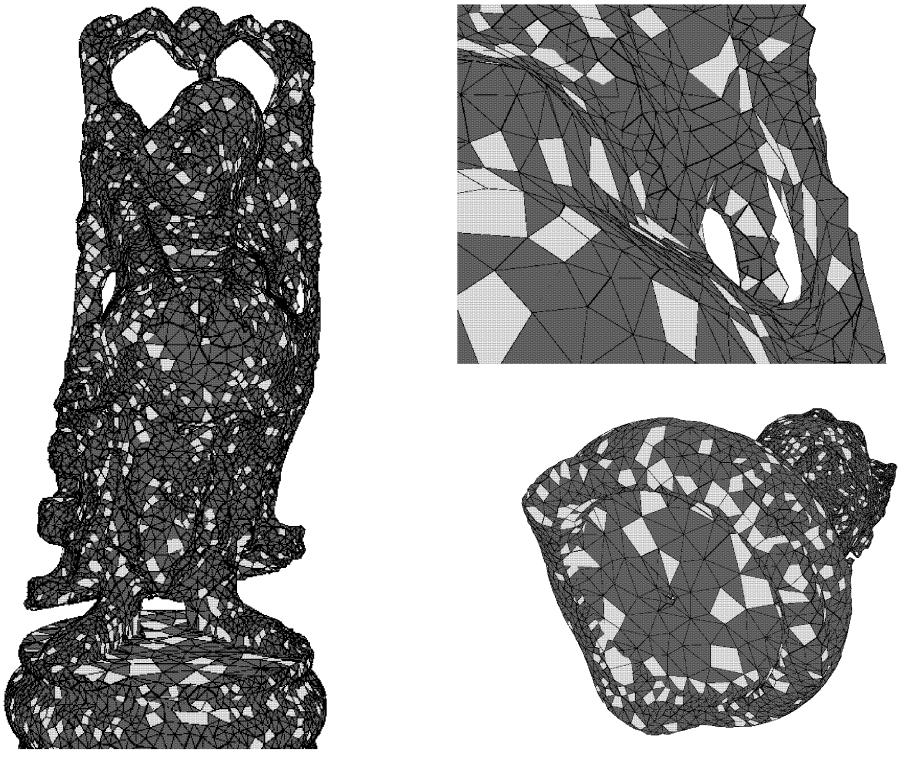


Figure 7: Happy Buddha: full view (left), side holes and bottom ridge (right)

k	faces	genus value
2	6415	62.5
3	3199	5
4	1620	1
5	976	0
7	457	1.5
14	94	0
15	80	0

a	faces	triangles	genus value
2	1024	99.4%	43
3	1023	99.5%	24
6	998	97.3%	1.5
7	976	95.3%	0
11	898	87.0%	0
15	778	73.3%	3.5
20	543	48.4%	18.5

Table 2: Dependence on k (left, for $a = 7$) and a (right, for $k = 5$)

not form a 2-manifold). Increasing the value causes a drastic drop in adjacencies and finally leads to a 2-manifold with correct genus. With larger values of a, of course, there are more and more non-triangular faces. Too large values for a lead to too few adjacencies and hence the reconstruction does not form a 2-manifold.

Overall, we consider k the most crucial parameter for our algorithm, which essentially determines the 'coarseness' of the subsample S. According to the theory in [3], k must be chosen very large to guarantee sufficient density of $CDM(S)$; of course, this comes at the cost of a very coarse subsample S (see Figure 8, top right). A too large k might 'smooth out' important details of the model. A too small choice of k, on the other hand, even in practice allows only for very few certified adjacencies (see Figure 8, top left). This results in larger faces. In our experiments, a value of $k = 5$ has proven to perform very well in practice, see Figure 8, top center.

4.3 Further Examples In Figure 9 we can see the Dragon model. The spurs on front and back legs are visible, as well as the front claw clenched on a ball. On bottom, a close-up of the scale ridge on the back; we can notice that the individual scales are clearly delimited.

The output of our algorithm on the Stanford Bunny can be seen in Figure 10. The main features are correctly detected, including paws, snout, and ears. On bottom, a close-up on the carved ears, whose concavity can be easily noticed (dots represent the original sample

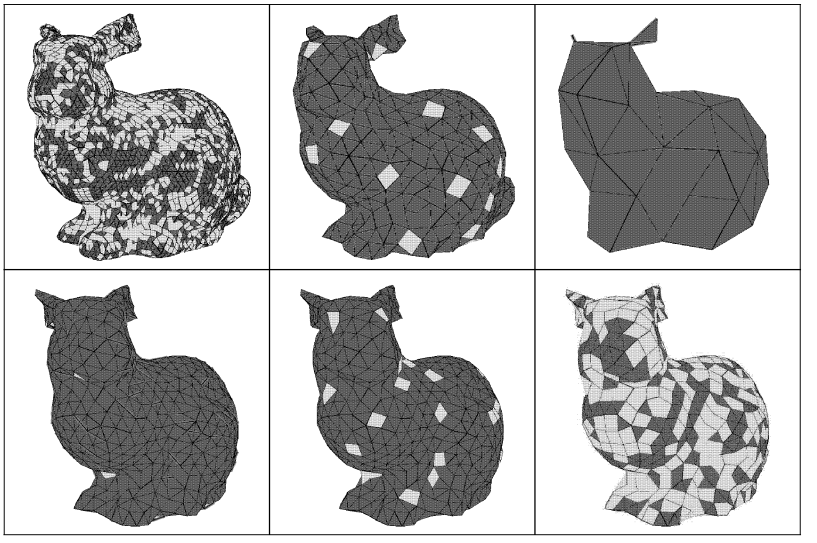


Figure 8: Effect of varying parameters k (top, for values $2, 5, 14$) and a (bottom, for values $2, 7, 15$); the thick red edges are not part of any face

points).

5 Outlook

Theoretically, our approach has the potential to work for reconstructing 2-manifolds even in higher dimensions; it does not extend to non-2-manifolds, though, as the "planarity property" of a graph that our algorithm crucially depends on (see [3] for more details) does not have an equivalent for non-2-manifolds.

On the practical side, we intend to complete our implementation and incorporate the pruned sample points into the reconstruction via weighted Delaunay triangulations. In the future, it might also be interesting to apply our algorithm to massive datasets; the inherently local computation and decision making exhibits nice locality properties which might be of use both in a parallel computing as well as an external memory scenario. Maybe the most interesting aspect of this paper is the fact that we were able to produce reasonable reconstructions of scanned 3D objects in a robust manner using only very simple geometric predicates (comparison of distances between points).

References

[1] N. Amenta and M. Bern. Surface reconstruction by voronoi filtering. In *Proc 14th annual Sympos on Computational geometry*, pages 39–48, New York, USA, 1998. ACM Press.

[2] N. Amenta, S. Choi, T. K. Dey, and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. In *Proc 16th annual Sympos on Computational geometry*, pages 213–222, New York, USA, 2000. ACM Press.

[3] D. Dumitriu, S. Funke, M. Kutz, and N. Milosavljevic. On the locality of reconstructing a 2-manifold in $\mathbb{R}^3$. unpublished manuscript. http://www.mpi-inf.mpg.de/∼funke/Papers/LocalRecon.pdf.

[4] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Symposium on Computational Geometry*, pages 334–341, 1991.

[5] S. Funke and N. Milosavljevic. Network Sketching or: "How Much Geometry Hides in Connectivity? – Part II". In *Proc. ACM-SIAM Sympos Discrete Algorithms*, pages 958–967, 2007.

[6] S. Funke and E. Ramos. Smooth-surface reconstruction in near-linear time. In *Proc. ACM-SIAM Sympos Discrete Algorithms*, pages 781–790. long version at http://www.mpi-sb.mpg.de/∼funke/Papers/SODA02/SSRINLT_ext.pdf, 2002.

[7] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *12th Annual European Symposium on Algorithms*, volume 3221 of *LNCS*, pages 702–713, Bergen, Norway, 2004. Springer.

[8] V.J. Milenkovic. Verifiable implementation of geometric algorithms using finite precision arithmetic. *Artif. Intell.*, 37(1-3):377–401, 1988.

[9] K. Sugihara, Y. Ooishi, and T. Imai. Topology-oriented approach to robustness and its applications to several Voronoi-diagram algorithms. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 36–39, 1990.

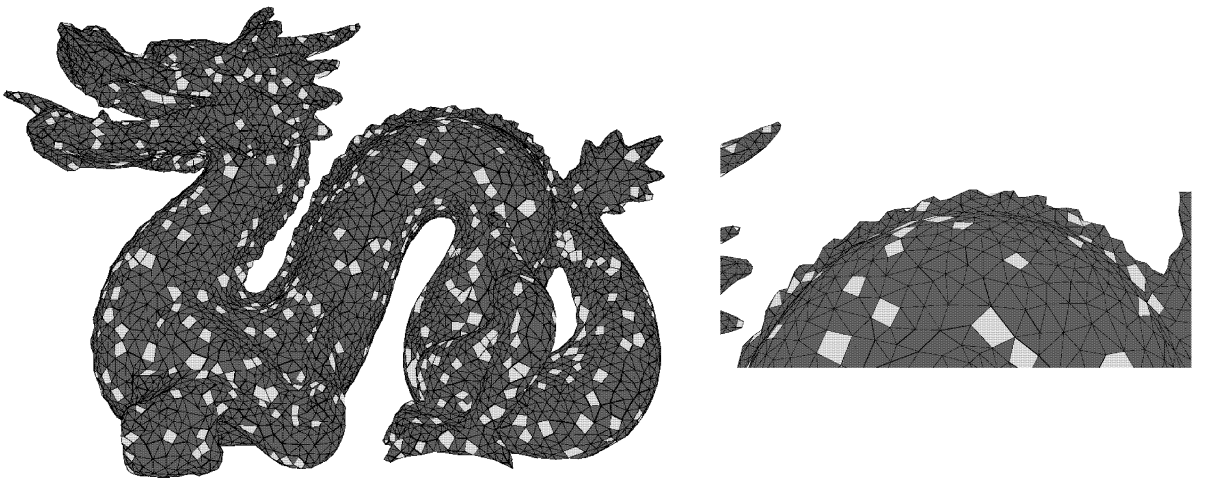


Figure 9: Dragon: full view (left), back scales (right)

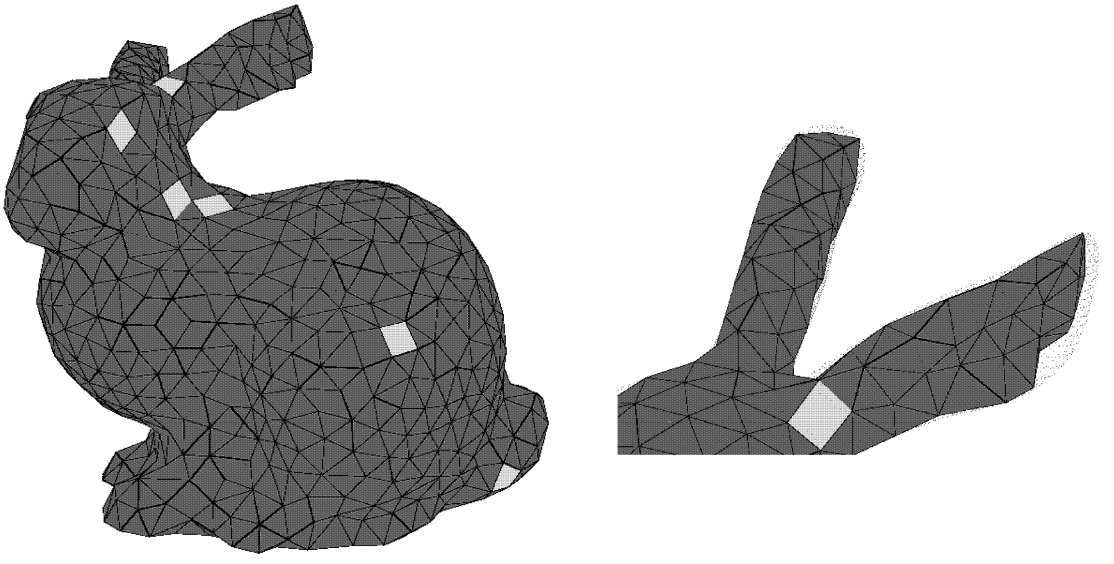


Figure 10: Stanford bunny: full view (left), carved ears (right)

Geometric Algorithms for Optimal Airspace Design and Air Traffic Controller Workload Balancing

Amitabh Basu* Joseph S. B. Mitchell† Girishkumar Sabhnani†
*Carnegie Mellon University, Pittsburgh, PA 15213
†Stony Brook University, Stony Brook, NY 11794

Abstract

The National Airspace System (NAS) is designed to accommodate a large number of flights over North America. For purposes of workload limitations for air traffic controllers, the airspace is partitioned into approximately 600 *sectors*; each sector is observed by one or more controllers. In order to satisfy workload limitations for controllers, it is important that sectors be designed carefully according to the traffic patterns of flights, so that no sector becomes overloaded. We formulate and study the airspace sectorization problem from an algorithmic point of view, modeling the problem of optimal sectorization as a geometric partition problem with constraints. The novelty of the problem is that it partitions data consisting of trajectories of *moving* points, rather than static point set partitioning that is commonly studied. First, we formulate and solve the 1d version of the problem, showing how to partition a line into "sectors" (intervals) according to historical trajectory data. Then, we apply the 1D solution framework to design a 2D sectorization heuristic based on binary space partitions. We also devise partitions based on balanced "pie partitions" of a convex polygon.

We evaluate our 2D algorithms experimentally. We conduct experiments using actual historical flight track data for the NAS as the basis of our partitioning. We compare the workload balance of our methods to that of the existing set of sectors for the NAS and find that our resectorization yields competitive and improved workload balancing. In particular, our methods yield an improvement by a factor between 2 and 3 over the current sectorization in terms of the time-average and the worst-case workloads of the maximum workload sector. An even better improvement is seen in the standard deviations (over all sectors) of both time-average and worst-case workloads.

1 Introduction

The National Airspace System (NAS) is a complex transportation system designed to facilitate the management of air traffic with safety as the primary objective and efficiency as the secondary objective. Airspace design engineers and air transportation policy makers are continually "tweaking" the system to adjust for changes in the demand patterns, changes in weather systems that disrupt the network, and changes in the air traffic management (ATM) policies that govern the safe operation of aircraft.

A key component of the NAS is the partitioning of airspace into managerial units. At the highest level, the NAS is partitioned into 20 Air Route Traffic Control Centers (ARTCC), each of which is partitioned into *sectors*, each one of which is managed by one air traffic controller (or a small team of 1-3 controllers) at any given time of the day. There are a total of about 600 sectors; the FAA employs about 15,000 controllers, of which over 7000 are due to retire within the next 9 years [21], suggesting a need to redesign the airspace for fewer controllers in the near future. There are roughly 60,000 daily flights within the NAS, interconnecting about 2000 airports. See Figure 1.

The capacity of the NAS to accommodate increases in traffic demand are being pushed to the limits. Both the FAA and NASA are backing initiatives to study how greater throughput can be accommodated safely through system redesign and new technologies for automation, communication, and ATM. The National Airspace Redesign (NAR) initiative [20] has been in place for the last few years to address this challenging problem. Airspace redesign is critical for anticipated future growth in the NAS. Current sector boundaries are largely determined by historical effects and have evolved over time; they are not the result of analysis of route structures and demand profiles, which have changed over the years, while the sector geometry has stayed relatively constant.

In this paper we study the automatic *sectorization* ("sector boundary design") of airspace problem from a formal and geometric perspective, while attempting to model precisely the system design constraints. In doing so, we have developed a tool, GeoSect, which allows us to explore algorithms and heuristics for automatic sectorization and load balancing.

More formally, *the sectorization problem* is to determine a decomposition of a given airspace domain $\mathcal{D}$ into a set of k *sectors*, $\sigma_1, \ldots, \sigma_k$, in an "optimal" manner. Optimality is defined in terms of the *workloads*, $w(\sigma_i)$, of the sectors, where $w(\sigma_i)$ is a numerical value indicating the amount of "effort" required to manage and control traffic in sector σ_i. The objective may be to minimize the maximum workload (min-max) or to minimize the average workload (min-avg) across sectors, subject to an upper bound, k, on the number of sectors. Alternatively, the objective may be to minimize k subject to a bound on the maximum or average workload across sectors.

The definition of workload is critical. It needs to take into account human factors issues, which include subjective estimations of psychological/physiological state and mental workload, such as issues of visual and auditory perception, memory, stress, and attention span. Many research studies (see, e.g., [14, 17, 25, 24]) have addressed the modeling and quantification of ATC workload.

We model the problem using a geometric and easily quantified approach to defining sector workload: Based on a given set of historical flight data, $w(\sigma)$ is defined to be the maximum (worst-case) or the time average number of aircraft in sector σ during a fixed time window $[0, T]$ (typically, the time window corresponds to a 24-hour day). This definition accounts directly for the traffic density/number of flights aspect of workload. While it does not include other components that often make up an aggregated workload estimate, we are able to quantify some these other factors and add them to our model. We have already done so for coordination workload between sectors (which accounts for the number of times a flight must be "handed off" between controllers), as we report later; other workload components for potential inclusion in our analysis include traffic mix, separation standards, aircraft speeds, crossing aircraft profiles, angle of intersection between routes, directions of flights, number of facilities in a sector, location of conflicts within a sector, number of altitude changes, etc (see [27] for more details). We also mention (Section 6) the extension of our methods to account for no-fly constraints and the location of airports within sectors.

The historical track data is assumed to be given. It gives a set of trajectories (each given by a sequence of *way points* with time stamps) for each recorded flight path in the NAS over the time window $[0, T]$. We are using the historical data to give a distribution (in space and time) of the typical trajectories of the aircraft in the NAS; on any given day, of course, the flight paths vary, with weather conditions and other events that disrupt the standard schedule. Thus, a potentially more desirable method of assessing workload is to use track data from an airspace simulation (such as NASA's Airspace Concept Evaluation System [30]), since this allows one to evaluate the "ideal" routes for a given set of demand, to incorporate new air traffic concepts (such as "Free Flight"), and to modify the demand according to predicted future growth. The methods we investigate, though, work equally well with input from a simulator or from historical data.

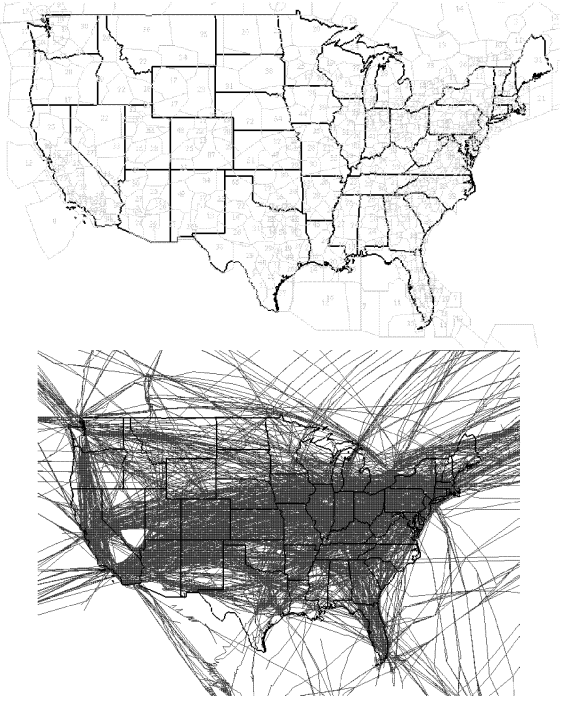


Figure 1: Top: The current sectorization of the airspace over the USA. Bottom: Historical track data for flights.

1.1 Related Work The sectorization problem has been studied most recently as a global optimization problem using techniques of integer programming; after discretizing the NAS into 2566 hexagonal cells, Yousefi and Donohue [29, 28] formulate and solve an extensive mathematical programming model that captures more of the sector workload issues than many prior methods. They use a large-scale simulation to compute en route

metrics that are combined to give a workload model. Delahaye et al. [11] use genetic algorithms for sectorization. Tran et al. [26] apply graph partitioning methods to sectorization.

In the algorithms literature, there has been related work on partitioning of rectangles and arrays for load balancing of processors; see, e.g., [5, 4, 6, 15, 16, 18, 19].

Geographical load balancing applications have arisen in political districting (to avoid gerrymandering); see Altman [1] (who proves NP-hardness of political districting), Altman and McDonald [2], and Forman and Yue [12]. Geographic load balancing also arises in electric power districting; see, e.g., Bergey, Ragsdale, and Hoskote [3]. Recent work in the computational geometry literature looks at minimum-cost load balancing in sensor networks; see Carmi and Katz [7].

What makes our sectorization problem novel compared with most geometric load balancing problems previously studied is that the input data consists of *trajectories* of *moving* points; typical geometric partitioning problems have addressed *static* point data. This implies that a 2D version of our problem is really best thought of in 3D (x, y, t), and it means that even the 1D version of our problem has interesting structure, as it maps to a 2D partitioning problem in space-time.

1.2 Summary of Contributions

(1) We model the airspace sectorization problem in algorithmic terms, as a precise computational geometric formulation.

(2) We provide an exact solution to some versions of the one-dimensional (1D) sectorization problem.

(3) We develop a suite of heuristics to solve the problem in two dimensions (2D), using the 1D solution as a subproblem, and we discuss algorithmic issues.

(4) We implement and conduct experiments to test the effectiveness of our methods on real flight data. We present extensive computational results comparing our methods and design choices in our heuristics. We compare also the results we obtain with the existing sectorization currently in use by the FAA.

Our results are quite promising: our best heuristic methods yield an improvement by a factor between 2 and 3 over the current sectorization in terms of the time-average and the worst-case workloads of the maximum workload sector. An even better improvement is seen in the standard deviations (over all sectors) of both time-average and worst-case workloads.

We have made various simplifications in developing the model and implementing our solutions. We are able to extend our solutions in several directions; see Section 6. Currently, our software works in only 2D; however, the methods extend to account for different altitudes and climb and descend trajectories. We also deal with only one altitude level of sectorization, the so-called "high altitude" sectors; there are also low altitude sectors for general aviation aircraft and ultra-high altitude sectors for military and certain transcontinental flights. Our experiments currently include all of the airspace in a contiguous region; additional experiments will consider only en route airspace and exclude from sectorization the Terminal Radar Approach Control (TRACON) areas near major airports. We model workload in terms of aircraft density (number of aircrafts in a sector), determined by a given set of track data, which may come from historical data (as ours did) or from the results of a simulation. We have extended the results to include coordination workload in the objective function as well, so that we take into account the number of times a flight must be handed off between sectors. We acknowledge that our current model is a simplification of workload estimation; it is, however, applicable to a broader model and it should help in understanding the bigger picture, including the operational constraints that impact the sectorization problem.

2 The 1D Sectorization Problem

We begin with a study of the 1D problem, which has interesting algorithmic aspects of its own; further, the 1D solution is used within the 2D heuristics we develop and implement.

Consider an airspace domain that is 1-dimensional, consisting of an interval, without loss of generality $\mathcal{D} = [0, 1]$, on the x-axis. Flights can take off at some point ("airport") of $\mathcal{D}$ and land at another point ("airport").

The input data consists of a set S of flight trajectories, each represented by a sequence of "waypoints", (x_i, t_i), where t_i is the timestamp when the flight is recorded to be at location $x_i \in [0, 1]$. We consider there to be a finite time horizon, $[0, T]$, containing all of the timestamps t_i. We generally assume that the flight speed between waypoints is constant; thus, a trajectory can be thought of as a t-monotone polygonal chain in the (x, t)-plane. In this "LineLand" model, it probably makes most sense to consider trajectories that consist of only two waypoints – the starting point and the destination point. (Other waypoints between the start and destination x-coordinates may be used to specify changes in speed; however, waypoints outside the interval of

(start, destination) correspond to seemingly unreasonable trajectories, which do some amount of doubling back.) Thus, we consider the input $S = \{s_1, \ldots, s_n\}$ to be a set of line segments in the (x,t)-plane, all of which lie within the 1-by-T rectangle, $[0,1] \times [0,T]$.

The sectorization problem asks us to partition $[0,1]$ into a set of k sectors, $\sigma_1, \sigma_2, \ldots, \sigma_k$; i.e., we desire partition points, $x_0 = 0 < x_1 < x_2 < \cdots < x_{k-1} < x_k = 1$, which define the sector intervals $\sigma_i = (x_{i-1}, x_i)$.

The *max-workload*, $w(\sigma_i)$, of a sector $\sigma_i = (x_{i-1}, x_i)$ is defined to be the maximum number of flights ever simultaneously in sector σ_i: this is given geometrically by the maximum number of segments of S intersected by a horizontal segment, $\overline{(x_{i-1},t)(x_i,t)}$, for $t \in [0,T]$. One can envision a sweep of the rectangle $[x_{i-1}, x_i] \times [0,T]$ by a horizontal segment – the max-workload of σ_i is the maximum number of segments of S intersected during the sweep. The *avg-workload*, $\bar{w}(\sigma_i)$, of a sector $\sigma_i = (x_{i-1}, x_i)$ is defined to be the *time-average* number of flights in the sector σ_i: this is given geometrically by the sum of the lengths of the t-projections of segments S clipped to the rectangle $[x_{i-1}, x_i] \times [0,T]$, divided by T. If we let $\xi_i(t)$ denote the number of segments of S crossed by the horizontal segment $\overline{(x_{i-1},t)(x_i,t)}$, then $w(\sigma_i) = \max_{t \in [0,T]} \xi_i(t)$ and $\bar{w}(\sigma_i) = \frac{1}{T} \int_0^T \xi_i(t) dt$.

The *min-k sectorization problem* is to determine a set of partition points x_i (and corresponding sectors σ_i) in order to minimize the number, k, of sectors in a partitioning of $[0,1]$, subject to a specified *workload bound*, B. The workload bound B stipulates that $w(\sigma_i) \leq B$, or that $\bar{w}(\sigma_i) \leq B$, for all $i = 1, \ldots, k$, in the max-workload or the avg-workload case, respectively.

The *min-B sectorization problem* is to determine a set of partition points x_i (and corresponding sectors σ_i) in order to minimize the upper bound, B, on the workloads of the sectors, subject to their being at most (and therefore exactly) k sectors, where k is specified as part of the input. In other words, we want to determine the x_i's, $i = 1, \ldots, k$, subject to $w(\sigma_i) \leq B$, or $\bar{w}(\sigma_i) \leq B$, for all $i = 1, \ldots, k$, in the max-workload or the avg-workload case, respectively.

Thus, we get four versions of our sectorization problem, depending if we are using max-workload or avg-workload measures, and depending on the choice of min-k or min-B in the optimization.

min-k, max-workload. We are given a budget B on the max-workload in each sector and wish to minimize the number, k, of sectors. We prove that the following greedy algorithm is optimal: At stage i, with partition points $x_1, \ldots, x_i$ already determined, we compute partition point x_{i+1} in order to make sector $\sigma_{i+1} = (x_i, x_{i+1})$ as large as possible, subject to the budget constraint B.

The determination of x_{i+1} according to this greedy rule is an interesting geometric subproblem in its own right, and it is related to the following problem: *Given a set of n line segments in the plane, determine the lowest point of the B-level.* Recall that the j-level of a set of line segments S is defined to be the locus of all points on S that have exactly j segments lying strictly below. In our setting, "below" means "leftward" in the (x,t)-plane, and "lowest" point on the B-level means the leftmost point of the B-level. The lowest point on the B-level in an arrangement of lines is solved in expected time $O(n \log n)$ by the randomized algorithm of Chan [9]. In fact, this algorithm is readily adapted to give the same expected running time $O(n \log n)$ for computing the lowest point on the B-level in an arrangement of line segments or x-monotone curves of constant complexity [8]. Below, we give a simple $O(n \log^2 n)$ deterministic algorithm; we are not aware of an $O(n \log n)$ deterministic algorithm for computing the lowest point on the B-level of an arrangement of lines or of segments.

Consider sweeping a vertical line ℓ rightwards from $x = x_i$. The max-workload of the sector between $x = x_i$ and ℓ can change only at certain events, when ℓ passes over a *critical point*, and it can only go up (by definition). See Figure 2. Each left endpoint of a segment of S is a potential critical point. A critical point may also occur at the intersection of two segments of S, if the signs of these segments' slopes are opposite (since, in this case, the t-projections of the segments within the vertical strip start to overlap, possibly causing the max-workload to change). A critical point may occur at the intersection $\rho_j \cap s_l$, for some segment $s_l \in S$, if the signs of the slopes of s_j and s_l are opposite; here, ρ_j is the rightwards ray from the right endpoint of segment $s_j \in S$. Finally, a critical point can occur at the intersection $\rho_{ij} \cap s_l$, for some segment $s_l \in S$, if the signs of the slopes of s_j and s_l are the same. Here, ρ_{ij} is the rightwards ray from the point $a_{i,j} = \{x = x_i\} \cap s_j$ on s_j intersected by the vertical line $x = x_i$.

We can now solve the geometric subproblem using binary search on the set of x-coordinates of potential critical points. Using slope selection (see Cole et al. [10]), we can, in $O(n \log n)$ time, compute the median x-coordinate, x', among vertices in the arrangement, $\mathcal{A}$, of the n lines containing each segment of S, the (at most n) lines containing each ray ρ_j, the (at most n) lines containing each ray ρ_{ij}, and the (at most n) vertical lines through left endpoints of segments in S. In fact, we compute x' to be the median x-coordinate among

vertices of the arrangement that lie between $x = x_i$ and $x = 1$. Now, we can "test" the value x', to see if x_{i+1} should lie to its left or its right, by computing the workload, $w([x_i, x'])$: If $w([x_i, x']) > B$, then we know that $x_{i+1} < x'$; otherwise, $x_{i+1} \geq x'$. Computing the workload $w([x_i, x'])$ is easily done in time $O(n \log n)$, e.g., by clipping the segments S to the strip $[x_i, x']$, projecting the clipped segments onto the t-axis, and sweeping in t to determine the depth of overlap among the projections. Since there are at most $O(n^2)$ candidate critical points, and each step of the binary search takes time $O(n \log n)$, we get that the overall algorithm to determine x_{i+1} greedily takes (deterministic) time $O(n \log n \log n^2) = O(n \log^2 n)$. Doing this for each stage of the greedy algorithm yields the following:

THEOREM 2.1. *The one-dimensional min-k, max-workload, sectorization problem can be solved exactly in (deterministic) time $O(kn \log^2 n)$, where k is the output optimal number of sectors. Using a randomized algorithm, it can be solved in expected time $O(kn \log n)$.*

Proof. We have described the algorithm and its running time already. In order to justify the correctness of the algorithm, consider an optimal partition $X^* = \{x_1^*, x_2^*, \ldots, x_{k^*}^*\}$. Let the output of the greedy solution be $X = \{x_1, x_2, \ldots, x_k\}$. Let i be the first index for which $x_i^* \neq x_i$. If $x_i^* > x_i$, then x_i could not have been the greedy output, since we could have pushed x_i further to the right (to x_i^*) without violating the budget constraint B. Thus, $x_i^* < x_i$. Now, we can replace x_i^* with x_i in X^*. The workload of the sector $[x_{i-1}^* = x_{i-1}, x_i^* = x_i]$ clearly cannot exceed the budget B (since the greedy sectors must be feasible), and the workload of the sector $[x_i^*, x_{i+1}^*]$ only went down with the replacement of x_i^* with $x_i > x_i^*$. Continuing this argument, we convert solution X^* into solution X, proving that the greedy algorithm produced an optimal partition.

min-B, max-workload. We are given an allowed number k of sectors and wish to determine a set of partition points, $x_1, \ldots, x_{k-1}, x_k = 1$, of $[0, 1]$ in order to minimize the maximum workload, $B = \max_i w(\sigma_i)$. We do this optimization using binary search, using the min-k solution above to test a particular value, B', of (integer) budget B. Note that the optimal B^* must lie between 1 and $B_0 \leq n$, where B_0 is the maximum number of segments of S intersected by a horizontal line. For each test value B', we run the greedy algorithm to determine the optimal number of sectors, $k^*(B')$, subject to budget B'. If $k^*(B') > k$, then we know that $B^* < B'$; otherwise, we know that $B^* \geq B'$. The binary search concludes in $O(\log n)$ steps, so we get

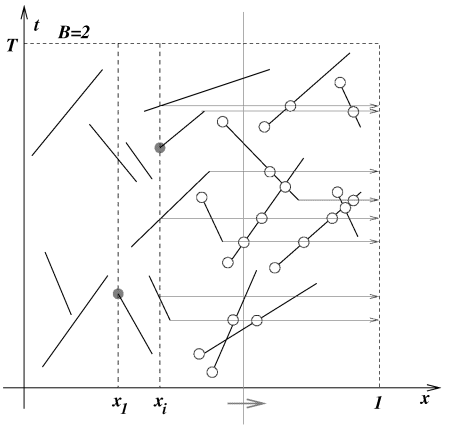


Figure 2: Sweeping ℓ (red) rightwards. The hollow blue circles indicate critical points where the max-workload might increase as ℓ sweeps.

THEOREM 2.2. *The one-dimensional min-B, max-workload, sectorization problem can be solved exactly in (deterministic) time $O(kn \log^3 n)$. Using a randomized algorithm, it can be solved in expected time $O(kn \log^2 n)$.*

min-k and min-B, avg-workload. In the average workload case, we consider the "cost" of a sector to be the time-average number of aircraft in a sector. Since the time-average $\bar{w}(\sigma_i)$ for sector $\sigma_i = (x_{i-1}, x_i)$ is simply the sum, $(1/T) \sum_{s \in S} \mu_{\sigma_i}(s)$, of the lengths $\mu_{\sigma_i}(s)$ of the t-projections of the segments $s \in S$ clipped to sector σ_i, each of which varies linearly with x_i, we see that the function $f(x) = \bar{w}((x_i, x))$ that measures the time-average workload of the interval (x_i, x) is a piecewise-linear (and continuous) function of x. The function $f(x)$ has breakpoints that correspond to the x-coordinates of endpoints of S. For the min-k avg-workload, k is exactly equal to $\lceil (1/T) \frac{\sum_{s \in S} \mu(s)}{B} \rceil$, where $\mu(s)$ is the length of the t-projection of segment s. The sector (interval) boundaries can be determined by greedily scanning from left to right the $O(n)$ possible critical values of x, between which the function $f(x)$ has an easy-to-describe (linear) formula, which we can threshold against the budget B. Thus, the overall running time becomes just $O(n \log n + k)$ for the min-k problem. For the min-B version, the avg-workload of each of the k sectors will be exactly $(1/T) \frac{\sum_{s \in S} \mu(s)}{k}$, and the running time of the algorithm to determine the sector boundaries remains the same, i.e., $O(n \log n + k)$. The correctness of the greedy approach is proven similarly as before and is omitted here. In summary,

THEOREM 2.3. *The one-dimensional min-k (and min-*

B), avg-workload, sectorization problem can be solved exactly in time $O(n \log n + k)$, where k is the output optimal number of sectors.

Remark. Note that the min-B problem is (trivially) always feasible, both for max-workload and for avg-workload. The min-k problem is always feasible for avg-workload and, for max-workload, it is feasible and results in a finite k provided that B is at least as large as δ_{max}, the maximum number of segments of S passing through a common point. (If $B < \delta_{max}$, no partitioning in the immediate x-vicinity of the high-degree vertex will suffice to meet the (max-workload) budget constraint; if $B = \delta_{max}$, then there needs to be an infinite sequence of partition points, converging on the x-coordinate of the high-degree vertex.)

3 The Sectorization Problem in Two Dimensions

In contrast with prior work on partitioning sets of (static) points in the plane, or elements of an array (e.g., see [15, 16, 18, 19]), our sectorization problem involves a third dimension (time): The input data consists of a set S of trajectories, which correspond to t-monotone polygonal chains in (x, y, t)-space. We let n denote the number of trajectories, and N the total number of waypoints (vertices) in the full set of n trajectories. Given a domain of interest, $\mathcal{D} \subset \Re^2$, we are to partition it into a small number of sectors, each of which has a small workload. As in the 1D problem, we can distinguish the min-k from the min-B problem, where k denotes the number of sectors in the partition and B denotes an upper bound on either the max-workload or the avg-workload of the sectors.

The max-workload for a sector $\sigma \subset \mathcal{D}$ is the maximum number of trajectories intersected by a "horizontal" (in t) polygon of shape σ, sweeping vertically through time, $t \in [0, T]$. Another way to view the problem is to clip the 3D trajectories to the vertical cylinder defined by σ, and project each clipped trajectory onto the t-axis. The maximum depth of this set of intervals is the max-workload for σ; the sum of the interval lengths, divided by T, is the avg-workload for σ.

3.1 Hardness Not surprisingly, the sectorization problem in two (or more) dimensions is NP-hard, in general. We sketch a proof of the special case in which sectors are required to be axis-aligned rectangles, and the goal is to minimize the max-workload upper bound B, subject to a bound k on the number of sectors. Hardness follows from the result of Khanna, Muthukrishnan et al [15], who proved that the following problem is NP-hard (and also NP-hard to approximate within a factor of $\frac{5}{4}$): Given an $n \times n$ array A of integers, find a rectangular partition of A into k rectangles, in order to minimize the maximum weight of a rectangle. The weight here is defined to be the sum of the array elements in the rectangle. For a given instance of the array partitioning problem, we construct an instance of the sectorization problem in which we form small "bundles" of straight trajectories associated with each entry of A, laid out in a regular grid in the xy-plane. Then, an optimal decomposition into k rectangular sectors of minimum upper bound B on max-workload corresponds exactly to a solution to the array partitioning problem. We have sketched:

THEOREM 3.1. *The optimal sectorization problem (min-k or min-B) for partitioning into rectangular sectors in two dimensions is NP-hard.*

3.2 Heuristics for 2D Sectorization Given the difficulty of solving the 2D sectorization problem exactly, we turn our attention to heuristics for its solution. We consider the min-k version, in which a budget B is given, and our goal is to partition $\mathcal{D}$ into a small number k of sectors.

Our heuristics for 2D sectorization are based on two forms of recursive partitioning: binary space partitions (BSP) and *pie-partitions*. BSP algorithms have been studied extensively in the computational geometry literature, starting with the work of Paterson and Yao [22, 23]. Pie-partitions are based on a multi-way partition into cones having a common apex; see below.

The use of recursive partitions heuristics is both natural and theoretically motivated. For sectorizations based on BSP partitions whose cuts come from fixed orientations (as ours do) with discretized intercepts (translations), we are able to solve the min-k problem (for given budget B) optimally, as well as the min-B problem (for given k) using dynamic programming. A subproblem is defined by a convex polygon having $O(1)$ sides; by selecting an optimal cut from among a discrete set of possibilities, and recursively optimizing on each side of the cut, we obtain an optimal BSP-based sectorization. This sketches the proof of the following theorem:

THEOREM 3.2. *The min-k and min-B optimal fixed-orientation, discrete intercepts BSP sectorization problem in 2D has an exact polynomial-time algorithm.*

Proof. Let c be the number of fixed orientations and d be the number of fixed intercepts. This gives us $O(cd)$

total number of polygons possible using these orientations and intercepts. A subproblem of the dynamic program is such a polygon and we maintain the optimal way to partition it in an array. The algorithm for min-k, with given budget B is as follows (it returns the number of sectors):

Partition_mink(Polygon P)

- If the max-workload of the polygon is B, simply return 1.
- Else for each pair of orientation and intercept (o, i), recursively solve the two polygons (say P_1 and P_2) which P is divided into and compute $w(o, i)$ = Partition(P_1) + Partition(P_2).
- Return $w(o, i)$ which is minimum over all choices of orientation and intercept.

Running time is clearly $O(c^2d^2n \log n)$.

For $min - B$, given k we have the following algorithm which returns the workload :

Partition_minB(Polygon P, k)

- If $k = 1$, simply return max-workload(P).
- Else for each pair of orientation and intercept (o, i), and every possible way to partition k into k_1 and k_2 such that $k = k_1 + k_2$, recursively solve the two polygons (say P_1 and P_2) which P is divided into and compute $B(o, i, k_1, k_2)$ = maxPartition(P_1,k_1), Partition(P_2,k_2).
- return $B(o, i, k_1, k_2)$ which is minimum over all choices of orientation and intercept and k_1 and k_2.

Running time is clearly $O(kc^2d^2n \log n)$.

It is easy to see why the above algorithms work. The first cut made by an optimal solution on the polygon P, is one of the cuts that is considered by the algorithm, and then the subproblems are recursively solved. Now look at the optimal solution on either side of this cut. The subproblems solved recursively on either side can be only better than this optimal. Moreover, the first cut found by the algorithm did at least as good as this cut. So the output from the algorithm is at least as good as the optimal partitioning.

If we do not restrict ourselves to BSP sectorizations, but still consider the class of allowable cuts to lie on a discrete set of lines, of dicrete slopes and intercepts, then we can obtain a polynomial-time constant-factor approximation for the (non-BSP-based) min-k sectorization problem, using the fact that an optimal sectorization can be converted into a BSP sectorization with a small factor increase in the number of sectors. In particular, this yields a 2-approximation for the rectangular (axis-parallel) case, by the results of [5] on the BSP of a packing of axis-aligned rectangles.

Throughout our discussion, we will interchangeably use the term "weight" and "workload" when referring to a sector.

3.3 BSP Heuristic Rather than implement a relatively high-degree polynomial-time dynamic programming algorithm, we have chosen to craft BSP heuristics based on computing a *most balanced cut* at each stage, which is defined as follows. Given a node of the current BSP subdivision, with associated sector σ, our algorithm finds a straight cut (from among a set of fixed orientations) to partition the convex polygon σ into two subpolygons, in order to minimize the maximum workload (either max-workload or avg-workload) of the two subpolygons. This strategy leads to the following simple consequence in the avg-workload case about the relative weights (workloads) of the sectors: In the final sectorization using most balanced cut BSP, the ratio of the (avg-workload) weight of the heaviest sector to the lightest sector is at most 2. In our experiments, as we describe later, we have further refined the most-balanced cut method for avg-workload in order to partition avg-workload exactly across the k sectors, while simultaneously attempting to control the max-workload balance; see Section 5.2.

In order to find the most balanced cut, we use a discrete set of ℓ *allowable orientations* for our cut. For each orientation, we find the most balanced cut with that orientation as follows. We project the line segments that make up the trajectories onto a plane perpendicular to the cut orientation, resulting in the 1D problem. We now use a binary search on the critical points (as defined in the previous section) to find the most balanced cut in the 1D case. Thus, each step of the BSP takes worst-case time $O(N^2\ell)$: $O(N)$ for projecting the segments, $O(N^2)$ for finding the critical points (which can be found in output-sensitive time, by standard techniques), and then finally the binary search for the most balanced cut. If we finally end up with K sectors, the entire procedure takes worst-case $O(KN^2\ell)$ time (again, with corresponding speed-up for using an output-sensitive segment intersection algorithm).

Since the calculation of the critical points becomes the bottleneck in this heuristic, even if using clever means of implementing nearly output-sensitive algorithms, in our experiments we decided to use a coarser set of points to search for the cut. We refer to this set as the *approximate critical set*. We empirically de-

cide the coarseness of this set and prove experimentally that for the real track data, this works just as well (in practice) as the original set of critical points and saves tremendously on the execution time.

3.4 Avoiding Bad Aspect Ratios The balanced BSP heuristic can clearly produce very skinny sectors, even while producing sectors with well-balanced workloads. Skinny sectors can be undesirable because air traffic passing through the sector may be in the sector for radically different time periods, depending on the orientation of the trajectory with the diameter of the sector.

To address this issue, we use the aspect ratio of the sector we are subdividing to guide us. For any rectangle, define α to be the ratio of the smaller side to the larger side. (Note that aspect ratio is often defined to be $1/\alpha$.) For any sector σ that we want to subdivide by the most balanced cut, we also use the following constraint for the cut. Consider a bounding rectangle with the smallest α for the region. We only consider cuts with an orientation within a small range of the orientation of the smaller side of this rectangle. In our implementation, we use a range of $\theta \mp \alpha\frac{\pi}{2}$, where θ is the orientation of the smaller side. Refer to Figure 3 (top). However, this heuristic can still result in bad aspect ratios as shown in the Figure 3 (bottom).

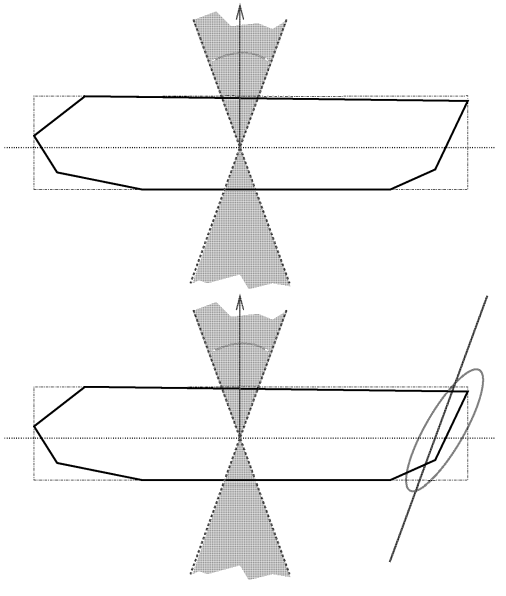


Figure 3: Top: the allowable range for cut orientations is shaded. Bottom: A cut within the allowable orientation range may result in a skinny polygon on the right.

We suggest another heuristic to circumvent this problem. We define $\beta < 0.5$ to be a user-specified lower bound on $\alpha(\sigma)$, which our algorithm is expected to respect in its decomposition. Given an orientation for the cut, we have to find a range for the cut so that the resulting two polygons have $\alpha \geq \beta$. In our experiments we naively search for this range by linearly searching through the approximate critical set. This range may clearly not exist (for example, set $\beta = 0.8$ and consider a square - no range exists for any orientation). However, for reasonable values of β this seems to work quite well. Empirically, we observed that for $\beta < 0.5$, if the original polygon has $\alpha \geq \beta$, this heuristic works extremely well.

3.5 Pie Cutting In addition to the BSP cuts, we consider another cutting operation to allow for more flexibility during sectorisation. This is the so-called "pie-cut". For this we fix a point within the region (called the *center*) and an orientation. We now wish to make a pie-cut which comprises three rays originating from the *center*. One of the rays is along the designated orientation. The other two are such that the resulting 3 pieces are all convex and as well-balanced as possible. We accomplish this pie-cut in two steps, obtaining one cut in each step. The line segments are first transformed to their polar coordinates, in the following sense. Consider any point p with polar coordinates (r, θ, z) with respect to the *center* and the given orientation (r is the distance from the *center*, θ is the angle which the line through p and the *center* makes with the given orientation. This point is transformed to (θ, z), resulting again in an instance of the 1-D sectorization problem. Then a cut is found which divides the workload in $1 : 2$ ratio. Then the second cut is chosen with range restrictions (so that the resulting regions are convex) to balance the workload in the $\frac{2}{3}$-sized region. See Figure 4.

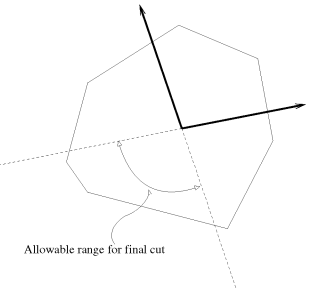


Figure 4: Range constraints for pie cutting.

For greater flexibility and control over α, we also use the pie-cut operation with more than 3 cuts. Note that its not desirable for many sectors to meet at one point (pie-cut center here) as it decreases the chance

of an airplane to stay in a sector for reasonable time, before leaving it. So there has to be an upper-bound on how many cuts should be allowed in pie-cut operation and it can easily be added as a constraint. If the current workload is W, and $P = \lfloor W/B \rfloor$, we start with $max(P, 5)$ cuts and if any of the resulting regions has α less than the threshold, we try with one less cut and so on, until we reach 3. At 3 however, even if one of the α values are bad, we make the cut anyway to maintain convexity of the regions. This is the disadvantage of pure pie-cuts. This can be remedied to a large extent if we combine BSP cuts with Pie-Cuts. That is our motivation for the final heuristic.

3.6 The Final Heuristic We formulate a method using both the operations of BSPs and Pie-cuts. The final heuristic first attempts to make a possible pie-cut. If the pie-cut is unable to find a partition respecting the β threshold, we use a BSP cut. The new regions are then inserted into a priority-queue according to the workloads. We recurse on the heaviest region until all the regions have a workload less than B.

subsectionOther heuristics Some other heuristics for 2D-partitioning are conceivable. For example, a partitioning resembling the Voronoi regions of some predetermined centers. There does not seem to be sufficient evidence to suggest that such combinatorial structures will optimize the workload as considered. A partitioning that does load balancing amongst different sectors according to the definition of workload in this paper does not seem to have any similarity to the structure of Voronoi regions. We feel our heuristics are natural strategies to try and implement when trying to optimize a function like the workload. A related, but perhaps of not much relevance, clustering idea appears in [13].

4 Experimental Setup

Implementation of our system GeoSect is done in C++ (Microsoft Visual Studio 6.0), using the OpenGL Graphics library for all visualizations. All the three heuristics, BSP, Pie-Cuts and the Final Heuristic were implemented as stated above. We also built the capability to search over an *approximate critical set* of points while solving the 1-D problem to save some time without compromising the balancing much. All experiments were run on machines with 3.2 GHz Pentium 4 processors, 1GB RAM.

Data is provided to us by Metron Aviation. The historical track data corresponds to (roughly) a 24-hour period from 04:00,June 27 to 05:00,June 28 2002, with 74588 flight tracks, and the average complexity (number of bends) of each track is 59.26. We compare our results to existing sector data. We are not using ultra high-altitude sectors, and in evaluating sector workloads, both in our sectors and in the existing sectors, we are assuming that all track data is relevant to the sectors. Note that some small fraction of the track data may correspond to ultra high-altitude sectors and may not be relevant to the workload of the high-altitude sectors. See Figure 1.

5 Experimental Results

5.1 Tuning the Parameters of the Heuristic For best performance of our heuristics, we first tune the user-specified parameters that are used at each stage of the heuristic. For tuning parameters in our heuristic, we use 5 different sub-regions of the NAS, shown in Figure 5 The selection of the regions was based on a visual inspection of the track data to correspond to both high and low traffic regions.

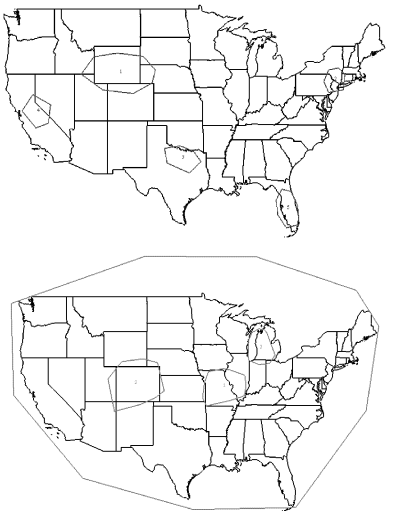


Figure 5: Top: Regions used for tuning the parameters Bottom: Regions used for testing the final heuristic

5.1.1 BSP We begin by selecting good choices of parameters for the BSP cuts. Statistics were generated for *Number of sectors* and *Max, Min, Average and Standard Deviation* for *Worst-case workload, Time-average workload,* α, for each of the 5 sub-regions of the NAS. We summarize our experimental findings:

- *Number of discrete orientations while searching for the most balanced cut.* We generated the above statistics for the following set of values :

{2,4,6,8,10,12,14,16,24,32}. The statistics show that increasing the number of orientations beyond 10 does not yield any significant change in the result. We choose to use 16 orientations in all future experiments.

- *Discretisation for balanced search in a given orientation.* Ideally, we should use the critical points of the projected tracks. However, as mentioned earlier, critical point computation is expensive and we use the *approximate critical set* instead. We examined the data for 5 different values of the parameter : $0.1, 0.01, 0.001, 0.0001, 0.00001$. Beyond 0.001, there is very slight fluctuation in the readings. We pick 0.0001 for our experiments.

- *Choice of β for aspect ratio control.* The goal is to obtain reasonably fat sectors without compromising too much on the quality of the sectorization (number of sectors, workload balance etc.). We looked at data for 10 uniformly spaced values between 0.01 and 0.4 for β. Arbitrary fluctuations are observed in the range 0.01 to 0.2. The experiments suggest predictable behavior for $\beta \geq 0.2$. In the final results we show the effect of β on the final workload-balancing.

Region	$WorstcaseWorkload$		
	Max	*Avg.*	*Std.Dev.*
1	5	4.237	0.501
2	5	4.571	0.564
3	5	4.559	0.537
4	5	4.442	0.573
5	5	4.414	0.553

Region	$TimeAverageWorkload$		
	Max	*Avg.*	*Std.Dev.*
1	0.891	0.344	0.133
2	0.893	0.403	0.151
3	0.766	0.409	0.138
4	1.206	0.405	0.169
5	0.771	0.351	0.142

Region	α		
	Avg.	*Min*	*Std.Dev.*
1	0.572	0.350	0.139
2	0.609	0.350	0.156
3	0.593	0.350	0.148
4	0.591	0.350	0.141
5	0.587	0.350	0.149

Figure 6: BSP Results for the 5 regions after parameter tuning.

Some results are presented in Figure 6. Here, we subdivide to balance the *Worstcase Workload*. We can see that the BSP cuts achieve highly balanced sectors in terms of workloads as reflected by the *standard deviation* of the Worstcase Workload. We could instead choose to balance the time-average workload, which would result in better standard deviation statistics for the time-average case. By the nature of the heuristic, we have a guarantee on the minimum value of the α.

5.1.2 Pure pie-cuts The only decision parameter for this class of heuristics is how to choose the orientation for the first cut of the pie. We compared two cases: (1) Use the α restriction, as in BSP cuts, i.e. the first cut is more or less perpendicular to the diameter; and, (2) Choose a random orientation uniformly in $[0, 2\pi]$. The aspect ratio readings fluctuate unpredictably for both cases. This happens because 3-pie cuts can result in regions with very bad aspect ratios, as mentioned previously.

We use the parameters derived in the previous two sections to build the final heuristic as described earlier.

5.2 Achieving the Balancing The main goal is to balance the time-average workload across all the sectors while controlling the worstcase-workload and also the aspect ratios of the sectors. It is easy to see that one can exactly balance the time-average workload, i.e. given k, the number of sectors, it is possible to achieve sectors with workload exactly equal to $TotalWL/k$. We control the worstcase-workload by iteratively choosing the sector with the maximum worstcase-workload as the candidate for splitting. Also, since we know the target workload for individual sectors($TotalWL/k$), we restrict ourselves to cuts that preserve integer multiples of target workload on both sides. For example, if we want to split a region with time-average workload 9 into 3 sectors, we do not split it into 4.5-4.5; we instead restrict to cuts that split it in 3-6 or 6-3, so that later the sector with time-average workload 6 can be split in 3-3. The aspect ratio of the sectors is controlled, as described previously, by avoiding the cuts that result in bad aspect ratios. There definitely is a trade-off between preserving good aspect ratios and optimally balancing the sectors. This trade-off is indicated in the final results; refer to Figure 8.

5.3 Comparing the Final Heuristic with Existing Sector Data While comparing our final heuristic with the existing sector data, experiments are run for two types of geographical domains: (1) ["Domain 1"] a

specific convex polygonal region, C, selected to contain approximately 10 current sectors; and (2) ["Domain 2"] a large convex polygonal region, U, selected to contain all of the continental USA. To be as accurate as possible, we purposefully select these regions so that they closely match existing boundaries of the sectors. See Figure 5.

While taking the statistics of the original sectors, we consider only the sectors which are completely inside the selected region of interest for partition. Also, the partitioning looks to balance the time-average workload.

5.3.1 Results for "Domain 1" Both the BSP method and the final heuristic performed competitively with the original sector data. The statistics for one of the regions are shown in Figure 7. Our heuristic achieves a significant improvement (by a factor of 10) over the original sectors in the standard deviation of the workloads and decreases substantially the maximum workload, while using the same number of sectors and having comparable average workloads. (The average workloads are slightly higher for our heuristic because it is applied to a *convex* region; thus, the (convex) region over which we include workload, the convex red polygon defining Domain 1, is slightly larger than the union of the original sectors for which the comparison is based – we count in the average workload of the original sectors only those that are *fully* contained in the convex Domain 1.) The BSP results are shown as well. This experimentally supports our claim that our heuristics achieve highly balanced workloads, while avoiding skinny sectors.

Sectorization	No. of Sectors	Time Average Workload		
		Max	*Avg.*	*Std.Dev.*
Original Sectors	10	12.33	6.899	2.578
Final Heuristic	10	7.289	7.226	0.054
BSP	10	7.9525	1.226	0.302

Sectorization	No. of Sectors	Worstcase Workload		
		Max	*Avg.*	*Std.Dev.*
Original Sectors	10	44	26.8	8.340
Final Heuristic	10	30	27.9	1.221
BSP	10	31	27	2.323

Sectorization	No. of Sectors	α		
		Avg.	*Min*	*Std.Dev.*
Original Sectors	10	0.548	0.264	0.169
Final Heuristic	10	0.534	0.323	0.159
BSP	10	0.522	0.25	0.171

Figure 7: The statistics for pure BSP, the final heuristic and the original sectors for Domain 1

5.3.2 Results for "Domain 2" We first obtained the statistics for the original sectors (fully) contained in our selected region. The goal of the experiment was to use the same number of sectors and balance the time-average workload, while still keeping the maximum worstcase-workload and the minimum aspect-ratio of a sector under check.

Sectorization	Constraint $\alpha \geq$	Time Average Workload		
		Max	*Avg.*	*Std.Dev.*
Original Sectors	0	24.519	6.283	3.378
Final Heuristic	0.15	7.335	6.365	0.157
Final Heuristic	0.25	9.283	6.365	0.294
Final Heuristic	0.30	8.938	6.365	0.457
BSP	0.15	7.343	6.365	0.0715
BSP	0.25	9.568	6.365	0.426
BSP	0.30	9.545	6.365	0.512
Pie-Cut	0	11.085	6.35	2.901

Sectorization	Constraint $\alpha \geq$	Worstcase Workload		
		Max	*Avg.*	*Std.Dev.*
Original Sectors	0	87	24.569	10.437
Final Heuristic	0.15	39	25.297	2.586
Final Heuristic	0.25	34	25.253	2.539
Final Heuristic	0.30	40	25.426	2.939
BSP	0.15	34	25.207	2.567
BSP	0.25	36	25.11	2.882
BSP	0.30	35	25.1	2.849
Pie-Cut	0	47	25.041	8.812

Sectorization	Constraint $\alpha \geq$	α		
		Avg.	*Min*	*Std.Dev.*
Original Sectors	0	0.316	0	0.241
Final Heuristic	0.15	0.45	0.15	0.185
Final Heuristic	0.25	0.506	0.25	0.152
Final Heuristic	0.30	0.532	0.30	0.151
BSP	0.15	0.588	0.15	0.188
BSP	0.25	0.60	0.25	0.181
BSP	0.30	0.578	0.30	0.164
Pie-Cut	0	0.286	0.021	0.175

Figure 8: The statistics for the Original Sectors, the Final Heuristic, pure BSP and pure Pie-Cut's for Domain 2. The number of sectors was 411 in all cases except Pie-Cut where it was 412

Clearly the final heuristic and the BSP give very nicely balanced sectors, again avoiding skinny sectors by producing α values above the β threshold. Notice that as the β is increased, the system becomes more constrained, hence decreasing the workload balancing (increasing the standard deviations of the workloads).

The standard deviation improved by an order of magnitude and max value of worstcase and time-average

workloads also improved by a factor of 2-3, while using essentially the same number of sectors (412 vs. 411; in the case of Pie-Cuts this is due to technical reasons)

The Pie-cut heuristic fails to keep the aspect ratio above the threshold and actually gives very poor values for α. Still, though, it does balance the workload better than the original sectors.

Again, our average workloads are slightly higher for our heuristic than the original sectors because we include in our sectorization the full (convex) Domain 2, while we only count the workloads for original sectors that are fully contained in the Domain 2 polygon. (This under-counting should not have much impact on the variation in the workloads across sectors – the variation is the main target of our investigation in load balancing.)

We have also implemented and experimented with a method that combines workload with *coordination workload*, which we define to mean the number of crossings between the trajectories and the sector boundaries (the hand-off points). By optimizing a linear combination of time-average workload (weighted 0.7) and coordination workload (weight 0.3) we were still able to balance the the time-average workload while preserving similar coordination workloads as the original sectors.

Our heuristics methods are thus seen to be very effective in global sectorization, in terms of balancing workload and producing sectors with good aspect ratios.

Refer to Figure 9 and Figure 10 for the screenshots of the results.

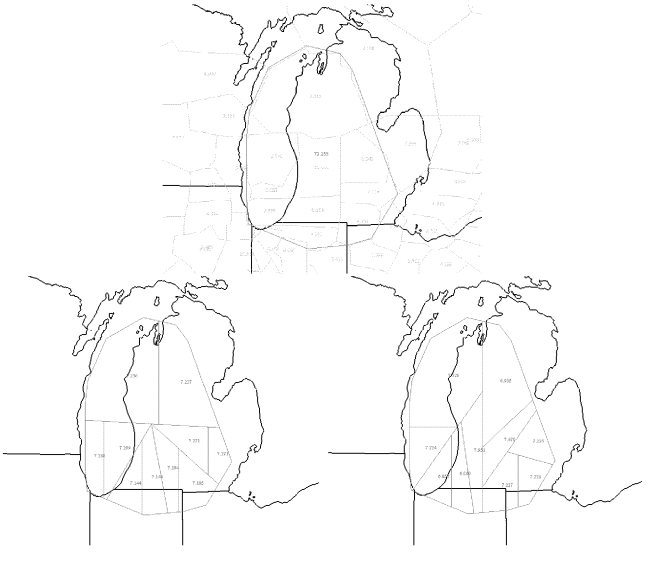


Figure 9: Partition results for Domain 1. Top: Original Sectors, Bottom Left: Final Heuristic partition, Bottom Right: BSP

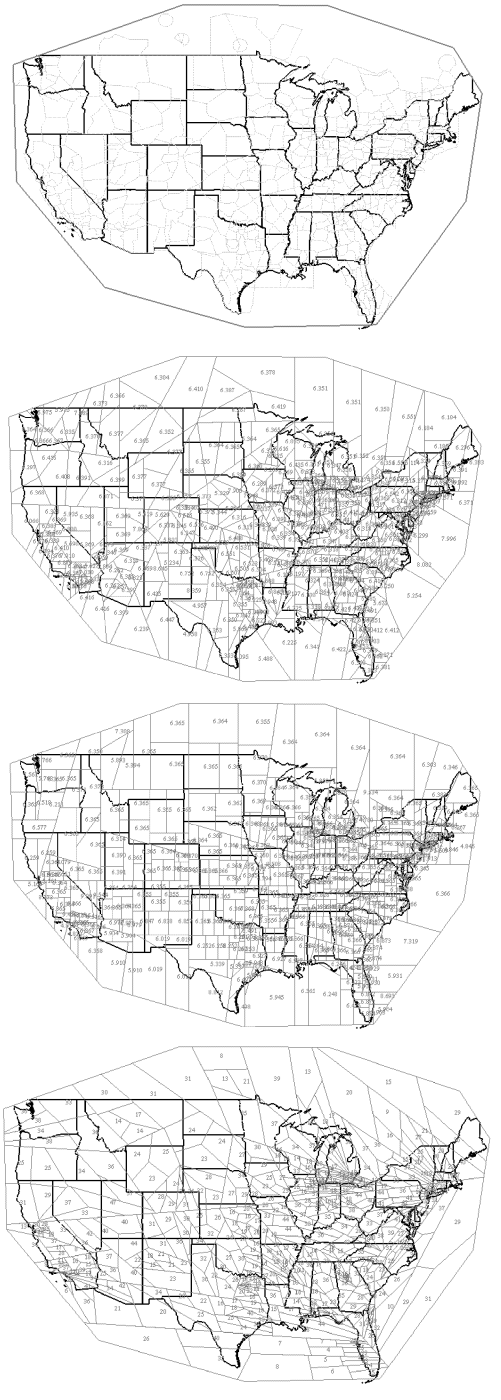


Figure 10: Partition results for the continental USA (Domain 2). From Top (i) Original Sectors (411) strictly within the selected region, (ii) Final Heuristic partition (411 sectors), (iii) BSP (411 sectors), (iv) Pie-Cut partition(412 sectors)

5.4 Comparing the Final Heuristic with Clustering/Integer Programming Method The Integer Programming (IP) method [28] begins with partitioning the entire NAS into smaller hexagonal cells. It then uses IP to cluster the cells in order to optimize certain workload metrics, in particular, coordination workload while constraining the deviation in the average workload. Below we present how the final heuristic results compare with the IP method while sectorizing ZFW (Dallas) center. The objective of the final heuristic in this experiment was to balance the average workload while keeping the aspect ratio of the sectors at least 0.30

Sectorization	No. of Sectors	Time Average Workload		
		Max	*Avg.*	*Std.Dev.*
IP Method	18	5.408	4.184	0.658
Final Heuristic	18	5.158	4.771	0.194

Sectorization	No. of Sectors	Worstcase Workload		
		Max	*Avg.*	*Std.Dev.*
IP Method	18	20	16.611	2.059
Final Heuristic	18	23	18.167	2.034

Sectorization	No. of Sectors	α		
		Avg.	*Min*	*Std.Dev.*
IP Method	18	0.210	0.442	0.148
Final Heuristic	18	0.319	0.600	0.173

Figure 11: The statistics for the IP Solutions and the Final Heuristic for sectorizing ZFW (Dallas) center

We see that the heuristic clearly does a better job at balancing the average workload of the resulting sectors and keeping their aspect-ratios high. The IP method though was able to keep the worstcase workload under check. Other problem with the IP method is that the resulting sectors have irregular boundaries because of the union of underlying hex-cells in the cluster. The running-time of the IP method is also considerably slower compared to the methods described in this paper. Refer to Figure 12 for the screenshots of these comparison.

6 Extensions to the Model and Conclusions

We have studied in detail the optimal sectorization problem that arises in air traffic management, giving theoretical formulation, algorithmic results in the 1D setting (which maps to an arrangements problem in the plane), and experimental results in the 2D setting, where we analyze the effectiveness of heuristics that leverage from the 1D solution.

The next step in our project is to extend our model

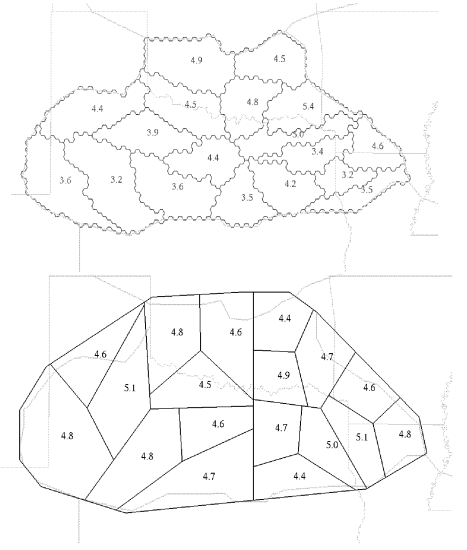


Figure 12: Partition results for ZFW Center. Top: IP Method, Bottom: Final Heuristic partition

and the implementation to take into consideration more of the factors that directly influence workload complexity in quantifiable ways, such as the coordination workload (proportional to how many flights cross the boundary of a sector), anticipated conflict avoidance, angles of crossing tracks, etc. Already, GeoSect includes the option to optimize a linear combination of workload and coordination workload.

Another important direction in which we have extended our model is to account for constraints in the NAS that affect the shapes of sectors. The "odd" shape of current sectors arises not only from historical artifacts but also partly from certain domain constraints on sector boundaries so that, e.g., they do not pass through no-fly zones or pass too close to airports, etc. (An airport and its immediate vicinity should be fully inside or fully outside a sector.) We have extended our model to include such constraints, allowing the partitioning to be done only with cuts that avoid constraints. In order to make a richer set of cuts available, we then also permit a cut to be a polygonal *chain* (based on a shortest constraint-avoiding path that has a limited number of bends) rather than a straight segment; thus, the sectors obtained no longer constitute a BSP, and sectors may be non-convex. We are implementing this feature into GeoSect for future experimentation.

We will be running experiments with GeoSect on track data given by wind-optimized routing data from NASA and Metron Aviation (in preparation).

Future experiments will also take into account ultra-high altitude sectors and will distinguish between track

data that fall in different altitude strata.

On the theoretical side, it would be interesting to investigate further provable approximation algorithms for the 2-dimensional sectorization problems studied here. Also, is it possible to give a deterministic $O(n \log n)$ algorithm to compute the lowest vertex of a k-level in an arrangement of lines or line segments?

Acknowledgements We thank an anonymous reviewer for helpful comments and corrections to an earlier draft. This work is supported under NASA Ames Research Center's Advanced Air Transportation Technologies (AATT) project under Task Order 20 for contract NNA04BA29T. We thank Parimal Kopardekar for guidance and technical input. We thank Jimmy Krozel, Arash Yousefi, Bob Hoffman, and Terry Thompson of Metron Aviation for useful discussions and domain expertise on air traffic management. We also thank Metron Aviation for providing us with the sector data and historical track data. During this investigation, J. Mitchell has been partially supported by grants from the National Science Foundation (CCR-0098172, ACI-0328930, CCF-0431030, CCF-0528209), the U.S.-Israel Binational Science Foundation (grant No. 2000160), Metron Aviation (NASA subcontract, NAS2-02075), and NASA Ames (NAG2-1620).

References

[1] M. Altman. Is automation the answer? The computational complexity of automated redistricting. *Rutgers Computer and Law Technology Journal*, 23(1):81–142, 1997.

[2] M. Altman and M. McDonald. A computation-intensive method for evaluating intent in redistricting. In *2004 Midwest Political Science Association Conference*, Chicago, IL, April 14-18 2004.

[3] P. K. Bergey, C. T. Ragsdale, and M. Hoskote. A simulated annealing genetic algorithm for the electrical power districting problem. *Annals of Operations Research*, 121(1-2):33–55, July 2003.

[4] P. Berman, B. DasGupta, and S. Muthukrishnan. Slice and dice: A simple, improved approximate tiling recipe. *Proc.13th ACM-SIAM Symposium on Discrete Algorithms*, pp. 455–464, 2002.

[5] P. Berman, B. DasGupta, and S. Muthukrishnan. On the Exact Size of the Binary Space Partitioning of Sets of Isothetic Rectangles with Applications, DIMACS report, 2000.

[6] P. Berman, B. DasGupta, S. Muthukrishnan, and S. Ramaswami. Improved approximation algorithms for rectangle tiling and packing. *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pp .427–436, 2001.

[7] P. Carmi and M. J. Katz. Minimum-cost load-balancing partitions. In *Proceedings of the 17th Canadian Conference on Computational Geometry (CCCG'05)*, pages 63–65, 2005.

[8] T. M. Chan. Personal communication, 2006.

[9] T. M. Chan. Geometric applications of a randomized optimization technique. *Discrete Comput. Geom.*, 22(4):547–567, 1999.

[10] R. Cole, J. Salowe, W. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM J. Comput.*, 18(4):792–810, 1989.

[11] D. Delahaye, M. Schoenauer, and J. M. Alliot. Airspace sectoring by evolutionary computation. In *Proc. IEEE International Congress on Evolutionary Computation*, 1998.

[12] S. L. Forman and Y. Yue. Congressional districting using a TSP-based genetic algorithm. volume 2724 of *Lecture Notes in Computer Science*, pages 2072–2083. Springer-Verlag, Jan. 2003.

[13] Sariel Har-Peled. Clustering motion In *IEEE Foundations of Computer Science*, 2003.

[14] K. C. Hendy, J. Liao, and P. Milgram. Combining time and intensity effects in assessing operator information and processing load. *Journal of Human Factors*, pages 30–47, 1997.

[15] S. Khanna, S. Muthukrishnan, and M. Paterson. On approximating rectangle tiling and packing. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 384–393, 1998.

[16] S. Khanna, S. Muthukrishnan, and S. Skiena. Efficient array partitioning. In *Automata, Languages and Programming*, pages 616–626, 1997.

[17] R. H. Mogford, J. A. Guttman, S. L. Morrow, and P. Kopardekar. The complexity construct in air traffic control: A review and synthesis of the literature. Technical Report DOT/FAA/CT-TN95/22, Department of Transportation, Federal Aviation Administration Technical Center, July 1995.

[18] S. Muthukrishnan, V. Poosala, and T. Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. *Lecture Notes in Computer Science*, 1540:236–256, 1999.

[19] S. Muthukrishnan and T. Suel. Approximation algorithms for array partitioning problems.

[20] National airspace redesign (NAR). Office of Air Traffic and Airspace Management, Federal Aviation Administration, http://www.faa.gov/ats/nar/.

[21] Occupational outlook handbook. Bureau of Labor Statistics, U.S. Department of Labor, http://stats.bls.gov/oco/.

[22] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[23] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *J. Algorithms*, 13:99–113, 1992.

[24] D. K. Schmidt. On modeling ATC work load and sector capacity. *Journal of Aircraft*, 13(7):531–537, 1976.

[25] E. S. Stein. Human operator or load on air traffic control. In M. W. Somlensky and E. S. Stein, editors, *Human factors in air traffic control*, pages 155–183. Academic Press, Mar. 1998.

[26] D. H. Tran, P. Baptiste, and V. Duong. Optimized sectorization of airspace with constraints. In *Proc. 5th FAA and EUROCONTROL ATM Conference*, Budapest, Hungary, June 2003.

[27] I. Wyndemere. Dynamic resectorization: Accommodating increased flight flexibility. Technical report, http://www.wynde.com/papers/atca97-2.pdf, Boulder, CO, 1997.

[28] A. Yousefi. *Optimum Airspace Design with Air Traffic Controller Workload-Based Partitioning.* PhD thesis, George Mason University, 2005.

[29] A. Yousefi and G. L. Donohue. Temporal and spatial distribution of airspace complexity for air traffic controller workload-based sectorization. In *AIAA 4th Aviation Technology, Integration and Operations (ATIO) Forum*, Chicago, Illinois, Sep. 20-22 2004.

[30] D. Sweet, V. Manikonda, J. Aronson, K. Roth, and M. Blake, Fast-Time Simulation System for Analysis of Advanced Air Transportation Concepts. In *AIAA Modeling and Simulation Technologies Conf.*, Monterey, CA, Aug., 2002

Better Approximation of Betweenness Centrality*

Robert Geisberger† Peter Sanders† Dominik Schultes†

Abstract

Estimating the importance or centrality of the nodes in large networks has recently attracted increased interest. *Betweenness* is one of the most important centrality indices, which basically counts the number of shortest paths going through a node. Betweenness has been used in diverse applications, e.g., social network analysis or route planning. Since exact computation is prohibitive for large networks, approximation algorithms are important. In this paper, we propose a framework for unbiased approximation of betweenness that generalizes a previous approach by Brandes. Our best new schemes yield significantly better approximation than before for many real world inputs. In particular, we also get good approximations for the betweenness of unimportant nodes.

1 Introduction

One of the most important aspects of automatic analysis of networks is the computation of *centrality indices* that measure the importance of a node in some well defined way. Recently, the focus of attention in network analysis has shifted to the analysis of ever larger networks that are rapidly becoming available in such diverse areas as transportation networks (e.g., public transportation or road networks), social networks (e.g., friendship circles, recommendation networks, or citation networks), computer networks (e.g., the internet or peer-to-peer networks), or networks in bioinformatics (e.g., protein interaction networks).

In this paper we consider *betweenness centrality* [8, 1], which is one of the most frequently considered centrality indices. Our results might also be applicable to related concepts such as *stress centrality* [15] that are also based on counting shortest paths. Consider a weighted directed (multi)-graph $G = (V, E)$ with $n = |V|$, $m = |E|$. Let SP_{st} denote the set of shortest paths between source s and target t and $SP_{st}(v)$ the subset of SP_{st} consisting of paths that have v in their interior. Then, the betweenness centrality for node v is

$$c_B(v) := \sum_{s,t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{1.1}$$

where $\sigma_{st} := |SP_{st}|$ and $\sigma_{st}(v) := |SP_{st}(v)|$.

This definition counts the number of shortest paths through v, counting paths with alternatives only fractionally.

Our original motivation for considering betweenness was to identify sets of important nodes that can define a highway-node hierarchy [14], which is used for (dynamic) routing in road networks. For this application, the requirement is to process huge networks with many million nodes in a few minutes. In this context, we also need reasonable approximations for the betweenness of *all* nodes since we have to decide which of several neighboring unimportant nodes will make it to the first level of the hierarchy.

1.1 Related Work.

Brandes [4] gives an exact algorithm for computing betweenness of all nodes that is based on solving a single source shortest path problem (SSSP) from each node. An SSSP computation from s produces a directed acyclic graph (DAG) encoding all shortest paths starting at s. By backward aggregation of counter values, the contributions of these paths to the betweenness counters can be computed in linear time (Section 3 gives more details). Depending on the graph model, the exact algorithm takes time between $\Theta(nm)$ (e.g., for unit edge weights) and $\Theta\big(nm + n^2 \log(n)\big)$ (comparison based general edge weights). Although this is polynomial time, it is prohibitive for networks with many millions of nodes and edges. Bader and Madduri [2] present a massively parallel implementation of the exact algorithm that can handle a few million nodes.

Brandes and Pich [5] investigate how the exact algorithm can be turned into an approximation algorithm by extrapolating from a subset of k starting nodes (*pivots*), otherwise using the same aggregation strategy as the exact algorithm. Subsequently, we refer to this approximation algorithm as "Brandes' algorithm". A random sample of starting nodes turns out to work well. In particular, the randomized method yields an unbi-

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

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

ased estimator[1] for betweenness. Unfortunately, this method produces large overestimates for unimportant nodes that happen to be near a pivot. For example, consider a degree-two node v connecting a degree-one node u to the rest of the network. If u is selected as a pivot, the betweenness of v is overestimated by a factor of n/k.

1.2 Our Contributions. Our main idea is to solve the problem described above by changing the scheme for aggregating betweenness contributions so that nodes do not unduly 'profit' from being near a pivot. Section 2 describes a general framework for this idea that yields unbiased estimators for betweenness.

Our framework also applies to a simpler variant of betweenness, *canonical-path betweenness centrality* $c_C(v)$, which we usually just call *canonical centrality*. We introduce this variant due to our original motivation of dealing with road networks, where shortest routes are 'almost unique'[2]. Note that some route planning methods even enforce unique shortest paths by perturbing the edge weights (e.g., [9]). Consequently, in case of canonical centrality, we consider only a single *canonical* shortest path between any source-target pair. Our approach does not require edge perturbation. It is sufficient that some deterministic tie breaking ensures that only a single shortest path is found.

Section 3 describes how to efficiently implement two instantiations of our framework—*linear scaling*, where the contribution of a sample depends linearly on the distance to the sample, and *bisection scaling*, where a sample only contributes 'on the second half' of a path. Linear scaling can be implemented using a slight variation of Brandes' original aggregation scheme. Bisection scaling requires a quite different approach and another level of random sampling. Section 4 reports on extensive experiments with a wide range of large graphs. The linear scaling is always better than [5]. (Sampled) bisection scaling is in many ways even better. In particular, it yields good approximation of the betweenness also for less important nodes already with a small number of pivots—an area in which the original method fails. Section 5 summarizes the results and outlines possible further improvements. For example, we have evidence that betweenness approximations can help to construct better highway-node hierarchies for road networks.

[1] That means the expectation of the estimated betweenness is the actual betweenness.

[2] Indeed, multiple shortest routes *do* appear in practice. However they usually share most edges so that in most cases, the term $\sigma_{st}(v)/\sigma_{st}$ is one or zero.

2 A Generalized Framework for Betweenness Approximation

Our estimator is parametrized by a *length function* $\ell : E \to \mathbb{R}$ on the edges[3] and a *scaling function* $f : [0,1] \to [0,1]$. For a path $P = \langle e_1, \ldots, e_k \rangle$ let $\ell(P) := \sum_{1 \le i \le k} \ell(e_i)$.

In each iteration, our algorithm performs one of $2n$ possible shortest path searches with uniform probability $1/2n$. Namely, forward search in $G = (V, E)$ from a pivot $s \in V$ ($|V| = n$ possibilities) or backward search from a pivot $t \in V$, i.e., a search from t in $(V, \{(v,u) : (u,v) \in E\})$ (another n possibilities). For each shortest path of the form

$$P = \langle \overbrace{s, \ldots, v}^{Q}, \ldots, t \rangle$$

found in this way, we define a *scaled contribution*

$$\delta_P(v) := \begin{cases} \frac{f(\ell(Q)/\ell(P))}{\sigma_{st}} & \text{for a forward search} \\ \frac{1-f(\ell(Q)/\ell(P))}{\sigma_{st}} & \text{for a backward search} \end{cases} .$$

Overall, v gets a contribution

$$\delta(v) := \delta_s(v) := \sum_{t \in V} \sum \{\delta_P(v) : P \in SP_{st}(v)\}$$

for a forward search, and

$$\delta(v) := \delta_t(v) := \sum_{s \in V} \sum \{\delta_P(v) : P \in SP_{st}(v)\}$$

for a backward search.

THEOREM 2.1. *$X := 2n\delta(v)$ is an unbiased estimator for $c_B(v)$, i.e., $E(X) = c_B(v)$.*

Proof. Summing over all $2n$ possible events with probability $1/2n$ each, we get

$$E(X) = 2n \frac{\sum_{s \in V} \delta_s(v) + \sum_{t \in V} \delta_t(v)}{2n}$$

$$= \sum_{s,t \in V} \sum \left\{ \frac{\overbrace{f\left(\frac{\ell(Q)}{\ell(P)}\right) + 1 - f\left(\frac{\ell(Q)}{\ell(P)}\right)}^{=1}}{\sigma_{st}} : P \in SP_{st}(v) \right\}$$

$$= \sum_{s,t \in V} \frac{\overbrace{|SP_{st}(v)|}^{=\sigma_{st}(v)}}{\sigma_{st}} \overset{(1.1)}{=} c_B(v) \ . \qquad \square$$

[3] ℓ may or may not be identical to the edge-weight function used for shortest-path calculations.

Hence, by averaging k independent runs of the above unbiased estimator, we obtain an approximation $\frac{X_1+\ldots+X_k}{k}$ of the betweenness value of node v.

In the following, we will instantiate the length function ℓ either with the original edge weight or with unit edge weight (hop counting). We will consider three variants for the choice of f. For constant $f(x) = 1/2$ we essentially get Brandes' algorithm.[4] Our new schemes use $f(x) = x$ for *linear scaling* and

$$f(x) = \begin{cases} 0 & \text{for } x \in [0, 1/2) \\ 1 & \text{for } x \in [1/2, 1] \end{cases}$$

for *bisection scaling*. The intuition behind these methods is to reduce the contributions for nodes close to the pivot. In a sense, bisection scaling is best for this purpose. However, it will turn out that it is easier to implement linear scaling.

3 Linear Time Computation of Contributions

A naive implementation of the definitions from Section 2 could take quadratic time even for a single pivot and canonical centrality. We will give algorithms that evaluate the shortest path DAG in time linear in its size. We only explain the computations for forward search from source s. Backward search works analogously.

3.1 Brandes' Algorithm. Brandes [4] already explains how to compute σ_{st} on the fly during the shortest path calculations: $\sigma_{ss} = 1$ and, for $s \neq t$, $\sigma_{st} = \sum_{v \in pred(t)} \sigma_{sv}$ where $pred(t)$ is a multiset containing the immediate predecessors of t in the shortest path DAG.[5] In a subsequent aggregation phase, the nodes are processed in reverse topological order, i.e., by non-increasing distance from s. We get,

$$\delta_s(v) = \sum_{w \in succ(v)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w))$$

where $succ(v)$ denotes the immediate successors of v in the shortest path DAG.[6] The factor $\frac{\sigma_{sv}}{\sigma_{sw}}$ takes care of distributing the contributions from w to possible multiple parents. The '1' is the contribution due to paths of the form $\langle s, \ldots, v, w \rangle$ and the $\delta_s(w)$ takes care of nodes reached indirectly via w.

3.2 Linear Scaling is easiest to implement by using the original edge weights as the length function ℓ. Then it is possible to implement it with only small deviations from Brandes' scheme. We have

$$\delta_s(v) = \sum_{w \in succ(v)} \frac{\mu(s, v)}{\mu(s, w)} \cdot \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w))$$

where $\mu(s, w)$ is the shortest path distance from s to w. Less formally, the modification compared to Brandes' scheme is to put values $1/\mu(s, w)$ into the aggregation rather than 1. At the end, multiplying with $\mu(s, v)$ yields the correct scaling of $\mu(s, v)/\mu(s, w)$ for all values put into the aggregation.

3.3 Bisection Scaling For Canonical Centrality. Here, we mostly use unit distances for the length function ℓ. (Recall that ℓ is not necessarily the same as the edge weight.) We do aggregation by a depth-first traversal of the shortest-path tree. This allows us to keep an array storing the path from s to the currently explored node. Aggregation works similarly to Brandes' algorithm with one major modification. When a node v at depth d is explored, let v' denote the node on position $\max(0, \lfloor d/2 \rfloor - 1)$. We decrement the current value for $\delta_s(v')$. This has the effect of dropping the contribution of v from the aggregation just where it is prescribed by the scaling function f. We have also implemented a variant for general length functions. Here, we have to search the stack for $\ell(\langle s, \ldots, v \rangle)/2$ starting at the position used for the predecessor of v. Although this has linear worst case complexity, it works reasonably well for road networks using travel time.

3.4 Bisection Scaling For General Betweenness Centrality. We have a direct implementation of *enumerative bisection scaling* that enumerates all paths by backtracking. This works well for graphs with few redundant paths but can be very slow in the worst case. What is more interesting is that the general case can be reduced to the canonical case: we randomly sample a parent pointer for each node t in the shortest path DAG. We call this variant *bisection sampling*. If a parent p of w is selected with probability $\frac{\sigma_{sp}}{\sigma_{sw}}$, we get an unbiased estimator. We can extract more information out of the shortest path DAG by performing several sampling steps for the same DAG, which does not invalidate the fact that the estimator is unbiased.

[4] One can also do just forward searches in this case.

[5] Since there may be an exponential number of paths, this recurrence might lead to arithmetic overflows if one would use integer arithmetics on machine words. However, we are only interested in approximations, so that it suffices to compute with floating point numbers with mantissas of logarithmically many bits. Our implementation uses double precision arithmetics that flags overflows as ∞-values. This never happened yet for the inputs we considered.

[6] In our framework with backward searches we have to divide by two at the end.

graph	nodes	edges	source	average time per pivot [ms]
Belgian road network	463 514	596 119	PTV AG	1 337
Belgian road network with unit distance	463 514	596 119	PTV AG	914
Actor co-starring network	392 400	16 557 451	[12]	6 242
US patent network	3 774 769	16 518 947	[10]	5
World-Wide-Web graph	325 729	1 497 135	[12]	144
CNR 2000 Webgraph	325 557	3 216 152	[11]	362
CiteSeer undir. citation network	268 495	2 313 294	[6]	1 711
CiteSeer co-authorship network	227 320	1 628 268	[6]	835
CiteSeer co-paper network	434 102	32 073 440	[6]	4 110
DBLP co-authorship network	299 067	1 955 352	[7]	1 323
DBLP co-paper network	540 486	30 491 458	[7]	5 024

Table 1: Overview of used graphs. A co-paper network has papers as nodes and edges between papers that share at least one author. The values in the last column refer to general betweenness centrality and Brandes' algorithm.

4 Experiments

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

We have approximated canonical centrality for several real-world road networks with up to 42 million edges using travel times as edge weights. We do most quality evaluations here with a subnetwork for Belgium with 463 514 nodes because for this we can still compute exact values.[7]

For general betweenness we used networks with unit edge weights stemming from a variety of applications including citation networks, coauthorship networks, web graphs and communication networks (AS graph, router level graphs). Table 1 gives additional information on these networks. Our focus was on large, real-world graphs where we can still compute exact betweenness in reasonable time. This includes the largest graphs from [2]. We do not use the graphs considered in [5] since they are mostly randomly generated and comparatively small. The evaluation here uses the most difficult of these instances[8]—a movie actor multigraph whose edges indicate costarring in some movie. This graph has 392 400 nodes, 16 557 451 edges, and many shortest paths between most pairs of nodes.

For assessing the quality of the estimation, we adopt the measures proposed by Brandes and Pich [5]—Euclidean distance of the normalized n-dimensional vectors for exact and estimated centrality, respectively, and the number of inversions[9] when comparing the rankings produced by exact and estimated centrality, respectively. The number of inversions is computed using a mergesort-based $\mathcal{O}(n \log n)$ algorithm. The Euclidean distance is mostly governed by the nodes with large betweenness, whereas the number of inversions treats all nodes equally. We have also looked at other measures. For example, the *average absolute betweenness error* turned out to give similar results as the Euclidean distance[10], whereas the *geometric mean of relative rank errors*[11] has similar characteristics as the number of inversions.

Since our new algorithms need slightly more time than Brandes' algorithm for evaluating the shortest path DAG, we ensure a fair comparison by giving our algorithms no more *time* than Brandes' algorithm; thus, they will perform less iterations. Effectively, the time needed by Brandes' algorithm for one pivot is our unit of time.

Figure 1 evaluates the approximation of canonical centrality for Belgium. With respect to the Euclidean distance, all algorithms show benign and predictable behavior. Still, our new methods fare uniformly better with bisection slightly ahead. In the same running time we get an about two times smaller distance or alternatively, we achieve about the same quality in eight times less running time. With respect to the number of inversions, the difference is much more dramatic. Even after

[7] Computing exact values for our complete Western European road network would take about 12 years.

[8] The US patent network is much bigger but relatively easy to solve exactly since most searches reach only a small number of nodes.

[9] i.e., the total number of pairs that are in the wrong order.

[10] Note that the normalization is likely to have little effect since all the compared methods are unbiased and hence the computed vectors are likely to have similar length anyway.

[11] Let r denote the ratio between the estimated rank and the true rank. Then the relative rank error is $\max\{r, 1/r\}$.

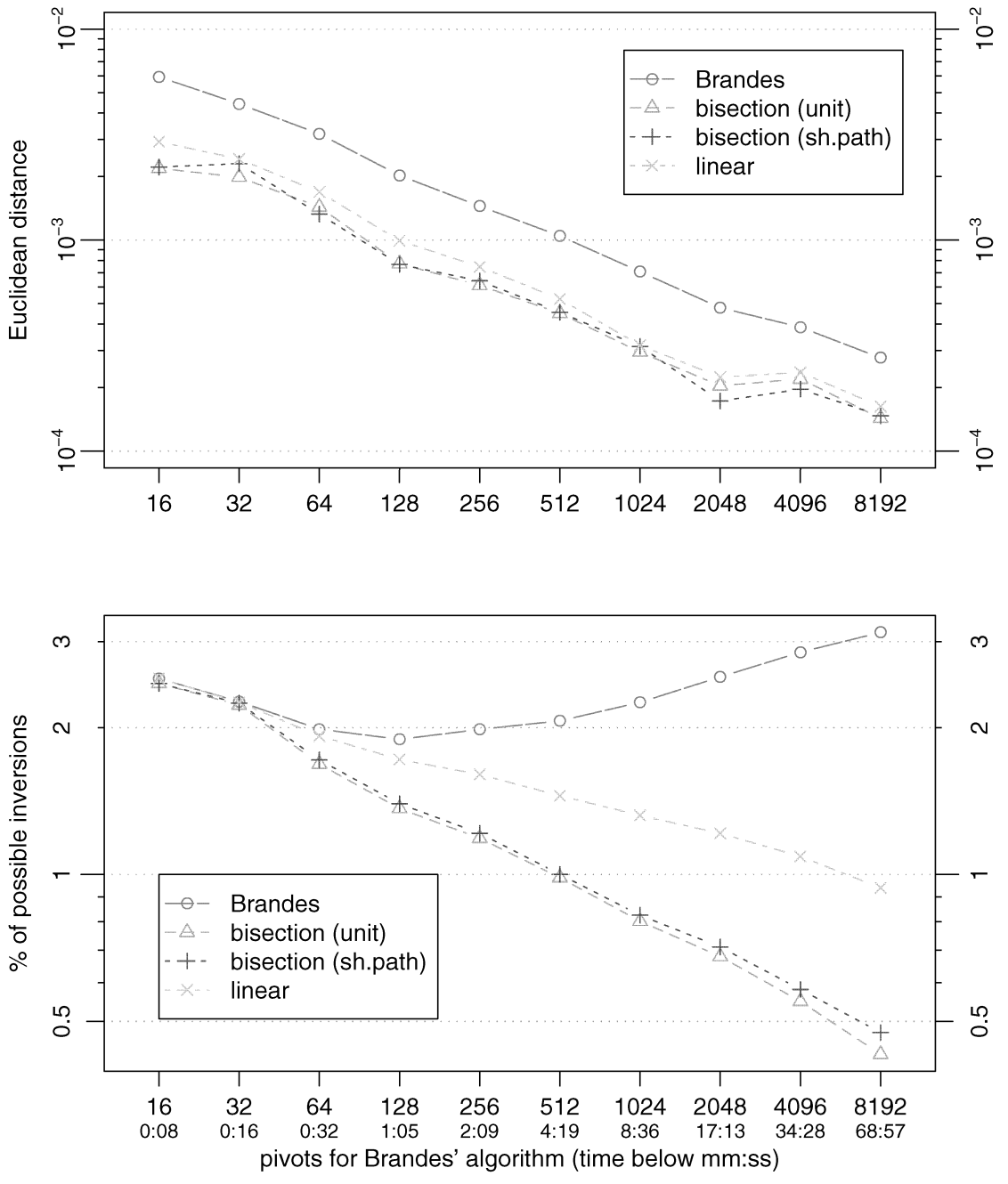


Figure 1: Euclidean distance and inversions for canonical centrality of Belgium.

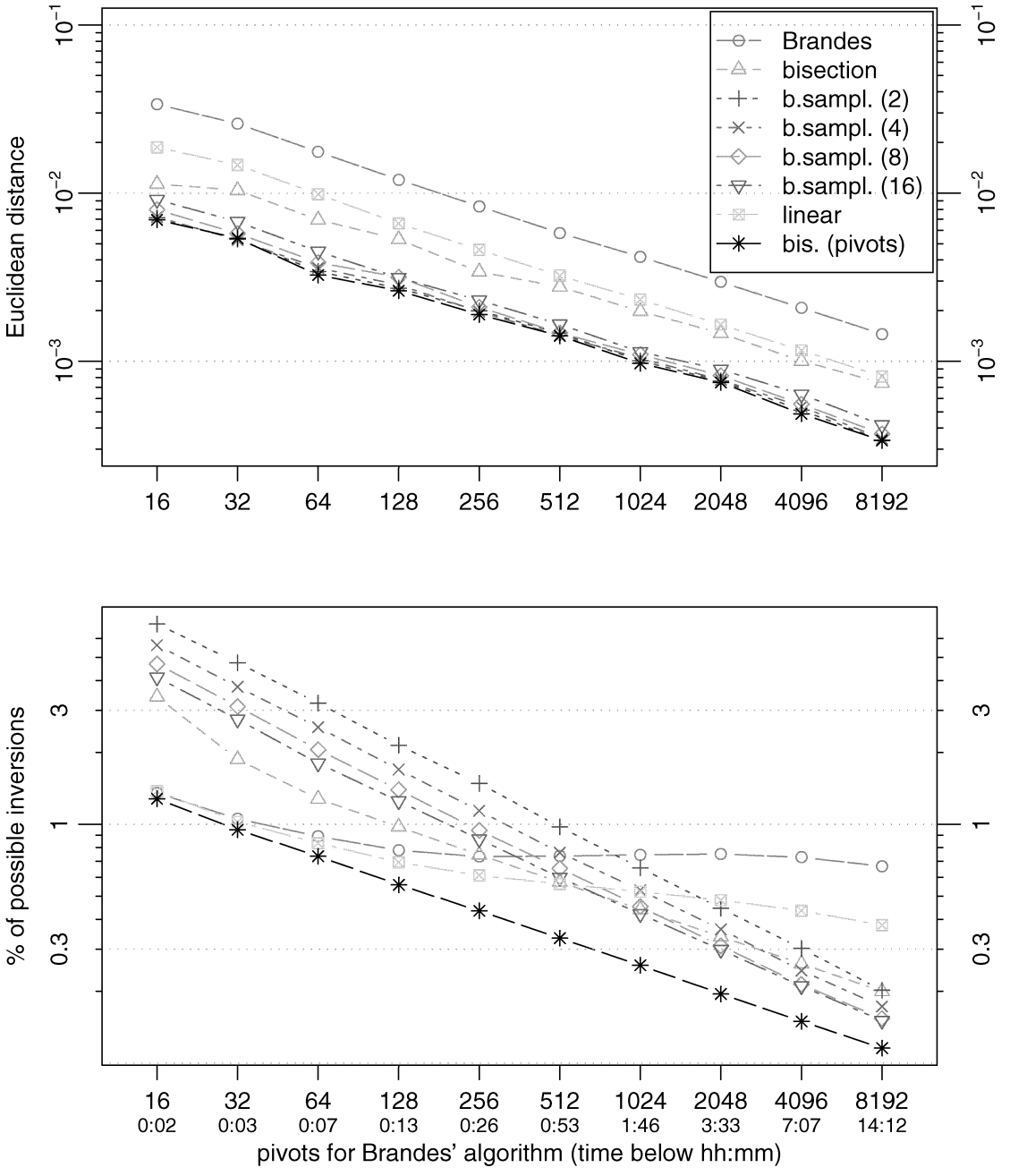


Figure 2: Euclidean distance and inversions for betweenness in the actor network.

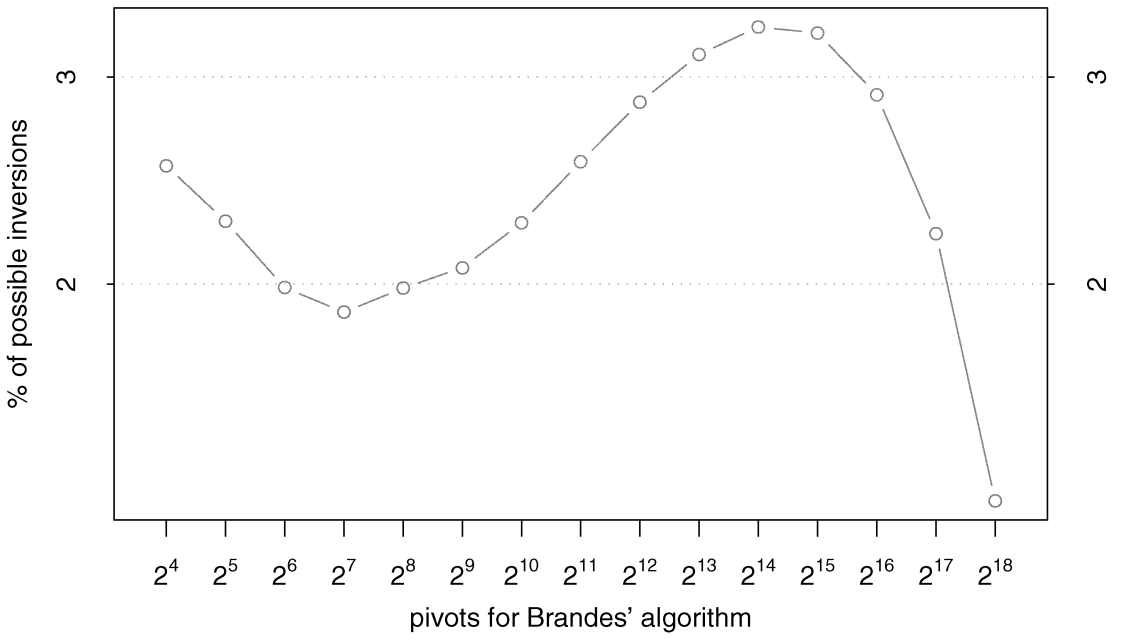


Figure 3: Inversions, canonical centrality of Belgium, Brandes' algorithm.

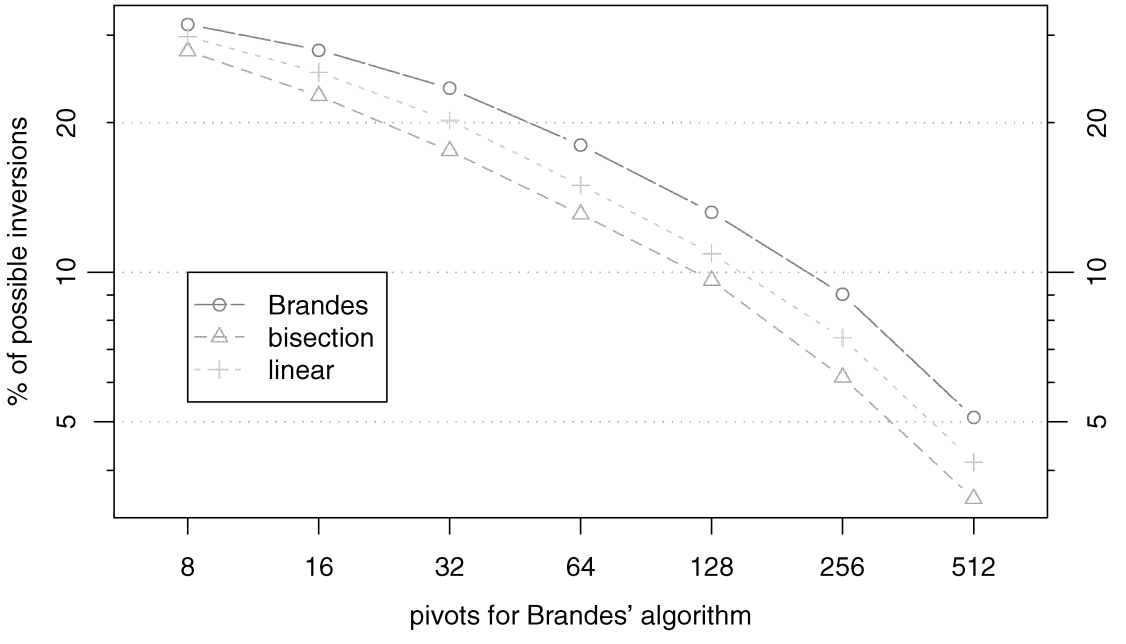


Figure 4: Inversions, canonical centrality of a random graph with 1 000 nodes and 10 074 edges.

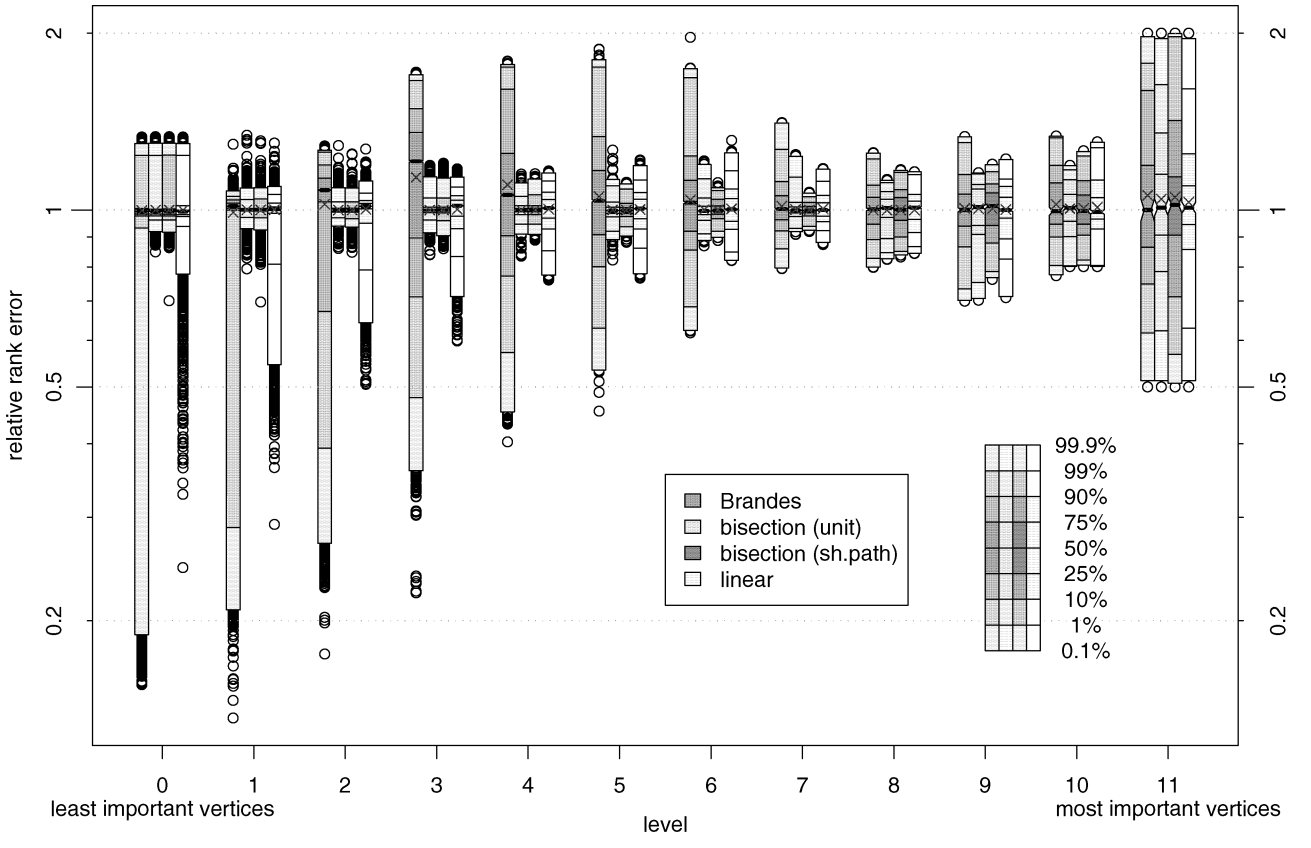


Figure 5: Distribution of the relative rank errors for 12 different levels (categories) of importance in the Belgian road network after 1024 iterations of Brandes' algorithm. Level 11 contains the 128 nodes with largest exact canonical centrality, Level 10 the next 256 most important nodes, and so on. The cross gives the average value. The circles denote outliers.

8192 iterations, Brandes' algorithm does not even begin to converge.[12] This behavior is unexpected since the random graphs used in [5] are much more 'well behaved' (see also Figure 4). A possible explanation is that the fate of most unimportant nodes is that their centrality is initially *slightly* underestimated. However, the more pivots are used, the more unimportant nodes near the pivots become *grossly* overestimated. The net effect on the number of inversions is positive. Our new algorithms show a nice, near linear behavior on a double-logarithmic plot. Bisection scaling clearly outperforms linear scaling. To understand what is going on, let us analyze in more detail how approximation errors are distributed over the nodes. Figure 5 illustrates the distribution of the relative rank errors for 12 different levels (categories) of importance. We can see that bisection scaling gives uniformly good approximation quality over all levels, while Brandes' algorithm has orders of magnitude larger errors for the less important levels, which constitute the majority of the nodes. All methods have comparatively large errors for the highest level of importance. The reason is that the betweenness values for these nodes are very similar so that already small errors in the absolute value can lead to large deviations in relative ranks.

Figure 2 evaluates betweenness approximation for the actor network. For the Euclidean distance, we get similar results as before. Interestingly, enumerative bisection fares quite well, although it is about four

[12] Figure 3 indicates that the numbers, in the very end, *do* go down.

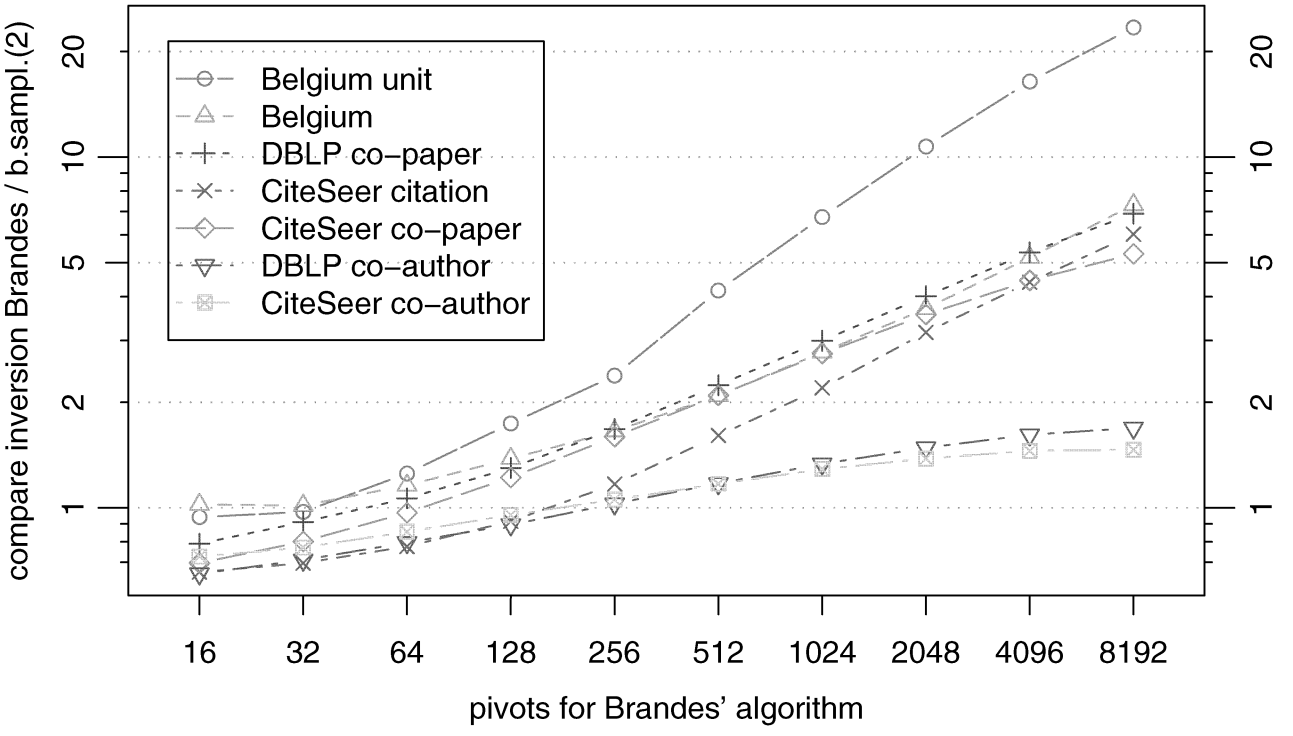


Figure 6: Number of inversions for Brandes' algorithm divided by number of inversions for bisection scaling with two samples for different networks.

times slower per iteration than the other methods. In particular, it outperforms linear scaling. Bisection sampling fares best. It has about four times smaller errors and needs about 16 times less time to achieve the same error bound as Brandes' algorithm with 8192 iterations. The number of sampled trees per pivot does not have a big influence on performance.

For the inversion number, the ranking of the algorithms is less clear. The linear scaling algorithm always outperforms Brandes' algorithm, which stagnates for many iterations before finally heading down after thousands of iterations. However, both Brandes and linear scaling have a better start than the other variants. Only after 1024 iterations, the bisection based algorithms start to take over. The main difference to the more clear-cut ranking for road networks seems to be the presence of highly redundant shortest paths. In this situation, bisection sampling throws away information that is essential if only few pivots are used. An analysis of a level plot similar to Figure 5 reveals that bisection sampling outperforms Brandes' algorithm on most levels but has deficits for the least important levels. Still, bisection sampling has a nice near-linear decrease of the inversions on a double-logarithmic plot that eventually, after a sufficient number of iterations, leads to superior performance. It is also interesting to see what would happen if we had a fast algorithm for exploiting all the information available. The lowest curve in Figure 2 plots the performance of our enumerative bisection algorithm based on just counting iterations of the outer loop. We see that this hypothetical algorithm outperforms all other algorithms from the beginning.

We see a similar pattern for many of the other networks. Figure 6 shows the improvement yielded by bisection sampling over Brandes' algorithm. For few pivots, Brandes' algorithm is up to a factor of two better. For many pivots, bisection sampling is up to a factor 20 better. With respect to the Euclidean distance, bisection sampling is always better for all of these networks. The only deviation from this pattern we found were the directed networks from Table 1—the US patent network and the web graphs. Here,

all approximation algorithms had problems to achieve good approximation. Linear scaling was always slightly better than Brandes' algorithm. Bisection sampling was sometimes better, sometimes worse. Shortest path searches turned out to be very local. This means that all approximation algorithms have trouble eliminating false zeroes for nodes of small betweenness. This is a disadvantage for bisection sampling since it is slower and since it produces zero contributions where Brandes and linear scaling produce positive contributions. On the other hand, the same effect makes these networks relatively easy for the *exact* algorithm. For example, our implementation solves the US patent graph in 127 minutes—only 2.5 times more time than Bader and Madduri [2] need using 16 IBM-P5 processor cores.

5 Conclusion and Future Work

Our new bisection scaling algorithm is a versatile method for approximating canonical centrality that works well even for huge networks. The algorithm has considerably better performance than previous methods for all graphs tried and for all global quality measures. This good evaluation largely extends to the original definition of betweenness and the sampled bisection algorithm. However, for a small number of pivots, the linear scaling variant of our framework is still better. Enumerative bisection might be the overall winner here if we found a near linear time implementation.

All algorithms discussed are easy to parallelize with up to k processors. We have not yet attempted any pragmatic tuning of the algorithms. For example, by using robust statistics (e.g., omitting the smallest and largest measurements) it might be possible to reduce the number of nodes with grossly overestimated betweenness. At least for road networks it should also help to do some preprocessing like getting rid of degree-one nodes. In multigraphs like the actor network, we might get improved performance by modeling parallel edges more explicitly.

For some directed networks, none of the approximation algorithms gives very convincing results since a good approximation takes almost as much time as an exact calculation. Although we did not find undirected graphs with similarly bad performance, better theoretical performance bounds[13] or a more detailed experimental investigation on the influence of graph structure on performance remain an interesting open problem.

[13] [5] gives a probabilistic tail bound which contains a bug however. Repairing this bug yields a bound only useful for nodes with near quadratic betweenness.

5.1 Path Sampling. Bias due to the selection of sampled source nodes can be avoided altogether by sampling both source and target node rather than attempting to gain as much information as possible from a single-source (all-target) shortest-path computation. The obvious drawback is that we get $n-1$ times fewer paths per sample. We can mitigate this problem by using speedup techniques for answering the queries. In [13] this is done using inexact heuristics so that it is unclear what is actually computed. At least for road networks we can do much better using recently developed exact speedup techniques that are very efficient. In particular, transit-node routing [3] achieves nearly constant query time and thus might lead to very good results. More precisely, transit-node routing computes *transit nodes* u and v such that there is a shortest path from source s via u and v to target t. We can process this information by incrementing counters for the access connections $\langle s, u\rangle$ and $\langle v, t\rangle$ and the transit connection $\langle u, v\rangle$. Only at the end, we have to propagate these counters down to the actual nodes (or edges) of the underlying network. This propagation cannot take longer than building the data structures during preprocessing. We have not implemented path sampling because it is complicated and unlikely to work better than bisection sampling for road networks. In particular, we need a huge number of samples before unimportant nodes get a significant contribution. On the other hand, if we were interested in a small selection of nodes with highest betweenness and the transit-node preprocessing would be needed anyway, then path sampling would be very efficient and we could even derive good performance guarantees using Chernoff bounds.

5.2 Including Local Searches. We have developed a generalization of our framework that allows a mix of local and global search and remains unbiased. This might lead to better estimates for not-so-important nodes. In particular, false zero values might be reduced by local searches around nodes with zero betweenness estimate. Preliminary experiments indicate that the general idea works but does not yield dramatic improvements for the instances tried.

5.3 Application to Highway-Node Routing. Highway-node routing [14] requires a nested hierarchy of more and more important nodes. We have preliminary results on how to use betweenness estimations to define improved highway-node hierarchies. At the same time, the resulting method is simpler than the approach used in [14]. However, a really good scheme seems to require additional heuristics beyond the scope of this paper.

Acknowledgments. Robert Görke has provided us with citation networks crawled from DBLP and Citeseer. Kamesh Madduri has provided the patent network and the actor network. We would like to thank Reinhard Bauer, Ulrik Brandes, Daniel Delling and Dorothea Wagner for interesting discussions. Several anonymous reviewers provided valuable suggestions.

References

[1] J. M. Anthonisse. The rush in a directed graph. Technical Report BN 9/71, Stichting Mathematisch Centrum, Amsterdam, 1971.

[2] David A. Bader and Kamesh Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *ICPP*, pages 539–550. IEEE Computer Society, 2006.

[3] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[4] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[5] Ulrik Brandes and Christian Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, to appear. special issue on Complex Networks' Structure and Dynamics.

[6] CiteSeer. Scientific Literature Digital Library. `http://citeseer.ist.psu.edu/`, 2007.

[7] DBLP. DataBase systems and Logic Programming. `http://dblp.uni-trier.de/`, 2007.

[8] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40:35–41, 1977.

[9] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A^*: Efficient point-to-point shortest path algorithms. In *Workshop on Algorithm Engineering & Experiments*, pages 129–143, Miami, 2006.

[10] A. B. Jaffe Hall, B. H. and M. Tratjenberg. The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools. *NBER Working Paper*, 8498, 2001.

[11] Laboratory for Web Algorithmics. `http://law.dsi.unimi.it/index.php?option=com_include&Itemid=65`.

[12] Notre Dame CNet resources. `http://www.nd.edu/~networks/`.

[13] M. J. Rattigan, M. Maier, and D. Jensen. Using structure indices for efficient approximation of network properties. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge Discovery and Data mining*, pages 357–366. ACM, 2006.

[14] D. Schultes and P. Sanders. Dynamic highway-node routing. In *6th Workshop on Experimental Algorithms*, volume 4525 of *LNCS*, pages 66–79. Springer, 2007.

[15] Alfonso Shimbel. Structural parameters of communication networks. *Bulletin of Mathematical Biophysics*, 15:501–507, 1953.

Decoupling the CGAL 3D Triangulations from the Underlying Space*

Manuel Caroli † Nico Kruithof ‡ Monique Teillaud §

Abstract

The *Computational Geometry Algorithms Library* CGAL currently provides packages to compute triangulations in $\mathbb{R}^2$ and $\mathbb{R}^3$. In this paper we describe a new design for the 3D triangulation package that permits to easily add functionality to compute triangulations in other spaces. These design changes have been implemented, and validated on the case of the periodic space $\mathbb{T}^3$. We give a detailed description of the realized changes together with their motivation. Finally, we show benchmarks to prove that the new design does not affect the efficiency.

1 Introduction

Computing Delaunay triangulations and the more general regular triangulations is a well-studied subject in Computational Geometry. There are many algorithms available [dBvKOS00, ES96] as well as several implementations [cga, She96, Hel01, qhu]. The algorithms work in general for the d-dimensional Euclidean space $\mathbb{R}^d$, and the implementations are usually restricted to $\mathbb{R}^2$ or $\mathbb{R}^3$. However, there are also applications for triangulations in spaces other than $\mathbb{R}^d$. For instance, in simulation, one is typically interested in having no boundary conditions. This can be achieved by computing a triangulation in the periodic space denoted by $\mathbb{T}^d$. Currently, the CGAL triangulation package is generic with respect to the implementation of the basic arithmetic operations and geometric tests and the triangulation data structure (Section 3) but the embedding space is bound to be $\mathbb{R}^3$. The goal of this work is to extend the triangulation package such that it is possible to easily add functionality to compute in other spaces. For the implementation, we will restrict ourselves to the case of three-dimensional triangulations, also referred to as tetrahedrizations.

*This work was partially supported by the ANR (Agence Nationale de la Recherche) under the Triangles project of the "Programme blanc" (No BLAN07-2_194137) http://www-sop.inria.fr/geometrica/collaborations/triangles/

†INRIA Sophia-Antipolis (Manuel.Caroli@sophia.inria.fr)

‡Nico Kruithof worked on this during his stay at INRIA Sophia-Antipolis (Kruithof@jive.nl)

§INRIA Sophia-Antipolis (Monique.Teillaud@sophia.inria.fr)

The CGAL 3D triangulation package implements two types of triangulations: *Delaunay triangulations* and *regular triangulations* [PT07b]. We shortly introduce both triangulations in the following section. The original triangulation package has been designed for computing in $\mathbb{R}^3$ only; we show how to add one more layer of genericity, which implies considerable modifications in the package design. However, we require the package to remain backward compatible and not to lose performance. The package modifications will be discussed in detail in the third section. The described changes enable us to add functionality for computing triangulations in the periodic space $\mathbb{T}^3$. In Section 4, we give more details about the properties of $\mathbb{T}^3$ and the implementation. Finally, we present some benchmarking results of the current design and the new design[1] and for different spaces in Section 5.

2 Delaunay Triangulation and Regular Triangulation

Given a set $\mathcal{S}$ of points, a triangulation partitions the space into cells (tetrahedra in 3D) whose vertices are the given points. The *Delaunay triangulation* has the property that the circumsphere of each cell does not contain any other point of $\mathcal{S}$ [BY98, dBvKOS00].

The regular triangulation generalizes the Delaunay triangulation by associating a weight with every point. Let $\mathcal{S}^{(w)}$ be a set of weighted points $p^{(w)} = (p, w_p) \in \mathbb{R}^3 \times \mathbb{R}$. The weighted point $p^{(w)}$ can also be seen as a sphere of center p and radius $\sqrt{w_p}$. To define the regular triangulation of $\mathcal{S}^{(w)}$, we first need to introduce the notion of *power product* and *power sphere*.

DEFINITION 2.1.

- *The* power product *of two weighted points* $p^{(w)} = (p, w_p)$ *and* $q^{(w)} = (q, w_q)$ *is defined as* $\Pi(p^{(w)}, q^{(w)}) := \|p - q\|^2 - w_p - w_q$.

[1] We will use the term *current* to refer to the implementation available in the public release 3.3 of CGAL, and the term *new* to refer to the implementation we present in this paper, which is available only in internal releases for the moment.

- *The* power test *checks the sign of the power product of two weighted points.*
  The power product is zero if the two spheres corresponding to $p^{(w)}$ and $q^{(w)}$ intersect orthogonally; in this case, the two weighted points are said to be orthogonal.
- *Four weighted points have a unique common orthogonal weighted point called the* power sphere.

A sphere $s^{(w)}$ is said to be *regular* if the inequality $\Pi(p^{(w)}, s^{(w)}) \geq 0$ holds for all $p^{(w)} \in \mathcal{S}^{(w)}$. A triangulation of $\mathcal{S}^{(w)}$ is regular if the power sphere of each cell of the triangulation is regular.

A cell and a point $p^{(w)}$ are said to be *in conflict* if $\Pi(p^{(w)}, s) < 0$, where s is the power sphere of the tetrahedron. See Figure 1 for an illustration in 2D.

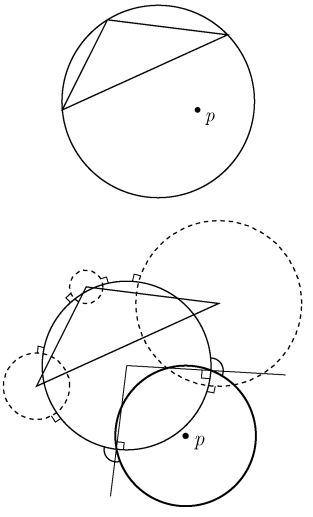


Figure 1: A cell and a point in conflict in Delaunay triangulation (top) and regular triangulation (bottom)

It is easy to see that if all points have equal weights, a sphere is regular if it does not contain any other point, and a sphere is in conflict with a point if it contains it; in this case, the regular triangulation is the Delaunay triangulation of the set of non weighted points. The regular triangulation is also referred to as *weighted Delaunay triangulation* or *power triangulation.* To know more about regular triangulations, see for example [ES96, Aur87].

2.1 Algorithm. The current implementation of the CGAL triangulation package uses an incremental approach to compute the triangulation. This means the points are added one by one. The algorithm for inserting a single point works as follows:

- `locate`: Locate the cell containing the point. Also report degeneracies, e.g. point on a facet or edge.
- `find_conflicts`: Mark all the cells that are in conflict with the newly added point.
- `insert`: Call `find_conflicts`, delete all cells in conflict, which creates a "hole" in the triangulation, then create new cells to fill the hole.

Additionally, there are many auxiliary functions providing access to the triangulation. Most of these functions slightly modify already computed and internally stored properties [PT07b]. Therefore, they are not considered in the further discussion.

2.2 Particularities of $\mathbb{R}^3$. In the current CGAL triangulation package, several aspects are specialized to $\mathbb{R}^3$:

A triangulation in $\mathbb{R}^3$ partitions the convex hull[2] of the point set. To handle the unbounded cell outside the convex hull, an infinite vertex is added to the triangulation, and all the facets of the convex hull are connected with this infinite vertex. In this way, all cells have four vertices and four adjacent cells. The underlying combinatorial graph of the triangulation of $\mathbb{R}^3$ can be seen as a triangulation of the topological sphere $\mathbb{S}^3$ in $\mathbb{R}^4$.

Furthermore, we have to deal with degenerate dimensions when computing triangulations in $\mathbb{R}^3$. For instance, if all the points lie in a plane, a two-dimensional triangulation must be computed and stored. This requires additional implementation effort for some functions.

We want to allow implementations of triangulations in different spaces in the new triangulation package. One of them is homeomorphic to the hypersurface $\mathbb{T}^3$ of a torus in 4D (see Section 4 for more details). The previously mentioned properties of $\mathbb{R}^3$, currently hard-coded, do not hold in $\mathbb{T}^3$.

3 Design

In this section, we first describe the design of the current triangulation package. Then we introduce the changes that have to be done in order to enable triangulations in different spaces.

Like the overall CGAL library, the triangulation package follows the Generic Programming paradigm [BKSV00, FT06]. The policy of CGAL is to provide code that is generic, flexible, and easily adaptable to specific needs of the user. The main class `Triangulation_3` is

[2] The convex hull of a set of points is the smallest convex set containing the points.

built as a layer on top of the triangulation data structure that stores the combinatorial structure of the computed triangulation. This allows a separation between the geometry and the combinatorics, which is reflected by the fact that the triangulation class takes two template parameters:

- the **geometric traits** class, which defines basic geometric objects (e.g. points, segments, triangles, tetrahedra) and predicates (e.g. orientation test, in_sphere test for Delaunay triangulation and power test for regular triangulation). A default traits class is provided in the package, but it can also be substituted by user provided traits.
- the **triangulation data structure** class, which stores the combinatorial structure and takes care of its validity [PT07a]. It is described in more detail in the following section.

3.1 The Triangulation Data Structure. To explain a data structure for storing triangulations, we need a more precise definition of a triangulation. The notion of simplicial complex must be recalled first. More details can be found in [Zom05, RV06].

Definition 3.1.

- *A* k-simplex *is the convex hull of a set of* $k+1$ *affinely independent points. A 0-simplex is called a* vertex.
- *If* σ *is a simplex defined by a finite point set* $\mathcal{S}$*, then any simplex* τ *defined by* $\mathcal{T} \subset \mathcal{S}$ *is called a* face *of* σ.
- *A* simplicial complex K *is a finite collection of non-empty simplices such that the following two conditions hold:*
  1. *if* $\sigma \in K$ *and* τ *is a face of* σ*, then* $\tau \in K$,
  2. *if* $\sigma_1, \sigma_2 \in K$*, then their intersection* $\sigma_1 \cap \sigma_2$ *is either empty or a face of both* σ_1 *and* σ_2.

Now we can define a triangulation as follows:

Definition 3.2. *Let* $\mathcal{S}$ *be a finite point set in some space* $\mathbb{X}$*. A simplicial complex* K *is a* triangulation *of* $\mathcal{S}$*, if*

1. *each point in* $\mathcal{S}$ *is a vertex of* K,
2. $\bigcup_{\sigma \in K} \sigma$ *is homeomorphic to* $\mathbb{X}$.

The triangulation data structure stores a purely combinatoric graph without any geometric information. It consists of a container of cells (3-faces) and a container of vertices (0-faces).

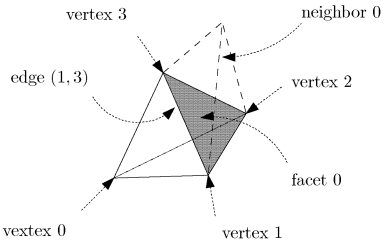


Figure 2: Representation.

Each cell stores pointers to its four vertices and its four adjacent cells. Each vertex stores one of its incident cells (Figure 2).

In order to achieve a high flexibility of design, the classes that store the cells and vertices can be specialized or even completely replaced by the user.

3.2 The Triangulation Class and its Specializations. The main class `Triangulation_3` is specialized to `Delaunay_triangulation_3` and `Regular_triangulation_3` as shown in the derivation diagram in Figure 3.

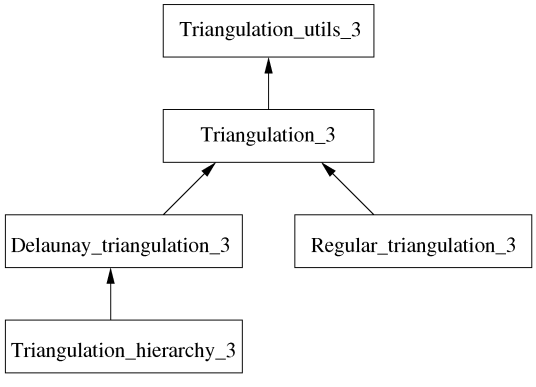


Figure 3: The current design.

These three classes provide high-level geometric functionality as member functions, such as location of a point in the triangulation [DPT02], point insertion, and vertex removal [DT03, DT06], and are responsible for the geometric validity. As mentioned before, they are parameterized by the geometric traits and by the triangulation data structure. This diagram shows two other classes:
- `Triangulation_utils_3` provides a set of tools operating on the indices of vertices in cells,
- `Triangulation_hierarchy_3` implements a hierarchy of triangulations suitable for speeding up point location [Dev02].

The geometric functions in the triangulation

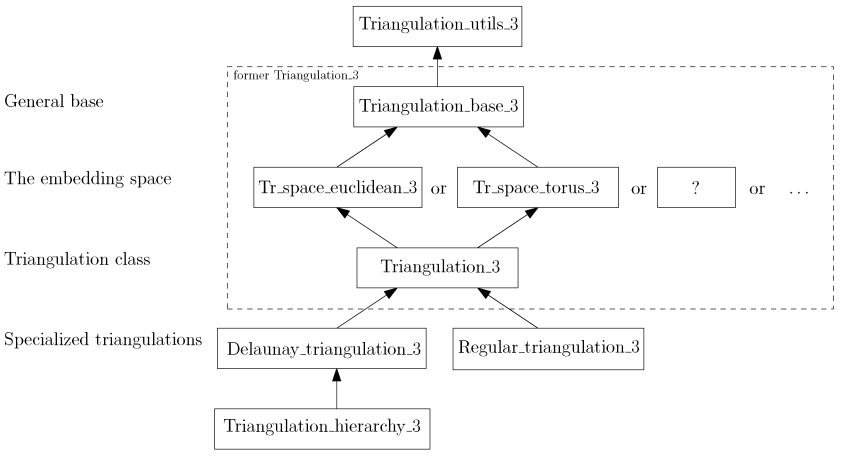


Figure 4: The new design.

class then trigger the combinatorial functions in `Triangulation_data_structure_3`. For instance, `insert` first performs the point location and then computes the conflict region (Section 2.1). These tasks involve only geometric predicates and do not change the combinatorial triangulation. Once the conflict region is known, its cells are deleted and replaced by new cells, which is a purely combinatorial operation performed by `Triangulation_data_structure_3`.

3.3 The New Design. The goal of the new design is to make it possible to easily extend the current implementation by several different spaces with the least possible redundancy in code. The basic idea of the new design is to split the current class `Triangulation_3` into three classes related by inheritance (cf. Figure 4). The embedding space class provides all functionality that depends on the space. It is now a template parameter of the triangulation, together with the triangulation data structure and the geometric traits.

More details are given in Section 3.4. Let us now list the affected classes in more detail and emphasize on the changes to the current design:

Triangulation data structure: The triangulation data structure does not change in the new design, since the invariant of storing a simplicial complex does not depend on the space.

Geometric traits: The geometric traits class can be substituted by an appropriate traits class if needed by the embedding space, in the same way as it can be modified by the user in the current design.

Triangulation embedding space: This class is new in the design and is used to handle everything that depends on the embedding space of the triangulation. There is functionality that depends on both space and triangulation type. In these cases we use visitors that modify the functionality relevant to the triangulation type in the space class. This design pattern is described in more detail in Section 3.5.

Triangulation: The Triangulation class provides the same interface as in the current design, independent of the embedding space. It provides generic algorithms for point location, point insertions and flips. This class is finally specialized to Delaunay triangulation and regular triangulation.

Introducing new spaces becomes easy with this design: only the space class (and possibly the geometric traits) is needed, since the algorithms provided by the triangulation classes are fully generic.

3.4 Redistributing the Functionality of `Triangulation_3`. In the current triangulation package, the functionality is distributed as shown in Figure 5.

In the Delaunay triangulation as well as in the regular triangulation, the `find_conflicts` function and the `insert` function overload the corresponding functions of `Triangulation_3`. Additionally, both specialized triangulation classes implement their own `remove` method: it removes the given vertex from the data structure and uses `insert` to retriangulate the hole.
In the new design, the functionality is redistributed as shown in Figure 6. All of the four functions mentioned in the figures depend on the space used. That means that we must provide a different implementation for each space. Therefore, the class `Triangulation_base_3` provides only basic functionality that is independent of space and triangulation type used. The space classes

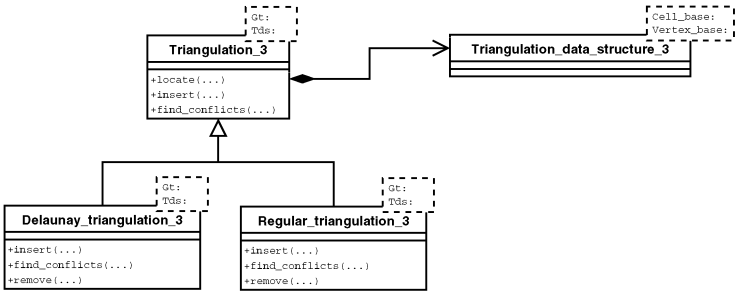


Figure 5: Distribution of functions in the current package (simplified).

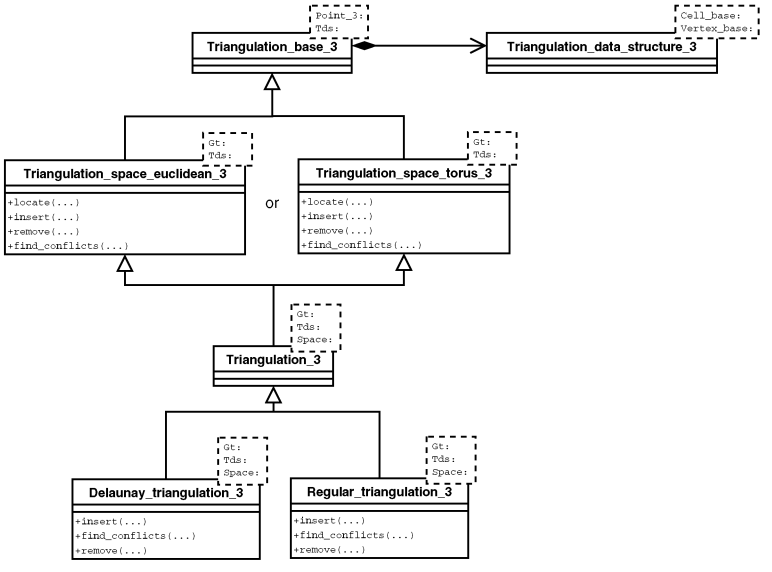


Figure 6: Distribution of functions in the new design (simplified).

contain the actual geometric functionality: point location, conflict search, point insertion, and point removal. Out of these four functions, only the point location does not depend on the triangulation type. This means that we need to implement each of the remaining three functions twice for each space (once for Delaunay and once for regular triangulation). To minimize code redundancy, we use the design pattern of visitors as explained in the next section.

3.5 Visitors. In a class hierarchy, it is usually hard to extend subclasses with new functions, because the new functions must be added to *each* subclass. The idea of the visitor design pattern [GHJV95] is to move functions operating on objects from the class hierarchy into their own classes, called *visitors*. In this way, functions can be reused for several types of objects, and adding a new function corresponds to only adding a new visitor.

In our implementation we modify the presented ideas of the visitor pattern to meet the generic programming paradigm. Visitors have been used in CGAL before, e.g. to permit the user to provide his own functions that must be applied during the algorithm run [WFZH07]. In our approach, we use visitors to exchange triangulation type dependent functionality in the functions from the space class.

The naive solution to our problem would be to implement four different classes for

- Delaunay triangulation in $\mathbb{R}^3$,
- Regular triangulation in $\mathbb{R}^3$,
- Delaunay triangulation in $\mathbb{T}^3$,
- Regular triangulation in $\mathbb{T}^3$.

This solution would require another pair of triangulation classes for each further space. Instead, we implement two space classes providing functionality for both triangulation types. The space classes correspond to the object in the above description of the visitor pattern. Now, we factor out the parts of the code that have to be implemented differently depending on the triangula-

```
class Conflict_tester {
  typename Conflict_vertices_iterator;
  typename Hidden_points_iterator;

  // Constructor: the visitor is initialized with the point to test
  Conflict_tester(Point, Embedding_space<Traits, Tds>)

  // returns true if the point is in conflict with the cell
  bool operator()(Cell_handle);

  // access functions
  Conflict_vertices_iterator conflict_vertices_begin();
  Conflict_vertices_iterator conflict_vertices_end();
  Hidden_points_iterator hidden_points_begin();
  Hidden_points_iterator hidden_points_end();
};

template <class ConflictTester, class Iter1, class Iter2, class Iter3>
Triple<Iter1,Iter2,Iter3> find_conflicts(
    Cell_handle,
    ConflictTester,
    Triple<Iter1,Iter2,Iter3>);
```

Figure 7: Pseudo-code listing to illustrate the visitor `ConflictTester`.

tion type. This functionality is coded in separate visitor classes.

An example for a visitor class is the conflict tester (Figure 7). The `find_conflicts` function needs to be able to decide whether or not a cell and a point are in conflict. In the Delaunay triangulation, this corresponds to an `in_sphere` test whereas in the regular triangulation, it corresponds to a `power_test` (cf. Section 2). Therefore, the `find_conflicts` function in the space class is templatized by a class of the model `Conflict_tester` and receives an object of this class. We only need to implement a class `Delaunay_conflict_tester` and a class `Regular_conflict_tester` instead of implementing the whole `find_conflicts` function twice.

Further visitors are needed:

- `[Delaunay|Regular]_point_hider`: In a regular triangulation it may happen that the insertion of a heavy-weighted point hides lighter-weighted points nearby. The hidden points disappear from the triangulation. But we need to store them in some data structure to reinsert them again in the case that the heavy-weight point that caused their disappearance is removed. The task of the point hider is to manage this data structure. The `Delaunay_point_hider` exists only for consistency reasons and does nothing.
- `[Delaunay|Regular]_point_remover`: The point remover extracts the hidden points from removed cells. Those vertices can be read by the remove function to be reinserted to the triangulation. This visitor is also empty for the Delaunay case.

4 The Periodic Space

In this section, we show how the design presented previously can be used for computing triangulations in the periodic space $\mathbb{T}^3$. We only sketch the basic ideas here. More details about this specific space will be developed in a forthcoming paper.

The periodic space $\mathbb{T}^3$ is represented here as $[0,1]^3$. It can be embedded in $\mathbb{R}^3$ using a rectangular tiling. We denote points in $\mathbb{R}^3$ and their coordinates by $p = (x,y,z)$ and points in $\mathbb{T}^3$ by $q = (u,v,w)$. Any point $q = (u,v,w) \in \mathbb{T}^3$ is mapped onto a regular point lattice in $\mathbb{R}^3$ given by:

$$g(q) := \{(u,v,w) + (i,j,k) \mid i,j,k \in \mathbb{Z}\},$$

In this definition, we map the periodic space into $\mathbb{R}^3$ by repeating it infinitely in all three directions of space.

Definition 4.1. (domain) *The* domain (i,j,k) *is defined as the set of points in* $\mathbb{R}^3$ *with the same constant* (i,j,k) *in the above defined mapping* g.

As mentioned above, a triangulation is defined as

a simplicial complex (cf. Section 3.1). As long as the subdivision we compute is not a simplicial complex, it is not a valid triangulation. As an example, let us subdivide $\mathbb{T}^2$ using one vertex such that the subdivision meets the Delaunay property when we embed it into $\mathbb{R}^2$. It consists of 3 edges and 2 facets (see Figure 8). This subdivision is not a simplicial complex because the vertices of all simplices are equal. The corresponding subdivision of $\mathbb{T}^3$ with one vertex has 7 edges, 12 facets and 6 cells. This subdivision cannot be used as a starting point for an incremental construction of a Delaunay triangulation.

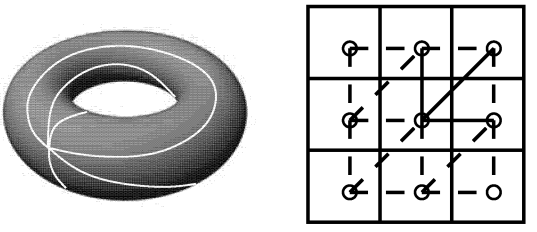


Figure 8: Left: invalid triangulation of $\mathbb{T}^2$ with a single vertex. Right: 9 periodic copies are shown.

Coverings. We propose to construct the triangulation of the torus in two stages. First, we construct a Delaunay triangulation of a 3-sheeted covering of the underlying space [Zom05]. In other words, the domain is explicitly copied 3 times in each direction of space (see Figure 8 for $\mathbb{T}^2$). It can be shown that the subdivision computed as described in Sections 2.1 and 3.4 is always a valid triangulation in such a covering. However, for the three-dimensional space we now have to cope with 27 times as many points as the triangulation contains. It is clear that this becomes very inefficient when computing triangulations of large point sets. Once the triangulation contains enough points, it is likely that it is a simplicial complex in the 1-sheeted covering as well. We could prove that after checking some precise criteria, we can fall back to the 1-sheeted covering.

Offsets. As mentioned above, the triangulation is stored as a set of vertices and a set of cells, where a cell contains pointers to its four neighboring cells and to its four incident vertices. In a periodic space, we also need to store whether or not a cell is wrapped around the domain. To do so, we introduce offsets: Each vertex of a cell is endowed with an offset, which is a three-dimensional vector of non-negative[3] integer entries. To get the vertex coordinates of a cell, we take the point coordinates attached to the respective vertex and add the offset multiplied by the domain size (cf. Figure 9). To use the offsets in the implementation we specialize the cell class in the triangulation data structure. Additionally, the space class and the respective visitors need to be modified to handle offsets.

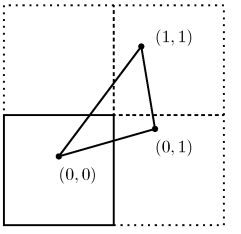


Figure 9: The concept of offsets in 2D space.

5 Benchmarks

The main constraint on our work is that the implementation should not lose efficiency after the redesign of the triangulation package. To ensure this, we tested the performance of both current and new implementations for Delaunay triangulations as well as regular triangulations in $\mathbb{R}^3$.

No. of points	current design	new design
1000	0.0190	0.0190
10000	0.204	0.205
100000	2.11	2.10
1000000	21.5	21.4

No. of points	current design	new design
1000	0.0880	0.0907
10000	1.00	1.01
100000	10.3	10.5
1000000	104	106

Table 1: Benchmarks for Delaunay triangulation (left) and regular triangulation (right).

All benchmarks have been run with the CGAL internal release CGAL-3.4-I-85 on an Intel Pentium 4 CPU clocked at 3.6 GHz. The used operating system is Linux Fedora Core 5 and `gcc` version 4.1.1 with the optimization option `-O2`. The given results are obtained by using `CGAL::Timer` and computing the average of the run time of three runs rounded to three significant digits. All computations have been performed on a random point set, uniformly distributed in a half-open unit cube. The results are given in seconds.

Table 1 allows us to compare the two designs. As we see, no significant loss in efficiency is identifiable. The point insertion uses spatial sorting[4] [Del07], which

[3] We can always move the reference coordinate system such that no negative entries are needed.

[4] The idea is to sort the points so that geometrically close points

accelerates the point location considerably. We can observe that the asymptotic behavior is almost linear.

The last benchmark shows a preliminary comparison of the computation of triangulations in $\mathbb{R}^3$ and in $\mathbb{T}^3$, made possible by the new design. To be able to compare the two triangulations, we precompute a triangulation of the first 1000 points. This is enough for the torus triangulation to switch back to the 1-sheeted covering (cf. Section 4). It does not make much sense to compare with computing in the 3-sheeted covering. Table 2 shows that with the first version of the space class for $\mathbb{T}^3$, the computation is clearly slower than in $\mathbb{R}^3$, but the difference will be reduced after improvements in the implementation.

No. of points	$\mathbb{R}^3$	$\mathbb{T}^3$
1000	0.0190	0.0500
10000	0.204	0.498
100000	2.11	4.93
1000000	21.5	49.5

Table 2: Benchmarks for Delaunay triangulation in $\mathbb{R}^3$ and $\mathbb{T}^3$.

References

[Aur87] F. Aurenhammer. Power diagrams: properties, algorithms and applications. *SIAM J. Comput.*, 16:78–96, 1987.

[BKSV00] Hervé Brönnimann, Lutz Kettner, Stefan Schirra, and Remco Veltkamp. *Applications of the Generic Programming Paradigm in the Design of* CGAL, volume 1766 of *Lecture Notes in Computer Science*, pages 206–216. Springer, Berlin, Germany, January 2000.

[BY98] Jean-Daniel Boissonnat and Mariette Yvinec. *Algorithmic Geometry*. Cambridge University Press, UK, 1998. Translated by Hervé Brönnimann.

[cga] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[Del07] Christophe Delage. Spatial sorting. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.3 edition, 2007.

[Dev02] Olivier Devillers. The Delaunay hierarchy. *Internat. J. Found. Comput. Sci.*, 13:163–180, 2002.

[DPT02] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. *Internat. J. Found. Comput. Sci.*, 13:181–199, 2002.

will be close in the insertion order with high probability.

[DT03] Olivier Devillers and Monique Teillaud. Perturbations and vertex removal in a 3D Delaunay triangulation. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 313–319, 2003.

[DT06] Olivier Devillers and Monique Teillaud. Perturbations and vertex removal in Delaunay and regular 3D triangulations. Research Report 5968, INRIA, Sophia-Antipolis, 2006.

[ES96] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15:223–241, 1996.

[FT06] Efi Fogel and Monique Teillaud. Generic programming and the CGAL library. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*. Springer-Verlag, Mathematics and Visualization, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[Hel01] M. Held. Vroni: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments. *Comput. Geom. Theory Appl.*, 18:95–123, 2001.

[PT07a] Sylvain Pion and Monique Teillaud. 3d triangulation data structure. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.3 edition, 2007.

[PT07b] Sylvain Pion and Monique Teillaud. 3d triangulations. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.3 edition, 2007.

[qhu] Qhull. http://www.qhull.org.

[RV06] Günter Rote and Gert Vegter. Computational topology: An introduction. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*. Springer-Verlag, Mathematics and Visualization, 2006.

[She96] J. R. Shewchuk. Triangle: Engineering a 2d quality mesh generator and Delaunay triangulator. In *First Workshop on Applied Computational Geometry*. Association for Computing Machinery, May 1996.

[WFZH07] Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin. Advanced programming techniques applied to CGAL's arrangement package. *Computational Geometry: Theory and Applications*, 38:37–63, 2007. Special issue on CGAL.

[Zom05] Afra Zomorodian. *Topology for Computing*. Cambridge University Press, Cambridge, 2005.

Consensus Clustering Algorithms: Comparison and Refinement

Andrey Goder* Vladimir Filkov†

Computer Science Department
University of California
Davis, CA 95616

Abstract

Consensus clustering is the problem of reconciling clustering information about the same data set coming from different sources or from different runs of the same algorithm. Cast as an optimization problem, consensus clustering is known as *median partition*, and has been shown to be NP-complete. A number of heuristics have been proposed as approximate solutions, some with performance guarantees. In practice, the problem is apparently easy to approximate, but guidance is necessary as to which heuristic to use depending on the number of elements and clusterings given. We have implemented a number of heuristics for the consensus clustering problem, and here we compare their performance, independent of data size, in terms of efficacy and efficiency, on both simulated and real data sets. We find that based on the underlying algorithms and their behavior in practice the heuristics can be categorized into two distinct groups, with ramification as to which one to use in a given situation, and that a hybrid solution is the best bet in general. We have also developed a refined consensus clustering heuristic for the occasions when the given clusterings may be too disparate, and their consensus may not be representative of any one of them, and we show that in practice the refined consensus clusterings can be much superior to the general consensus clustering.

1 Introduction

Clustering is a very useful technique for mining large data sets since it organizes the data, based on similarity, into smaller groups which are easier to handle in a downstream analysis. Often, different clusterings of the same data can be obtained either from different experimental sources or from multiple runs of non-deterministic clustering algorithms. *Consensus clustering* formalizes the idea that combining different clusterings into a single representative, or consensus, would emphasize the common organization in the different data sets and reveal the significant differences between them. The goal of consensus clustering is to find a consensus which would be representative of the given clusterings of the same data set.

Multiple clusterings of the same data arise in many situations; here we mention two classes of instances. The first class is when different attributes of large data sets yield different clusterings of the entities. For example, high-throughput experiments in molecular biology and genetics often yield data, like gene expression and protein-protein interactions, which provide a lot of useful information about different aspects of the same entities, in this case genes or proteins. Specifically, in gene expression data different experimental conditions can be used as attributes and they can yield different clusterings of the genes. In addition to the individual value of each experiment, combining the data across multiple experiments could potentially reveal different aspects of the genomic system and even its systemic properties [1]. One useful way of combining the data from different experiments is to aggregate their clusterings into a consensus or representative clustering which may both increase the confidence in the common features in all the data sets but also tease out the important differences among them [2, 3].

A second class of instances results from situations where multiple runs of the same non-deterministic clustering or data mining algorithms yield multiple clusterings of the same entities. Non-deterministic clustering algorithms, e.g. K-means, are sensitive to the choice of the initial seed clusters; running K-means with different seeds may yield very different results. This is in fact desirable when the data is non-linearly separable, as the multiple *weak* clusterings could then be combined into a stronger one [4]. To address this, it is becoming more and more common to analyze jointly the resulting clusterings from a number of K-means runs, seeded with different initial centers. One way to aggregate all those clusterings is to compute a consensus among them, which would be more robust to the initial conditions.

*andrey.goder@gmail.com
†filkov@cs.ucdavis.edu

1.1 Consensus Clustering Formalization: the Median Partition Problem The version of the consensus clustering problem in which we are interested is the following: given a number of clusterings of some set of elements, find a clustering of those elements that is as close as possible to all the given clusterings. To formalize this problem we need two things: a representation of a clustering of a set, and a measure $d(,)$ of distance between clusterings.

We identify a clustering of a set A, $|A| = n$, with a set partition π of A, such that if two elements a, b are clustered together in the clustering of A then they are in the same cluster, or block, in π.

A natural class of distances between set partitions can be defined by counting the clustering agreements between them for each pair of elements in the set. Given two set-partitions π_1 and π_2 let a equal the number of pairs of elements co-clustered in both partitions, b equal the number of pairs of elements co-clustered in π_1, but not in π_2, c equal the number of pairs co-clustered in π_2, but not in π_1, and d the number of pairs not co-clustered in either partition (in other words a and d count the number of clustering agreements in both partitions, while b and c count the disagreements). As our distance measure we chose the *symmetric difference distance* (sdd), defined as

$$d(\pi_1, \pi_2) = b + c = \binom{n}{2} - (a + d).$$

This distance measure is a metric and has a nice property that it is computable in linear time (with respect to the number of elements n) [5, 2], which is one of the reasons why we chose it. Another reason is that it can be updated very fast following one element moves in set partitions, as related to the local search heuristics [2], which are described later.

It is of note also that the sdd is related to the popular *Rand Index*, which is defined as $R = (a+d)/\binom{n}{2}$. The sdd and the Rand are not corrected for chance, i.e. the distance between two independent set partitions is non-zero on the average, and is dependent on n. A version corrected for chance, known as the *adjusted Rand Index*, as well as other pair-counting measures (e.g. *Jaccard Index*) exist [6, 7]. The adjusted Rand and the Jaccard indexes are given by complex formulas, and although they can also be computed in linear time, we are not aware of fast update schemes for them. We will use the adjusted Rand in Sec. 5 where the algorithms' complexity is not an issue. In addition, several of the consensus clustering heuristics we use run efficiently only with the symmetric difference distance.

Now we can state the formal version of consensus clustering, known as the *median partition* (MP) problem. Given a set of k set partitions, $\{\pi_1, ..., \pi_k\}$, and $d(,)$ the symmetric difference distance metric, we want to find a set partition π^* such that

$$\pi^* = \operatorname*{argmin}_{\pi} \sum_{i=1}^{k} d(\pi_i, \pi). \tag{1.1}$$

The median partition problem with the symmetric difference distance is known to be NP-complete [8]. For more on the history of the problem and variants see [2].

1.2 Prior work A number of algorithms have been published for approximating the median partition problem [3, 9]. Various theoretical results, including performance guarantees have been derived, although the gap between the performance of these heuristics and their upper bounds is wide [3, 9, 10].

The existing algorithms can be naturally grouped into three groups: the first consists of only one algorithm, the simplest non-trivial heuristic for MP: choose one of the input partitions as the consensus. The second group is comprised of clustering-based algorithms [10], and the third of local-search heuristics [3]. Ailon et al. [10] and Filkov and Skiena [2] provide some theoretical upper bounds as well as performance comparisons for some of those heuristics. Bertolacci and Wirth [9] give a more extensive comparison, while only focusing on clustering-based algorithms and using sampling of the data for better performance.

1.3 Our Contributions In this paper we compare a comprehensive collection of existing median partition heuristics and provide definitive comparisons of their performance in variety of situations. In particular, our contributions are:

- we have written a C++ library for set partitions, supporting many common operations. We have implemented six heuristics that find approximate solutions to the median partition problem: Best Of K, Best One Element Moves, Majority Rule, Average Link, CC Pivot, and Simulated Annealing, with several variations of each. The code is publicly available from *http://www.cs.ucdavis.edu/∼filkov/software/conpas*.
- We provide a statistical approach to evaluating the algorithms' performance by standardizing the distribution of the sum of distances between given partitions and a consensus.
- We assess the efficacy of the six algorithms on both real and simulated data, and find that clustering-based algorithms are generally faster while local-

search heuristics give generally better results. A hybrid algorithm between the two is the best bet in practice.

- Finally, we provide a new approach of refined consensus clustering, that resolves the key problem of determining whether the computed consensus is a good one. A general problem with finding the median partition is that the given partitions may be dissimilar enough that it does not make sense to find a median. In such cases particular subsets of the partitions may in fact be similar, but a challenge is to find subsets which should be grouped together, and to provide medians for just those subsets. We take a statistical approach to this problem, using significance testing to determine when partitions are similar enough to be grouped together.

This paper is organized as follows. In the next section we describe the algorithms and the theory necessary for their evaluation. In Section 3 we present the data and the experimental methodology we employed to compare the algorithms. The results and discussion are given in Section 4. In Section 5 we report on the refined consensus clustering problem, and we conclude in Section 6.

2 Theory and Algorithms

Let S denote the sum of distances $\sum_{i=1}^{k} d(\pi_i, \pi)$ between a proposed consensus π and a set of k given partitions $\{\pi_1, \ldots, \pi_k\}$, where $d(,)$ is the symmetric difference distance. Each of the algorithms we implemented finds a consensus partition which is an approximation to π^*, the set partition that minimizes S. To compare the performance of the algorithms on sets of different number of elements n and number of partitions k, in this section we show how to normalize S in order to make it independent of n and k. In addition, we restate a lower-bound on the minimum sum of distances, which has been derived and reported previously [9].

2.1 Normalization of the Sum of Distances The idea is to relativize S with respect to purely random choices for the initial set partitions and the consensus. To correct for the number of set partitions, k, we simply divide S by it, since we are assuming that random set partitions are independent. The correction for n is more subtle, and involves standardizing S by transforming its distribution into a normal distribution. We define the Normalized distance, d_n, as a z-score

$$d_n(\pi_1, \pi_2) = \frac{d(\pi_1, \pi_2) - \mathbf{E}(D_n)}{\sqrt{\mathbf{Var}(D_n)}}. \tag{2.2}$$

To calculate d_n we give approximations for the expectation and variance of $d(,)$, as follows.

First, we calculate the probability that the co-clustering of two elements i and j is in disagreement in two different partitions (i.e. i and j are co-clustered in one partition and not co-clustered in the other, or vice versa). Let π_1 and π_2 be two set partitions of size n chosen uniformly at random from the B_n partitions of n elements, where B_n is the n-th Bell number (B_n can be computed in a number of ways, e.g. using the recurrence $B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k$, where $B_0 = 1$ [11]). We define the Bernoulli random variable $X_{ij}^{(m)}$ as 1 if i and j are co-clustered in π_m and 0 otherwise. First we give the distribution of $X_{ij}^{(m)}$.

Lemma 2.1.

$$\mathbf{P}(X_{ij}^{(m)} = 1) = \frac{B_{n-1}}{B_n}. \tag{2.3}$$

Proof. There are B_{n-1} ways to cluster the $n-1$ elements other than j. Then there is exactly one way to place j in the same cluster as i. Thus there are B_{n-1} set partitions where i and j are co-clustered.

Now let $Z_{ij} = |X_{ij}^{(1)} - X_{ij}^{(2)}|$. Z_{ij} is 1 if the clustering of i and j disagrees between π_1 and π_2, and 0 otherwise. Then, by definition of the symmetric difference distance,

$$D_n = d(\pi_1, \pi_2) = \sum_{i<j} Z_{ij}. \tag{2.4}$$

We observe that the Z_{ij} are identically distributed Bernoulli random variables. Then we have that

Lemma 2.2.

$$\mathbf{E}(Z_{ij}) = 2\frac{B_{n-1}}{B_n}\left(1 - \frac{B_{n-1}}{B_n}\right), \tag{2.5}$$

and

$$\mathbf{E}(D_n) = 2\binom{n}{2}\frac{B_{n-1}}{B_n}\left(1 - \frac{B_{n-1}}{B_n}\right). \tag{2.6}$$

Proof. Z_{ij} equals 1 exactly when one of the $X_{ij}^{(m)}$ is 1 and the other 0. Thus,

$$\begin{aligned} \mathbf{E}(Z_{ij}) &= \mathbf{P}(Z_{ij} = 1) \\ &= 2\mathbf{P}(X_{ij}^{(m)} = 1)(1 - \mathbf{P}(X_{ij}^{(m)} = 1)), \end{aligned}$$

in which substituting (2.3) yields (2.5). To show (2.6) we take the expected value of (2.4) and substitute (2.5).

While the Z_{ij}'s are not pairwise independent in general (for all i and j), most of the pairs are. Therefore this suggests that D_n should have a normal distribution for large enough n. (We confirmed that this approximate normality holds for all our data sets by running an Anderson-Darling test [12] to test for normality, and we found that D_n is normal with p-value (significance) < 0.01 for $n \geq 50$).

A normally distributed variable that is a sum of n Bernoulli random variables has a variance that is approximately $np(1-p)$. This suggests that

$$\mathbf{Var}(D_n) \approx \left(1 + \frac{2B_{n-1}^2}{B_n^2} - \frac{2B_{n-1}}{B_n}\right)\mathbf{E}(D_n).$$

While these values are slow to compute for large n using recurrences for B_n, we can use approximation formulas for the ratio B_n/B_{n+1}, like the following asymptotic one, derived from an asymptotic formula for B_n given in [13].

$$\frac{B_n}{B_{n+1}} \to \frac{\log n}{n} \text{ as } n \to \infty.$$

Applying this approximation above we find, as $n \to \infty$,

$$\mathbf{E}(D_n) \to \frac{n\log(n-1)}{n-1}\left(n - \log(n-1) - 1\right), \text{and}$$

$$\begin{aligned}\mathbf{Var}(D_n) &\to \mathbf{E}(D_n) \\ &+ \mathbf{E}(D_n)\frac{2\log(n-1)(1-n+\log(n-1))}{(n-1)^2}.\end{aligned}$$

As d_n is approximately normally distributed, then so is the sum

$$S_n = \sum_{i=1}^{k} d_n(\pi_i, \pi),$$

as well as the quantity S_n/k. S_n/k has the nice property that it is a z-score and can be compared to other z-scores and to tables of z-scores to find out the significance of such a score compared to a random event. It is this last quantity, S_n/k, that we will use when comparing the algorithms' performance over sets of data of different number of elements n. We refer to it as the *Average Normalized Sum of Distances*. When the number of elements is the same we use a simpler quantity, the *Average Sum of Distances*, where

$$Avg.\ SOD = \frac{S}{k \cdot \binom{n}{2}}.$$

2.2 Theoretical Lower Bound Since the median partition problem is NP-complete, it is useful to have performance bounds on the heuristics. While upper bounds of the heuristics' performance exist [3, 10] and are useful, lower bounds on the optimal solution for a given input are more important in judging how well any heuristic does in practice. A good lower bound is given in [9], and we restate it here. Let $\{\pi_1, ..., \pi_k\}$ be a set of set partitions, and let $X_{ij}^{(m)}$ be defined as in Sec. 2.1. Then for any set partition π, the sum of distances from it to the k given ones is at least

$$\sum_{i<j} min\left(\sum_{p=1}^{k} X_{ij}^{(p)},\ k - \sum_{p=1}^{k} X_{ij}^{(p)}\right). \tag{2.7}$$

We use this lower bound to estimate the worst-case distance between our solutions and the optimal solution.

2.3 Algorithms All of the algorithms we implemented and tested produce a consensus set partition from an input set of k set partitions $\{\pi_1, ...\pi_k\}$. Recall that the optimal consensus is the set partition which minimizes S.

- **Best Of K (BOK)** In this algorithm we compute S using each input partition π_i as the consensus, in turn. Then we choose that π_i that gives the minimum value for S. It can be shown that this algorithm is a 2-approximation and has running time $O(k^2 n)$ [2].

- **Clustering-based Algorithms** For these three algorithms we first compute a distance matrix $M = (m_{ij})$, where m_{ij} is the proportion of the k input partitions that have i and j clustered apart. Then we choose a cutoff value, τ. This preprocessing step takes $O(n^2 k)$ time.

  In the **Majority Rule (MR)** algorithm (also called quota rule [3]) we begin with every element clustered separately. Then for every pair (i, j) where $m_{ij} < \tau$, we join together the blocks containing i and j. The running time of this algorithm is dominated by the preprocessing step.

  In **Average Link (AL)**, which is the standard average-link agglomerative clustering algorithm, we likewise begin with every element clustered separately. Then we find the two blocks with the smallest average distance, computed using M, and merge them together. We continue doing this until the two closest blocks have a distance $\geq \tau$. This algorithm can be implemented to run in $O(n^2(k + \log n))$, including the preprocessing.

In the **CC Pivot algorithm** [10] we repeatedly choose a pivot element p uniformly at random from the unclustered elements. Then we form a block containing p and every element i where $m_{pi} < \tau$. We continue recursively on the remaining elements, until everything has been clustered. Ailon *et al.* [10] show that this gives an expected $\frac{11}{7}$-approximation algorithm. The running time of this algorithm is dominated by the preprocessing.

- **Local Search-based Heuristics** Given a set partition π, a one element move is any transformation $\pi \mapsto \pi'$ that moves some element a from one block into another one. For example, $\{\{1,2,4,6\},\{3,4\}\}$ is transformed into $\{\{2,4,6\},\{1,3,4\}\}$ with a one element move that sends element 1 from the first block into the second. Given that we allow moves into the "empty" block, we can reach any set partition from any other one with some sequence of at most $n-1$ one element moves.

  Thus, we can begin with some initial candidate partition π and apply one element moves to π to improve it as a consensus. Filkov and Skiena [3] give an efficient way to compute the change in S that results from a one element move, allowing for the two following algorithms to be implemented with $O(n^2k)$ preprocessing time and $O(n)$ time for each one element move.

  In the **Best One Element Moves (BOEM)** algorithm we start with an initial consensus partition–we test starting with the results of BOK or AL. Then we repeatedly perform the best one element move as long as it decreases S.

  In the **Simulated Annealing (SA)** algorithm we use simulated annealing with one element moves. Thus at any given step we always accept any move that decreases S and those that increase S with some decreasing probability. The actual running time depends on the cooling schedule of the SA.

3 Implementation, Data Sets, and Testing Methodology

We have written a C++ library for set partitions, supporting many common operations. Our library has procedures for enumerating all set partitions and selecting set partitions uniformly at random, using algorithms from [14]. It also generates the noisy set partitions described in Sec. 3.1. We have implemented six heuristics that find approximate solutions to the median partition problem: Best Of K, Best One Element Moves, Majority Rule, Average Link, CC Pivot, and Simulated Annealing, with several variations of each. The code is available from *http://www.cs.ucdavis.edu/∼filkov/software/conpas*.

		Mushrooms $n = 8124, k = 22$ Lower Bound: 0.3812	
	Heuristic	Avg. SOD	Time/s
	BOK	0.4068	**1**
Clustering	MR ($\tau = 0.5$)	0.5066	22
	MR ($\tau = 0.25$)	0.3963	21
	AL ($\tau = 0.5$)	0.3940	3727
	AL ($\tau = 0.25$)	0.4117	3641
	CCPivot ($\tau = 0.5$)	0.3993	1222
	CCPivot ($\tau = 0.25$)	0.4366	1210
Local Search	BOEM (BOK seed)	0.3992	21
	BOEM (AL seed)	**0.3919**	3899
	SAFast (BOK seed)	0.4913	47
	SAMedium (BOK seed)	0.4065	542
	SASlow (BOK seed)	**0.3919**	5780
	SASlow (AL seed)	0.3929	9428

Table 1: Results and running times for the heuristics on the Mushrooms data set

In our implementation, we tested the clustering algorithms for two different thresholds, $\tau = 0.5$ and $\tau = 0.25$. For the local search algorithms, we choose three different cooling schedules for SA, which we call SAFast, SAMedium, and SASlow, each starting with the results of BOK. We also test two hybrid algorithms: SASlow and BOEM starting with the consensus partition produced by Average Link. All tests were done an AMD Opteron 2.6 GHz processor with 4 GB of RAM.

3.1 Data Sets In testing the above algorithms we considered both artificial and real data. The real data includes the Mushrooms data set from the UCI repository [15], which has 8124 mushrooms and 22 attributes for each mushroom, such as cap color, stalk shape, habitat, etc. We form one cluster for each attribute, where every mushroom with a particular value for that attribute is clustered together. Any mushrooms with missing values were also clustered together. The other dataset used was the yeast stress experiment [1], which has the responses of 6152 yeast genes to 170 different stress conditions. We use KNNimpute [16] to fill in missing values in the data. Then we clustered it in two ways: by experiment and by gene, using standard agglomerative average-link clustering with a cutoff of 0.5.

To describe the artificial data, we first introduce the concept of noise. Given a set partition π of n elements, we can introduce some percentage of noise x into π by performing xn random one element moves on π. We

		Yeast (by Gene) $n = 170, k = 6152$ Lower Bound: 0.4015	
	Heuristic	Avg. SOD	Time/s
	BOK	0.4217	826
Clustering	MR ($\tau = 0.5$)	0.4773	3
	MR ($\tau = 0.25$)	0.4434	**2**
	AL ($\tau = 0.5$)	0.4102	24
	AL ($\tau = 0.25$)	0.4442	11
	CCPivot ($\tau = 0.5$)	0.4094	119
	CCPivot ($\tau = 0.25$)	0.4438	121
Local Search	BOEM (BOK seed)	0.4119	834
	BOEM (AL seed)	0.4086	25
	SAFast (BOK seed)	0.4085	844
	SAMedium (BOK seed)	0.4083	850
	SASlow (BOK seed)	**0.4082**	954
	SASlow (AL seed)	0.4083	136

Table 2: Results and running times for the heuristics on the yeast dataset, clustered by gene

		Yeast (Experiment) $n = 6152, k = 170$ Lower Bound: 0.4031	
	Heuristic	Avg. SOD	Time/s
	BOK	0.4314	**20**
Clustering	MR ($\tau = 0.5$)	0.5326	87
	MR ($\tau = 0.25$)	0.4331	85
	AL ($\tau = 0.5$)	0.4154	1778
	AL ($\tau = 0.25$)	0.4341	1692
	CCPivot ($\tau = 0.5$)	0.4169	4824
	CCPivot ($\tau = 0.25$)	0.4341	4908
Local Search	BOEM (BOK seed)	0.4256	149
	BOEM (AL seed)	**0.4118**	1832
	SAFast (BOK seed)	0.4335	160
	SAMedium (BOK seed)	0.4302	572
	SASlow (BOK seed)	0.4122	4850
	SASlow (AL seed)	0.4122	6540

Table 3: Results and running times for the heuristics on the yeast dataset, clustered by experiment

choose an element uniformly at random and a block to move it to uniformly at random, excluding its own block but including the "empty" block. Thus, for example, if $n = 100$ at 10% noise we would perform 10 such moves.

Our simulated data consists of generating k noisy partitions of n elements, with x noise, all based upon the same randomly chosen partition. We generated two artificial noise studies to test our algorithms. In the first, we fixed k and x and varied n from 50 to 500, and in the second we fixed n and x and varied k from 25 to 150.

4 Results and Discussion

The results for the real data sets are given in Tables 1, 2, and 3. The values given are the Avg. SODs: sums of Rand distances, averaged over 100 experiments, and averaged by the number of set partitions, k and all pairs, $\binom{n}{2}$. The best values in each column are shown in bold. The running time is given in seconds. For each data set we also provide a lower bound on Avg. SOD, as calculated by the formula in Sec. 2.2. For the three data sets the best scoring algorithms, BOEM/AL (twice) and SASlow get, respectively, to within 2.8%, 1.7%, and 2.1%, of the lower bound, and hence of the optimum.

Fig. 1 and Fig. 2 show the results of the noise study for the 4 best algorithms. We ran each algorithm 100 times on random data and averaged the results. Shown on the y-axis is the Average Normalized SOD, i.e. S_n/k averaged over 100 runs. The y-axis values can be also thought of as z-scores, and their magnitude for our algorithms indicates that all our results are very reasonable and are much better than chance.

Fig. 1 compares the performance (in terms of the Average Normalized SOD) of the four top algorithms over data sets of varying sizes. The stability of the results indicate with confidence that the greedy algorithm BOEM, seeded with the result from Average Link performs the best. The apparent non-linear trend of the results is probably due to at least two factors: (1) our asymptotic approximation and normality assumption from Sec. 2.1 fail for smaller values of n, and (2) the algorithms do better for smaller values of n.

Fig. 2 shows that the performance of the algorithms is not dependent on k, the number of given set partitions. Still, it is interesting to examine the variability in the results, and although some algorithms appear to have a smaller variability than others, the differences in the absolute values of the results are too small to claim that with any statistical certainty.

Overall, BOEM/AL was not our slowest algorithm, and often it is not the fastest, but it performed best here and in the above study on the real data, so we recommend it for any general use.

5 Refined Consensus Clustering

It is unlikely that any real large dataset is composed of clusterings that are very similar. Thus a consensus clustering will likely not give very useful information. A challenge is to determine in what situations a consensus clustering is appropriate. To address this problem here we propose to refine the set of clusterings to

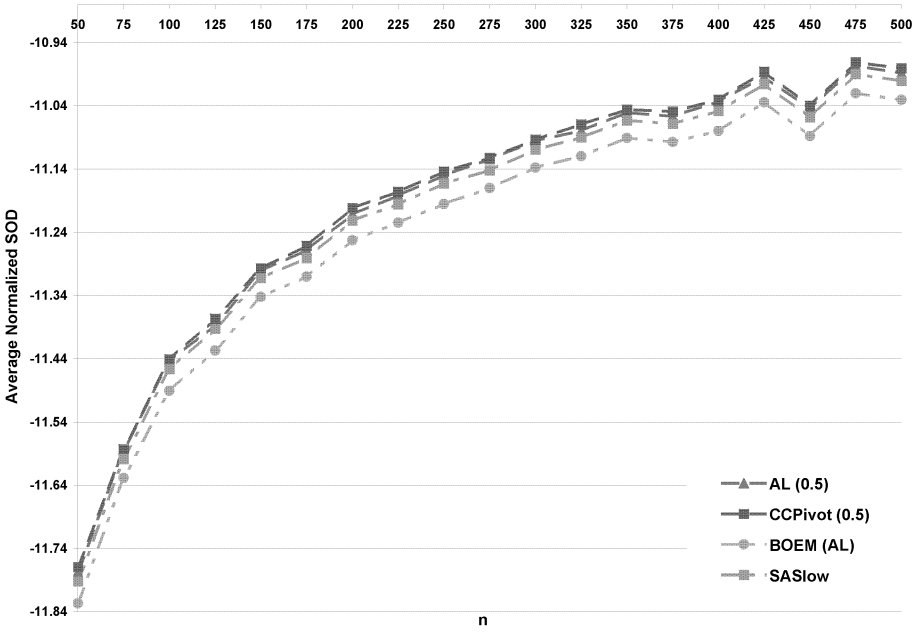


Figure 1: Evaluation of the behavior of the top 4 algorithms with varying n. Here $k = 50$ and the noise is 10%.

smaller sets, each of which will have a better overall consensus. Our approach, *refined consensus clustering*, effectively amounts to clustering the given clusterings and calculating consensuses for the resulting clusters of set partitions. While previous approaches to cluster clusterings exist [17], they are application specific and do not benefit from the full median partition framework that we have built.

We use an agglomerative clustering approach (the Average Link method) to cluster the set partitions. We employ the *adjusted Rand* [6] distance as our dissimilarity measure of choice, because it is a normalized distance, and in this algorithm it does not affect the running time adversely. We begin with all clusterings in separate groups and join together those groups with the smallest average distance. The average is taken over all pairs between the two groups.

In the standard agglomerative clustering method we would continue joining groups until the smallest average distance exceeded some threshold, e.g. 0.5. Instead we present a novel approach, using the idea of the "noise level" of a group of clusterings. Recall from Sec. 3.1 that we generated a noisy set of partitions with a particular noise level. In general, every group of set partitions can be thought of as having been generated around some consensus with a given noise level. We formalize this by taking the sums of the pairwise distances between set partitions, i.e. for some set of partitions $\Pi = \{\pi_1, \pi_2, ..., \pi_k\}$ we define

$$SP(\Pi) = \sum_{i<j} d(\pi_i, \pi_j).$$

Considering all groups of a given noise level, their sum of pairwise distances SP form a distribution. Thus, given any particular group of set partitions, we can calculate a p-value to determine whether it has a given level of noise or more. In practice we do this by sampling the distribution (which depends on n and k) as we do not have a way to compute it analytically.

In our algorithm we have a cutoff noise level, c, and we continue clustering until the next cluster formed would have a noise level of c or higher. This ensures that the resulting clusters will be reasonably similar and that their consensus will be useful.

5.1 Implementation We have implemented this procedure as part of our general set partition library.

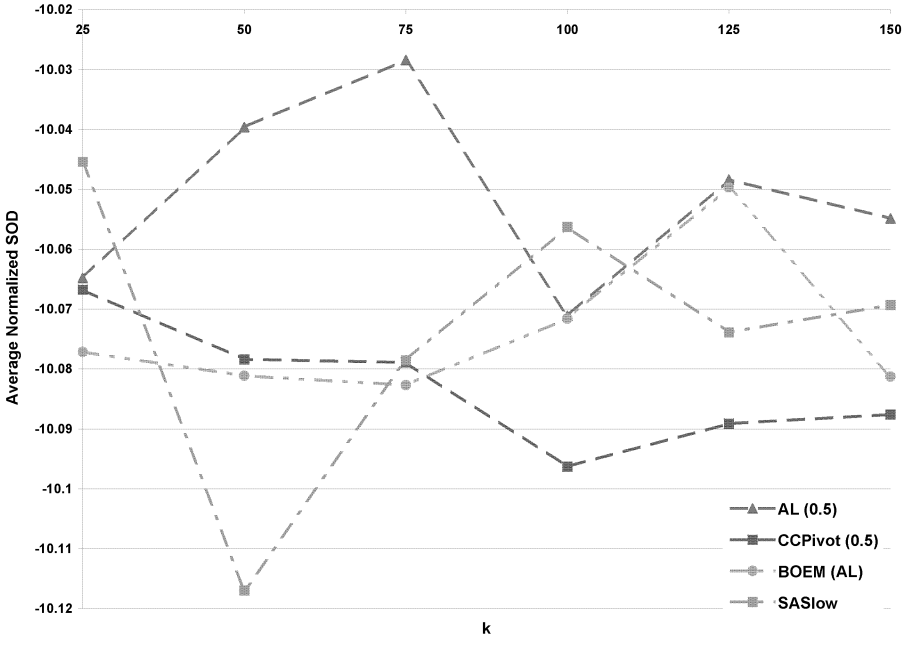


Figure 2: Evaluation of the behavior of the top 4 algorithms with varying k. Here $n = 100$ and the noise is 10%.

It takes as input a list of set partitions and an optional noise threshold and generates as output the clustering of those partitions along with a consensus partition, as calculated by the BOEM/AL algorithm, for each non-trivial cluster (size > 2). The default threshold we use is 0.25, as experimentally we have found it to be roughly the point at which the partitions become mostly random and a consensus is no longer useful. We leave a detailed study of determining this threshold for future work.

5.2 Refined Consensus Results As an example application of our refined clustering method, we ran our algorithm on the Mushroom dataset and the Yeast dataset, both described in Sec. 3.1. The results are found in Tables 4 and 5. For the Mushroom data, the algorithm found two subsets of attributes to be significant at the level of noise of 0.25. For the Yeast data, the algorithm found only one subset of experiments to be significant at the level of noise of 0.25. We found a consensus clustering for each, and computed its Avg. SOD. These clusters have much better consensuses than the general datasets, suggesting that the respective consensuses are more representative.

6 Conclusion and Future Work

The problem of consensus clustering is important because of the prevalence of large data sets and their availability from different sources. A number of heuristics exist for solving a version of this problem called median partition, but no extensive studies have been done comparing their results over a variety of real and artificial data sets.

In this paper we reported on the development of software libraries for solving the median partition problem, performed a comparison of the algorithms, and proposed a refined consensus clustering as a more objective way of finding consensuses. Our results indicate that of the two classes of heuristics that we implemented, the clustering-based methods are generally faster, while the local-search heuristics result in a better performance. We propose a hybrid algorithm, a greedy local search method starting from an initial consensus provided by Average Link Clustering (i.e. BOEM/AL) which has an excellent performance/cost ratio in practice, and most often is the very best performer. Our code is publicly available as a resource for the community, and can be used for various applications.

Subset of Mushroom Attribs.	Avg. SOD	Lower Bound
ring number, gill spacing, veil color, ring type	0.0591	0.0591
population, stalk shape, bruises	0.1551	0.1287

Table 4: Results of refined consensus clustering on the Mushroom dataset

Subset of Experiments	Avg. SOD	Lower Bound
Nitrogen Depletion 12 h 1.5 mM diamide (90 min) Nitrogen Depletion 1 d Nitrogen Depletion 2 d YPD 12 h ypd-2 dtt 015 min dtt-2 constant 0.32 mM H2O2 (40 min) rescan DBY7286 + 0.3 mM H2O2 (20 min)	0.0678	0.0678

Table 5: Results of refined consensus clustering on the Yeast dataset

Several future directions naturally fall out of this study. We are aware that there is at least one other class of algorithms for solving the consensus clustering problem, which use information theoretic metrics [18, 19]. It would be useful to compare those with the combinatorial ones in our library. The refined consensus clustering is still a work in progress. A most immediate improvement would be an automated way to determine the cutoff threshold for the significance. Also, the results at this point are difficult to read out. A visual representation, maybe in terms of some heatmaps of the clusters, would greatly aid their interpretation.

References

[1] A. Gasch, P. Spellman, C. Kao, O. Carmen-Harel, M. Eisen, G. Storz, D. Botstein, and P. Brown. Genomic expression programs in the response of yeast cells to environment changes. *Mol. Bio. Cell*, 11:4241–4257, 2000.

[2] V. Filkov and S. Skiena. Integrating microarray data by consensus clustering. *International Journal on Artificial Intelligence Tools*, 13(4):863–880, 2004.

[3] V. Filkov and S. Skiena. Heterogeneous data integration with the consensus clustering formalism. *Proceedings of Data Integration in the Life Sciences*, pages 110–123, 2004.

[4] Alexander Topchy, Anil K. Jain, and William Punch. Combining multiple weak clusterings. In *ICDM '03: Proceedings of the Third IEEE International Conference on Data Mining*, page 331, Washington, DC, USA, 2003. IEEE Computer Society.

[5] M. Bender, S. Sethia, and S. Skiena. Efficient data structures for maintaining set partitions. *Proceedings of Seventh Scandinavian Workshop on Algorithm Theory*, pages 83–96, 2000.

[6] L. Hubert and P. Arabie. Comparing partitions. *J. Class.*, 2:193–218, 1985.

[7] E.B Fawlkes and C.L. Mallows. A method for comparing two hierarchical clusterings. *Journal of the American Statistical Association*, 78:553–584, 1983.

[8] Y. Wakabayashi. The complexity of computing medians of relations. *Resenhas IME-USP*, 3:323–349, 1998.

[9] M. Bertolacci and A. Wirth. Are approximation algorithms for consensus clustering worthwhile? In *Proceedings of the Seventh SIAM ICDM*, 2007.

[10] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: ranking and clustering. In *Proceedings of the thirty-seventh annual ACM Symposium on Theory of Computing*, pages 684–693, 2005.

[11] H. Wilf. *generatingfunctionology*. A K Peters, 3rd edition, 2006.

[12] H.C. Thode Jr. *Tests for Normalcy*. Marcel Dekker, NY, 2002.

[13] J. Pitman. Some probabilistic aspects of set partitions. *American Mathematical Monthly*, 104:201–209, 1997.

[14] A. Nijenhuis and H. Wilf. *Combinatorial Algorithms*. Academic Press, 1978.

[15] D. Newman, S. Hettich, C. Blake, and C. Merz. UCI repository of machine learning databases. 1998.

[16] O. Troyanskaya et al. Missing value estimation methods for DNA microarrays. *Bioinformatics*, 17:520–525, 2001.

[17] A.D. Gordon and M. Vichi. Partitions of partitions. *Journal of Classification*, 15:265–285, 1998.

[18] D. Cristofor and D. Simovici. Finding median partitions using information-theoretical-based genetic algorithms. *Journal of Universal Computer Science*, 8(2):153–172, 2002.

[19] A. Strehl and J. Ghosh. Cluster ensembles - a knowledge reuse framework for combining partitionings. In *Proc. of AAAI*, pages 93–98. AAAI/MIT Press, 2002.

Shortest Path Feasibility Algorithms: An Experimental Evaluation

Boris V. Cherkassky[1] Loukas Georgiadis[2] Andrew V. Goldberg[3]
Robert E. Tarjan[4] Renato F. Werneck[3]

Abstract

This is an experimental study of algorithms for the shortest path feasibility problem: Given a directed weighted graph, find a negative cycle or present a short proof that none exists. We study previously known and new algorithms. Our testbed is more extensive than those previously used, including both static and incremental problems, as well as worst-case instances. We show that, while no single algorithm dominates, a small subset (including a new algorithm) has very robust performance in practice. Our work advances state of the art in the area.

1 Introduction

The *shortest path feasibility problem* (FP) is to find a negative-length cycle in a given directed, weighted graph, or to present a proof (a set of feasible potentials) that no such cycle exists. This is closely related to the problem of finding shortest path distances in a network. A solution to the shortest path problem is also a solution to FP. Furthermore, given a feasible set of potentials, one can solve the shortest path problem in almost linear time with Dijkstra's algorithm [9]. These are classical network optimization problems with many applications, such as program verification [25] and real-time scheduling [5].

The classical Bellman–Ford–Moore (BFM) algorithm [1, 11, 18] achieves the best known strongly polynomial time bound for FP: $O(nm)$ (where n and m denote the number of vertices and arcs in the graph, respectively). With the additional assumption that arc lengths are integers bounded below by $-N \leq -2$, the $O(\sqrt{n}m \log N)$ bound of Goldberg [14] is an improvement, unless N is very large. The BFM bound can also be improved in the expected sense for many input distributions; see *e.g.* [17].

Previous studies of shortest path algorithms include [2, 3, 8, 13, 19, 20, 21, 26]. The earliest experiments concluded that BFM performs poorly in practice, and favored alternatives such as incremental graph algorithms [20, 21] and the threshold algorithm [8]. These methods have relatively bad worst-case bounds, however, and are not robust in practice. More recently, Cherkassky and Goldberg [2] showed that two $O(nm)$ algorithms, BFCT [23] and GOR [15], are much more robust. These methods, which have been studied further in [19, 26], serve as the basis of our experimental comparison.

Our first contribution is an algorithm design framework that generalizes BFM and is a natural basis for new algorithms.

As our second contribution, we implement several new algorithms and reimplement existing ones, in one case correcting a heuristic used in previous implementations. Our experiments show that a new algorithm, RD (Robust Dijkstra), is not only very robust, but also significantly outperforms previous ones on some problem classes.

Finally, we propose a new set of benchmark instances that is more extensive than existing ones. It includes natural problems, bad-case instances for all algorithms, and experiments modeling incremental scenarios where one solves a sequence of feasibility problems, each a perturbation of the previous one.

This paper is organized as follows. Section 2 presents some definitions and notation. Section 3 reviews the scanning method and gives a general framework for designing $O(n)$-pass algorithms. The negative cycle detection techniques used by the algorithms we implemented are discussed in Section 4. Section 5 reviews existing algorithms used in our study, including their improved implementations, and introduces some new algorithms. Section 6 describes families of instances that elicit the worst-case behavior of these methods. Section 7 describes our experimental setup and presents an empirical comparison of the various algorithms studied.

[1]Central Economics and Mathematics Institute of the Russian Academy of Sciences. E-mail: bcherkassky@gmail.com. Part of this work has been done while this author was visiting Microsoft Research Silicon Valley.

[2]Hewlett-Packard Laboratories, Palo Alto, CA, 94304. E-mail: loukas.georgiadis@hp.com.

[3]Microsoft Research Silicon Valley, Mountain View, CA 94043. E-mail: {goldberg,renatow}@microsoft.com.

[4]Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08540 and Hewlett-Packard Laboratories, Palo Alto, CA, 94304. E-mail: ret@cs.princeton.edu.

Finally, Section 8 presents concluding remarks.

2 Definitions and Notation

The inputs to FP are a directed graph $G = (V, E)$ and a length function $\ell : E \to \mathbf{R}$. A *potential function* maps vertices to values in $\mathbf{R} \cup \{\infty\}$. We refer to these values as *potentials*. Given a potential function d, we define the *reduced cost function* $\ell_d : E \to \mathbf{R} \cup \{\infty\}$ by $\ell_d(v, w) = \ell(v, w) + d(v) - d(w)$. (If $d(v) = d(w) = \infty$, we define $\ell_d(v, w) = \ell(v, w)$.) We refer to arcs with negative reduced cost as *negative arcs*. The potential function is *feasible* if there are no negative arcs.

Note that replacing ℓ with ℓ_d may change the length of the shortest path between two distinct vertices, but the path itself (the sequence of arcs) will remain the same. It is also easy to see that this change (a *potential transformation*) preserves the length of every cycle.

The goal of FP is to find a feasible set of potentials or a negative-length cycle (or simply *negative cycle*) in G. A feasible potential function exists if and only if there is no negative cycle.

We say that an arc a is *admissible* if $\ell_d(a) \leq 0$, and denote the set of admissible arcs by E_d. The *admissible graph* is defined by $G_d = (V, E_d)$.

To deal with the fact that the input graph may not be strongly connected and that some algorithms for FP maintain a tentative shortest path tree, we add a *root vertex* s to V and connect s to all other vertices with zero-length arcs. Further references to G will thus be to this transformed graph. A *shortest path tree* of G is a spanning tree rooted at s such that for any $v \in V$, the s-to-v path in the tree is a shortest path from s to v in G.

We say that $d(v)$ is *exact* if the distance from s to v in G is equal to $d(v)$, and *inexact* otherwise.

3 Scanning Algorithms

This section briefly outlines the general *labeling method* [10] for solving shortest path and feasibility problems, on which all algorithms studied in this paper are based. (See *e.g.* [24] for more details.) We maintain for every vertex v its potential $d(v)$ and parent $p(v)$ (possibly *null*). Typically, all potentials are initially zero, $p(s)$ is set to *null* and the remaining vertices v have $p(v)$ set to s. At every step, we perform a *labeling operation*: pick an arc (u, v) such that $d(u) < \infty$ and $\ell_d(u, v) < 0$ and set $d(v) = d(u) + \ell(u, v)$ and $p(v) = u$. If G has no negative cycle, eventually there will be no negative arcs, at which point d will be feasible. As Section 4 will show, a simple modification ensures that this method also terminates in the presence of a negative cycle.

The *scanning method* is a variant of the labeling method based on the SCAN operation. The method maintains for each vertex v a *status* $S(v) \in \{\text{unreached}, \text{labeled}, \text{scanned}\}$. Given a labeled vertex v, SCAN(v) examines all arcs (v, w), and if $d(v) + \ell(v, w) < d(w)$ then $d(w)$ is set to $d(v) + \ell(v, w)$, $p(w)$ to v, and $S(w)$ to labeled. A labeled vertex becomes scanned when a SCAN operation is applied to it, and a vertex of any status becomes labeled whenever its potential decreases. Scanning algorithms for FP typically start with all potentials set to zero, which requires the initial set of labeled vertices to contain all vertices with outgoing negative arcs.

3.1 $O(n)$-pass algorithms.

As described, the scanning algorithm does not necessarily run in polynomial time. We can, however, define a general class of scanning methods that perform $O(nm)$ labeling operations. This class generalizes BFM and GOR, among other algorithms.

We partition the sequence of vertex scans into *passes* (subsequences of scans) with the following properties: (a) each vertex is scanned at most once during a pass; and (b) each vertex that is already labeled when the pass begins must be scanned during the pass. Note that a vertex that becomes labeled during the pass can also be scanned, but it does not need to be.

LEMMA 3.1. *If there is a shortest path from s to v containing k arcs, then after at most k passes $d(v)$ is exact. Thus in the absence of negative cycles, the algorithm terminates after at most $n - 1$ passes.*

The proof is by induction on the number of passes: one shows that after pass i, every vertex whose shortest path from s has i arcs will have exact potential.

BFM can be seen as an implementation of this general algorithm. It maintains labeled vertices in a queue: the vertex to be scanned next is removed from the front, and newly labeled vertices are inserted into the back. Note that every vertex that is currently labeled will be scanned before any vertex that is not, as required by a pass.

4 Negative Cycle Detection

To ensure that the labeling method terminates in the presence of negative cycles, we must use a *cycle detection strategy*. This section discusses some of them.

Let the *parent graph* G_p be the subgraph of G induced by the arcs $(p(v), v)$, for all v such that $p(v) \neq \textit{null}$. This graph has several useful properties (see *e.g.* [24]). In particular, the arcs in G_p have nonpositive reduced costs and (if G_p is acyclic) form a tree rooted at s. If G_p does have a cycle, it will correspond to a negative cycle in the original graph. Conversely, if G

contains a negative cycle, then G_p will provably contain a cycle after a finite number of labeling operations.

Although a cycle can appear in G_p after one labeling operation and disappear after the next, it can be detected as soon as it appears. Suppose a labeling operation is applied to an arc (u, v) and G_p is acyclic: a cycle will appear in G_p if and only if u is a descendant of v in the current tree. This could be tested by following parent pointers from u, but this takes $\Theta(n)$ worst-case time and does not work well in practice [2].

A more efficient solution is *subtree disassembly*, proposed by Tarjan [23]. When the labeling operation is applied to (u, v), one traverses the entire subtree rooted at v. If u is found, the algorithm terminates with a negative cycle. Otherwise, all vertices visited (except v itself) are removed from the current tree and marked as unreached. They will only be scanned after becoming labeled again. Disassembling a subtree takes linear worst-case time if we maintain a doubly-linked list to represent a preorder traversal of the current tree [16]. Fortunately, this cost can be charged to the work of building the subtree, which makes the amortized cost of each labeling operation constant. In particular, applying subtree disassembly to an $O(n)$-pass algorithm does not change its worst-case complexity. Marking a labeled vertex as unreached may delay its next scan, but this is arguably positive because its potential is known to be inexact. The net effect is usually a decrease in the total number of scans.

A slight variant of this method is *subtree disassembly with updates* [2]: if the potential of v decreases by δ, potentials of all proper descendants w of v can be updated to $d(w) - \delta$. Because the new potentials may be exact, additional bookkeeping is required to make sure these vertices are scanned at least once more.

Here we propose a simpler alternative for the case when arc lengths are integral. Instead of decreasing the potentials by δ, we decrease them by $\delta - 1$, to $d'(w) = d(w) - \delta + 1$. Because the new potentials are not exact, the updated vertices are guaranteed to be scanned again. Moreover, the optimization is still effective: after an update, only paths to w with length equal to $d(w) - \delta$ (or shorter) will cause w to become labeled.

An alternative to methods based on subtree traversal is *admissible graph search* [14], which uses the fact that if $p(w) = v$, then (v, w) is in G_d. Therefore, if G_p contains a cycle, the admissible graph G_d contains a negative cycle. Since the arcs in G_d have nonpositive reduced cost, a negative cycle can be found in $O(n+m)$ time using an augmented depth-first search. As we will see, admissible graph search is a natural strategy for GOR: since it already performs a depth-first search of G_d at each iteration, cycle detection has very little overhead.

5 Algorithms

This section describes the algorithms we tested, including both existing and new ones.

5.1 BFCT. This is the BFM algorithm with subtree disassembly, including updates. Initially, all vertices are labeled and have zero potential. Labeled vertices are maintained in a FIFO queue. A vertex to be scanned is removed from the front of the queue, and newly labeled vertices are inserted at the back. When an arc (u, v) is scanned and the potential of v is reduced, we disassemble and update the subtree rooted at v. Any labeled vertices found in the subtree are deleted from the queue, since they are no longer labeled. BFCT runs in $O(nm)$ worst-case time.[1]

5.2 GOR. Proposed by Goldberg and Radzik [15], GOR is an $O(n)$-pass algorithm that works as follows. Define a *source* as a vertex with outgoing negative arcs. Suppose the admissible graph G_d is acyclic. Intuitively, improvements in potential values will propagate along the arcs of this graph. In each pass, GOR sorts in topological order all sources and the vertices reachable from them in G_d, then scans them in this order.

Because G_d is not necessarily acyclic, we compute its strongly connected components (SCCs). If there is an SCC containing a negative arc, it also contains a negative cycle, and GOR terminates. Otherwise, there is a zero-reduced-cost path between any two vertices within each SCC. Therefore, we can make G_d acyclic by contracting each SCC into a single vertex (until the end of the computation). Because of its performance overhead and implementation complexity, however, graph contraction should be avoided in practice.

The implementation of [2] avoids both the contraction and the SCC computation by running a simple depth-first search (DFS). When a back arc is discovered, it checks if the corresponding cycle is negative. If it is, the algorithm terminates; otherwise, it ignores the back arc and resumes the DFS. If no negative cycle is found, the DFS produces a topological ordering of the graph obtained from G_d when the ignored arcs are deleted. After a careful analysis of this procedure, however, we discovered that there are rare situations in which the DFS may fail to find an existing negative cy-

[1] In [2], the abbreviation "BFCT" refers to an implementation of the algorithm that does no updates, and a more complicated implementation of updates is considered. Our new implementation of updates has essentially no overhead, making the implementation with no updates obsolete.

cle in G_d, which could cause the algorithm to terminate later or not at all (although the latter never happened in the experiments reported in [2, 26]). Note that the problem is not in the original GOR algorithm, but in the way this particular implementation avoids contractions.

We propose and use a new implementation of GOR that corrects this problem while still avoiding contractions. It runs the SCC algorithm of Tarjan [22], which performs a single DFS while maintaining a second stack (in addition to the one used by the DFS) to store SCC-related information. When an SCC is fully processed, its vertices are at the top of the second stack. While the algorithm pops vertices of the SCC from the stack, our implementation of GOR examines their adjacency lists to check if there is a negative arc inside the SCC. If there is, the algorithm terminates with a negative cycle; otherwise, it produces a topological ordering of G_d with back arcs deleted. Vertices will be scanned in that order. The resulting implementation still runs in $O(nm)$ time and is guaranteed to find a negative cycle in G_d if there is one. In our experiments, we observed that this new implementation has an additional overhead of about 20% on graphs without negative cycles.

As a heuristic to speed up the detection of "obvious" cycles, we also check for negative back arcs during the DFS, and stop immediately if one is found.

5.3 RD. We now discuss a new $O(n)$-pass algorithm, which we call *robust Dijkstra* (RD). Unlike previous methods, RD takes potentials into account when deciding which vertex to scan next. It does so by associating to each labeled vertex v a *key*, defined as the (strictly positive) difference between its *previous potential* (the value of $d(v)$ during the last scan of v) and its current potential. If v has not yet been scanned, we use its initial potential as its *previous potential*. Note that the *previous potential* does not change when a vertex is labeled.

The algorithm partitions the labeled vertices into two sets, Q and S: Q contains those that have yet to be scanned in the current pass, and S contains the rest. Set Q is maintained as a priority queue ordered by key values, and S as an ordinary queue. Initially, all vertices are labeled and have zero potential. At each step, we extract a vertex with maximum key from Q and scan it. When a vertex becomes labeled, it is added to S if it has been scanned during the current pass or to Q otherwise.[2] If the potential of a labeled vertex v in Q decreases, its key increases and an increase-key operation is performed on v. When Q becomes empty, a new pass starts by moving vertices from S to Q and heapifying Q. The algorithm uses subtree disassembly with updates, which causes vertices in the affected subtrees to be removed from S or Q.

It may seem more natural to give priority to vertices with the lowest potentials (instead of highest improvement). In practice, however, it does not work well because a potential transformation may affect the vertex selection order in every pass. For example, if for some vertex v we decrease the length of incoming arcs by a large value and increase the length of outgoing arcs by the same amount, v will be scanned immediately after it is added to Q.

As its name suggests, RD is a generalization of Dijkstra's algorithm: on the shortest path problem, these algorithms will scan the same sequence of vertices if arc lengths are nonnegative. (RD initialization is slightly different for the shortest path problem: the source potential is still set to zero, but all others must be set to M, a finite number larger than the length of any possible shortest path.) Note that, if there are negative input arcs, RD cannot use a monotone priority queue, such as multilevel buckets [7]; a general (and potentially less efficient) data structure must be used instead. Using Fibonacci heaps [12], the algorithm runs in $O(n(m + n \log n))$ time. Our implementation, which we call RDH, uses a 4-heap [24] (which is more efficient in practice) and runs in $O(nm \log n)$ time.

In an effort to reduce the data structure overhead, we also implemented RDB, which maintains an approximate bucket-based priority queue. Each bucket consists a doubly-linked list of vertices that behaves as a queue with support for arbitrary deletions.[3] Bucket 0 contains vertices with key 0, and bucket i contains vertices with keys in $[2^{i-1}, 2^i)$. We also maintain an upper bound μ on the index of the biggest-key nonempty bucket. This data structure supports the insert, delete, and increase-key operations in $O(\log \log M)$ time and extract-max in $O(\log M)$ time. This leads to an $O(n(m \log \log M + n \log M))$ time bound for RDB.

5.4 MBFCT. Wong and Tan's MBFCT algorithm [26] is a "local" variant of BFCT. All vertices start the algorithm unreached and with zero potential. The following procedure is executed while there are unreached vertices: Pick an unreached vertex v, mark it labeled, set $p(v) = s$, and run BFCT; terminate if a negative cycle is found, otherwise proceed to the next iteration. The potential function modified by BFCT is maintained throughout MBFCT, which means that only vertices whose potentials improve are visited in each new

[2] A variant that adds all newly labeled vertices to S proved to be less robust in our tests.

[3] We also tested buckets as stacks, with largely similar results.

iteration.

Since each call to BFCT can only reduce the number of unreached vertices, the algorithm terminates after at most n calls. It runs in $O(n^2m)$ worst-case time, and our experiments include a family of instances that match this bound. Although this is not an $O(n)$-pass algorithm, in practice it is often faster than BFCT [26], especially on graphs with several small negative cycles. Unlike the original implementation of MBFCT, ours does updates during subtree disassembly, which sometimes makes it more efficient.

Note that there is some similarity between MBFCT and Pallottino's algorithm [20]. The latter also works "locally" using the Bellman-Ford-Moore algorithm.

Next we sketch Pallottino's algorithm. It partitions vertices into low- and high-priority sets. Initially every vertex has low priority, and priorities can only change from low to high. The algorithm maintains two queues, H and L, which contain labeled vertices of high and low priority, respectively. Initially, L contains all labeled vertices. The algorithm works in phases. At the beginning of each phase H is empty. The algorithm removes a vertex from the head of L, changes its priority to high, and adds it to H. Then the vertices in H are scanned in FIFO order. A scan can add vertices to both H and L, depending on the priority of the labeled vertex. The phase terminates when H becomes empty. (One can show that at this point the current potentials are feasible for the subgraph induced by the set of high-priority vertices.)

Two main differences between MBFCT and Pallottino's algorithm are that MBFCT does not restrict scans to high-priority vertices and uses subtree disassembly (though Pallottino's algorithm can also use subtree disassembly, as shown in [2]). Both algorithms run in $O(n^2m)$ time, however, and our bad-case example for MBFCT, described in Section 6, is also bad for Pallottino's algorithm.

5.5 Other algorithms. We experimented with several other algorithms. This section discusses some that, although natural, ended up being relatively less competitive in practice.

A natural approach is to combine the ideas of topological sort and subtree disassembly. One can either do GOR-like topological sorting at each pass of BFCT, or perform subtree disassembly (which may make mark some labeled vertices as unreached) while running GOR. We tried several variants of these ideas, but failed to get a robust performance improvement on our best implementations.

We also considered an algorithm that has potentially useful theoretical performance guarantees in the incremental context. The *arc-fixing algorithm* (AF) is motivated by the chain-elimination procedure of [14]. To gain intuition, suppose the input graph as only one negative arc (v, w) with $\ell(v, w) = -x$. We can change $\ell(v, w)$ to zero and apply Dijkstra's algorithm to the resulting graph with w as the source. If the distance to v is less than x, the input graph has a negative cycle formed by (v, w) and the computed shortest path from w to v. Otherwise one can use the computed distances to get feasible potentials, "fixing" the arc (v, w).

Next we describe the arc-fixing algorithm. All vertices start with zero potential. Each iteration begins by checking if the admissible graph G_d has a negative cycle. If so, the algorithm terminates. Otherwise, it tries to update the potentials by running two shortest path computations.

Let G' be the graph induced by the arcs out of the root vertex s (with length zero) together with all arcs (v, w) with $\ell_d(v, w) < 0$. This graph is acyclic. In linear time, we compute the distance $\pi(v)$ in G' from s to each vertex v. Note that $\pi(v) \leq 0$.

The algorithm then builds a new graph G'', which has the same topology as G, but different arc lengths: arcs out of s have length $\pi(v)$, and the remaining ones have length $\max\{0, \ell_d(a)\}$. Since all negative arcs of G'' are adjacent of s, Dijkstra's algorithm correctly finds the distances $p(v)$ in G'' from s to each vertex v while scanning each vertex once. Note that $p(v) \leq 0$.

The algorithm completes an iteration by updating $d(v)$ to $d(v) + p(v)$.

One can show that the arc-fixing algorithm has the following properties: (i) once an arc becomes nonnegative, it remains nonnegative; (ii) in each iteration, either there is a vertex that has outgoing negative arcs in the beginning but not at the end, or the next iteration discovers a negative cycle. Therefore, if the number of vertices with outgoing negative arcs in G is k, the algorithm terminates in k iterations. Although this is a promising bound for incremental scenarios, this algorithm was not competitive with the best algorithms in our study.

6 Worst-Case Instances

This section introduces graph families that elicit the worst-case behavior of the algorithms we implemented. We concentrate on the feasibility problem, but we note that simple modifications of these families give worst-case instances for the single-source shortest path problem as well. Within a family, the number of vertices in each network is a linear function of a parameter k, which indicates how many times a given gadget is repeated. The number of arcs is also linear in k for most families, but there are two cases in which it is

quadratic. To the best of our knowledge, these are the first families to match the theoretical worst-case bounds of any of the algorithms we study.

6.1 BFCT. The network BAD-BFCT(k) contains $n = 4k - 1$ vertices and $m = 5k - 3$ arcs. Vertices 1 to $3k - 2$ together with the arcs $(i + 1, i)$, $1 \le i \le 3k - 3$, form a path P. Every third vertex on P is connected to vertex $3k-1$, i.e., we have the arcs $(3(i-1)+1, 3k-1)$ for $1 \le i \le k$. Finally, $3k-1$ is connected to vertices $3k$ to $4k-1$. All arcs in this graph have length -1. Figure 1 gives an example for $k = 4$.

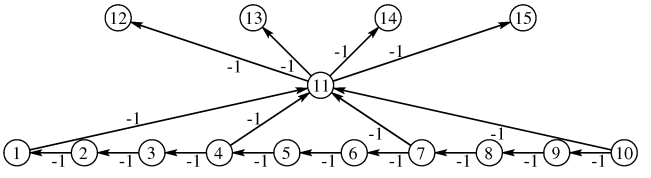


Figure 1: Worst-case input for BFCT ($k = 4$).

Assume that the vertices are initially ordered in the BFCT queue by their identifier, from smallest to largest. During the first pass, the vertices on P are scanned in increasing order. After the first pass, the vertices on P (except the last vertex) are scanned once more, in decreasing order. Because of subtree disassembly, only one vertex on P is in the queue at a time. Consider what happens when vertex $3i$ ($1 \le i < k$) is scanned for the second time. At this point only $3i$ and $3k - 1$ are in the queue. After scanning $3i$, vertex $3i - 1$ is inserted into the queue following $3k - 1$. After scanning $3k-1$, vertices $3k, \ldots, 4k-1$ are inserted into the queue and scanned after $3i - 1$. Scanning $3i - 2$ disassembles the subtree of $3k - 1$ and produces a similar state as before scanning $3i$. The same pattern continues for $j = i - 1, \ldots, 1$. This implies a total of $\Theta(k^2) = \Theta(n^2)$ scans.

We note that by adding the arcs $(3i - 1, 3k - 1)$, the subtree of $3k - 1$ is disassembled before scanning vertices $3k$ to $4k - 1$, which results to $O(1)$ scans per vertex. It is also interesting to note that without subtree disassembly the subgraph induced by P alone is a worst-case instance.

6.2 MBFCT. With a similar network we can construct a worst-case instance for MBFCT. To obtain BAD-MBFCT(k) from BAD-BFCT(k) we first reverse the orientation of the path P. Let $G(k)$ be the resulting graph. Then we add new vertices $4k, \ldots, 6k-1$, which are connected to the extremes of P as follows: the even vertices are connected to vertex 1, and the odd vertices are connected to $3k - 2$. The length of the arc leaving vertex $4k + i$ is $-4k(i + 2)$, $0 \le i \le 2k - 1$. Figure 2 gives an example for $k = 4$. Note that $n = 6k-1$ and $m = 7k-3$.

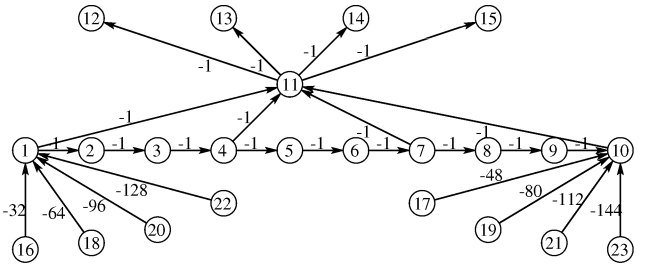


Figure 2: Worst-case input for MBFCT ($k = 4$).

After vertex $4k+2i$ is scanned, MBFCT needs $\Theta(k^2)$ scans to process $G(k)$. The role of each vertex $4k+2i+1$ is to disassemble the subtree rooted at $3k - 1$, so that the next pass over $G(k)$ will still require $\Theta(k^2)$ scans. This gives a total of $\Theta(k^3) = \Theta(n^3)$ scans.

We note that, if we connect an artificial source to all original vertices with arcs of length zero, then we obtain a worst-case ($\Theta(n^3)$ scans) instance for Pallottino's shortest paths algorithm [20].

6.3 GOR. The worst-case network BAD-GOR(k) consists of a path P of k vertices $1, \ldots, k$, a vertex $k+1$ with k incoming and k outgoing arcs, and k vertices $k+2, \ldots, 2k+1$. Set $\ell(1, 2) = -3k$, $\ell(1, k+1) = -1$, and $\ell(i, i+1) = 1$ for $2 \le i \le k-1$. Also, $\ell(k+1, k+1+i) = -1$ for $1 \le i \le k$, and $\ell(i, k+1) = 2(k-i)$ for $2 \le i \le k$. We have $n = 2k + 1$ and $m = 3k - 1$. Figure 3 gives an example for $k = 7$.

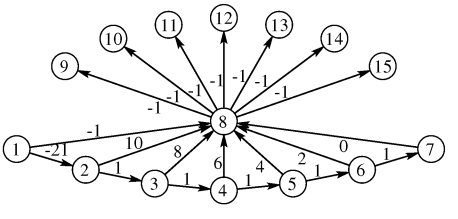


Figure 3: Worst-case input for GOR ($k = 7$).

Note that GOR needs $\Theta(k)$ passes to process P: Initially the only admissible arc on P is $(1, 2)$, and scanning vertex 2 makes $(2, 3)$ admissible. These are the only arcs on P that are considered during the first pass of the algorithm. Each subsequent pass makes two more arcs on P admissible. The arcs leaving vertex $k+1$ remain admissible in each pass, because the length of the path $(1, 2, .., i, k + 1)$ is smaller than the length of

$(1, 2, .., i, i+1, k+1)$ for $i = 2, \ldots, k-1$. This causes GOR to scan vertices $k+1, \ldots, 2k+1$ in each pass. It follows that the total number of scans is $\Theta(k^2) = \Theta(n^2)$.

6.4 RD. The elaborate rule that RD employs for selecting the next vertex to scan (combined with subtree disassembly with updates) makes it hard to construct worst-case instances. We can simplify the problem slightly by making additional assumptions on how the priority queue breaks ties. This is the case in our implementation of RDB, since each bucket is implemented as a queue. But in a heap-based implementation of RDH it is hard to predict which element among the ones with maximum key will be returned. Our implementations of RDB and RDH do exhibit quadratic running time for the worst-case families we created, even though the latter does not use any special tie-breaking rule. The reader should be aware, however, that other implementations may behave differently.

For an intuition on the construction of the worst-case networks, consider the graph $G(k)$ shown in Figure 4. It consists of k arcs $(x_i, y_i) = (2i-1, 2i)$ for $1 \leq i \leq k$, together with the arcs (x_i, x_{i+1}) and (y_i, x_{i+1}) for $1 \leq i < k$. We require each path (x_i, y_i, x_{i+1}) to be shorter than (x_i, x_{i+1}). To that end we can set $\ell(x_i, y_i) = 0$, $\ell(y_i, x_{i+1}) = -2$ and $\ell(x_i, x_{i+1}) = -1$. In the first pass of RD, vertex x_{i+1} has priority over y_i; hence all the x_i's will be scanned first, and then all the y_i's. In the following passes, when x_i is scanned y_i and x_{i+1} get the same priority, equal to the distance improvement for x_i. Suppose that (x_i, x_{i+1}) is processed before (x_i, y_i). Then, assuming that the priority queue of RD favors vertices that are inserted more recently, x_{i+1} will be scanned first. In this case, RD needs $\Theta(k)$ passes[4] to compute the shortest path from 1 to $2k$. The total number of scans is therefore $\Theta(k^2) = \Theta(n^2)$.

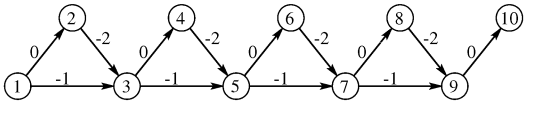


Figure 4: Worst-case input for RD ($k = 5$) when the priority queue returns the most recently inserted vertex among all those with maximum key.

The graph $G(k)$ is actually a worst-case input for the version of RDB where each bucket is implemented as a stack, since among all vertices in the same bucket the one most recently inserted will be preferred. If we represent each bucket as a queue (as in our implementation), however, RDB can process this graph with a linear number of scans. We can get a worst-case instance for this version of RDB as follows.

First, we swap the order in which the arcs leaving each odd vertex in $G(k)$ are processed, so that when x_i is scanned, x_{i+1} is inserted into the queue before y_i. To simplify the discussion, consider a shortest path computation starting from x_1 and assume that all labeled vertices are inserted in a single bucket. (Similar arguments to the ones below apply to the actual RDB algorithm for the feasibility problem.) After x_1 is scanned, x_2 and y_1 are inserted, in this order, into the queue, and x_2 is scanned before y_1. When y_1 is scanned, the distance to x_2 is improved but the current pass ends, as x_2 was already scanned. Then, the second pass begins with x_2 being scanned again. The same pattern continues until all odd vertices (except x_1) are scanned twice and all even vertices are scanned once, giving a total of k passes and $\Theta(k)$ scans in total.

To make the number of scans quadratic, we augment this graph with a set of new vertices that will be scanned in every pass. We connect each even vertex y_i to a new vertex $2k+1$ and connect $2k+1$ to k new vertices $2k+2, \ldots, 3k+1$. All new arcs have length -1. The new graph has $n = 3k+1$ vertices and $m = 5k-2$ arcs. Figure 5 gives an example for $k = 5$. The new vertices will be scanned in each pass, thus resulting in $\Theta(k^2)$ scans.

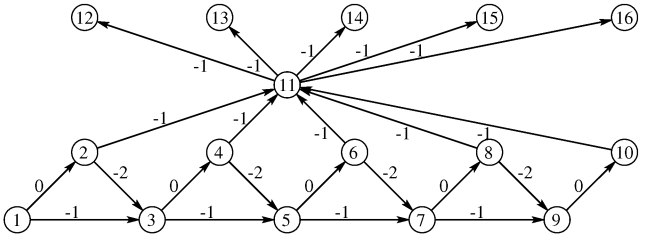


Figure 5: Worst-case input for RDB and RDH ($k = 5$).

The experimental results in Section 7.2 (Table 6) suggest that our implementation of RDH also exhibits quadratic number of scans for the graph of Figure 5. We refer to the this family as BAD-RD(k).

It is interesting to note that another graph inducing $\Theta(k^2)$ scans for RDB is a complete directed acyclic graph (DAG) on k vertices. In this graph we have an arc (i, j) with length -1, for each $i < j$. Also, the experimental results suggest that RDH requires a superlinear number of scans for this family, which we call COMP-DAG(k). Note that, unlike all previous families, these graphs are dense.

[4] The number of passes is exactly $k+1$ if subtree disassembly is not used, and $k/2+1$ if it is.

6.5 Arc-fixing algorithm. This is another family of dense graphs. We have $n = 3k + 2$ and $m = k^2 + 4k + 1$. To construct a graph, we start with a path $P = (1, 2, \ldots, 2k + 2)$ where the arcs lengths alternate between -1 and 1. Then we add k vertices, $2k + 3$ to $3k + 2$. For $0 \leq i \leq k - 1$, vertex $2k + 3 + i$ has an incoming arc of length $k+1-i$ from $2i+1$, and $2(k-i)$ unit-length outgoing arcs to $2i + 3, 2i + 4, \ldots, 2k + 2$. Figure 6 gives an example for $k = 3$.

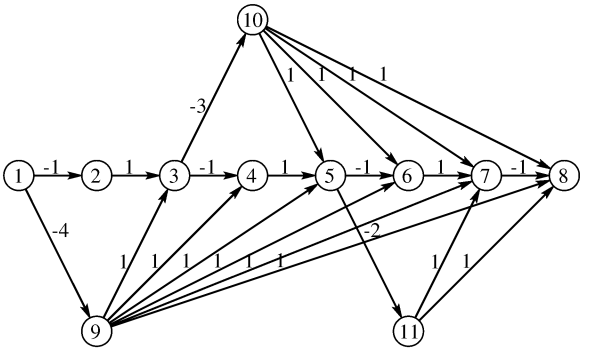


Figure 6: Worst-case input for the arc-fixing algorithm ($k = 3$).

Each iteration of the arc-fixing algorithm causes the reduced cost of one negative arc on P to become zero. At the end of the i-th iteration all vertices on the path $P_i = (2i + 3, 2i + 4, \ldots, 2k + 2)$ have equal potentials, so the reduced costs of the arcs on P_i do not change. Therefore, the algorithm requires $\Theta(k) = \Theta(n)$ iterations. Since each iteration runs Dijkstra's algorithm, the total running time is $\Omega(mn)$.

7 Experiments

7.1 Experimental setup. The improved implementations of BFCT and GOR, the most robust algorithms in previous studies, are the baseline of our experimental evaluation. We also include MBFCT, for which [26] gives encouraging results. Finally, we test our new algorithms, RDH and RDB, which are superior on some problem families.

Our main measure of performance is the number of scans per vertex, which is machine-independent. It includes both shortest path scans and auxiliary scans (such as those during DFS). All results reported are averages taken over 10 (or more, when noted) instances with different pseudorandom generator seeds.

We also report some running times to measure the data structure overhead for different algorithms. Our timed experiments were run on a 3 GHz Intel Xeon with 32 GB of RAM running Red Hat Enterprise Linux 5. For readability and succinctness, however, we omit running times for most experiments.

All algorithms were implemented in the same style and made as efficient as possible. They were coded in C++ and compiled with `gcc` v. 4.1.2 with the `-O4` flag. Arc lengths and potentials were represented as 32-bit signed integers.

7.2 Feasibility. We start with the standard (non-incremental) feasibility problem. As seen in [2], the number of negative cycles and their cardinality (number of arcs) can have a significant effect on performance. For a tight control over the structure of the instances, we created a filter called NEGCYCLE. It takes as input an arbitrary base graph with no negative arcs and adds vertex-disjoint negative cycles to it. Every arc on these cycles has length zero, except for one with length -1. As in [2], from each base graph family we create five subfamilies: (01) no negative cycles; (02) one negative cycle with three arcs (small cycle); (03) many small negative cycles; (04) a constant number of medium negative cycles; and (05) one Hamiltonian negative cycle.

After adding the negative cycles (or not, in case 01), we apply a potential transformation to "hide" them. A potential transformation with parameter X selects, for each vertex v, an integer $d(v)$ uniformly at random from $[0, X)$, adds $d(v)$ to each of v's incoming arcs, and subtracts $d(v)$ from the outgoing arcs; the effect is to replace lengths by reduced costs. Unless otherwise noted, we use $X = 1000$.

Our testbed includes the graph families tested in [2], augmented with road networks and worst-case examples. We discuss each family in turn.

7.2.1 Random graphs. The SPRAND generator [3] creates an instance with n vertices and m arcs by building a Hamiltonian cycle on the vertices and adding $m - n$ arcs at random. All arc lengths are selected uniformly at random from $[L, U]$.

In our first experiment, we set $L = 0$ and $U = 1000$ and fix $m = 5n$ (results were similar with other graph densities). We vary n, and apply the NEGCYCLE filter in each case. Table 1 shows the average number of scans per vertex for this family (which we call RAND5).

Note that the number of scans per vertex barely increases (if at all) with graph size, which suggests that on these graphs the algorithms perform much better than their worst-case bounds. For the 03 subfamily (which has many cycles of size 3), most algorithms actually do better (relative to n) as the graph size increases. The main exception is GOR, which performs a first pass that visits essentially every arc in the graph,

Table 1: Feasibility: Scans per vertex on RAND5.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262144	1.0105	1.0107	1.1016	1.0107	1.0180
	524288	1.0105	1.0107	1.1013	1.0106	1.0180
	1048576	1.0105	1.0107	1.1019	1.0106	1.0181
	2097152	1.0105	1.0106	1.1019	1.0106	1.0182
02	262144	0.9813	0.5803	1.0939	0.9311	0.7954
	524288	0.9713	0.4916	1.0934	0.7703	0.7805
	1048576	1.0087	0.6095	1.0939	0.9508	0.7082
	2097152	1.0078	0.5313	1.0938	0.7200	0.7851
03	262144	0.0798	0.0001	1.2186	0.0001	0.0001
	524288	0.0667	$<10^{-4}$	1.1947	$<10^{-4}$	$<10^{-4}$
	1048576	0.0397	$<10^{-4}$	1.2191	$<10^{-4}$	$<10^{-4}$
	2097152	0.0319	$<10^{-4}$	0.9882	$<10^{-4}$	$<10^{-4}$
04	262144	2.1171	0.2050	3.3318	1.9126	1.8541
	524288	2.1399	0.3257	3.3741	1.9669	1.8826
	1048576	2.1677	0.4349	3.4460	1.9929	1.9222
	2097152	2.1388	0.3724	3.5164	1.9872	1.9083
05	262144	5.2216	3.0555	10.6780	4.6662	4.6957
	524288	5.1291	2.5635	10.7117	4.6237	4.5978
	1048576	5.1830	3.0098	10.9752	4.6047	4.5592
	2097152	4.9904	2.5356	10.9011	4.5558	4.6200

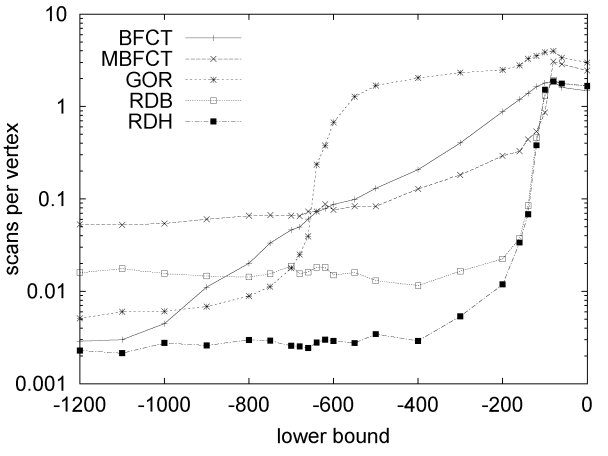


Figure 7: Feasibility on random graphs with 65536 vertices and arc lengths in $[L, 1000]$, with L variable.

but only detects a cycle when processing the SCCs in the second pass. For this family, MBFCT, RDH and RDB, which tend to concentrate on a small region of the graph, do better than BFCT, which takes a more global approach. For the 04 subfamily, MBFCT is the winner.

The results of our second experiment, also on random graphs, are shown in Figure 7. Here we fix $n = 65536$, $m = 5n$, the maximum arc length U at 1000, and vary the minimum allowed arc length L from 0 to -1200. We apply a potential transformation with parameter 2000 to the resulting graph (this is not done in [2], but it leads to more interesting results). The figure shows the average number of scans per vertex (over 50 different random seeds) as a function of L. Every instance with $L \leq -100$ had a negative cycle.

All algorithms are faster for lower values of L, when negative cycles are more numerous. RDH needs the fewest scans to find such cycles, in some cases winning by several orders of magnitude—the priority queue helps finding "obvious" cycles by focusing on a small number of candidate paths. The bucket-based version of the same algorithm, RDB, cannot find a cycle as fast: since buckets are organized as queues, the algorithm tends to expand many candidate paths at once. In fact, preliminary experiments have shown that organizing buckets as stacks (which tends to favor the most recently improved path) would make the algorithm about as fast as RDH for this problem family, but the stack-based implementation is less robust in general.

Both BFCT and GOR are not much slower than RDH when L is very negative, but degrade as it increases. GOR does so suddenly, at around $L = -650$, when the cycle-detection heuristic (which looks for negative back arcs during the DFS) ceases to be effective. BFCT degrades more slowly, but soon becomes worse than MBFCT.

7.2.2 Grids. We also tested the algorithms on *grid networks*, which are grids embedded on a torus, created by the TOR generator [2]. Specifically, vertices correspond to points on the plane with integer coordinates $[x, y]$, $0 \leq x < X$, $0 \leq y < Y$. These points are connected "upward" by *layer* arcs of the form $([x, y], [x, y + 1 \bmod Y])$, and "forward" by *inter-layer* arcs of the form $([x, y], [x + 1 \bmod X, y])$. In addition, there is a source connected to all vertices with $x = 0$. Lengths are chosen uniformly at random from $[1, 100]$ for layer arcs, and from $[1\,000, 10\,000]$ for inter-layer arcs (including those from the source). Once the grid is formed, we apply the NEGCYCLE filter to create distinct subfamilies.

Table 2 shows the average number of scans per vertex on *square grids with negative cycles* (SQNC), for which $X = Y$. All five subfamilies are shown.

This family is slightly harder than RAND5 for all algorithms, but their relative performance remains largely unchanged. An important exception is BFCT becoming faster than MBFCT on subfamily 05, which has a Hamiltonian negative cycle. Also, RDB outperforms RDH on the first four subfamilies.

Experiments with *long grids with negative cycles* (LNC), which have $X = n/16$ layers with $Y = 16$ vertices each, yielded very similar results, as Table 3

Table 2: Feasibility: Scans per vertex on SQNC.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262145	1.0153	1.6324	4.0449	1.7912	2.3171
	524177	1.0109	1.6303	4.0443	1.7906	2.3209
	1048577	1.0077	1.6276	4.0418	1.7908	2.3237
	2096705	1.0054	1.6255	4.0411	1.7905	2.3260
02	262145	0.9897	0.2255	3.1748	0.9189	1.2664
	524177	0.9213	0.3262	3.1536	0.6701	1.0183
	1048577	0.9667	0.5502	3.2424	0.4878	1.1787
	2096705	0.8345	0.3608	3.1213	0.6290	1.4416
03	262145	0.1973	0.0011	2.7220	0.0011	0.0023
	524177	0.1760	0.0011	2.7227	0.0010	0.0014
	1048577	0.2068	0.0003	2.7232	0.0003	0.0008
	2096705	0.1078	0.0008	2.7234	0.0008	0.0007
04	262145	2.1214	2.6874	6.0961	2.4650	2.7870
	524177	1.8846	1.5407	5.4931	2.3570	2.6913
	1048577	1.6985	1.1161	5.2201	2.2186	2.6100
	2096705	1.7382	0.6888	5.4070	2.2453	2.6151
05	262145	17.1357	48.5386	37.0674	12.2037	11.2036
	524177	17.7833	49.9466	37.6437	12.6346	11.6750
	1048577	18.8587	52.1875	39.6365	12.9990	12.0022
	2096705	19.8007	54.4070	42.1887	13.4241	12.2391

Table 3: Feasibility: Scans per vertex on LNC.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262145	1.3564	1.9224	4.0411	1.7901	2.0752
	524289	1.3566	1.9262	4.0428	1.7919	2.0769
	1048577	1.3566	1.9264	4.0427	1.7916	2.0737
	2097153	1.3567	1.9257	4.0433	1.7918	2.0761
02	262145	1.0063	0.3684	2.9553	0.3818	0.6307
	524289	0.9934	0.6405	2.9375	0.6405	1.4967
	1048577	0.9716	0.5137	3.1860	0.7144	1.3092
	2097153	1.0624	0.5253	3.2567	0.8406	0.5776
03	262145	0.1508	0.0005	2.6767	0.0003	0.0006
	524289	0.0965	0.0004	2.6765	0.0002	0.0001
	1048577	0.0829	0.0001	2.6762	0.0001	0.0001
	2097153	0.0762	$<10^{-4}$	2.6763	$<10^{-4}$	0.0001
04	262145	3.7127	7.1427	8.9451	3.5805	3.6780
	524289	3.9816	8.3143	9.4718	3.8881	3.8592
	1048577	5.0501	9.5579	11.7692	4.5792	4.4755
	2097153	4.4770	10.8935	10.2916	4.3081	4.2728
05	262145	16.9175	46.6560	36.0015	12.1477	11.0841
	524289	18.1748	49.0632	38.1384	12.8522	11.7341
	1048577	19.2769	52.5656	40.2583	13.2109	11.9235
	2097153	19.9966	53.2295	41.8637	13.5788	12.1767

shows. An important difference is that subfamily 04 is harder for all algorithms, especially MBFCT.

7.2.3 Layered networks. These graphs, also created by TOR, are partitioned into layers $0, \ldots, X-1$, each consisting of a cycle of length 32 plus 64 random arcs. Arc lengths within a layer are chosen uniformly at random from the interval $[1, 100]$. In addition, each vertex has five arcs to forward layers (including the next one). The length of an inter-layer arc that goes x layers forward is picked uniformly at random from $[1, 10\,000]$ and multiplied by x^2. A special source is connected to all vertices in the first layer by zero-length arcs. Finally, we introduce negative cycles with the NEGCYCLE filter. We call this family PNC (*p-hard with negative cycles*). Table 4 reports the results for this family. Note that, compared to other families, PNC is particularly bad for MBFCT when no cycles are present (subfamily 01).

7.2.4 Road networks. We also tested our algorithms on road networks, available at the 9th DIMACS Implementation Challenge web site [6]. Vertices represent intersections and arcs represent road segments (with lengths proportional to the travel distance). We used the NEGCYCLE filter to add negative cycles and perturb arcs lengths. Results for the CAL instance, which represents California and Nevada and has 1.9 million vertices, are shown in Table 5. (Results for other road networks were very similar.) Here MBFCT is usually the best algorithm, especially for the 04 subfamily.

7.2.5 Hard instances. Although the instances tested so far shed light on the feasibility problem, they are not very hard for any of the algorithms studied. In no case was the average number of scans higher than 55, even for graphs with more than 2 million vertices. All algorithms have essentially linear behavior. For a more complete analysis, we created the BAD family, with the worst-case instances described in Section 6.

We tested instances from this family with $k = 200$, 400, 800, and 1600. Recall that the number of vertices in each graph is a linear function of k. The number of arcs is quadratic in k for BAD-AF and COMP-DAG, and linear for the remaining families. Table 6 shows the number of scans per vertex in each case.

The table clearly shows the quadratic worst case of BFCT, GOR, RDB, and RDH in their respective families. Interestingly, MBFCT not only has a cubic worst case on its own subfamily, but also has quadratic behavior in some of the other families. In this sense, it is less robust than the other algorithms. RDB and RDH also have superlinear running time on BAD-AF and COMP-DAG.

It should be noted that all worst-case instances are somewhat sensitive to perturbations, such as permutations of vertex identifiers or potential transformations. This is especially true for BAD-RD, the hardest family to create.

7.2.6 Running times. So far, we have compared the algorithms only in terms of operation counts, which are machine-independent. We now discuss running times.

Table 4: Feasibility: Scans per vertex on PNC.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262145	2.0239	8.5484	5.8403	2.8259	2.5846
	524289	2.0243	8.5580	5.8373	2.8268	2.5822
	1048577	2.0245	8.5655	5.8356	2.8265	2.5830
	2097153	2.0244	8.5624	5.8347	2.8262	2.5825
02	262145	1.4243	2.1267	3.8669	0.7111	1.5076
	524289	1.0842	2.3483	4.2981	1.1895	1.1215
	1048577	1.4933	2.1911	3.8810	0.9444	1.1428
	2097153	1.3424	2.5123	4.1195	0.8123	1.2812
03	262145	0.1303	0.0005	2.3687	0.0004	0.0005
	524289	0.0968	0.0003	2.3681	0.0001	0.0002
	1048577	0.0969	0.0005	2.3674	0.0002	0.0002
	2097153	0.0764	0.0001	2.3672	$<10^{-4}$	0.0001
04	262145	6.3613	20.0428	14.0924	6.5855	6.0718
	524289	6.8502	20.0095	15.6653	6.9911	6.4192
	1048577	6.5174	25.3448	13.9551	6.6889	6.3908
	2097153	7.2917	26.2656	15.9607	7.0950	6.6919
05	262145	8.2741	22.0434	16.1135	7.1392	6.8433
	524289	8.7537	17.7014	17.7396	7.3837	6.9304
	1048577	8.9492	28.6557	18.1954	7.5952	6.8891
	2097153	9.5213	21.6117	18.9055	7.9059	7.3997

Table 5: Feasibility: Scans per vertex on CAL.

FAM	BFCT	MBFCT	GOR	RDB	RDH
01	1.0500	1.0533	1.3251	1.0513	1.0519
02	1.0213	0.4729	1.3218	0.6290	0.8492
03	0.0482	$<10^{-4}$	1.3897	$<10^{-4}$	$<10^{-4}$
04	2.4935	0.4038	4.0262	2.2669	2.1212
05	5.6818	3.3757	12.3628	5.2562	5.2181

Table 6: Feasibility: Scans per vertex on hard instances.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
BAD-AF	602	34.7	34.4	3.0	3.2	6.6
	1202	68.1	67.7	3.0	3.9	7.5
	2402	134.7	134.4	3.0	4.4	13.2
	4802	268.1	267.7	3.0	5.0	16.0
BAD-BFCT	799	51.8	374.6	2.3	2.3	2.3
	1599	101.8	749.6	2.3	2.3	1.3
	3199	201.8	1499.6	2.3	2.3	2.2
	6399	401.8	2999.6	2.3	2.3	1.3
BAD-GOR	401	1.0	1.2	103.6	1.0	1.0
	801	1.0	1.2	203.6	1.0	1.0
	1601	1.0	1.2	403.6	1.0	1.0
	3201	1.0	1.2	803.6	1.0	1.0
BAD-MBFCT	1199	1.7	6840.0	2.3	1.7	1.7
	2399	1.7	27012.3	2.3	1.7	1.7
	4799	1.7	107356.7	2.3	1.7	1.7
	9599	1.7	428045.6	2.3	1.7	1.7
BAD-RD	601	1.0	2.7	3.0	61.6	35.4
	1201	1.0	2.7	3.0	128.5	68.8
	2401	1.0	2.7	3.0	262.0	135.4
	4801	1.0	2.7	3.0	528.9	268.8
COMP-DAG	200	1.0	50.5	3.0	50.5	11.0
	400	1.0	100.5	3.0	100.5	15.7
	800	1.0	200.5	3.0	200.5	21.4
	1600	1.0	400.5	3.0	400.5	30.2

For reference, the total running time for RAND5 is shown in Table 7. For this family, the algorithms are usually within a factor of two or three from each other. The main exceptions are GOR, which is substantially slower than other algorithms in subfamily 03, and MBFCT, which is substantially faster in subfamily 04.

For a more detailed analysis, Table 8 shows the average time per scan on the same instance. Note that this table only reports the time spent during the main loop of each algorithm, during which all scans are performed. The time required to allocate and initialize the data structures specific to each method is not considered here. (The total times listed in Table 7 do include initialization.) As a side effect, we could not measure the average time per scan for subfamily 03 with enough precision, since the number of scans is very small. For that reason, the results for this subfamily are not presented in Table 8.

Based on the subfamilies for which we could measure times reliably, we can draw some conclusions. In general, and not surprisingly, MBFCT and BFCT take about the same time to scan a vertex. GOR, which ex-

Table 7: Feasibility: Average total time in seconds for RAND5.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262144	0.105	0.115	0.124	0.157	0.186
	524288	0.220	0.241	0.260	0.338	0.398
	1048576	0.459	0.505	0.541	0.711	0.880
	2097152	0.962	1.059	1.131	1.532	1.804
02	262144	0.102	0.069	0.124	0.144	0.145
	524288	0.211	0.123	0.259	0.260	0.307
	1048576	0.457	0.314	0.539	0.671	0.592
	2097152	0.958	0.579	1.124	1.109	1.387
03	262144	0.014	0.006	0.148	0.011	0.016
	524288	0.026	0.012	0.304	0.022	0.032
	1048576	0.042	0.024	0.651	0.043	0.063
	2097152	0.078	0.048	1.122	0.086	0.122
04	262144	0.413	0.047	0.588	0.508	0.584
	524288	0.909	0.153	1.294	1.187	1.307
	1048576	2.006	0.433	2.892	2.602	2.903
	2097152	4.205	0.820	6.403	5.638	6.293
05	262144	1.328	0.778	2.194	1.620	1.897
	524288	2.841	1.407	4.773	3.653	4.115
	1048576	6.242	3.602	10.745	7.867	9.035
	2097152	12.752	6.425	23.024	16.920	19.647

Table 8: Feasibility: Average time per scan in microseconds for RAND5.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262144	0.37	0.41	0.39	0.55	0.64
	524288	0.39	0.43	0.41	0.60	0.69
	1048576	0.41	0.45	0.43	0.63	0.77
	2097152	0.43	0.48	0.45	0.68	0.79
02	262144	0.37	0.41	0.39	0.55	0.62
	524288	0.39	0.43	0.41	0.59	0.67
	1048576	0.41	0.45	0.43	0.63	0.71
	2097152	0.43	0.48	0.45	0.68	0.76
04	262144	0.73	0.81	0.66	0.99	1.17
	524288	0.80	0.87	0.72	1.13	1.29
	1048576	0.87	0.91	0.79	1.22	1.41
	2097152	0.93	1.01	0.85	1.33	1.54
05	262144	0.97	0.96	0.78	1.32	1.53
	524288	1.05	1.04	0.84	1.50	1.69
	1048576	1.14	1.13	0.93	1.62	1.88
	2097152	1.21	1.20	1.00	1.76	2.02

ecutes cheaper scans during DFS, is often faster than BFCT. RDH is the only method whose time per scan may depend on n. In this family, however, it is less than twice as slow as BFCT, regardless of graph size, which indicates that the heap is often very small. RDB is faster (per vertex) than RDH, but by less than 20% when RDH does well. In contrast, for the SQNC family the dependence of RDH on n is more obvious, as Table 9 shows (once again, we omit subfamily 03). For RDH, scans are slower, especially when the heap is large, as one would expect.

Table 9: Feasibility: Average time per scan in microseconds for SQNC.

FAM	n	BFCT	MBFCT	GOR	RDB	RDH
01	262145	0.05	0.08	0.07	0.38	0.77
	524177	0.05	0.08	0.07	0.40	0.87
	1048577	0.05	0.08	0.07	0.42	0.98
	2096705	0.05	0.08	0.08	0.44	1.13
02	262145	0.05	0.06	0.05	0.27	0.52
	524177	0.05	0.08	0.05	0.22	0.43
	1048577	0.05	0.08	0.05	0.18	0.58
	2096705	0.05	0.08	0.05	0.23	0.77
04	262145	0.22	0.28	0.11	0.42	0.71
	524177	0.22	0.30	0.11	0.44	0.80
	1048577	0.22	0.31	0.10	0.46	0.91
	2096705	0.24	0.31	0.10	0.48	1.03
05	262145	0.75	0.74	0.48	1.09	1.27
	524177	0.83	0.81	0.53	1.18	1.39
	1048577	0.89	0.88	0.57	1.28	1.54
	2096705	0.97	0.96	0.62	1.43	1.72

7.3 Shortest paths. As already mentioned, the feasibility and shortest path problems are closely related. In fact, most of the algorithm discussed here can be used to solve the latter problem with minor modifications in the initialization phase. In particular, potentials are initialized to M (a value larger than the length of any shortest path) instead of zero, and the root is the only labeled vertex initially.

Table 10 shows the average number of scans per vertex required to solve the shortest path problem (as opposed to feasibility) on the main families tested. For RAND5, SQNC, LNC, and PNC, we set $n = 2^{20} + 1$.

Table 10: Shortest paths: Scans per vertex.

FAMILY		BFCT	GOR	RDB	RDH
RAND5	01	2.1210	4.4187	1.9610	1.0005
	02	1.2957	2.4887	1.1922	0.2349
	03	0.0003	0.0007	0.0003	$<10^{-4}$
	04	18.8214	38.3942	12.2307	4.7244
	05	21.9508	45.2572	14.4742	7.0170
SQNC	01	2.5172	7.2834	11.3351	13.5625
	02	0.2909	1.8643	0.2693	0.4220
	03	0.0022	0.0065	0.0022	0.0007
	04	5.6998	14.4458	5.9138	4.5004
	05	17.5253	38.1337	13.0139	11.8274
LNC	01	3.2505	6.2108	182.9656	9.7147
	02	0.8680	1.6579	0.2646	2.2448
	03	0.0001	0.0003	0.0001	$<10^{-4}$
	04	10.6546	26.1818	9.7350	8.5018
	05	17.4264	37.3533	12.8423	11.7624
PNC	01	10.4459	17.1255	21.8359	28.3594
	02	2.2423	3.6602	3.9441	2.2051
	03	0.0008	0.0044	0.0004	0.0001
	04	8.4816	19.1209	9.3675	8.5197
	05	6.9598	14.7687	6.2270	6.5254
CAL	01	2.9274	5.9919	12.2258	1.3121
	02	0.5841	1.5298	0.4309	0.3067
	03	$<10^{-4}$	$<10^{-4}$	$<10^{-4}$	$<10^{-4}$
	04	17.5119	37.0022	11.6149	6.2877
	05	24.0836	52.7844	15.9154	10.5812

Comparing these results with those obtained for the feasibility problem, we see that they can be significantly different. When there are no cycles (in subfamily 01), feasibility algorithms are consistently faster than their SP counterparts, since the former do not need to build the full shortest path tree. The difference is considerable for RDB on subfamily 01 of LNC. (In this case, representing buckets as stacks instead of queues would not improve performance; in fact, it would be much worse, according to our preliminary experiments.)

Even when negative cycles are present, there are cases in which feasibility algorithms are significantly faster than the corresponding SP version. This often happens when the negative cycles are relatively long, as in subfamilies 04 and 05.

An interesting case is that of subfamily 03, with numerous but very small cycles. We have seen that

almost all feasibility algorithms can find a negative cycle while visiting a very small portion of the graph. The exception was GOR, which had to traverse the entire graph before finding a negative cycle. This only happens because all vertices are initially labeled; in the SP version of GOR, in contrast, only the source is labeled in the beginning. As a result, the algorithm is about as fast as the other methods.

The results of this section suggest that one should not compare feasibility and shortest path algorithms directly, and explains the difference between our results and those of [26].

7.4 Incremental feasibility. The experiments reported up to this point have assumed that one must solve the feasibility problem from scratch. There are cases, however, when one must simply adjust a feasible set of potentials after a small modification in the underlying graph. We call this the *incremental feasibility problem* (IFP). In general, we want to study a dynamic situation in which one solves a sequence of feasibility problems, each a perturbation of the previous one. This occurs in applications such as the minimum cycle mean problem [4].

To assess how an FP algorithm would behave in the incremental context, we give it two inputs: the graph itself and a list of all vertices with negative outgoing arcs. Since all potentials are pre-initialized to zero, the algorithm needs to focus initially only on these vertices. Unlike the standard FP case, the measurements we report (operation counts) do *not* include the initialization phase, since our goal is to model a situation in which the algorithm is restarted after computing a feasible set of potentials.

To create an IFP instance, we start with a graph containing only nonnegative arcs and make a random subset of them negative. To change the length of an arc (v, w), we first use Dijkstra's algorithm to determine the distance D from w to v with respect to the current arc lengths. If v is not reachable from w, we set $\ell(v, w)$ to -1. Otherwise, we decrease $\ell(v, w)$ to $-2D-1$, $-D-1$, $-D$, or $-D/2$ to create a very negative cycle, a slightly negative cycle, a zero cycle, or a positive cycle. We denote these cases by -2, -1, 0, and 1, respectively.

We must be careful when changing the lengths of more than one arc. In cases 0 and 1, we perform these changes sequentially: when computing the distance between the endpoints of the i-th arc, we use the potentials from previous Dijkstra's computations to maintain arc nonnegativity. In cases -1 and -2, which create negative cycles, we perform all shortest path computations on the original graph, and change all arc lengths at once.

Table 11: Incremental feasibility: Scans per vertex on RAND5 ($n = 65536$).

CASE	ARCS	BFCT	MBFCT	GOR	RDB	RDH
-2	1	0.122	0.122	0.905	0.145	0.112
	2	0.143	0.132	1.115	0.123	0.140
	64	0.221	0.146	0.405	0.038	0.032
	1024	0.361	0.223	0.785	0.021	0.007
-1	1	0.222	0.222	0.833	0.157	0.176
	2	0.365	0.335	1.326	0.246	0.172
	64	0.571	0.579	0.610	0.061	0.037
	1024	0.513	0.200	0.769	0.018	0.007
0	1	0.512	0.512	1.073	0.431	0.423
	2	1.229	1.251	2.503	1.004	0.979
	64	3.543	5.711	8.911	2.828	2.710
	1024	4.199	9.171	12.795	3.230	3.083
1	1	0.004	0.004	0.008	0.004	0.004
	2	0.012	0.012	0.024	0.012	0.012
	64	1.034	1.137	1.933	0.960	0.850
	1024	5.151	8.930	9.207	3.724	3.391

Table 11 shows the average number of scans per vertex required to find feasible potentials starting from an SPRAND-generated graph with $n = 2^{16}$ and $m = 5n$. For each case, we vary the number of arcs made negative: 1, 2, 64 or 1024. Tables 12, 13 and 14 present the average number of scans per vertex for SQNC, LNC, and PNC instances, all with 65537 vertices.

Table 12: Incremental feasibility: Scans per vertex on SQNC ($n = 65537$).

CASE	ARCS	BFCT	MBFCT	GOR	RDB	RDH
-2	1	1.433	1.433	30.412	0.693	0.701
	2	1.355	1.185	13.544	0.733	0.796
	64	1.938	0.350	8.763	0.365	0.396
	1024	2.619	0.489	5.138	0.622	0.217
-1	1	1.597	1.597	23.622	3.756	0.703
	2	1.286	1.452	16.768	1.270	0.797
	64	1.844	0.290	8.709	0.371	0.338
	1024	2.635	0.502	5.043	0.591	0.219
0	1	1.612	1.612	22.908	3.817	0.704
	2	2.066	2.068	26.526	4.929	0.973
	64	2.326	2.288	14.749	5.124	2.271
	1024	2.515	4.962	10.007	5.793	3.918
1	1	0.612	0.612	5.831	1.024	0.349
	2	1.093	1.086	9.256	1.923	0.663
	64	2.738	2.545	14.984	5.280	4.243
	1024	3.227	4.382	14.738	6.717	5.668

Here RDH is the most robust algorithm. It is never much worse than any other method, and often much better. RDB has comparable behavior for random graphs, but is noticeably worse for TOR-based instances. Unsurprisingly, BFCT and MBFCT have the exact same

Table 13: Incremental feasibility: Scans per vertex on LNC ($n = 65537$).

CASE	ARCS	BFCT	MBFCT	GOR	RDB	RDH
-2	1	0.650	0.650	2.430	1.278	0.200
	2	0.390	0.800	1.541	3.291	0.315
	64	0.057	0.079	0.167	0.295	0.029
	1024	0.500	0.004	0.754	0.590	0.001
-1	1	0.650	0.650	2.506	5.518	0.200
	2	0.390	0.800	1.590	3.640	0.315
	64	0.056	0.079	0.165	0.246	0.029
	1024	0.497	0.004	0.757	0.589	0.001
0	1	0.650	0.650	2.232	5.416	0.200
	2	2.908	2.908	8.509	23.461	0.969
	64	3.233	3.231	6.169	26.759	1.563
	1024	3.209	3.285	5.762	25.394	1.655
1	1	0.324	0.324	1.114	1.883	0.100
	2	1.585	1.531	5.089	9.016	0.493
	64	4.377	3.727	9.457	23.142	3.320
	1024	4.492	3.994	9.061	23.392	4.120

Table 14: Incremental feasibility: Scans per vertex on PNC ($n = 65537$).

CASE	ARCS	BFCT	MBFCT	GOR	RDB	RDH
-2	1	5.183	5.183	8.425	6.924	0.493
	2	3.623	3.492	5.992	4.566	0.337
	64	0.002	0.117	0.019	0.058	0.010
	1024	0.016	0.031	0.059	0.057	0.003
-1	1	4.853	4.853	8.407	6.159	0.497
	2	3.645	3.489	5.975	4.507	0.337
	64	0.003	0.122	0.032	0.133	0.010
	1024	0.017	0.035	0.061	0.099	0.003
0	1	4.853	4.853	7.743	6.183	0.497
	2	7.735	7.735	12.254	9.904	0.861
	64	9.174	9.900	15.224	11.704	4.782
	1024	6.678	16.118	14.239	9.288	4.050
1	1	2.248	2.248	3.542	2.744	0.249
	2	4.535	4.257	7.098	5.518	0.473
	64	11.239	10.779	18.913	13.911	5.131
	1024	9.279	22.839	18.486	12.183	5.701

performance when there is only one original negative arc, since they reduce to the same algorithm. With more arcs, MBFCT is better than BFCT when negative cycles are obvious, but BFCT is clearly superior when they are absent. Once again, GOR is not competitive.

Table 15 presents the corresponding results for CAL. Since all algorithms perform very few scans per vertex, we report the actual number of scans in this case. In particular, MBFCT, RDB and RDH are remarkably good when there are negative cycles. It is not hard to see why. In road networks, an arc tends to be the shortest path between its endpoints. In addition, the input instance is undirected: for each arc (v, w) there is a corresponding arc (w, v) with the same length. As a result, the negative cycles created by our filter tend to have only two arcs, and are quickly discovered by MBFCT, RDB and RDH.

8 Concluding Remarks

We developed an experimental framework for the feasibility problem that is more extensive than the previous one [2], which was actually designed for the shortest path problem. This led to interesting results. For the feasibility problem, where in some cases it suffices to scan a small subset of the vertices, GOR is not as robust as BFCT. Another previously studied algorithm, MBFCT [26], performs poorly on some problem classes but outperforms BFCT when negative cycles can be found locally. Our new algorithm, RDH, performs even better than MBFCT on many of the classes where the latter works well, although MBFCT still wins on some classes.

Although no single algorithm is dominant, BFCT, MBFCT, and RDH are the most practical among those studied in this and previous papers. But there is no reason to think that these are the best. It is quite possible that better algorithms exist, even if we restrict ourselves to algorithms within our $O(n)$-pass framework.

References

[1] R. E. Bellman. On a Routing Problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[2] B. V. Cherkassky and A. V. Goldberg. Negative-Cycle Detection Algorithms. *Math. Prog.*, 85:277–311, 1999.

[3] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Prog.*, 73:129–174, 1996.

[4] A. Dasdan. Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms. *ACM Trans. on Design Automation of Electronic Systems*, 9(4):385–418, 2004.

[5] R. Dechter, I. Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.

[6] C. Demetrescu, A.V. Goldberg, and D.S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths. http:// www.dis.uniroma1.it/~challenge9/, 2007.

[7] E. V. Denardo and B. L. Fox. Shortest–Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.

[8] R. B. Dial, F. Glover, D. Karney, and D. Klingman. A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees. *Networks*, 9:215–248, 1979.

[9] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959.

Table 15: Incremental feasibility: Total number of scans on CAL.

CASE	ARCS	BFCT	MBFCT	GOR	RDB	RDH
-2	1	2.0	2.0	13.3	2.0	2.0
	2	3.0	2.0	19.1	2.0	2.0
	64	65.0	2.0	327.4	2.0	2.0
	1024	1000.1	2.0	5125.3	2.0	2.0
-1	1	2.0	2.0	11.4	2.0	2.0
	2	3.0	2.0	17.2	2.0	2.0
	64	65.0	2.0	325.6	2.0	2.0
	1024	1000.1	2.0	5123.4	2.0	2.0
0	1	4.3	4.3	9.1	4.3	4.3
	2	7.3	7.3	16.0	7.3	7.3
	64	262.8	262.8	575.3	262.1	261.8
	1024	4144.8	4144.8	9079.3	4140.5	4136.8
1	1	2.7	2.7	6.7	2.7	2.7
	2	5.0	5.0	11.6	5.0	5.0
	64	172.1	172.1	391.2	171.8	171.8
	1024	2728.0	2728.5	6254.5	2728.0	2727.9

[10] L.R. Ford, Jr. Network Flow Theory. Technical Report P-932, The Rand Corporation, 1956.

[11] L.R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ, 1962.

[12] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[13] G. Gallo and S. Pallottino. Shortest Paths Algorithms. *Annals of Oper. Res.*, 13:3–79, 1988.

[14] A. V. Goldberg. Scaling Algorithms for the Shortest Paths Problem. *SIAM J. Comput.*, 24:494–504, 1995.

[15] A. V. Goldberg and T. Radzik. A Heuristic Improvement of the Bellman-Ford Algorithm. *Applied Math. Let.*, 6:3–6, 1993.

[16] J. L. Kennington and R. V. Helgason. *Algorithms for Network Programming.* John Wiley and Sons, 1980.

[17] S.G. Kolliopoulos and C. Stein. Finding Real-Valued Single-Source Shortest Paths in $o(n^3)$ Expected Time. In *Proc. 5th Int. Programming and Combinatorial Optimization Conf.*, 1996.

[18] E. F. Moore. The Shortest Path Through a Maze. In *Proc. of the Int. Symp. on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[19] M. Nonato, S. Pallottino, and B. Xuewen. SPT_L Shortest Path Algorithms: Reviews, New Proposals, and Some Experimental Results. Technical Report TR-99-16, Dipartmento di Informatica, Pisa University, Italy, 1999.

[20] S. Pallottino. Shortest-Path Methods: Complexity, Interrelations and New Propositions. *Networks*, 14:257–267, 1984.

[21] U. Pape. Implementation and Efficiency of Moore Algorithms for the Shortest Root Problem. *Math. Prog.*, 7:212–222, 1974.

[22] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[23] R. E. Tarjan. Shortest Paths. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1981.

[24] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[25] G. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1970.

[26] C-H. Wong and Y-C Tam. Negative Cycle Detection Problem. In *Proc. 13th Annual European Symposium on Algorithms*, pages 652–663, 2005.

Ranking Tournaments: Local Search and a New Algorithm*

Tom Coleman† Anthony Wirth‡

Abstract

Ranking data is a fundamental organizational activity. Given advice, we may wish to rank a set of items to satisfy as much of that advice as possible. In the Feedback Arc Set (FAS) problem, advice takes the form of pairwise ordering statements, 'a should be ranked before b'. Instances in which there is advice about every pair of items is known as a *tournament*. This task is equivalent to ordering the nodes of a given directed graph to minimize the number of arcs pointing in one direction.

In the past, much work focused on finding good, effective heuristics for solving the problem. Recently, a proof of the NP-completeness of the problem (even when restricted to tournaments) has accompanied new algorithms with approximation guarantees, culminating in the development of a PTAS (polynomial time approximation scheme) for solving FAS on tournaments.

In this paper we re-examine many of these existing algorithms and develop some new techniques for solving FAS. The algorithms are tested on both synthetic and Rank Aggregation-based datasets. We find that, in practice, local-search algorithms are very powerful, even though we prove that they do not have approximation guarantees. Our new algorithm is based on reversing arcs whose nodes have large indegree differences, eventually leading to a total ordering. Combining this with a powerful local-search technique yields an algorithm that beats existing techniques on a variety of data sets.

1 Introduction

1.1 The Feedback Arc Set Problem The Feedback Arc Set (FAS) problem is a key ranking problem, asking us to rank items in a set given only advice about the best way to order specific pairs. A ranking π is simply a permutation of that set. Thus the only information we have to help us to form our ranking is a set of statements of the form 'a should be ranked before b'. These statements may be contradictory: the aim of the FAS problem is to produce a π that violates as few of these pairwise statements as possible.

The natural graph representation of the problem uses a vertex for each item and a directed arc from a to b for each demand 'a should be ranked before b'. In this context, the aim is to order the vertices from left to right so that the number of arcs pointing left (back-arcs) is as small as possible.

Given some ranking π, if we remove the set of back-arcs, we will eliminate all cycles in the graph. We call such a set a 'Feedback Arc Set'. An equivalent formulation of the problem is therefore: given a digraph G, find the smallest subset of the arcs of G that intersects all cycles in G (whose removal therefore renders G acyclic). In this paper we focus on the special case of *tournament* graphs, in which there is an arc between every pair of nodes. We also consider *weighted* tournaments, in which the weights on arcs $u \to v$ and $v \to u$ must sum to 1.

Originally motivated by problems in circuit design [13], FAS has found applications in many areas, including computational chemistry [21, 19], and graph drawing [12]. An application to Rank Aggregation in particular has seen much focus solving the problem on directed complete graphs (tournaments).

1.2 The Rank Aggregation Problem Closely related to the FAS problem is the Rank Aggregation problem. Given a set of rankings, we must find a single ranking that *best* represents the consensus. Dwork et al. [10] outline the problem and motivate it as a method for aggregating data from search engines. There is a significant body of work studying this problem, which is known as MetaSearch [5].

Precisely what *best* means is a difficult issue, and it relates to how we interpret the similarity of rankings. The Kemeny distance [16] $K(\pi, \sigma)$ between two rankings, π and σ, is defined as the number of pairs i, j for which π ranks i above j, yet σ ranks j above i. In this paper, the Rank Aggregation problem seeks a ranking τ that minimizes the *sum* of the Kemeny distances from τ to each of the input rankings.

This problem is a special case of FAS on weighted tournaments. There is a fairly simple reduction from Rank Aggregation to FAS: for each i, j, if i is ranked above j in some fraction w of the input rankings, place an arc between i and j with weight w.

1.3 The Linear Ordering Problem A very similar problem to the Feedback Arc Set problem is the Linear Ordering problem. Here we have a matrix (c_{ij}), and we want to choose an ordering σ such that $\sum_{i <_\sigma j} c_{ij}$ is minimized. The weighted tournament version of FAS is a special case of this, under the

*This work was supported by the Australian Research Council, through Discovery Project grant DP0663979.

†The University of Melbourne

‡The University of Melbourne

constraint that $c_{ij} = 1 - c_{ji}$.

1.4 Hardness of these problems The FAS problem is one of Karp's [15] original NP-complete problems (in the general case). Dwork et al. [10] prove the RANK AGGREGATION problem is NP-hard, even with as few as 4 input rankings. Recently, the FAS problem has been shown to be NP-hard even on unweighted tournaments [7]. Given this, it is natural to ask for an approximation algorithm which runs in polynomial time, yet is guaranteed to differ in cost from the optimal solution by a small factor.

1.5 Algorithms with Guarantees The first constant factor approximation algorithm for FAS, designed primarily for tournaments, was the pivoting algorithm of Ailon et al. [1]. Intuitively similar to quicksort, it uses on average $O(n \log n)$ comparisons for an expected 3-approximation on *tournament graphs*.

Coppersmith et al. [9] show that ordering the nodes by their indegrees is a 5-approximator on tournament instances, regardless of how ties are broken. The vast majority of sporting leagues use this heuristic to rank teams (though often with elaborate tie-breaking mechanisms).

Finally, Kenyon-Mathieu and Schudy completed the approximation picture for tournaments picture with a PTAS (polynomial-time approximation scheme) [18]. This scheme comprises a simple moves-based local search heuristic—which we investigate in isolation below—and the PTAS of Arora, Frieze and Kaplan [3] for the MAXIMUM ACYCLIC SUBGRAPH problem.[1]

1.6 Heuristics Viewing the input as a directed graph, we define the *Kendall score* of a node to be its indegree. One of the simplest and earliest heuristics developed for the FAS problem ranks the items according to their *Kendall scores*, breaking ties arbitrarily [17]. Ali et al. [2] and Cook et al. [8] improve this technique by considering various methods of breaking ties. Let us define the ITERATED KENDALL algorithm to be:

> Rank the nodes by their Kendall scores. If there are nodes with equal score, break ties by recursing on the subgraph defined by these nodes. If there is a subgraph whose elements all have equal indegree, rank them arbitrarily.

Eades et al. [11] created an algorithm that is in fact quite similar to ITERATED KENDALL, possibly inspired by selection sort. We define the EADES algorithm to be:

> Select the node that has the smallest Kendall score and place it at the left, breaking ties arbitrarily. Recurse on the remainder of the nodes, having recomputed the indegrees on the remaining subgraph.

In retrospect, the tie breaking could be done in a more sophisticated way.

Chanas and Kobylanski [6] present an algorithm for the LINEAR ORDERING problem based on repeated application of a procedure analogous to insertion sort. In Section 2.2 we outline the algorithm, and compare it to other sort-based methods.

Saab [23] presents an algorithm using a divide-and-conquer approach. The idea is to split the input into two halves, minimizing the number of back-arcs between the halves, and then recurse on each half. However this minimization task is a difficult one: it is a directed version of the MIN BISECTION problem, and solving it would solve FAS. We do not explore this algorithm further.

1.7 Our Contributions / Paper Organization In Section 2 we provide details of the algorithms we will test. Some are existing procedures (especially local search approaches), others are our improvements to them, still others are based on our scheme of reversing arcs (in an organized manner) to destroy directed triangles, and thus produce a total ordering.

We then show in Section 3 that some of the existing heuristic algorithms are not guaranteed approximators for the FAS problem.

We tested all algorithms on not only synthetic data (as has generally been the method of testing heuristics in the past), but also on three sets of FAS problems generated from RANK AGGREGATION data. This work is outlined in Section 4.

The results of these tests and further analyses are presented in Section 5. We conclude and supply ideas for further work in Section 6.

2 Algorithm details

2.1 Improving the Eades Algorithm The EADES algorithm focuses on the left side of the ranking. We improve this by allowing the selection of a node to either end of the ranking. Let $\text{In}(v)$ stand for the indegree of node v, with $\text{Out}(v)$ its outdegree. We define the algorithm EADES IMPROVED to be:

> Select the node u that maximizes $|\text{In}(u) - \text{Out}(u)|$ and place it at the appropriate end of the ranking. Recurse on the subgraph induced by removing u. Again, break ties arbitrarily.

[1] We did not implement this algorithm during this experimental evaluation. Although theoretically very powerful, the algorithm is complicated and impractical to implement.

2.2 Sorting Algorithms As suggested above, Ailon et al.'s algorithm [1] is much like quicksort. For this paper QUICKSORT is defined to be:

> Choosing pivots uniformly at random, run the quicksort algorithm with the *comparison* function: $u < v$ if and only if $u \to v$.

Unlike traditional sorting problems, in which we assume there is a total order on the data, the difficulty in FAS is the lack of transitivity, which sorting algorithms are designed to exploit. Nevertheless, sorting algorithms provide schemes for deciding which of the advice to *believe*.

Cook et al. [8] focus on ensuring a Hamiltonian path exists in along the final ordering of the nodes; any sensible algorithm should achieve this. The method they use to reach this is in effect a BUBBLESORT of the tournament, using the same comparison function as QUICKSORT.

To our knowledge, no paper has studied the MERGESORT approach. In this paper we test its performance.

As noted above, the EADES algorithm is the obvious analogy to selection sort (with our improved version being a two-sided selection sort). Chanas and Kobylanski [6] apply an insertion technique to the LINEAR ORDERING problem that is more involved than the usual insertion sort. Their 'SORT' procedure is:

> Make a single pass through the nodes from the left to the right. As each node is considered, it is moved to the left, inserted into the position that minimizes the number of back-arcs.

Since executing SORT cannot increase the number of back-arcs, the authors first propose an algorithm SORT* which repeatedly applies SORT until there is no improvement in the number of back-arcs. They also show that the composition of two steps SORT $\circ$ REVERSE (where REVERSE simply reverses the order of the nodes) cannot increase the number of back-arcs. The CHANAS algorithm is therefore (SORT* $\circ$ REVERSE)*, which Chanas and Kobylanski show outperforms the original version in practice.

2.3 Local Search Algorithms One approach that has been used successfully for many optimization problems is to begin with some solution and then iteratively improve that solution until no further improvement is possible. Researchers have met with success in proving approximation bounds for local search schemes for the k-median [4] and k-means [14] problems, along with related problems. Here we consider two such local improvement schemes for FAS:

The SWAPS heuristic, which swaps the position of two nodes in the order.

The MOVES heuristic, which moves one node to any position in the order, leaving the relative order of the other nodes unchanged.

In this paper we show that neither algorithm can provide an approximation guarantee. We found that the MOVES heuristic performs well in practice, but that SWAPS does not: so the latter was omitted from our experiments.

One particular advantage of a local search techniques is that the initial solution it is given can be the output of an approximation algorithm. Consequently, the local search approach inherits the approximation guarantee.

Chanas An application of the SORT step of CHANAS has the effect of checking, for each vertex of the graph, from left to right, if a move to the left in the order is possible. This is essentially a scheme for selecting which MOVES-style changes to make. The operation SORT $\circ$ REVERSE does the same thing, but with moves to the right. So CHANAS is simply a method for investigating MOVES in a particular order. We developed an alternative algorithm, termed CHANAS BOTH: the SORT procedure is allowed to move a node *either* left or right, to the position that results in the fewest back-arcs. A consequence of this modification is that some nodes may be moved more than once in a single SORT pass.

2.4 Triangle-Destroying Algorithms A tournament only has a cycle if it has a directed triangle ($\vec{\triangle}$). We therefore considered algorithms that destroy directed triangles by selecting arcs to reverse. It might seem more natural to delete arcs, but this would make the digraph no longer a tournament, creating the possibility of cycles without the presence of $\vec{\triangle}$s. Our algorithms work in the following way:

> While the tournament is not acyclic, *choose* an arc and reverse its orientation. Once the tournament is acyclic, use the ordering of the nodes as the solution to the original problem.

The choice of arc to be reversed affects the performance and running time of this procedure; the remainder of this section examines various heuristics.

We call the number of $\vec{\triangle}$s an arc is involved in its *triangle count*. The first algorithm, TRIANGLE COUNT, simply chooses the arc with the highest triangle count. There is a pitfall here though: reversing an arc can create a new $\vec{\triangle}$ that did not previously exist.

To avoid this problem we instead can choose the arc for which the number of $\vec{\triangle}$s after the reversal is least.

We call this heuristic Triangle Delta. A potential problem with Triangle Delta could be the existence of a tournament that was not acyclic (and thus still had triangles), yet contained no arcs whose reversal would lower the number of $\vec{\triangle}$s. Lemma 2.1 proves that this situation is impossible.

Lemma 2.1. *Let T be a tournament. Then if T has a cycle, there exists an arc $e \in T$ such that reversing e will reduce the triangle count of T.*

Proof. See Appendix A.

We also considered a third approach, called Triangle Both: choose the arc with the highest triangle count, provided that it reduces the number of $\vec{\triangle}$s. Note that calculating the triangle count, and the change in $\vec{\triangle}$s, for every arc of the digraph requires $O(n^3)$ operations.

In a weighted tournament, the weight of a $\vec{\triangle}$ is the sum of the weights of its arcs. Therefore in such graphs, the triangle count of an arc is the sum of the weights of the $\vec{\triangle}$s it is involved in.

2.5 Degree Difference Algorithms We designed a new algorithm Degree Difference, that selects an arc to reverse based on a criterion that may be related to the triangle count. We select the arc $u \to v$ for which the difference between u's indegree and v's indegree is greatest. Unfortunately, it may take $\Theta(n)$ time to find such an arc at each iteration. Nevertheless, this algorithm always makes progress towards a total ordering (which we use as our solution), as the value of $\sum_v \mathrm{In}(v)^2$ increases whenever an arc from a higher-degree to a lower-degree node is reversed.

In an effort to speed up the Degree Difference algorithm, we used a sampling technique. We sample $\log n$ vertices (favoring high indegree) to potentially be the 'tail' of the arc, and another $\log n$ (favoring low indegree) to potentially be the 'head'. We then check each of the $\log^2 n$ arcs between sampled vertices, choosing the back-arc of highest degree difference. We resample if no back-arc of non-negative degree difference is found. This algorithm is called Degree Difference Sampled 1, and it takes approximately $O(n^2 \log^2 n)$ time on average.

A further variation, Degree Difference Sampled 2, maintains two lists: one of potential head nodes, and one of potential tail nodes. A node v is a potential head if its indegree, $\mathrm{In}(v)$, is not unique or there is no node of indegree $\mathrm{In}(v)-1$. Similarly, a node u is a potential tail if $\mathrm{In}(u)$ is not unique or there is no node of indegree $\mathrm{In}(u)+1$. The motivation for this is to increase $\sum_v \mathrm{In}(v)^2$. We sample $\log n$ nodes from each list uniformly, and from those pairs select the arc with the largest indegree difference to reverse.

3 Approximation counter-examples

We now show that various algorithms for FAS cannot guarantee reasonable factor approximations.

A word on notation: All graphs shown are tournaments (complete graphs), so in the interest of readability, not all arcs are drawn. In the figures below, only back-arcs, with respect to the given ordering, are displayed. So all pairs of nodes with no arc displayed are assumed to have a right pointing arc between them.

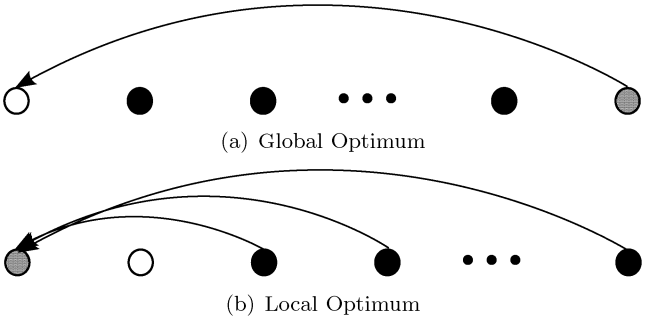


Figure 1: Standard Bad Example

3.1 Standard bad example This example consists of a completely transitive tournament of size n, with one minor perturbation—there is a single back-arc, from the last node (node n) to the first (node 1). Figure 1(a) shows this (global) optimum configuration; a local optimum for the Swaps heuristic, shown in Figure 3.1, has cost $n-2$.

Note also that there is no guarantee that Bubblesort will start with node n placed after node 1. Without this, it will also reach the costly local optimum.

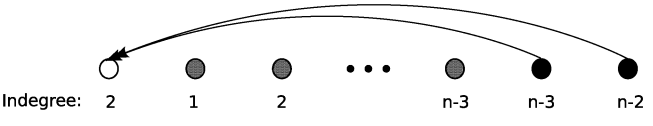
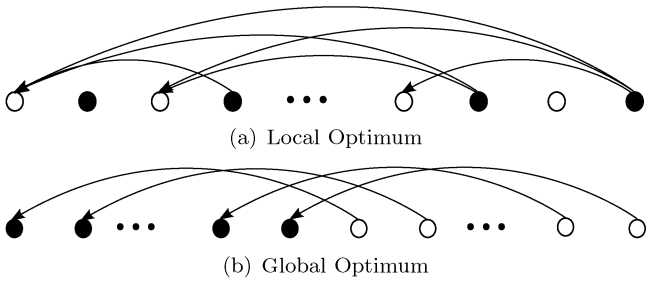


Figure 2: The Counterexample for the Eades algorithm.

3.2 The Eades algorithms Figure 2 shows a modification to the standard bad example of Section 3.1, in which the optimum solution has two back-arcs: from nodes $n-1$ and n to node 1. Displayed below each node is its indegree. The Eades and Eades Improved algorithms both place node 2 at the left of their solution (as it has the lowest indegree); with that node removed,

the induced subgraph is precisely the same as the original one, albeit one node smaller. The final order will therefore be $(2, 3, 4, 5, \ldots, n-1, n, 1)$, which has a cost of $n-3$.



(a) Local Optimum

(b) Global Optimum

Figure 3: Counterexample family (n even) for the MOVES local search heuristic (and thus for CHANAS & CHANAS BOTH). (a) There is a back-arc from each even-numbered node (black) to each odd-numbered node (white) that is at least 3 positions preceding. (b) Shows the global optimum. The black nodes have the same relative ordering to one another, as do the white nodes.

3.3 Moves and Chanas The configuration on the left of Figure 3 is a local optimum for the MOVES heuristic. Following the discussion at the end of Section 2.3, it is also a configuration that CHANAS and CHANAS BOTH can be stuck in.

The spacing of the back-arcs ensures that it is never an improvement to move a single node. The cost of the local optimum is $n^2/8 - n/4$. The global minimum, shown in Figure 3(b), places all black nodes before all white nodes, without changing the relative order within the color group, thus incurring a cost of $n/2$. So the *locality gap* here is in $\Omega(n)$.

4 Experiments

We conducted a series of experiments to validate the empirical performance of these algorithms. All experiments were conducted on 4-core Intel Xeon 3.2GHz machine, with 8 gigabytes of physical memory. All algorithms were compiled by `gcc` version 3.4.6 with the `-O3` optimization flag.

In order to investigate the significance of initial solution quality to the effectiveness of local search techniques, we first tested each algorithm in isolation—for the local search algorithms, this meant starting from a random ordering—and then passed the output of each algorithm into both the CHANAS and MOVES algorithms.

Note that passing CHANAS as input to CHANAS is an interesting case—although CHANAS is a local search algorithm (which would imply that a second run would have no further effect), the SORT* $\circ$ REVERSE step can significantly change the ordering without affecting the solution quality (in terms of number of back-arcs). So repeated calls to CHANAS can sometimes get the algorithm out of a local plateau. However, it is difficult to tell when this is going to happen.

4.1 Datasets We tested the FAS algorithms on the following synthetic dataset.

Biased Starting with a total order from nodes 1 to n, we reverse each arc independently with probability p. In particular, with $p = 0.5$, we have a random tournament.

The following datasets are based around the idea of aggregating inconsistent rankings of a set of datapoints into a FAS-style tournament.

WebCommunities Our colleague, Laurence Park, provided us with a set of 9 different complete rankings of a large set of documents (25 million) [22]. From this we took 50 samples of 100 documents and considered the rankings of each of those subsets.

EachMovie We used the EachMovie collaborative filtering dataset [20] to generate tournaments of movie rankings. The idea here was to identify subgroups (we used simple age/sex demographics) of the users, and then generate tournaments that represented the 'consensus view' of those groups.

The EachMovie dataset consists of a vote (on a scale of one to five) by each user for some set of the movies. To form a tournament from a group we took the union of movies voted for by that group and then set the arc weight from movie a to movie b to be the proportion of users who voted a higher than b. For consistency, we sampled each tournament down to size 100.

5 Discussion of Results

We tested all of the algorithms on a large number of data sets. We selected just four of the data sets to display in Table 1: these show a variety of performance characteristics. We first note the striking performance of the CHANAS local search procedure. Despite coming with an approximation guarantee, the QUICKSORT procedure performs relatively poorly, as does the MERGESORT algorithm. BUBBLESORT does surprisingly well on the **WebCommunities** dataset, but only after the CHANAS procedure is applied; otherwise it is possibly the worst of the algorithms. The EADES and EADES IMPROVED algorithms are strong, but there should be a slight preference for the latter due to its lower running time.

As expected, the TRIANGLE BOTH algorithm is very slow, though it is a strong performer when combined

Table 1: Each algorithm is tested on the Biased data set with $p = 0.6$ and 0.95. We also tested them on the WebCommunities and EachMovie data sets, as described above. In addition to the basic algorithms, we ran MOVES and CHANAS-style local search procedures on the outcomes of each of these algorithms. We report the average of the *percentage* (%) relative difference between the number of back-edges (**Errors**), compared to the CHANAS heuristic, over all problem instances in the dataset. We also report the percentage (%) of times each algorithm wins (produces the best amongst the solutions generated), with the win split between algorithms that are equal first on a particular instances (**Wins**). Finally, the average running time (in seconds) of each procedure (including the local search cleanups) is reported separately **Time**.

		0.6			0.95			laurence			eachmovie		
	Variant	**Errors**	**Wins**	**Time**	**Errors**	**Wins**	**Time**	**Errors**	**Wins**	**Time**	**Errors**	**Wins**	**Time**
	—	12.02	0.0	2.1	29.84	0.0	1.9	15.70	0.0	0.1	28.96	0.0	0.3
IT. KEND.	MOVE	0.37	6.7	4.9	0.30	5.2	3.8	0.00	3.3	0.4	0.40	9.8	0.6
	CHAN	-0.31	9.7	6.7	0.00	7.3	4.5	0.00	3.7	0.3	-0.01	9.9	0.8
	—	9.89	0.0	4.7	62.17	0.0	4.7	31.49	0.0	0.2	34.78	0.0	0.7
EADES	MOVE	0.58	3.7	7.5	0.31	5.3	6.9	0.01	2.8	0.6	0.35	6.8	1.0
	CHAN	-0.25	8.3	9.7	0.00	7.3	7.3	-0.00	3.6	0.4	0.00	5.7	1.2
	—	8.65	0.0	1.9	52.23	0.0	1.9	19.45	0.0	0.1	37.65	0.0	0.3
EADES IMP.	MOVE	0.61	2.8	4.7	0.32	5.0	4.7	0.00	4.5	0.3	0.47	2.5	0.6
	CHAN	-0.17	6.6	6.8	0.00	7.3	4.7	0.00	3.8	0.3	0.04	10.0	0.8
	—	0.00	74.0	5.6	0.00	27.8	2.8	0.00	11.8	0.1	0.00	75.7	0.6
CHANAS	MOVE	0.00	32.8	7.5	0.00	12.1	5.3	0.00	5.0	0.3	0.00	35.3	0.8
	CHAN	0.00	6.3	7.8	0.00	7.3	5.5	0.00	3.7	0.3	0.00	6.0	0.9
	—	35.28	0.0	1.6	731.12	0.0	1.6	75.39	0.0	0.1	210.08	0.0	0.2
BUBBLE.	MOVE	0.76	3.9	4.6	0.21	6.7	3.4	0.00	2.9	0.3	0.25	11.8	0.6
	CHAN	0.02	4.2	7.2	0.00	7.2	3.6	-0.00	4.7	0.3	0.01	6.9	0.8
	—	23.25	0.0	1.6	130.91	0.0	1.6	0.87	0.7	0.1	65.73	0.0	0.2
MERGE.	MOVE	0.80	2.4	4.6	0.23	6.1	5.2	0.00	3.2	0.2	0.39	10.6	0.6
	CHAN	0.02	5.5	7.2	0.00	7.2	4.5	0.00	3.5	0.3	-0.01	14.4	0.9
	—	23.66	0.0	1.6	135.85	0.0	1.6	0.84	0.8	0.1	61.68	0.0	0.2
QUICK.	MOVE	0.79	3.4	4.6	0.22	6.6	4.4	0.00	3.3	0.2	0.39	7.2	0.6
	CHAN	0.03	4.8	7.1	0.01	7.2	4.5	0.00	3.5	0.2	-0.01	12.2	0.8
	—	2.12	0.2	345.4	0.06	21.5	208.9	0.01	9.6	2.7	1.13	7.2	31.2
TRI. BOTH	MOVE	0.24	11.6	347.9	0.05	9.9	211.1	0.00	4.7	2.8	0.05	17.9	31.5
	CHAN	-0.36	12.7	349.8	0.03	6.8	211.8	0.00	3.8	2.8	-0.16	19.5	31.7
	—	7.47	0.0	158.0	0.03	24.2	27.8	0.02	5.3	0.4	2.27	2.7	8.5
D. D.	MOVE	0.58	4.8	160.7	0.03	11.1	29.5	0.00	5.1	0.5	0.17	11.3	8.8
	CHAN	-0.17	5.0	162.8	0.00	7.3	29.8	-0.00	4.0	0.5	-0.10	17.6	9.0
	—	11.25	0.0	9.4	48.17	0.0	4.0	0.41	0.0	0.1	23.33	0.0	0.8
D. D. SAM. 1	MOVE	0.31	9.2	12.2	0.29	5.4	5.8	0.00	3.7	0.3	0.38	9.9	1.1
	CHAN	-0.36	11.1	14.0	0.00	7.2	6.1	-0.00	4.0	0.3	0.01	9.2	1.3
	—	10.75	0.0	15.6	100.18	0.0	6.0	0.84	0.0	0.1	27.05	0.0	1.2
D. D. SAM. 2	MOVE	0.35	7.6	18.3	0.28	5.7	8.4	0.00	3.3	0.3	0.36	3.9	1.6
	CHAN	-0.34	10.4	20.1	0.00	7.2	8.7	-0.00	3.9	0.3	-0.07	12.7	1.8
	—	0.85	11.8	17.6	0.28	12.0	11.5	0.00	12.5	0.7	0.45	29.4	1.9
MOVES	MOVE	0.87	1.9	19.4	0.28	5.5	14.3	0.00	4.7	0.8	0.47	8.4	2.2
	CHAN	-0.03	3.8	21.6	0.03	6.7	14.2	-0.00	3.9	0.8	0.08	10.1	2.3
	—	0.78	14.0	3.0	0.22	14.3	3.6	0.00	9.3	0.2	0.37	31.0	0.4
CHAN. BOTH	MOVE	0.79	3.3	4.8	0.22	6.5	5.4	0.00	3.5	0.3	0.38	10.6	0.7
	CHAN	-0.05	4.9	7.0	0.03	6.7	5.9	-0.00	4.0	0.3	0.04	11.7	0.8

with local search. The Degree Difference algorithm is similar, though at a different point on the performance/speed tradeoff. The sampling methods for Degree Difference, Degree Difference Sampled 1 and Degree Difference Sampled 2, seem a better compromise.

5.1 The Time-Effectiveness Tradeoff Figure 4 highlights the relative performance and efficiency of selected algorithms. On the Biased ($p = 0.6$) data set, the two algorithms that cannot be said to be worse than others are the hybrid of Degree Difference Sampled 1 and Chanas, and the hybrid of Iterated Kendall and Chanas. Chanas by itself, not shown in this picture, unsurprisingly takes less time than these two algorithms. These three algorithms are therefore the subject of further study.

There is a certain random component to all of these algorithms. For Iterated Kendall, there is less randomness in the algorithm, and this is borne out in the results. But for Chanas, the input is in fact a random ordering of the items. In Figures 5(a) and 5(b), we repeat each algorithm twice, four times, eight times, etc. to see whether spending greater computation time produces better solutions. Unfortunately for Iterated Kendall, there seems to be a relative stagnation in its effectiveness, especially on the **EachMovie** data. It is hard to differentiate between the hybrid Chanas and Degree Difference Sampled 1 algorithms. Naturally, one could run Wilcoxon-style non-parametric tests to show that one algorithm is significantly better than the other in a pure statistical sense. However, the difference may not be important, and it can be hard to compare algorithms that take slightly different running times. We leave the graphs themselves as the strongest evidence of the similarity.

6 Conclusion

In this paper we outlined the operation of a number of algorithms which aim to solve the FAS problem, extending them where possible and developing a variety of new algorithms. We analyzed some of algorithms from the perspective of algorithmic approximability, proving that they cannot be good approximators. These results complement existing results about algorithms which are proven approximators.

Additionally we examined each algorithm from a practical perspective, testing their performance on two sets of real world data, as well as synthetic data. We found in practice Chanas is a very effective algorithm, however using the output of a different algorithm as input to Chanas is more effective again.

The most effective algorithms to do this with were Iterated Kendall and Degree Difference Sampled 1, with the first being faster, whereas the second was more effective. We gave these algorithms more time (by repeated application), and found that Degree Difference Sampled 1 became more effective, though only slightly better than Chanas.

6.1 Further Work The Chanas Both algorithm runs in significantly reduced time in comparison to Chanas. However its performance is not as impressive. Perhaps there is a better order to investigate local moves than both algorithms that will run quickly, yet perform as well as Chanas.

The Degree Difference Sampled 1 algorithm performs well, yet there are many other ways of sampling the vertices to check indegrees. These could be investigated—leading potentially to both better performing algorithms and theoretical results about them.

Most of the algorithms outlined here work unmodified on non-tournament digraphs. However, some, for example Triangle Both and Degree Difference, will not work as currently specified, but perhaps analogous versions could be found that will. It would be profitable to test all these algorithms on non-tournaments in much the same way as in this paper.

An issue that has not been addressed in great detail in this paper is how the algorithms scale for larger data sets. Further work investigating the large-scale time-performance of these algorithms would be of value.

7 Acknowledgements

Many thanks to Laurence Park for providing the dataset for the **WebCommunities** set of experiments. Thanks to Compaq for the **EachMovie** dataset. Thanks to Peter Stuckey, Adrian Pearce and Nick Wormald for helpful advice and comments.

References

[1] Ailon, N., Charikar, M., and Newman, A. Aggregating inconsistent information: ranking and clustering. *Proceedings of the thirty-seventh annual ACM symposium on Theory of Computing* (2005), 684–693.

[2] Ali, I., Cook, W., and Kress, M. On the Minimum Violations Ranking of a Tournament. *Management Science 32*, 6 (1986), 660–672.

[3] Arora, S., Frieze, A., and Kaplan, H. A new rounding procedure for the assignment problem with applications to dense graph arrangement problems. *Mathematical Programming 92*, 1 (2002), 1–36.

[4] Arya, V., Garg, N., Khandekar, R., Meyerson, A., Munagala, K., and Pandit, V. Local Search Heuristics for k-Median and Facility Location Problems. *SIAM Journal on Computing 33*, 3, 544–562.

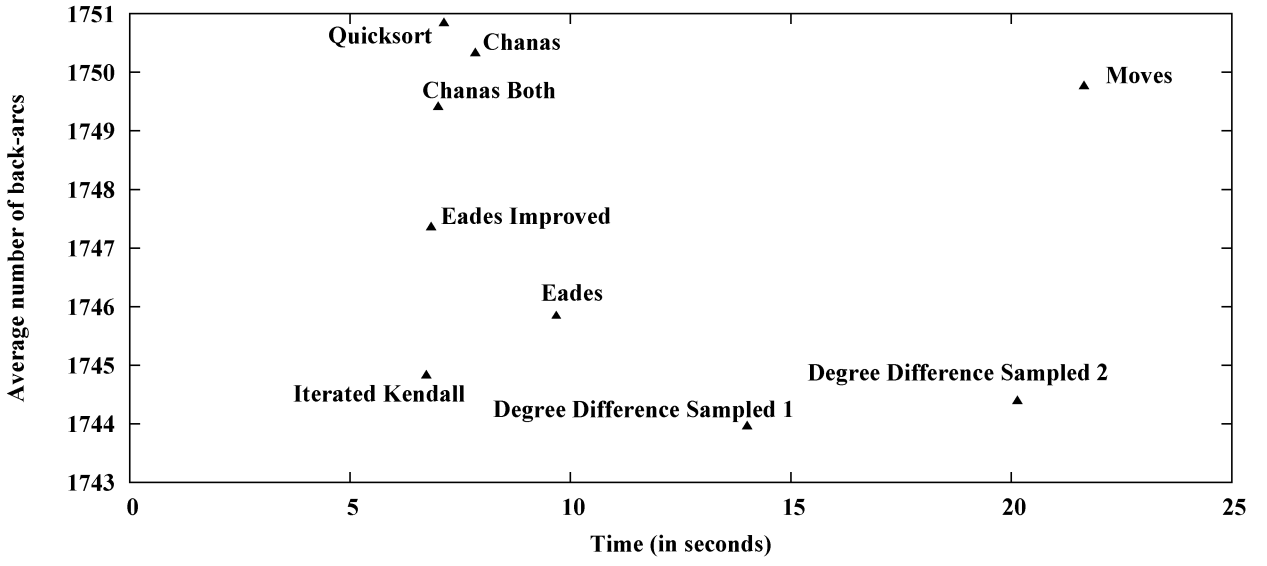
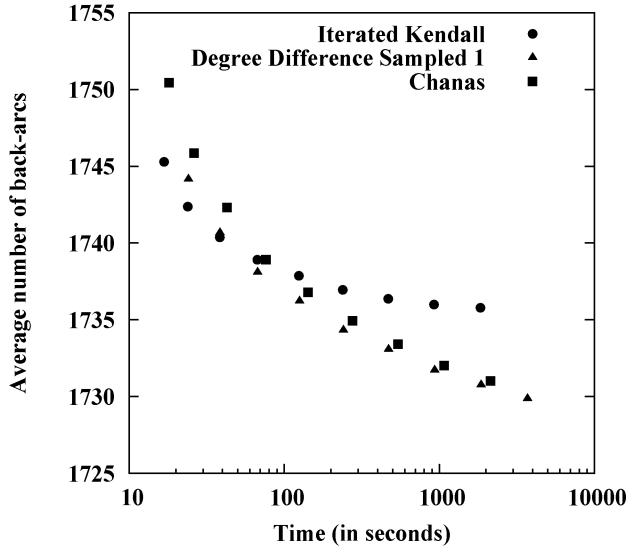
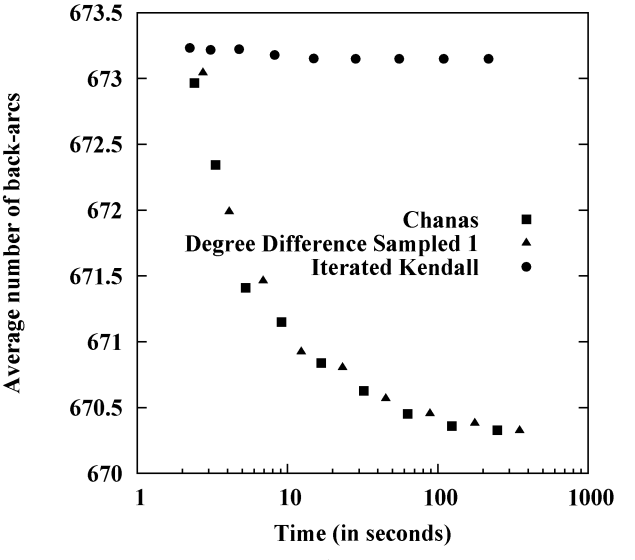


Figure 4: The amount of time taken compared to the effectiveness of the various algorithms as inputs to CHANAS. A point to the left indicates reduced running time; further downwards indicates less errors in the output ranking. The data shown is from the **Biased** dataset, $p = 0.6$, from a run of 1000 instances of size 100.



(a) For the **Biased** dataset, $p = 0.6$.



(b) For the **EachMovie** dataset (146 tournaments of size up to 100)

Figure 5: The effect of repeated calls to a hybrid of each algorithm and CHANAS. We ran each algorithm once, twice, four times, etc up to 256 times and took the best output ordering. The effectiveness and time taken (in seconds) are displayed.

[5] Aslam, J., and Montague, M. Models for metasearch. *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval* (2001), 276–284.

[6] Chanas, S., and Kobylański, P. A new heuristic algorithm solving the linear ordering problem. *Computational Optimization and Applications 6*, 2 (1996), 191–205.

[7] Charbit, P., Thomassé, S., and Yeo, A. The Minimum Feedback Arc Set Problem is NP-Hard for Tournaments. *Combinatorics, Probability and Computing 16*, 01 (2006), 1–4.

[8] Cook, W., Golan, I., and Kress, M. Heuristics for ranking players in a round robin tournament. *Computers and Operations Research 15*, 2 (1988), 135–144.

[9] Coppersmith, D., Fleischer, L., and Rudra, A. Ordering by weighted number of wins gives a good ranking for weighted tournaments. *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm* (2006), 776–782.

[10] Dwork, C., Kumar, R., Naor, M., and Sivakumar, D. Rank aggregation revisited. *Proceedings of WWW10* (2001), 613–622.

[11] Eades, P., Lin, X., and Smyth, W. A Fast and Effective Heuristic for the Feedback Arc Set Problem. *Information Processing Letters 47*, 6 (1993), 319–323.

[12] Eades, P., and Wormald, N. Edge crossings in drawings of bipartite graphs. *Algorithmica 11*, 4 (1994), 379–403.

[13] Johnson, D. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput. 4*, 1 (1975), 77–84.

[14] Kanungo, T., Mount, D., Netanyahu, N., Piatko, C., Silverman, R., and Wu, A. A local search approximation algorithm for k-means clustering. *Computational Geometry: Theory and Applications 28*, 2-3 (2004), 89–112.

[15] Karp, R. M. Reducibility among combinatorial problems. *Complexity of Computer Computations* (1972), 85–103.

[16] Kemeny, J. Mathematics without numbers. *Daedalus 88* (1959), 571–591.

[17] Kendall, M. Further Contributions to the Theory of Paired Comparisons. *Biometrics 11*, 1 (1955), 43–62.

[18] Kenyon-Mathieu, C., and Schudy, W. How to rank with few errors. *Proceedings of the thirty-ninth annual ACM symposium on Theory of Computing* (2007).

[19] Klein, D., and Randić, M. Innate degree of freedom of a graph. *Journal of Computational Chemistry 8*, 4 (1987), 516–521.

[20] McJones, P. Eachmovie collaborative filtering data set. *DEC Systems Research Center 249* (1997).

[21] Pachter, L., and Kim, P. Forcing matchings on square grids. *Discrete Mathematics 190*, 1-3 (1998), 287–294.

[22] Park, L., and Ramamohanarao, K. Mining web multi-resolution community-based popularity for information retrieval. In *Proceedings of the 2007 ACM 16th Conference on Information and Knowledge Management* (2007), pp. 545–52.

[23] Saab, Y. A Fast and Effective Algorithm for the Feedback Arc Set Problem. *Journal of Heuristics 7*, 3 (2001), 235–250.

A Proof of Lemma 2.1

Let σ be an ordering on the vertices of T that induces a minimum Feedback Arc Set. Let $a = v \leftarrow w$ be a back-arc of maximal length under σ, that is maximizing $\sigma(w) - \sigma(v)$. We claim that reversing a will lower the triangle count.

Firstly, we note that reversing a will not create any $\vec{\triangle}$s of the form v-w-x, where x is to the right of both v and w, and this would imply a back-arc $v \leftarrow x$ that is 'longer' than $v \leftarrow w$; this is impossible by our choice of a. Similarly, no $\vec{\triangle}$ x-v-w can be created where x is to the left of both vertices. So any $\vec{\triangle}$ created must involve an x between v and w.

Consider the four possibles for a node x that is placed between v and w by σ:

1. $v \leftarrow x \leftarrow w$ (reversing a will create a triangle). Say there are A such x's.
2. $v \rightarrow x \rightarrow w$ (reversing a will delete a triangle). B of these.
3. $v \rightarrow x \leftarrow w$ (reversing a will have no effect). C of these.
4. $v \leftarrow x \rightarrow w$ (no effect). D of these.

Since σ is optimal, moving v to the position after w will not reduce the back-arc count. So the number of back-arcs into v from such x's must be less than the number of forward arcs from v to such x's (strictly, as there is a back-arc from w to v). So we have

$$A + D < B + C\,.$$

Similarly, moving w before v will not improve the order, so we have

$$A + C < B + D\,.$$

Combining these quickly gives

$$2A + D + C < 2B + C + D \implies A < B\,,$$

and therefore the number of $\vec{\triangle}$s will decrease.

An Experimental Study of Recent Hotlink Assignment Algorithms

Tobias Jacobs*

Abstract

The concept of *hotlink assignment* aims at enhancing the structure of web sites such that the user's expected navigation effort is minimized. We concentrate on sites that are representable by trees and assume that each leaf carries a weight representing its popularity.

The problem of optimally adding at most one additional outgoing edge ("hotlink") to each inner node has been widely studied. A considerable number of approximation algorithms have been proposed and worst-case bounds for the quality of the computed solutions have been given. However, only little is known about the practical behaviour of most of these algorithms yet.

This paper contributes to close this gap by evaluating all recent strategies experimentally. Our experiments are based on trees extracted from real websites as well as on synthetic instances. The latter are generated by a new method that simulates the growth of a web site over time. We also propose a memory-efficient way to implement an optimal hotlink assignment algorithm, making it possible to compute optimal solutions for larger instances than before. Finally, we present a new approximation algorithm that is easy to implement and exhibits an excellent behaviour in practice.

1 Introduction

The design of web sites typically aims at making a large amount of information conveniently accessible. Web designers cannot arbitrarily distribute the contents among the pages as this would make information retrieval too complex for the users. The site's structure must rather somehow represent structural properties of the information. On the other hand, the interests of users on most web sites are highly correlated. Typically, about 80% of the users access only about 20% of the pages (cf. [14]). Moreover, the popularity of pages is likely to change over time.

By automatically moving popular pages closer to the home page one can both reduce web traffic and increase user-friendliness. However, restructuring the whole web site is not practicable for reasons mentioned above. In contrast, the concept of adding additional hyperlinks called *hotlinks* to the pages is a non-destructive approach which preserves the original site's structure.

We concentrate on web sites that are representable by a rooted tree, where the root is the home page, inner nodes represent navigation pages and the information is contained in the leaves, each having a certain popularity. This model has been widely studied in literature ([2, 3, 4, 5, 6, 9, 10, 11, 12, 13]) and a number of algorithms with provable good worst-case behaviour have been proposed. Our main purpose is to evaluate these algorithms experimentally and to make a statement about which of them are recommendable in practice.

Problem definition. A weighted tree T is a triple (V, E, ω), where (V, E) is a tree rooted at a node $r \in V$. Let $L \subseteq V$ be the set of leaves. The weight function $\omega : L \to \mathbb{R}_0^+$ assigns a non-negative weight to each leaf.

For nodes $u \in V$ we also write $u \in T$. The set of ancestors (descendants) of u, not including u itself, is denoted $\mathrm{anc}(u)$ ($\mathrm{desc}(u)$), $\mathrm{ch}(u)$ is the set of u's direct children and $\mathrm{par}(u)$ denotes the parent of u.

A *hotlink* (u, v) is an additional directed edge between nodes in T. We say that u is the *hotparent* of v and v is the *hotchild* of u. We further say that the hotlink *starts* in u and *ends* in v. A set $A \subset V \times V$ of hotlinks is called a *hotlink assignment (HLA)*.

We assume that a user only knows about hotlinks that start in nodes she has already visited, and she always uses any hotlink taking her closer to her destination leaf. This natural behaviour is called the *greedy user model* or *obvious navigation assumption*.

Referring to [13], a hotlink assignment A is called *feasible* if it satisfies the following properties:

(i) $v \in \mathrm{desc}(u)$ for any $(u, v) \in A$.

(ii) If $(u, v) \in A$, then there is no $(u', v') \in A$ with $u' \in \mathrm{desc}(u) \cap \mathrm{anc}(v)$ and $v' \in \mathrm{desc}(v)$.

(iii) For any node $u \in T$ there is at most one $(u, v) \in A$.

Properties (i) and (ii) exclude hotlinks that would never be taken by a greedy user. Property (iii) reflects the requirement that the number of hotlinks on a concise web page must be somehow limited. However, many algorithms naturally generalize to relaxations of that property. In the remainder of the paper, when talking about hotlink assignments, we always mean feasible HLAs.

*Department of Computer Science, University of Freiburg.

A hotlink assignment A is optimal if the *path length*

$$p(A) = \sum_{l \in L} \omega(l) \mathrm{dist}^A(r, l)$$

is minimal among all hotlink assignments for the tree under consideration, where $\mathrm{dist}^A(r, l)$ denotes the length of the greedy user's path from r to l in the graph $(V, E \cup A)$. We abbreviate $\mathrm{dist}^{\emptyset}(u, v)$ by $\mathrm{dist}(u, v)$. The task of finding an optimal hotlink assignment for a given tree is called the *Hotlink Assignment Problem*.

Equivalently, one can also maximize the *gain*

$$g(A) = p(\emptyset) - p(A) \ .$$

In spite of equivalence with respect to optimal solutions, the two optimization terms are not equivalent when considering approximation ratios. Most known polynomial algorithms for the hotlink assignment problem approximate only one of these terms up to a good ratio (cf.Table 3). On the other hand it is possible to combine the approximation properties of several algorithms by subsequently running each of them and then choosing the best among the computed hotlink assignments.

Related work. The hotlink assignment problem has first been formulated by Bose et. al. in [2]. They have shown that is is NP-hard when considering general graphs instead of trees.

To the author's best knowledge it is yet unknown if the optimal solution for trees can be computed in polynomial time. An optimal algorithm whose runtime and space requirements are exponential in the depth of the tree (and thus polynomial for trees of logarithmic depth) has been independently discovered by Gerstel et. al. ([7]) and Pessoa et. al. ([12]). An implementation of this algorithm has been confirmed in [13] to be able to find optimal solutions for trees having ten thousands of nodes and a depth up to 14.

The first approximation algorithm for the hotlink assignment problem has been presented in [10]. As in [4] and [13] that algorithm has already been shown to be outperformed by greedy-like strategies, we do not include it in our experiments.

Algorithm GREEDY has first been formulated in [3]. In [9] we have proven that it achieves at least half of the gain of an optimal solution in the greedy user model. Another 2-approximation in terms of the gain has been presented by Matichin and Peleg in [11]. Their algorithm assigns only hotlinks that bypass exactly one node. In [9] we have generalized this approach by showing that restricting hotlinks to some length h leads to a guaranteed approximation ratio of $\frac{h}{h-1}$ in terms of the gain. Using a modified version (called LPATH) of the optimal algorithm mentioned above, such hotlink assignments can be computed in polynomial time for constant h, so LPATH is a PTAS. The abovementioned algorithm of Matichin and Peleg is implicitly covered by the evaluation of LPATH for different values of h, including $h = 2$. By assigning the tree's depth to h we also obtain an efficient method to compute optimal solutions.

Another family of recent algorithms is more tailored to approximate the resulting path length. A lower bound for that optimization term has been given in [2]: Let Δ be the outdegree of the tree under consideration. Then no hotlink assignment can achieve a path length better than $H(\omega)/\log(\Delta + 1)$, where $H(\omega) = \sum_{l \in L} \omega(l) \log(1/\omega(l))$ is the entropy of the probability distribution among the leaves (assuming that the weights are normalized to sum up to 1). Douïeb and Langerman have presented two algorithms that guarantee a path length of $O(H(\omega))$ and are thus constant factor approximations for trees of constant degree. In [9] we have introduced a 2-approximation algorithm in terms of the path length for arbitrary trees.

Our contribution. The main intention of this work is to close the gap between theory and practice concerning the hotlink assignment problem. We have implemented all algorithms mentioned above and evaluated their performance by applying them to two different sets of trees. The first set contains instances that have already been used for the tests in [13]. We have extended this set by extracting 13 additional trees from German university web sites. These new instances contain up to 217.000 nodes and are thus about a factor of 5 larger than the instances from the original set. The trees contained in the second set have been generated by a new random construction method that is based on a model of Barabási and Albert ([1]) and generates more realistic instances than the methods used in previous experiments on hotlink assignment ([4]).

We have also implemented and evaluated a completely new approximation algorithm which is based on a heuristic given in [9]. It turns out that this method, in spite of its simplicity, performs excellently in practice. We will see that it is only up to 5% away from the optimal solution for almost all instances, which is only beaten by the PTAS.

Concerning the approximation scheme LPATH, we give an implementation whose memory requirements only depend on the depth of the tree. Thus we have been able to compute optimal solutions for all tree instances considered in [13] within 500MB of RAM. This improves upon the implementation of an optimal algorithm proposed in that work.

This paper is organized as follows: Some further

notation is given Section 2. That section also introduces the hotlink assignment algorithms we have evaluated experimentally. Section 3 describes the acquisition of our test instances and the experimental setup. The results of our study are presented in Section 4. Section 5 concludes.

2 Further notation and algorithms

In this section we introduce some further notation for our optimization problem. We then describe the hotlink assignment algorithms we have evaluated in our experiments and discuss some implementation issues. We use the terms *algorithm*, *strategy* and *method* interchangeably throughout the paper.

2.1 Notation For $u \in T$ we denote by $T(u)$ the maximal subtree of T rooted at u. For any set V' of nodes, $T(-V')$ is the maximal subtree of T that is obtained from T by removing all subtrees rooted at an element of V'. Furthermore, for any hotlink assignment A, let $T(u)(-A) = T(u)(-\{v \mid \exists (u', v) \in A, u' \notin T(u)\})$ be the subtree obtained from $T(u)$ by omitting all subtrees rooted at hotchildren of u's ancestors.

We define the weight of an inner node u as the sum over the weights of those leaves having u on the (greedy user's) path from r to them. Note that weights of inner nodes possibly change when hotlinks are assigned or modified. Formally,

$$\omega^A(u) = \begin{cases} \omega(u) & \text{if } u \text{ is a leaf} \\ \sum_{v \in \mathrm{ch}^A(u)} \omega^A(v) & \text{otherwise} \end{cases},$$

where $\mathrm{ch}^A(u)$ is the set of u's children and hotchildren in $T(u)(-A)$. We abbreviate $\omega^{\emptyset}$ by ω.

Next we introduce an order among the children of a node. The relation "$\succ$" is defined as a total order among siblings such that $\omega(u) > \omega(u') \Rightarrow u \succ u'$ for any pair u, u' of siblings. Ties are broken arbitrarily. The direct successor sibling of u with respect to that order is denoted by $\mathrm{succ}(u)$. The *first child* of an inner node is its unique child having no predecessor.

Finally, we define the *heavy path* of a node u. If u is a leaf, then its heavy path only consists of u. Otherwise, it is the path obtained by appending the heavy path of u's first child to u.

2.2 Algorithms We begin by a number of simple *top-down methods*. The latter term has been introduced in [6] (as well as the term "heavy path") and denotes hotlink assignment algorithms that assign a hotlink (r, v) and then recursively apply themselves to $T(v)$ and all $T(u)(-\{v\}), u \in \mathrm{ch}(r)$. Thus any top-down method is fully characterized by the choice of v.

GREEDY: Algorithm GREEDY has first been studied in [3] (where it is called "recursive"). It is a top-down method where the root's hotchild v is chosen such that $g(\{(r, v)\})$ is maximized.

GREEDY is a 2-approximation in terms of the gain ([9]). It has exhibited the best performance among the approximation algorithms studied experimentally so far ([3, 4, 13]).

H/PH: The H/PH-strategy is also a top-down method. Let h be the node whose weight is closest to $\omega(r)/2$. If $\omega(h) > \alpha\omega(r)$, then h is chosen as the hotchild of r. Otherwise the parent p_h of h becomes r's hotchild. The threshold α is given as the solution of $(\frac{\alpha}{1-\alpha})^{2(1-\alpha)} = \alpha$, i.e. $\alpha \approx 0.2965$.

The H/PH-algorithm has been proposed in [6], where it is proved to guarantee a path length of at most $1.141H(\omega) + 1$. Thus it is asymptotically a $(1.141 \log(\Delta + 1))$-approximation in terms of the path length, where Δ is the outdegree of the tree (cf. Section 1).

In [6] the authors also show that h is always located on the root's heavy path. Applying that observation makes the algorithm considerably faster.

PMIN: In [9], $p_{\min}(T)$ has been defined as the sum over the weights of all nodes in T except the root and except the heaviest child of each node. In this work we use a slightly modified definition: $p_{\min}$ is the sum over the weights of all nodes in T except the root and except the heaviest child of each node *but the root*. Formally,

$$p_{\min}(T) = \sum_{u \in V \setminus (L \cup \{r\})} (\omega(u) - \max_{v \in \mathrm{ch}(u)} \omega(v)) + \sum_{l \in L} \omega(l) \ .$$

PMIN is a top-down method which chooses the hotchild v of r such that

$$p_{\min}(T(v)) + \sum_{u \in \mathrm{ch}(r)} p_{\min}(T(u)(-\{v\}))$$

is minimized. It is a new algorithm and no bounds for its approximation ratios are known yet.

HEAVY PATH: Algorithm HEAVY PATH has been proposed in [5]. It works as follows: First split the tree into the set of heavy paths. This can be done in linear time by recursively computing the set of heavy paths of the subtrees rooted at the children of r, uniting these sets and appending r to the path containing the first child of r. Then interpret each of these paths as a list of weighted elements. The weight $W(u)$ of each such element u is $\omega(u) - \omega(v_f)$, where v_f is the first child of u in the tree. Then a HLA for each such list $u_1 \dots u_n$ is computed as follows: Assign a hotlink (u_1, u_i) such that $\sum_{1 \le j < i} W(u_j)$ and $\sum_{i < j \le n} W(u_j)$ are both at most

$\frac{1}{2}\sum_{1\le j\le n} W(u_j)$ and recursively apply the algorithm to the sublists $u_2 \dots u_{i-1}$ and $u_i \dots u_n$.

The whole algorithm can be implemented such that it takes linear time. Thus the heavy path strategy has the lowest worst-case runtime among all strategies evaluated in this work. It guarantees a maximum path length of $3H(\omega)$, making it a $(3\log(\Delta+1))$-approximation in term of the path length.

LPATH: The *length* of a hotlink (u, v) is defined as $\text{dist}(u, v)$. The best assignment of hotlinks that have a maximum length of h is a $\frac{h}{h-1}$-approximation in terms of the gain (cf. [9]). Algorithm LPATH, which for any given h computes a HLA that is at least as good as the best length h assignment in worst-case time $O(|V|3^h)$, is thus a PTAS in terms of the gain. In our experiments we also use LPATH as an optimal algorithm by setting h to the tree's depth.

LPATH is a dynamic programming algorithm. Subproblems are determined by a subtree together with a number of hotlinks starting outside that subtree that have already been decided to end in it. More formally, subproblems are defined by a triple (q, a, u), where $q = q_1 \dots q_n$ is a directed path, $a = a_1 \dots a_n a_{n+1} b \in \{0,1\}^{n+2}$ is a binary vector and u is node in T. It represents the tree qT_u obtained by appending the tree T_u to q_n. $T_u = T(\text{par}(u))(-\{u' \mid u' \succ u\})$ (not to be confused with $T(u)$) is the tree obtained by deleting all subtrees from $T(\text{par}(u))$ that are rooted at a sibling $u' \succ u$. The vector a represents a number of restrictions concerning HLAs for qT_u. Path nodes q_i are only allowed to have a hotchild if $a_i = 1$. They are not allowed to be hotchildren themselves. The node $r = \text{par}(u)$ is only allowed to be a hotparent if $a_{n+1} = 1$, and may only be a hotchild if $b = 1$.

For a given subproblem (q, a, u), LPATH deletes the first $|q|-h$ components of q and a in case of $|q| > h$. This only excludes hotlinks that would have a length greater than h. Then all possible configurations of hotlinks starting in nodes from q are compared. A configuration is determined either by one hotlink from q pointing at r, or by a selection of hotlinks starting at nodes in q that end somewhere in $T(u)$. In the latter case the remaining hotlinks end in $T_{\text{succ}(u)}$.

Each such configuration results in new subproblems, see [9] for details. As the solution of (q, a, u) only depends on a and u, the number of subproblems is limited by $|V|2^{h+2}$. Summing up the number of configurations for all subproblems results in a runtime of $O(|V|3^h)$ (cf. [13]).

Next we discuss a number of practical improvements to LPATH. The dynamic programming approach for the algorithm has been adopted from the PATH algorithm of Pessoa. Laber and Souza [12]. The only difference to LPATH is that, whenever the path q of a subproblem becomes longer than h, PATH gives up. In [13] Pessoa et. al. have proposed two improvements to their algorithm. The first is to increase the number of considered hotlink assignments by always cutting the first component from q and a until $a_1 = 1$. That improvement is already included in the original definition of LPATH, as the latter strategy always cuts the first components q and a when they become too long. The second improvement is a consequence of the observation, that for any subproblem (q, a, u) the total number of hotlinks starting from path nodes q_i is limited by the number of leaves in T_u. Let $\hat{a} = \sum_{1\le i\le|a|} a_i$ and let l_u be the number of leaves in T_u. We adopt the second improvement by inverting the $\hat{a} - l_u$ components of a having the highest possible indices from 1 to 0 whenever the $\hat{a} > l_u$.

In our experiments we have observed that the memory requirements of LPATH are by far more critical than the runtime is. For all tree instances the algorithm either terminated in reasonable time (2-3 minutes or less), or the memory requirements exceeded our hardware limit. So the purpose of our main improvement is to reduce space consumption.

The total number of subproblems considered by LPATH is $\Theta(|V|2^h)$. Fortunately, we do not need to store all of them simultaneously. For any fixed node u, let S_u be the set of solutions for all possible values of (a, u). Let u_f be the first child of u. Then $S_{\text{succ}(u)}$ and S_{u_f} suffice to compute any element of S_u. Furthermore, $S_{\text{succ}(u)}$ and S_{u_f} are not required for any further computation, so these sets can be removed from memory after computing S_u.

Let $u_1, \dots, u_n$ be the unique postorder sequence of $V - \{r\}$ where each node appears before its successor with respect to "$\succ$". We successively compute $S_{u_1}, \dots, S_{u_n}$ and delete any set as soon as it is not required anymore. It is easy to observe that with this policy any solution of a subproblem is guaranteed to be available when it is required. Note that the solution of the original problem is contained in S_{u_n}. The following lemma bounds the number of solution sets that are simultaneously stored.

LEMMA 2.1. *Let d be the depth of the tree. At any time during the execution of* LPATH*, for $1 \le x \le d$ there is at most one node u with $\text{dist}(r, u) = x$ whose solution set S_u is currently stored and not currently computed.*

Proof. Let u and u' be two nodes that have the same distance to the root and let u' occur after u in the postorder sequence. Let $S_{u'}$ be currently stored and completely computed. If u and u' are siblings, then S_u has been used for the computation of $S_{u''}$, where u'' is either u' or another sibling between u and u', so S_u has already

been deleted. If u and u' are not siblings, then the parent v of u occurs in the postorder sequence before u'. In that case S_v has already been computed, which implies that S_u has been used and thus deleted. □

As only one solution set is computed at a time, it follows directly from Lemma 2.1 that at most $d+1$ solution sets are simultaneously stored and thus the memory requirements of our implementation are $O(d2^h)$. The improvement is significant in practice, as the depth is typically small compared to the number of nodes (cf. Table 1).

CENTIPEDE: In a *centipede hotlink assignment* only the first child of each inner node is allowed to be bypassed by hotlinks. The best centipede HLA is a 2-approximation in terms of the path length ([9]).

From the centipede restriction follows that for each non-heaviest inner node u of a tree T the partial hotlink assignment for $T(u)$ is independent from that for T', where T' is obtained from $T(-\{u\})$ by appending a leaf of weight $\omega(u)$ to the the parent of u. So, when computing the best centipede hotlink assignment for T, the trees $T(u)$ and T' can be considered separately.

By applying this observation to each non-heaviest inner node, the algorithm CENTIPEDE splits the tree into a set of *centipede trees*, which are trees whose inner nodes have at most one non-leaf child. Optimal hotlink assignments for centipede trees are then computed in polynomial time by a dynamic programming algorithm, see [9] for a detailed description.

3 Experimental setup

3.1 Real instances Our set of real instances consists of 33 trees. 20 of them represent Brazilian university sites and have been made available by the authors of [13]. Their size (depth) ranges between 48 (4) and 57,877 (16).

The remaining thirteen instances represent websites of German universities, ranging from size (depth) 26,194 (5) to 217,213 (88). We have extracted the trees from the corresponding sites using breadth-first search, which implies that, for any page v in the original structure, a shortest path from the home page to v becomes the unique path from r to v in the resulting tree. In order to guarantee that no page occurs more than once, our algorithm maintains two hash tables: One for the web addresses already visited and one for the contents of the corresponding pages.

3.2 Synthetic instances As we want to make reliable statements about the algorithms' behaviour for different tree sizes, we need a large number of test instances. The available 33 trees extracted from web sites

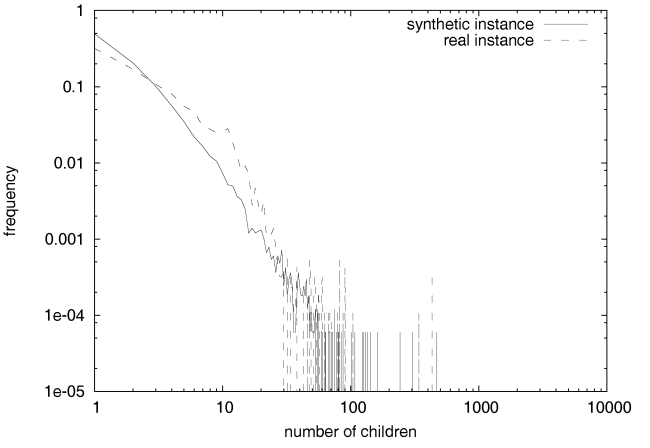


Figure 1: Distribution of the number of children in real and synthetic instance.

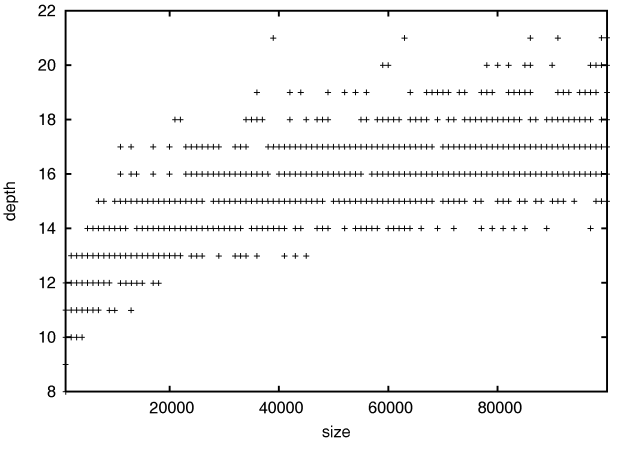


Figure 2: Relation between size and depth in the synthetic trees.

are not sufficient for that purpose. In [13], Pessoa et. al. increase the number of instances by considering each subtree of minimum depth 3 that is rooted at a node in one of their original trees. This approach causes strong dependencies in the data set and is therefore problematic in our opinion.

Instead, we randomly generate a large number of synthetic trees. Czyzowicz et. al. ([4]) have observed that the outdegree Δ of inner nodes follows a power-law, i.e. $P[\Delta = i] \sim i^{-\Delta}$. They have chosen the exponent $k = 2.72$ from literature about the graph structure of the World Wide Web. Their proposed algorithm maintains a FIFO-queue where all new nodes are stored. In each iteration step a node is removed from that queue and is assigned a random number i of new nodes as its children, where i is chosen according to the abovementioned power-law. The algorithm terminates

as soon as the desired number of nodes has been generated.

In our opinion there are two problems with that method. The first is that the leaves of any possible output tree are basically all on the same level, which is a consequence of the breadth-first and top-down manner of the construction. The second problem is that, according to our given data sets, the chosen value of k does not seem to be adequate. Maximum-likelihood-estimation of that parameter (see e.g. [8]) based on our real instances resulted in $k \approx 1.78$. This value is also problematic as for $k \leq 2$ the expectation of the number of children is infinity, which is unrealistic for trees having a limited number of nodes.

We have therefore developed a new algorithm that is based on a model of Barabási and Albert ([1]). In their model a graph, initially containing a small number of m_0 nodes, is built by in each iteration step i adding a new node v_i adjacent to m existing nodes. The probability for any node v to be chosen as one of v_i's neighbors is proportional to the number of nodes already adjacent to v. The authors show that for large i the number of neighbors of a node converges against a power-law distribution. In our case $m_0 = m = 1$, so that the resulting graph is a tree. Unlike the algorithm of Czyzowicz et. al., our construction generates trees with leaves at all levels.

The data set generated for our experiments consists of trees having sizes $1000, 2000, \ldots, 100000$. For each of these size values ten instances were generated, so their total number is 1000.

In these synthetic trees the distribution of the nodes' outdegree is similar to the typical distribution in real instances, as Figure 1 shows. Figure 2 displays the depth of the generated trees. It turns out that the depth tends to grow only very slowly with the size.

3.3 Assigning weights As no user access patterns are available to us, we have randomly assigned weights to the leaves of both the synthetic and the real trees. We did so using Zipf distribution, i.e. the ith heaviest leaf is assigned a weight of $\frac{1}{iH_m}$, where H_m is the mth harmonic number and m is the number of leaves in the tree. Zipf distribution is considered as the typical access distribution, see e.g. [14]. The same approach has also been employed in [3] and [13].

3.4 Test environment We have run all algorithms described in Section 2 on both the real and the synthetic tree instances. In case of LPATH we have applied each configuration of $h = 2 \ldots 15$.

We have measured the path length of the resulting hotlink assignments as well as the runtime of the algorithms.

$\|V\|$	d	OPT	GREEDY	PMIN	H/PH	HEAVY PATH	CENTI-PEDE
48	6	53.41	53.41	53.41	50.58	49.10	51.14
117	4	21.77	21.77	21.77	14.12	0.00	15.38
320	7	42.73	40.88	42.68	29.68	31.72	41.66
369	5	30.39	30.39	30.25	12.15	12.74	24.13
443	10	49.72	49.54	44.81	37.37	30.97	40.24
542	6	26.53	26.03	24.86	7.15	6.44	22.35
556	4	12.70	10.98	12.70	0.32	0.00	9.69
646	5	28.12	27.26	28.12	14.91	11.99	26.35
746	9	44.74	44.56	44.52	31.24	29.70	41.16
842	10	51.67	50.25	50.70	38.02	33.41	41.58
1016	4	18.35	18.35	18.35	5.76	5.12	17.93
1100	8	46.07	45.96	45.21	27.36	34.14	43.81
1151	7	35.51	35.03	33.63	19.91	19.59	33.35
1158	7	26.53	25.93	25.88	16.81	6.07	22.30
1743	3	13.15	13.15	13.15	6.15	6.12	8.63
1892	9	36.99	35.98	36.28	19.14	22.40	32.98
2275	6	30.37	30.29	30.29	22.18	21.32	27.99
2317	6	21.38	20.75	21.36	8.43	5.57	18.65
10484	16	55.21	54.22	54.41	43.29	41.23	50.24
24986	16	48.22	47.07	47.85	33.27	31.56	45.72
26194	5	23.43	22.77	22.92	11.90	11.64	20.59
30890	56	38.81	37.98	38.14	27.83	25.50	34.54
45439	23	35.89	35.33	35.32	20.95	19.48	32.24
57877	10	39.99	37.20	39.71	25.21	24.74	36.39
78472	45	?	54.80	56.92	47.01	45.95	54.15
89894	41	?	45.86	45.99	32.25	31.02	43.66
96288	26	?	42.61	43.04	29.02	30.47	39.52
107591	37	?	49.12	49.09	39.70	34.29	49.21
110892	58	?	38.87	39.43	26.40	27.04	36.61
115044	42	?	45.27	44.14	28.54	31.23	40.68
120330	88	?	59.23	58.40	45.97	48.45	56.95
163237	40	?	48.28	48.39	34.73	35.39	?
217213	15	43.76	42.32	43.36	31.91	32.03	41.02

Table 1: Relative gain (%) concerning the real instances.

Based on the path length we have calculated a number of additional values. As defined in Section 1, the gain $g(A)$ of an assignment A is $p(\emptyset) - p(A)$. For reasons of comparability between different instances we have also computed the *relative gain* as $g(A)/p(\emptyset)$.

The main focus of our study lies on the approximation ratios that occur in practice. Therefore, for all instances where an optimal solution OPT could be computed by our implementation of LPATH, we have calculated $p(A)/p(OPT)$ as well as $g(OPT)/g(A)$. Observe that for the latter ratio it is irrelevant if the absolute or the relative gain is used.

4 Results

4.1 Relative gain Table 1 gives an overview of the algorithms' results on the real trees. With our improved implementation of LPATH optimal hotlink assignments for most of the trees could be computed in less than 1 hour without exceeding our memory limit of 500MB.

$\|V\|$	d	LPATH $h=2$	LPATH $h=3$	LPATH $h=4$	LPATH $h=5$	LPATH $h=6$	LPATH $h=7$
48	6	38.07	52.28	53.41	53.41	53.41	53.41
117	4	21.77	21.77	21.77	21.77	21.77	21.77
320	7	31.68	41.25	42.65	42.73	42.73	42.73
369	5	26.11	30.25	30.39	30.39	30.39	30.39
443	10	37.12	45.09	49.29	49.70	49.72	49.72
542	6	23.51	25.95	26.53	26.53	26.53	26.53
556	4	12.70	12.70	12.70	12.70	12.70	12.70
646	5	23.98	28.12	28.12	28.12	28.12	28.12
746	9	34.05	41.57	44.63	44.74	44.74	44.74
842	10	37.80	48.14	51.48	51.67	51.67	51.67
1016	4	18.35	18.35	18.35	18.35	18.35	18.35
1100	8	34.86	44.44	45.62	45.98	46.07	46.07
1151	7	29.20	33.47	35.51	35.51	35.51	35.51
1158	7	24.86	26.53	26.53	26.53	26.53	26.53
1743	3	13.15	13.15	13.15	13.15	13.15	13.15
1892	9	32.28	36.07	36.99	36.99	36.99	36.99
2275	6	28.75	30.37	30.37	30.37	30.37	30.37
2317	6	21.02	21.38	21.38	21.38	21.38	21.38
10484	16	39.49	51.93	55.03	55.19	55.20	55.21
24986	16	35.47	45.69	47.91	48.17	48.21	48.22
26194	5	22.30	23.43	23.43	23.43	23.43	23.43
30890	56	30.89	37.04	38.71	38.80	38.81	38.81
45439	23	28.99	34.10	35.74	35.87	35.88	35.88
57877	10	32.06	38.57	39.84	39.98	39.99	39.99
78472	45	38.61	51.06	55.56	57.29	57.61	57.79
89894	41	34.46	44.40	46.33	46.51	46.55	46.55
96288	26	32.43	40.52	43.18	43.48	43.59	43.59
107591	37	35.26	46.36	49.36	50.39	50.58	50.64
110892	58	30.03	37.12	39.35	39.67	39.71	39.72
115044	42	32.95	41.19	44.52	45.70	45.93	45.93
120330	88	39.23	53.71	57.63	59.16	59.69	59.94
163237	40	34.33	45.51	47.81	48.55	48.63	48.65
217213	15	32.72	41.00	43.41	43.67	43.73	43.75

Table 2: Relative gain (%) of LPATH concerning the real instances.

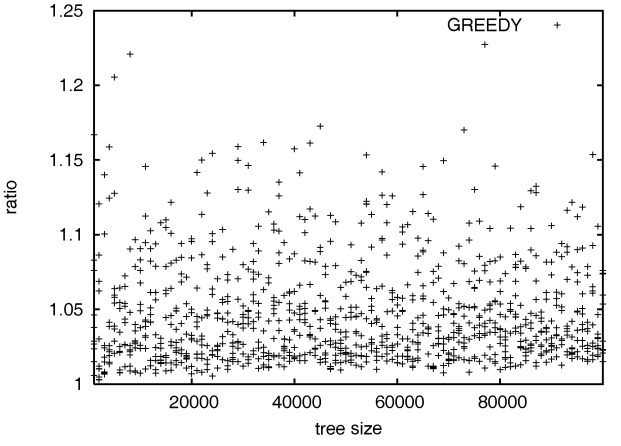


Figure 3: Ratio (gain) of GREEDY and PMIN for different tree sizes.

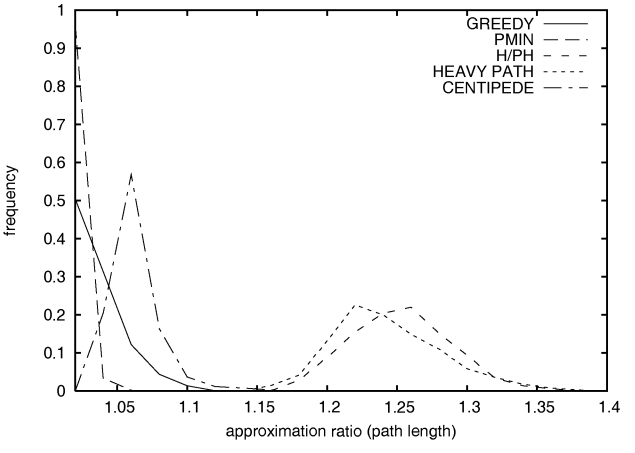


Figure 4: Histogram of the approximation ratio in terms of the path length

This significantly improves the best optimal algorithm previously known: EX-PATH, given in [13], even with a greater amount of memory available, failed for some instances that LPATH has been able to handle. However, for some of our new, larger instances, also LPATH could not compute an optimal assignment.

At first sight one can recognize that GREEDY and PMIN by far yield the best results, always being close to the optimum. In contrast, HEAVY PATH, H/PH and CENTIPEDE are far-off from being near-optimal. Interestingly, these are exactly the algorithms tailored to approximate the resulting path length. The performance of LPATH is only comparable to first-mentioned two strategies for values of h greater than 3.

4.2 Approximation ratios Table 3 compares the theoretical worst-case approximation ratios to the ratios that have been achieved on the real trees. For algorithms H/PH and HEAVY PATH no bound in terms of the gain had been given yet. However, trees having a depth of 2 where no hotlink at all is assigned by these strategies are easy to construct, so their ratios in terms of the gain do not exist. Almost all algorithms approximate the optimal solution's gain and path length up to the factor 1.5 or better. The only exceptions are H/PH and HEAVY PATH, partially having high factors for the gain.

The experiments based on the random instances allow for more detailed findings. Figure 3 exemplarily shows for the gain of GREEDY that the algorithms' solution quality is basically independent from the size of the tree. The picture looks similar for all algorithms and types of ratios.

The histogram in Figure 5 reveals that, concerning the gain, algorithm PMIN performs even better than GREEDY. For almost all instances PMIN is less and 5%

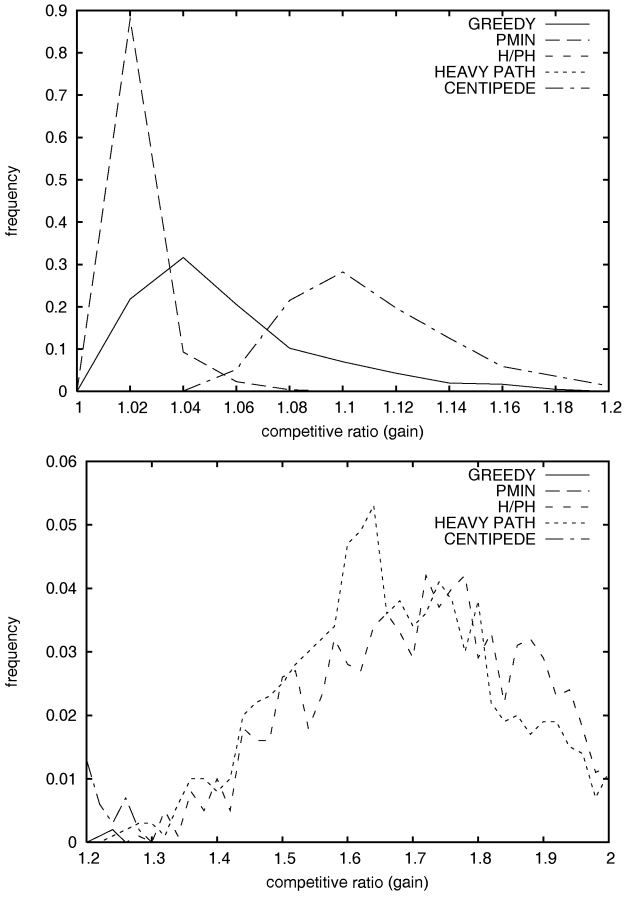


Figure 5: Histograms of the approximation ratio in terms of the gain

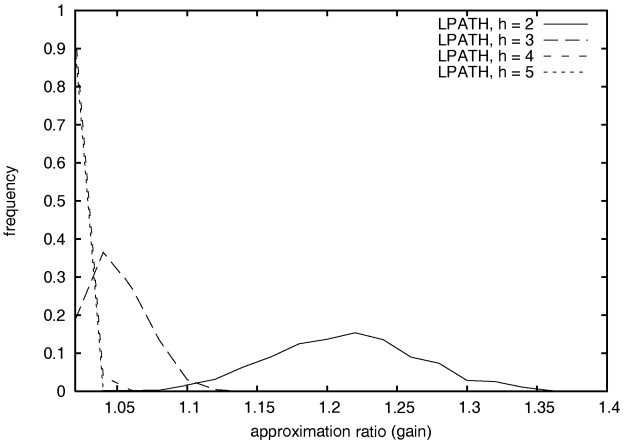


Figure 6: Histograms of the approximation ratio in terms of the gain for LPATH.

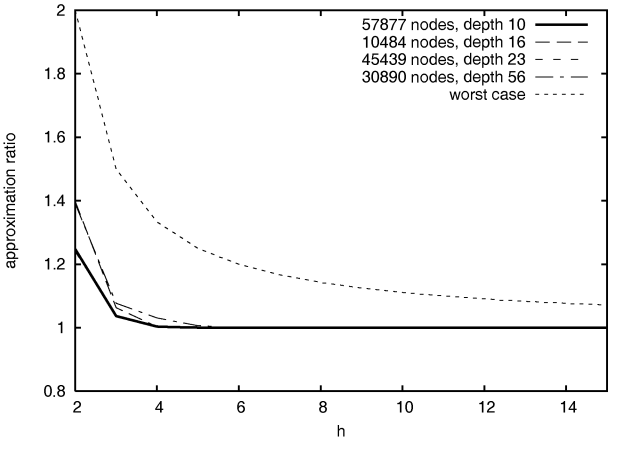


Figure 7: Approximation ratio resulted from running LPATH on tree instances for different values of h .

away from the optimal solution. The plots also show that CENTIPEDE is still quite robust compared to H/PH and HEAVY PATH. All these observations also hold for the path length (Figure 4).

Comparing Figure 5 and 6 confirms that LPATH, for $h = 2$, performs significantly worse than GREEDY. Recall that the former also simulates the approximation algorithm by Matichin and Peleg ([11]) and both strategies are 2-approximations in terms of the gain. For $h = 3$ the results are comparable to those of GREEDY. For $h > 3$ LPATH is very close to the optimal solution. This can also be observed in Figure 7. On each of the four selected instances of real trees the approximation ratio converges to 1 much faster than in theory. However, it seems that the speed of convergence is lower for trees having a greater depth d. Recall that for $h \geq d$ a ratio of 1 is guaranteed.

4.3 Runtime The runtime of LPATH, as expected, grows exponentially with h, up to some characteristic value (Figure 8). For h greater than that value the runtime remains constant. The characteristic value depends on the tree instance. It is typically slightly smaller than the tree's depth, when there are only few possibilities for long hotlinks.

The runtime of the other algorithms for processing the synthetic instances is depicted in Figure 9. The vertical axis is scaled logarithmically here in order to make all runtimes visible in one picture. Apparently, for each algorithm the runtime values form a point cloud that is quite compact, i.e. the runtimes are reliable. For almost all strategies the runtime grows linearly with the tree size, which for most of them is some orders of magnitude better than their worst case. The gradients, however, vastly differ. The fastest algorithm, as expected, is HEAVY PATH, followed by GREEDY, H/PH

	approximation ratio (gain)		
algorithm	worst case	min	max
GREEDY	2	1.000	1.156
PMIN	?	1.000	1.109
H/PH	∞	1.056	39.967
HEAVY PATH	∞	1.088	4.372
CENTIPEDE	∞	1.024	1.524
LPATH, $h = 2$	2	1.000	1.403
LPATH, $h = 3$	1.5	1.000	1.103
LPATH, $h = 4$	1.333	1.000	1.010
	approximation ratio (path length)		
algorithm	worst case	min	max
GREEDY	2	1.000	1.146
PMIN	?	1.000	1.098
H/PH	∞	1.061	1.347
HEAVY PATH	∞	1.081	1.378
CENTIPEDE	∞	1.005	1.209
LPATH, $h = 2$	2	1.000	1.351
LPATH, $h = 3$	1.5	1.000	1.092
LPATH, $h = 4$	1.333	1.000	1.009

Table 3: Theoretical worst-case approximation ratios and experimentally observed ratios. Infinity means that no constant bound exists.

and PMIN. The only algorithm having a superlinearly growing runtime is CENTIPEDE, as the right side of Figure 9 reveals.

5 Summary and conclusion

We summarize the most important findings of our study.

Algorithm PMIN performs excellent on all instances. It is also easy to implement, but its running time is quite high compared to other strategies. The analysis of its theoretical approximation ratios and the development of a more efficient implementation would be an interesting issue for future research.

The GREEDY strategy also exhibits a very good performance. It is slightly worse than PMIN in practice, but the guaranteed approximation ratio of 2 concerning the gain makes this algorithm a good choice. It is also easy to implement and runs faster than PMIN.

Although the path length seems to be the more natural optimization term, strategies tailored to approximate it are not the first choice in practice. The CENTIPEDE algorithm has a very high running time. As the quality of its solutions is only moderate, this algorithm is only advisable in practice if one wants a guaranteed approximation factor of 2 for the path length

Algorithms H/PH and HEAVY PATH exhibit the worst performance among all algorithms studied in this work. Anyhow, HEAVY PATH is extremely fast, making it interesting for quickly computing hotlink assignments for large trees.

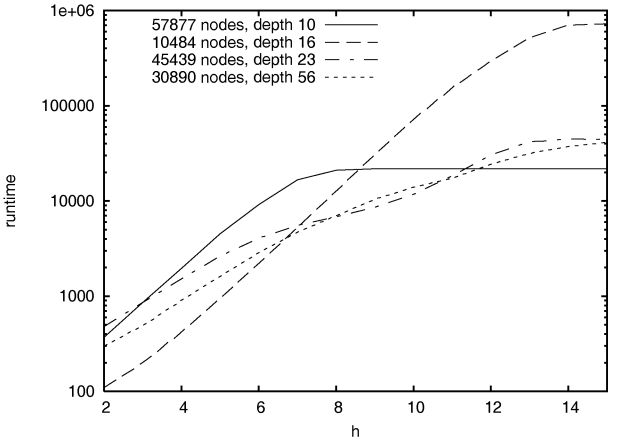


Figure 8: Runtime resulted from running LPATH on tree instances for different values of h .

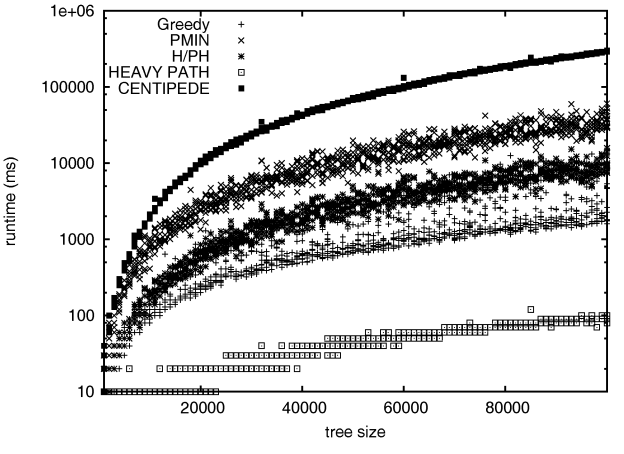


Figure 9: Runtime of the approximation algorithms, logarithmically scaled.

For small values of h the assignments achieved by LPATH are also not as good as the results of GREEDY and PMIN. So this PTAS is only recommendable in practice for $h > 3$.

Note that it is always possible to combine the properties of several algorithms. For example, by running PMIN, GREEDY and CENTIPEDE and then choosing the best among the three resulting assignments one would obtain a solution having the quality of PMIN with a guaranteed approximation ratio of 2 in terms both of the path length and the gain.

Acknowledgments: The author likes to thank Artur Alves Pessoa, Eduardo Sany Laber and Críston de Souza for making the dataset used in [13] available.

References

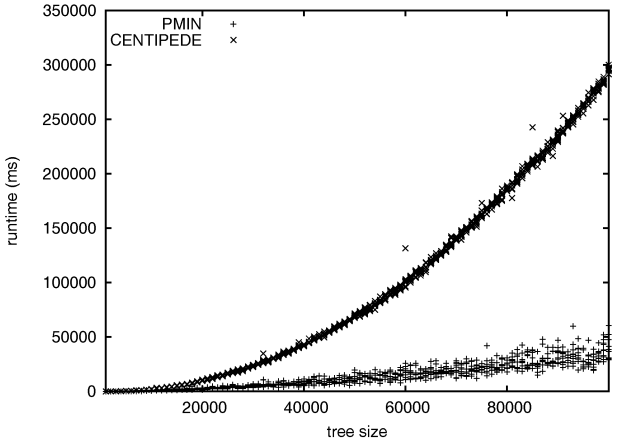


Figure 10: Runtime of CENTIPEDE and GREEDY, linearly scaled.

[1] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509, 1999.

[2] P. Bose, E. Kranakis, D. Krizanc, M. V. Martin, J. Czyzowicz, A. Pelc, and L. Gasieniec. Strategies for hotlink assignments. In *International Symposium on Algorithms and Computation*, pages 23–34, 2000.

[3] J. Czyzowicz, E. Kranakis, D. Krizanc, A. Pelc, and M. Martin. Evaluation of hotlink assignment heuristics for improving web access.

[4] J. Czyzowicz, E. Kranakis, D. Krizanc, A. Pelc, and M. V. Martin. Enhancing hyperlink structure for improving web performance. *J. Web Eng.*, 1(2):93–127, 2003.

[5] K. Douïeb and S. Langerman. Dynamic hotlinks. In F. K. H. A. Dehne, A. López-Ortiz, and J.-R. Sack, editors, *WADS*, volume 3608 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 2005.

[6] K. Douïeb and S. Langerman. Near-entropy hotlink assignments. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA 2006)*, volume 4168 of *LNCS*, pages 292–303, Zürich, Switzerland, 2006. Springer Berlin / Heidelberg.

[7] O. O. Gerstel, S. Kutten, R. Matichin, and D. Peleg. Hotlink enhancement algorithms for web directories: (extended abstract). In T. Ibaraki, N. Katoh, and H. Ono, editors, *ISAAC*, volume 2906 of *Lecture Notes in Computer Science*, pages 68–77. Springer, 2003.

[8] M. L. Goldstein, S. A. Morris, and G. G. Yen. Problems with fitting to the power-law distribution. *The European Physical Journal B - Condensed Matter and Complex Systems*, 41(2):255–258, 2004.

[9] T. Jacobs. Constant factor approximations for the hotlink assignment problem. In *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS 2007)*, Halifax, Canada, 2007.

[10] E. Kranakis, D. Krizanc, and S. Shende. Approximate hotlink assignment. *Inf. Process. Lett.*, 90(3):121–128, 2004.

[11] R. Matichin and D. Peleg. Approximation algorithm for hotlink assignment in the greedy model. *Theor. Comput. Sci.*, 383(1):102–110, 2007.

[12] A. A. Pessoa, E. S. Laber, and C. de Souza. Efficient algorithms for the hotlink assignment problem: The worst case search. In R. Fleischer and G. Trippen, editors, *ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 778–792. Springer, 2004.

[13] A. A. Pessoa, E. S. Laber, and C. de Souza. Efficient implementation of hotlink assignment algorithm for web sites. In L. Arge, G. F. Italiano, and R. Sedgewick, editors, *ALENEX/ANALC*, pages 79–87. SIAM, 2004.

[14] J. E. Pitkow. Summary of www characterizations. *World Wide Web*, 2(1-2):3–13, 1999.

Empirical Study on Branchwidth and Branch Decomposition of Planar Graphs

Zhengbing Bian* Qian-Ping Gu* Marjan Marzban* Hisao Tamaki† Yumi Yoshitake†

Abstract

We propose efficient implementations of Seymour and Thomas algorithm which, given a planar graph and an integer β, decides whether the graph has the branchwidth at least β. The computational results of our implementations show that the branchwidth of a planar graph can be computed in a practical time and memory space for some instances of size about one hundred thousand edges. Previous studies report that a straightforward implementation of the algorithm is memory consuming, which could be a bottleneck for solving instances with more than a few thousands edges. Our results suggest that with efficient implementations, the memory space required by the algorithm may not be a bottleneck in practice. Applying our implementations, an optimal branch decomposition of a planar graph of practical size can be computed in a reasonable time. Branch-decomposition based algorithms have been explored as an approach for solving many NP-hard problems on graphs. The results of this paper suggest that the approach could be practical.

Key words: Graph algorithms, branch-decomposition, planar graphs, algorithm engineering, computational study.

1 Introduction.

The notions of branchwidth and branch decompositions are introduced by Robertson and Seymour [20] in relation to the more celebrated notions of treewidth and tree decompositions [18, 19]. A graph of small branchwidth (or treewidth) admits efficient dynamic programming algorithms for a vast class of problems on the graph [4, 6]. There are two major steps in a branch/tree-decomposition based algorithm for solving a problem: (1) computing a branch/tree decomposition with a small width and (2) applying a dynamic programming algorithm based on the decomposition to solve the problem. Step (2) usually runs in exponential time in the width of the branch/tree decomposition computed in Step (1). So it is extremely important to decide the branch/tree-width and compute the optimal decompositions. It is NP-complete to decide whether the width of a given general graph is at least an integer β if β is part of the input, both for branchwidth [22] and treewidth [3]. When the branchwidth (treewidth) is bounded by a constant, both the branchwidth and the optimal branch decomposition (treewidth and optimal tree decomposition) can be computed in linear time [7, 9]. However, the huge constants behind the Big-Oh make the linear time algorithms only theoretically interesting.

One hurdle for applying branch/tree-decomposition based algorithms in practice is the difficulty of computing a good branch/tree decomposition because of the NP-hardness and huge hidden constants problems. Recently, the branch-decomposition based algorithms with practical importance for problems in planar graphs have been receiving increased attention [10, 11]. This is motivated by the fact that an optimal branch decomposition of a planar graph can be computed in polynomial time by Seymour and Thomas algorithm [22] and the algorithm is reported efficient in practice [13, 14]. Notice that it is open whether computing the treewidth of a planar graph is NP-hard or not. The result of the branchwidth implies a 1.5-approximation algorithm for the treewidth of planar graphs. Readers may refer to the recent papers by Bodlaender [8] and Hicks et al. [15] for extensive literature in the theory and application of branch/tree-decompositions.

Given a planar graph G of n vertices and an integer β, Seymour and Thomas give a decision algorithm (called ST Procedure for short in what follows) which decides if G has a branchwidth at least β in $O(n^2)$ time [22]. Using ST Procedure as a subroutine, they also give an algorithm which constructs an optimal branch decomposition of G. The algorithm calls ST Procedure $O(n^2)$ times and runs in $O(n^4)$ time. Gu and Tamaki [12] give an improved algorithm which calls ST Procedure $O(n)$ times and runs in $O(n^3)$ time to construct the branch decomposition. Hicks proposes a divide and conquer heuristic algorithm to reduce the number of calls for ST Procedure [14]. Computational studies show that the heuristic is effective in reducing the calls but has the time complexity of $O(n^4)$ [14]. All known algorithms for computing the optimal branch decomposition of a planar graph rely on ST Procedure; thus, an efficient implementation of the procedure plays a key role in com-

*School of Computing Science, Simon Fraser University, Burnaby BC, Canada V5A 1S6. {zbian/qgu/mmarzban}@cs.sfu.ca.

†Department of Computer Science, Meiji University, Kawasaki, 214-8571 Japan. {tamaki/yyoshi}@cs.meiji.ac.jp.

puting the branch decompositions. A straightforward implementation of ST Procedure requires $O(n^2)$ bytes of memory which is reported in [13] a bottleneck for solving large instances with more than 5,000 edges. Hicks proposes memory friendly implementations in the cost of performing re-calculations and increasing the running time of ST Procedure to $O(n^3)$ [13].

In this paper, we propose efficient implementations of ST Procedure. Our implementations can be classified into two groups. Group (1) does not perform re-calculations and runs in $O(n^2)$ time. The most memory efficient implementation in this group can compute the branchwidth of some instance of size up to one hundred thousand edges with 500Mbytes memory and in a couple of hours. Group (2) performs re-calculations and can compute the branchwidth of the instance of one hundred thousand edges with 200Mbytes of memory. The implementations in Group (2) may run in $O(n^3)$ time in the worst case. All of our implementations still use $O(n^2)$ bytes of memory. However, the constants behind the Big-Oh are much smaller than those in a straightforward implementation. In contrast, the results of this paper and those of [13] show that straightforward implementations can only handle instances of size up to about 5,000 edges within 1Gbytes of memory. Our most time efficient implementation is faster than the straightforward one by a factor of $3 \sim 15$. Compared with the previous memory friendly implementations of [13], our most memory efficient implementations of Group (1) and Group (2) use at most 1/4 memory and 1/8 memory and run faster by a factor of $100 \sim 400$ and a factor of $100 \sim 200$, respectively. Notice that the CPU used in [13] has frequency 194MHz and the CPU used for testing our implementations has frequency 3.06GHz, so we need to keep in mind this difference of speed when we compare the running time.

The results of this paper suggest that the memory size required by ST Procedure may not be a bottleneck for computing the branchwidth and optimal branch decomposition of a planar graph in practice. Our implementations also imply more efficient algorithms which call ST Procedure to find the optimal branch decompositions. We implemented the $O(n^4)$ time algorithm of Seymour and Thomas and the $O(n^3)$ time algorithm of Gu and Tamaki. The computational results show that the optimal branch decompositions of planar graphs with a few thousands edges can be computed in a couple of hours.

The rest of the paper is organized as follows. In Section 2, we give the preliminaries of the paper. We review ST Procedure and give some observations which provide the base of our efficient implementations in Section 3. Section 4 describes our implementations. Computational results are presented in Section 5. The final section concludes the paper.

2 Preliminaries.

Readers may refer to a textbook on graph theory (e.g., the one by West [24]) for basic definitions and terminology on graphs. In this paper, graphs are unweighted undirected graphs (i.e., each edge has a unit length) unless otherwise stated. Let G be a graph. We use $V(G)$ for the vertex set of G and $E(G)$ for the edge set of G. A *branch decomposition* of G is a tree T_B such that the set of leaves of T_B is $E(G)$ and each internal node of T_B has node degree 3. For each edge e of T_B, removing e separates T_B into two sub-trees. Let E' and E'' be the sets of leaves of the subtrees. The width of e is the number of vertices of G incident to both an edge in E' and an edge in E''. The width of T_B is the maximum width of all edges of T_B. The *branchwidth* of G is the minimum width of all branch-decompositions of G.

The algorithms of Seymour and Thomas [22] for branchwidth and branch decomposition are based on another type of decompositions called *carving decompositions*.

A carving decomposition of G is a tree T_C such that the set of leaves of T_C is $V(G)$ and each internal node of T_C has node degree 3. For each edge e of T_C, removing e separates T_C into two sub-trees and the two sets of the leaves of the sub-trees are denoted by V' and V''. The width of e is the number of edges of G with both an end vertex in V' and an end vertex in V''. The width of T_C is the maximum width of all edges of T_C. The *carvingwidth* of G is the minimum width of all carving decompositions of G. Notice that the carving decomposition is defined for more general graphs in [22]. The definition allows positive integer lengths on edges of the graphs. The width of e in T_C for the weighted graph is defined as the sum of lengths of edges with an end vertex in V' and an end vertex in V''.

Let G be a planar graph with a fixed embedding. Let $R(G)$ be the set of faces of G. The *medial graph* [22] $M(G)$ of G is a planar graph with an embedding such that $V(M(G)) = \{u_e | e \in E(G)\}$, $R(M(G)) = \{r_s | s \in R(G)\} \cup \{r_v | v \in V(G)\}$, and there is an edge $\{u_e, u_{e'}\}$ in $E(M(G))$ if the edges e and e' of G are incident to a same vertex v of G and they are consecutive in the clockwise (or counter clockwise) order around v. $M(G)$ in general is a multigraph but has $O(|V(G)|)$ edges. Seymour and Thomas [22] show that the carvingwidth of $M(G)$ is exactly twice the branchwidth of G and an optimal carving decomposition of $M(G)$ can be translated into an optimal branch decomposition of G in linear time. To decide whether a planar graph G has

the branchwidth at least an integer β, ST Procedure actually decides whether $M(G)$ has the carvingwidth at least 2β.

A face $r \in R(G)$ and an edge $e \in E(G)$ are incident to each other if e is a boundary of r in the embedding. Notice that an edge e is incident to exactly two faces. For a face $r \in R(G)$, a vertex v is incident to r if v is an end vertex of an edge incident to r. For a face $r \in R(G)$, let $V(r)$ and $E(r)$ be the sets of vertices and edges incident to r, respectively. For a vertex $v \in V(G)$, let $E(v)$ be the set of edges incident to v.

The *planar dual* G^* of G is defined as that for each vertex $v \in V(G)$, there is a unique face $r_v^* \in R(G^*)$; for each face $r \in R(G)$, there is a unique vertex $v_r^* \in V(G^*)$; and for each edge $e \in E(G)$ incident to r and r', there is a unique edge $e^* = \{v_r^*, v_{r'}^*\} \in E(G^*)$ which crosses e.

A *walk* in a graph G is a sequence of edges $e_1, e_2, ..., e_k$ of G, where $e_i = \{v_{i-1}, v_i\}$ for $1 \leq i \leq k$. A walk is *closed* if $v_0 = v_k$. The length of a walk is the number of edges in the walk. For two vertices u and v in a graph G, the distance $d(u, v)$ is the minimum length of all walks between u and v. The walk with distance $d(u, v)$ is a shortest path between u and v.

3 Seymour and Thomas procedure.

We give a brief review of ST Procedure and readers may refer to [22] for more details of the decision procedure. ST Procedure is often called the *rat-catching* algorithm because it can be intuitively described by a rat catching game introduced in [22]. We first review the game and then give a formal description of ST Procedure.

3.1 Rat catching game. In this game, there are two players, a rat and a rat-catcher. The game is on a planar graph G of a fixed embedding, with a face and an edge of G interpreted as a room and a wall of a room, respectively. The rules for the game are as follows.

(R1) The rat-catcher selects a room.

(R2) The rat selects a corner of a room (a vertex of G).

(R3) The rat-catcher selects a room adjacent to the current room and moves to the wall between the two rooms (the edge of G incident to the current face and the selected face). The rat-catcher generates a noise of a fixed level that may make walls noisy. The condition of making a wall noisy will be given later.

(R4) The rat moves to a different corner via walls or stays at the current corner. The rat can not use a noisy wall but can use as many quiet walls as possible in one move.

(R5) The rat-catcher moves to the room it selected and can not change its mind to move back to the previous room. The rat-catcher keeps making noise.

(R6) If the rat is in a corner, all walls incident to the corner are noisy, and the rat-catcher is in a room with this corner, then the rat-catcher catches the rat and wins the game. Otherwise goto (R3).

Now we give the condition on a wall becoming noisy. For the planar dual G^* of G, let v_r^* and e^* be the vertex and edge of G^* corresponding to the face r and edge e of G, respectively. Let k be the noise level produced by the rat-catcher. When the rat-catcher is on edge e, edge f is noisy if and only if there is a closed walk of length smaller than k containing edges e^* and f^* in G^*. Similarly, when the rat-catcher is in face r, edge f is noisy if and only if there is a closed walk of length smaller than k containing vertex v_r^* and edge f^* in G^*. The rat-catcher wins the game if the rat is at a vertex v with node degree smaller than k and the rat-catcher is in a face incident to v. The rat wins the game if there is a scheme by which the rat can escape from the rat-catcher for ever. We use $RC(G, k)$ to denote the rat catching game on G and k. Seymour and Thomas show that the rat wins the game $RC(G, k)$ if and only if G has carvingwidth at least k and give ST Procedure which, given G and k, computes the outcome of the game $RC(G, k)$ [22].

3.2 ST Procedure. Now we present ST Procedure using the language of the game $RC(G, k)$. Our presentation is different from the original one which is based on a notion called antipodality [22]. For a graph G with maximum node degree at least k, the rat always wins the game $RC(G, k)$ because the rat will never get caught if it stays at a vertex with node degree at least k. So we assume that G has maximum node degree smaller than k in the following discussion. Given G and k, ST Procedure computes an escaping scheme for the rat or decides no such scheme exists. The escaping scheme is represented by a collection of vertex subsets and subgraphs of G by which the rat can escape from the rat-catcher for ever. The collection contains a non-empty subset of vertices of G (a subset of corners) for every face and a non-empty subgraph of G (a subset of corners and quiet walls) for every edge.

Given G and k, we define G_e to be the subgraph of G obtained by deleting noisy edges from G when the rat-catcher is on edge e. More specifically, $V(G_e) = V(G)$

and

$$E(G_e) = \{f \mid \text{every closed walk of } G^* \text{ containing } e^* \text{ and } f^* \text{ has length at least } k\}.$$

Notice that every edge of G_e is quiet when the rat-catcher is on e. For each face $r \in R(G)$, we define

$$S_r = \{(r, v) | v \in V(G)\} \text{ and } S = \cup_{r \in R(G)} S_r.$$

For each edge $e \in E(G)$, we define

$$T_e = \{(e, C) | C \text{ is a connected component of } G_e\}.$$

Let $T = \cup_{e \in E(G)} T_e$. Then the game $R(G, k)$ can be described by a bipartite graph $H(G, k)$, where the vertex set of $H(G, k)$ is $S \cup T$ and there is an edge between $(r, v) \in S$ and $(e, C) \in T$ if face r is incident to edge e and v is a vertex of C. The vertices of $H(G, k)$ can be interpreted as the states of the game: a $(r, v) \in S$ represents that the rat-catcher is in face r and the rat is at vertex v, and a $(e, C) \in T$ expresses that the rat-catcher is on edge e and the rat is at a vertex of C and can use the edges of C to move. The edge between $(r, v) \in S$ and $(e, C) \in T$ indicates the possible state transitions of the game: when the rat is at v and the rat-catcher moves from face r to edge e, the game state transits from (r, v) to (e, C); or when the rat is at some vertex of C and moves to v, and the rat-catcher moves from edge e to face r, the game state transits from (e, C) to (r, v).

A game state of $R(G, k)$ is called a *losing state* if the rat will lose the game at the state. To compute an escaping scheme for the rat, ST Procedure deletes the losing states from $H(G, k)$. For a face $r \in R(G)$ and a vertex v incident to r ($v \in V(r)$), (r, v) is a losing state because the rat gets caught if the rat is at v and the rat-catcher is in r. For an edge e incident to face r, (e, C) is a losing state if for every vertex v of C, (r, v) is a losing state. To see this, assume that the rat-catcher is on edge e and the rat is at some vertex of C. The rat-catcher may move to r or r', the other face incident to e, in the next step. In either of the moves, the rat can only move to a vertex v of C. If the rat-catcher moves to r then the game transits to (r, v) at which the rat will get caught. If the rat-catcher moves to r' then the rat is at some vertex of C. The rat-catcher can move back to e and then to r, and the rat will get caught. Similarly, if (e, C) is a losing state then for every face r incident to e and every vertex v of C, (r, v) is a losing state.

For every edge $e \in E(G)$, ST Procedure initializes set X_e to include all states of T_e. For every face $r \in R(G)$, ST Procedure initializes set X_r to include all states of S_r and then deletes (r, v) from X_r for every $v \in V(r)$. After this initial deletion step, for each face r and each edge e incident to r, if there is a state (e, C) such that for every vertex v of C state (r, v) has been deleted, then the state (e, C) is deleted from X_e. If this deletion is done then for the other face r' incident to e, state (r', v) is deleted from $X_{r'}$ for every vertex v of C. This deletion may result in further deletions of losing states. The deletion process is repeated until no further deletion is possible. It is shown in [22] that graph G has carvingwidth at least k if and only if after the deletion process finishes, X_r and X_e are not empty for every $r \in R(G)$ and every $e \in E(G)$. The collection of non-empty X_r and X_e for every face r and every edge e is an escaping scheme for the rat. Below is a simplified version of the formal description of ST Procedure [22]. We remark that ST Procedure decides if the carvingwidth is at least k for more general planar graphs. It allows weighted input graphs with positive integer lengths on edges.

ST_Procedure

Input: A non-null connected planar graph G with a fixed embedding, a planar dual G^* of G, an integer $k \geq 0$.

Output: Decides if G has carvingwidth at least k.

1. If the maximum node degree of G is at least k then output G has carvingwidth at least k and terminate.

2. For each face $r \in R(G)$, let $X_r = S_r$.

   For each edge $e \in E(G)$, compute G_e and let $X_e = T_e$. For each $(e, C) \in X_e$ and the faces r and r' incident to e, let $c(r, e, C) = |V(C)|$ and $c(r', e, C) = |V(C)|$, where $V(C)$ is the set of vertices of C.

3. For each face r and each state $(r, v) \in X_r$ with $v \in V(r)$, put (r, v) to a stack L and delete (r, v) from X_r.

4. If L is empty then goto the next step.

   Otherwise, remove a state x from L.

   Assume that $x = (r, v)$ is a state for a face ($x \in S$). For each edge e incident to r, find the state $(e, C) \in X_e$ such that C contains v. Decrease $c(r, e, C)$ by one. If $c(r, e, C)$ becomes 0 and $(e, C) \in X_e$ then put (e, C) to L and delete (e, C) from X_e.

   Assume that $x = (e, C)$ is a state for an edge ($x \in T$). If there is a face r incident to e such that $c(r, e, C) > 0$ then for each vertex v of C and $(r, v) \in X_r$ put (r, v) to L and delete (r, v) from X_r.

   Repeat this step.

5. If X_r is non-empty for every $r \in R(G)$ and X_e is non-empty for every $e \in E(G)$ then output G has carvingwidth at least k, otherwise output G has carvingwidth smaller than k.

Notice that we can stop ST Procedure and conclude that the rat loses the game when some X_r becomes empty. The reason is that if all states of X_r are deleted, all states of X_e for e incident to r will be deleted; then all states for face r' incident to e will be deleted; and finally all states for every face and edge will be deleted. Similarly, the rat loses the game if some X_e becomes empty.

To compute G_e for each e, ST Procedure needs to find the quiet edges when the rat-catcher is on edge e. An edge f is quiet and will be included in G_e if every closed walk in G^* that contains e^* and f^* has length at least k. More specifically, let $e^* = \{u^*, v^*\}$ and $f^* = \{x^*, y^*\}$. Edge f is included in G_e if and only if $d(u^*, x^*)+d(v^*, y^*)+2 \geq k$ and $d(u^*, y^*)+d(v^*, x^*)+2 \geq k$. A solution for the all-pairs shortest path problem of G^* will suffice for the distances required in computing G_e for all $e \in E(G)$.

THEOREM 3.1. *(Seymour and Thomas [22])*
Given a planar graph G of n vertices and integer $k \geq 0$, ST Procedure decides if G has carvingwidth at least k or not using graph $H(G, k)$ in $O(n^2)$ time and $O(n^2)$ bytes of memory.

To decide the branchwidth of G, the input to ST Procedure is the medial graph $M(G)$ and the branchwidth of G is $k/2$ if the carvingwidth of $M(G)$ is k.

3.3 Observations for efficient implementations.

We give some observations on the game $RC(G, k)$ that can be used for efficient implementations of ST Procedure. By the definition of the game $RC(G, k)$, a state (r, v) is a losing state if $v \in V(r)$ for G with maximum node degree smaller than k. ST Procedure makes use of this sufficient condition to delete the losing states at the initial step of the deletion process for each face r. We observe that if we can find and delete more losing states at the initial step for each face r, then ST Procedure may run faster and use less memory. We prove the following sufficient condition for finding more losing states.

LEMMA 3.1. *For a face r and a vertex v in graph G with maximum node degree smaller than k, (r, v) is a losing state if there exist two faces s and t incident to v such that there are*
(1) a closed walk W_1 in G^* with length smaller than k that consists of the shortest path from v_r^* to v_s^*, the clockwise walk from v_s^* to v_t^* around r_v^*, and the shortest path from v_t^* to v_r^*; and (2) a closed walk W_2 in G^* with length smaller than k that consists of the shortest path from v_r^* to v_s^*, the counter-clockwise walk from v_s^* to v_t^* around r_v^*, and the shortest path from v_t^* to v_r^*.

Proof. Assume that W_1 and W_2 exist. Then for every edge e incident to v in G, e^* is either in W_1 or W_2 and e is noisy when the rat-catcher is in r. Let $e_1^*, ..., e_j^*$ be the edges in the shortest path from v_r^* to v_s^*. Assume that the rat is at v and the rat-catcher is in r. Since all edges incident to v are noisy, the rat can not move away from v. Next, the rat-catcher can move to edge e_1. Since all edges incident to v are noisy when the rat-catcher is on e_1, the rat has to stay at v. Similarly, the rat has to stay at v when the rat-catcher is on edge $e_i, 1 \leq i \leq j$. So the rat-catcher can move to face s using edges $e_1, ..., e_j$ and catch the rat at v. □

Once the shortest paths from v_r^* to all other vertices of G^* have been computed, it is easy to see the time for checking if (r, v) is a losing state by the condition of Lemma 3.1 is proportional to the node degree of v. Therefore, it takes $O(n)$ time to check (r, v) for a face r and all $v \in V(G)$. For each face r, let $U(r)$ be the set of vertices that for every $v \in U(r)$, (r, v) is a losing state computed by the sufficient condition of Lemma 3.1. From Theorem 3.1, we have the following result.

THEOREM 3.2. *Given a planar graph G of n vertices and $k \geq 0$, ST Procedure decides if G has carvingwidth at least k in $O(n^2)$ time and $O(n^2)$ bytes of memory when the losing states (r, v), $v \in U(r)$, are deleted at the initial step of the deletion process for each face r.*

For each face $r \in R(G)$, we define G_r to be the subgraph of G obtained by deleting the noisy edges from G when the rat-catcher is in face r. That is, $V(G_r) = V(G)$ and

$$E(G_r) = \{f \mid \text{every closed walk of } G^* \text{ containing } v_r^* \text{ and } f^* \text{ has length at least } k\}.$$

Notice that every edge of G_r is quiet when the rat-catcher is in r. Recall that G_e is the quiet subgraph of G when the rat-catcher is on edge e. Our next observation is that for every edge e incident to face r, $E(G_r) \subseteq E(G_e)$, because v_r^* is an end vertex of e^* and therefore the set of closed walks of G^* containing vertex v_r^* and edge f^* includes all closed walks of G^* containing edges e^* and f^*. From this, a component of G_r is a subgraph of some component of G_e. Hence, when the rat-catcher moves from face r to edge e and the rat is at any vertex of some component D of G_r, the component of G_e on which the rat can move is the

same one which contains D as a subgraph. Thus, when the rat-catcher is in face r, the states of the game can be expressed by

$$S'_r = \{(r, D) | D \text{ is a connected component of } G_r\}.$$

Let $S' = \cup_{r \in R(G)} S'_r$. The game $RC(G, k)$ can be described by a bipartite graph $H'(G, k)$, where the vertex set of $H'(G, k)$ is $S' \cup T$ and there is an edge between $(r, D) \in S'$ and $(e, C) \in T$ if face r is incident to edge e and D is a subgraph of C. For a face r and a component D of G_r, (r, D) is a losing state if for every vertex v of D, (r, v) is a losing state. For an edge e incident to face r, state (e, C) is a losing state if for every component D of G_r that is a subgraph of C, (r, D) is a losing state. Similarly, if (e, C) is a losing state then for every face r incident to e and every component D of G_r that is a subgraph of C, (r, D) is a losing state. Summarizing the above and from Theorem 3.1, the following result holds.

THEOREM 3.3. *Given a planar graph G of n vertices and $k \geq 0$, ST Procedure decides if G has carvingwidth at least k using graph $H'(G, k)$ in $O(n^2)$ time and $O(n^2)$ bytes of memory.*

When graph $H'(G, k)$ is used for the game $RC(G, k)$, X_r is initialized as S'_r for each face $r \in R(G)$ in ST Procedure. Compared with S_r, S'_r may have less game states and thus require less memory.

During the deletion process of ST Procedure, losing states are deleted from sets X_r and X_e. Our another observation is that the elements of X_e for an edge e incident to faces r and r' at a step of ST Procedure can be computed in $O(n)$ time from the elements of X_r and $X_{r'}$ at that step. This gives an option for implementing ST Procedure that does not keep but dynamically computes X_e from X_r and $X_{r'}$ during the deletion process.

THEOREM 3.4. *Given a planar graph G of n vertices and $k \geq 0$, ST Procedure can decide if G has carvingwidth at least k or not in $O(n^3)$ time and $O(n^2)$ bytes of memory if for each edge e, X_e is not kept but dynamically computed during the deletion process.*

Proof. For an edge e incident to faces r and r', the set X_e is needed when an element of X_r or $X_{r'}$ is deleted during the computation and when ST Procedure terminates. So, we can compute X_e in $O(n)$ time once there is an element deleted from X_r or $X_{r'}$. Since there are $O(n)$ elements in $X_r \cup X_{r'}$, X_e is computed $O(n)$ times. The total time for computing X_e for all $e \in E(G)$ is $O(n^3)$. From Theorem 3.1, the theorem holds. □

The re-calculation of edge data considered in this paper is different from the re-calculation in the previous study of [13], where face data are re-calculated for some faces and each re-calculation for a face r involves a computation of T_e for each e incident to r.

Finally, it is easy to see that if all states of S_r (or S'_r) for some face r are losing states then for every face r', all states of $S_{r'}$ ($S'_{r'}$) are losing states and the rat loses the game.

OBSERVATION 3.1. *If X_r becomes empty for some face r during the deletion process then graph G has carvingwidth smaller than k.*

By this observation we can terminate ST Procedure when some X_r becomes empty. This may save the computation time when the rat loses the game.

4 Efficient implementations.

Let G be a connected planar graph with a given embedding and $V(G) = \{v_1, ..., v_n\}$. We first describe a straightforward implementation (called *Naive*) of ST Procedure and then propose several improvements on the implementations of ST Procedure. Those improvements try to reduce both the memory space and running time of ST procedure.

4.1 Naive implementation.

A straightforward implementation of ST Procedure would use graph $H(G, k)$ for deciding the outcome of the game $RC(G, k)$. We use the following data structure for graph $H(G, k)$ in Naive.

- For each face $r \in R(G)$, a Boolean array B_r (of n elements) is assigned such that $B_r[i]$ is used to indicate if $(r, v_i) \in X_r$ or not. A list of $|E(r)|$ elements is used to keep the edges incident to r.
- For each edge $e \in E(G)$, the two faces r and r' incident to e are kept. All components of G_e are kept in a list. Each component of G_e is given an index and component C_j is kept in the jth element of the list. The element of the list for C_j contains the set of vertices of C_j, $c(r, e, C_j)$, $c(r', e, C_j)$, and a Boolean variable indicating if (e, C_j) has been deleted from X_e or not. An integer array I_e (of n elements) is used to indicate which component a vertex is in. If v_i is a vertex of C_j then $I_e[i]$ is set to j.
- In addition to the face and edge data, a stack L is used and a distance matrix is kept for the all pairs shortest distances in the dual graph G^* of G.

It is easy to check that the Naive implementation runs in $O(n^2)$ time. A simple calculation shows that Naive

implementation requires about $40n^2$ bytes of memory when G is a medial graph. Since there are many single vertex components in G_e and the operating system may have a minimum memory allocation size of 16 bytes, the memory usage in practice is close to $50n^2$ bytes.

4.2 Common improvements. We first describe two common improvements which are used in all of our efficient implementations. When we say *processing* a face r or an edge e, we mean deleting a losing state from X_r or X_e.

The first common improvement is that we define a processing order of the faces in our implementations. We put losing states (r, v) to the stack for only one face at a time. When there are losing states of multiple faces to be included to the stack, we group the losing states according to the faces, and give an order on the groups to be put to the stack. Only the group at the top of the order is put to the stack at a time. A face which has been processed is given a higher priority to be put to the stack. The processing order on the faces is used to define a subset $Q \subseteq R(G)$ and to restrict the rat-catcher moving within the faces of Q. Given a subset Q of $R(G)$, let $S_Q = \cup_{r\in Q} S_r$, $S'_Q = \cup_{r\in Q} S'_r$, and $T_Q = \cup_{e\in E(r), r\in Q} T_e$. We start with a small Q and perform the deletion process for the subgraph of $H(G, k)$ induced by the vertices of $S_Q \cup T_Q$ (or the subgraph of $H'(G, k)$ induced by the vertices of $S'_Q \cup T_Q$) until no deletion is possible. Then we enlarge Q by including a new face and repeat the deletion process. Q is enlarged gradually until $Q = R(G)$. By Observation 3.1, ST Procedure may stop at a small Q when the rat-catcher wins the game. Also, for a given subset Q, the losing states are deleted from X_r and X_e $(r \in Q, e \in E(r))$, and after the deletion, the data for X_r and X_e can be compressed before Q is enlarged. This helps in reducing the time and memory of ST Procedure.

The second common improvement is that we use a parsimonious data structure for edge data. We observe that there are many single vertex components in edge data. This makes the list of components for each edge very big. We keep the same face data as those in Naive. For each edge e, a component of G_e is called *non-trivial* if it has at least one edge otherwise called *trivial*. We only assign an index to a non-trivial component and keep a list of non-trivial components. We decide the integer type for I_e based on the number of non-trivial components in G_e. The length of the integer type for I_e is just big enough to encode the indices of non-trivial components of G_e. A trivial component $C = \{v_i\}$ is not kept in the list and $I_e[i]$ is used to indicate if (e, C) has been deleted from X_e or not. Further, if a non-trivial C_j has at least a constant fraction of n (δn) vertices then the set of vertices of C_j is not kept in the list. If there are at most a constant number (c) of non-trivial components then the sets of vertices of the components are not kept in the list. In these cases, when an access to vertices of a non-trivial component is needed, we check I_e to find the vertices of the component. It is easy to see that this does not increase the order of the time complexity of the implementation. A smaller δ saves more memory but may give a larger running time. Similarly, a larger c saves more memory but may increase the running time. We have chosen $\delta = 1/100$ and $c = 100$ in this study. A distance matrix is used to keep the all pairs shortest distances. We decide the integer type for the distance matrix based on the input integer k to ST Procedure. When G is a medial graph, we can reduce the required memory size to about $4n^2$ bytes if one-byte integer arrays are used for each I_e and the distance matrix, and to about $7n^2$ bytes if two-byte integer arrays are used.

4.3 More improvements.

Improvement A_1 This improvement is based on Theorem 3.2. In A_1, the elements (r, v), $v \in U(r)$, are deleted from X_r and put to the stack at the initial step of the deletion process for face r. From Lemma 3.1, $U(r) \supseteq V(r)$ and computational studies show that $|U(r)|$ is usually much larger than $|V(r)|$. Therefore, A_1 gives a room for improving both the running time and memory space.

Improvement D_1 The features of D_1 can be expressed by dynamic data creation and data compression. In D_1 the data for a face (edge) are created only when ST Procedure starts to process the face (edge). When some losing states are deleted, the face/edge data are compressed. More specifically, when ST Procedure is to perform the first deletion for a face r, $U(r)$ is computed and array B_r of n elements is created. After the losing states (r, v_i), $v_i \in U(r)$, are deleted from X_r, vertices of $V(G) \setminus U(r)$ are re-indexed and array B_r is compressed to indicate if (r, v_i) has been deleted for vertices v_i of $V(G) \setminus U(r)$ only. Similarly, when ST Procedure is to perform the first deletion for an edge e, G_e is computed and the edge data are created. Let r and r' be the two faces incident to e. We create two integer arrays I_e and I'_e for e. If the vertices of $V(G) \setminus U(r)$ have been re-indexed and B_r has been compressed then I_e is compressed accordingly. Similarly, array I'_e is compressed for the vertices of $V(G) \setminus U(r')$.
To calculate $U(r)$ and G_e, the shortest distances from vertex v_r^* to all other vertices in the planar dual graph G^* of G are needed for each face r in G.

When a distance matrix is used to keep the shortest distances, we need to solve $|R(G)|$ single source shortest path problems. In D_1, the distance matrix is discarded. When we process a face r, we create the data for r and the data for I_e for all e incident to r. Since each edge is incident to two faces r and r', the total number of single source shortest path calculations is bounded by $2|E(G)|$. When G has n vertices and is a medial graph, $|R(G)| = n + 2$ and $|E(G)| = 2n$. From this, if the distance matrix is used, we need to solve $n + 2$ single source shortest path problems while we need to solve at most $4n$ such path problems if D_1 is applied.

Combining D_1 with A_1, the required memory size is now about $5n \times q$ bytes if one-byte integer arrays are used for I_e and I'_e and about $9n \times q$ if two-byte integer arrays are used, where q is the average of $|V(G) \setminus U(r)|$. For the Delaunay triangulation instances tested, q is less than $0.3n$ (instances dependent).

Improvement A_2 This improvement is based on Theorem 3.3. For each face r, instead of S_r, A_2 initializes X_r to include all states of S'_r.

Improvement A_3 This improvement is based on Theorem 3.4 and performs re-calculation for edge data. A_3 keeps the face data once they are created but keeps the edge data for only a pre-defined maximum number of edges. Once this number is reached A_3 starts to delete the entire X_e for some edge e. If a deleted X_e is needed again, X_e is re-computed from X_r, where r is incident to e.

Improvement D_2 In D_2, we use a bit vector B_r for the data of face r, with one bit for one element of X_r. The memory size for face data is 1/8 of that when a one-byte Boolean array is used. But more complex bit operations have to be used.

It is easy to check that all improvements except A_3 do not change the order of running time of ST Procedure. However, applying A_3, the running time of ST Procedure may become $O(n^3)$.

5 Computational results.

All of our efficient implementations use common improvements. In our implementations with any of improvements A_2, A_3 and D_2, improvements A_1 and D_1 are always used. We do not mention A_1 and D_1 explicitly in those implementations. We test Naive and Implementations A_1, A_1D_1, A_2, A_2D_2, A_3, A_3D_2, A_2A_3, and $A_2A_3D_2$. Three classes of instances are used in the test. Class (1) of instances includes Delaunay triangulations of point sets taken from TSPLIB [17]. Those instances are used as test instances in the previous studies [13, 14]. The instances in Class (2) are generated by the LEDA library [2, 16]. LEDA generates two types of planar graphs. One type of the graphs are the randomly generated maximal planar graphs and their subgraphs obtained from deleting some edges. Since the maximal planar graphs generated by LEDA always have branchwidth four, the subgraphs obtained by deleting edges from the maximal graphs have branchwidth at most four. The graphs of this type are not interesting for the study of branchwidth and branch decompositions. The other type of planar graphs are those generated based on some geometric properties, including Delaunay triangulations and triangulations of points uniformly distributed in a two-dimensional plane, and the intersection graphs of segments uniformly distributed in a two-dimensional plane. We will report the results on the intersection graphs. The instances in Class (3) are generated by the PIGALE library [1]. PIGALE randomly generates one of all possible planar graphs with a given number of edges based on the algorithms of [21]. We use Naive and our implementations to compute the carving-width of the medial graphs of the instances (i.e., the input graph to ST Procedure is not an instance itself but the medial graph of the instance). Our implementations are tested on a computer with Intel(R) Xeon(TM) 3.06GHz CPU, 2Gbytes physical memory and 8Gbytes swap memory. The operating system is SUSE LINUX 10.0, and the programming language we used is C++.

We compute an upper bound on the carvingwidth as the initial guessed input integer k to call ST Procedure. It is known that the branchwidth of a planar graph of n vertices is at most $\sqrt{4.5n}$ [11]. From this, $2\sqrt{4.5n}$ is an upper bound on the carvingwidth of the medial graph of an instance of n vertices. We follow a similar approach in [13] to compute another upper bound l: Let $M(G)$ be a medial graph of a planar graph G of n vertices. For each face r of $M(G)$ which corresponds to a vertex in G, we compute the eccentricity of v_r^* (the length of the longest shortest paths from v_r^* to all other vertices) in the planar dual $M(G)^*$. We initialize l as twice as the minimum eccentricity among all v_r^*'s. Finally, we take $k = \min\{2\sqrt{4.5n}, l\}$. Either the linear search or the binary search can be used to find the carvingwidth starting from the initial guessed k. In the linear search, when the rat-catcher wins, k is decreased by two and ST Procedure is called again until the rat wins the game. In the binary search, we call ST Procedure to search for the carvingwidth between k (upper bound) and the node degree of $M(G)$ (which is four and a lower bound). For the instances in classes (1) and (2), the eccentricity-based guess is very close to the carvingwidth and k always takes the value of l. The linear search uses a

smaller number of iterations to find the carvingwidth than the binary search. For instances in Class (3), the eccentricity-based guess could be very large and k may take $2\sqrt{4.5n}$ for large instances. Since $2\sqrt{4.5n}$ is still far away from the carvingwidth, the binary search does a better job. One may run the linear search and binary search in parallel and take the results from the one which finishes earlier.

5.1 Computation time and memory. Table 1 shows the computation time of Naive and efficient implementations for the carvingwidth of the medial graphs of the instances in Class (1). In the table, *Itr* is the number of iterations in the linear search. Table 2 shows the memory size (in megabytes) of those implementations. Only the data for relatively large instances are given in the tables.

For the instances in Class (1), one-byte integer arrays are used for each edge and the distance matrix. The most time efficient implementation is A_1 which is faster than Naive by a factor of at least 10 and uses at most 1/10 memory of Naive. With more improvements, the memory requirement is further reduced but the running time is slightly increased. The effect of data compression in Improvement D_1 is significant. The memory used by A_1D_1 is only about $1/3 \sim 1/4$ of that by A_1. Improvement A_2 is effective in reducing the memory size. In general, the number of non-trivial components is small for both faces and edges, and thus the memory saving is big. The memory used by Improvement D_2 for face data is 1/8 of that when one-byte Boolean arrays are used. When the memory for face data becomes dominating, Improvement D_2 reduces memory requirement significantly. The most memory efficient implementation without re-calculation is A_2D_2 which is faster than Naive by a factor of $8 \sim 9$. For Instance *pla*33810 which has $101,367$ edges (corresponding to $101,367$ vertices in the input medial graph), A_2D_2 uses about 500Mbytes memory, which is about $\frac{1}{20}n^2$ bytes, where n is the number of vertices in the input medial graph. Compared with Naive, the memory saving is by a factor of about 1000.

Improvement A_3 performs re-calculation for edge data. The performance depends on the maximum number of edges that are kept. This maximum number can be chosen based on the size of available memory. In general, a larger maximum number gives an implementation which uses more memory but runs faster (less re-calculations). The maximum number of kept edges is 500 for the results in the paper unless otherwise stated explicitly. Among all implementations, $A_2A_3D_2$ is the most memory efficient one. $A_2A_3D_2$ is faster than Naive by a factor of $6 \sim 7$. For Instance *pla*33810, $A_2A_3D_2$ uses less than 200Mbytes memory, which is about $\frac{1}{50}n^2$ bytes. Compared with Naive, the memory saving is by a factor of about 2500. The memory used by Implementation $A_2A_3D_2$ can be further reduced to about 155Mbytes for Instance *pla*33810 with a slightly increase in the running time, if we keep at most 50 edges.

The instances in Class (2) are generated by the LEDA function *random_planar_graph* [2]. We have tested our implementations on instances of Delaunay triangulations and triangulations of points randomly distributed in a two-dimensional plane, and intersection graphs of segments. Our implementations have similar performances for the Delaunay triangulations and triangulations instances as those for the instances in Class (1). Table 3 gives the computation time and memory of Naive and Implementations A_1, A_2D_2, and $A_2A_3D_2$ for instances of intersection graphs of segments. In the table, *Itr* is the number of iterations in the linear search. The instances of intersection graphs of segments may have a large number of non-trivial components for edges and faces, and two-byte integer arrays are used to represent the edge data. Therefore, the memory usage is considerably larger than the Delaunay instances of the same size. As shown in the table, our efficient implementations are faster and use much less memory than Naive.

Instances of Class (3) are generated by the PIGALE library [1]. PIGALE provides a number of planar graph generators. Since 2-connected planar graphs are the most interesting class of graphs in the study of branchwidth and branch decompositions, we selected the function for generating 2-connected planar graphs. The function, given the number m of edges, randomly generates one of all possible 2-connected planar graphs of m edges. The output graph is usually a multigraph with parallel edges. Since parallel edges are not interesting for branchwidth finding, we specify the function to produce simple 2-connected graphs. With a given m, the function outputs a 2-connected random planar graph with m' edges. Normally m' is smaller than m, since parallel edges are not kept and there are performance considerations [1]. Table 3 gives the computation time and memory of Naive and Implementations A_1, A_2D_2, and $A_2A_3D_2$. In the table, *Itr* is the number of iterations in the binary search. The instances in Class (3) may have a small number of non-trivial edge components, but we still use two-byte integer arrays for the edge data. For this class of instances, the eccentricity-based guess is usually bad. For example, the medial graph of Instance PI37730 has carvingwidth 12, but the eccentricity-based guess is 8974. For large instances tested, the binary search always finishes earlier than the linear search. The

Table 1: Computation time (in seconds) of Naive and efficient implementations for Class (1) instances.

Instances	Number of edges	bw	Itr	Computation time (in seconds)								
				Naive	A_1	A_1D_1	A_2	A_2D_2	A_3	A_3D_2	A_2A_3	$A_2A_3D_2$
pr1002	2,972	21	2	51.2	4.23	5.38	6.07	6.35	5.58	5.83	6.52	6.98
rl1323	3,950	22	2	95.7	6.47	8.0	9.19	9.56	8.65	9.05	12.3	13.1
d1655	4,890	29	2	158	11.3	15.0	17.3	17.6	15.7	16.7	21.6	22.3
rl1889	5,631	22	2	195	13.8	16.6	20.1	20.8	19.2	20.4	29.5	30.5
u2152	6,312	31	3	X	24.5	35.5	41.8	42.4	39.9	42	61.6	62.4
pr2392	7,125	29	2	X	21.1	25.8	32.1	32.8	31.8	31.3	49.1	50.4
pcb3038	9,101	40	2	X	31.6	41.3	50.8	51.9	44.6	49.7	83.2	74.2
fl3795	11,326	25	2	X	63.7	80.2	99	104	86.3	98	155	163
fnl4461	13,359	48	2	X	67.4	92.4	116	119	97.1	110	185	185
rl5934	17,770	41	2	X	151	197	245	249	213	241	385	391
pla7397	21,865	33	2	X	246	296	376	385	393	453	606	629
usa13509	40,503	63	4	X	X	1,061	1,359	1,371	1,241	1,386	2,154	2,165
brd14051	42,128	68	2	X	X	1,061	1,418	1,417	1,226	1,361	2,274	2,282
d15112	45,310	78	3	X	X	2,070	2,810	2,852	2,379	2,598	4,549	4,603
d18512	55,510	88	2	X	X	X	2,315	2,321	2,100	2,241	3,752	3,756
pla33810	101,367	100	5	X	X	X	12,379	12,614	X	14,747	20,482	21,734

number of iterations in the binary search is about 10 for large instances, and this prohibits us from solving very large instances in a reasonable time. The memory usage for the PIGALE instances is very small, compared to the instances of Classes (1) and (2) even two-byte integer arrays are used for edge data.

For the instances in Class (3), Implementation A_2D_2 is very memory efficient. This indicates that the numbers of non-trivial components for faces in those instances are small. The gap between the running time of Implementation A_1 and that of Implementation A_2D_2 is a little bigger for instances of this class than the gap for instances in the other two classes. This can be explained by the following reasons. Naive and Implementation A_1 keep the shortest distance matrix, while A_2D_2 discards the matrix. As analyzed in Improvement D_1, Naive and A_1 need to solve $n+2$ single source shortest path problems while A_2D_2 may need to solve $4n$ such problems, where n is the number of vertices of the input medial graph. A_2D_2 also needs to calculate the components of G_r for every face r and there is no such computation in Naive and A_1. Notice that each of the shortest distances calculation, the computation of components of G_r for all r, and the deletion process takes $O(n^2)$ time. For instances of Classes (1) and (2), the time of deletion process is larger than the sum of the other two. However, the deletion process runs faster for the instances of Class (3) than for instances of Classes (1) and (2). In this case, the shortest distances calculation and the computation of components of G_r may become a dominating part of the total running time (see [5] for more details).

Among all implementations, the most time efficient one is A_1. Compared with Naive, A_1 is faster by a factor of $3 \sim 15$. The memory saving of A_1 is also significant. A_1 can solve an instance of about 20,000 edges in Class (1) and instances of about 15,000 edges in Classes (2) and (3), while Naive can only solve instances of size up to about 5,000 edges for all three classes. The most memory efficient implementation without re-calculation is A_2D_2. It can solve an instance of about 1,000,000 edges in Class (1) by about 3.5 hours and 500 Mbytes, an instance of about 60,000 edges in Class (2) by about 1.5 hours and 1.5Gbytes memory, and an instance of about 1,000,000 edges in Class (3) by about 14 hours and 200 Mbytes. Implementation $A_2A_3D_2$ is the most memory efficient one among all implementations. It can solve an instance of about 1,000,000 edges in Class (1) by about 6 hours and 200 Mbytes, an instance of about 1,000,000 edges in Class (2) by about 6 hours and 1.4Gbytes, and an instance of about 700,000 edges in Class (3) by by about 14 hours and 160 Mbytes. All implementations without using A_3 have time complexity $O(n^2)$ since no re-calculation for edge data is performed. In the worst case, Implementation $A_2A_3D_2$ may perform re-calculation repeatedly for some edges and has time complexity $O(n^3)$. However, the worst case scenario has not been observed and the running time of Implementation $A_2A_3D_2$ is at most as twice as that of Implementation A_2D_2 for most instances.

Table 2: Memory usage (in megabytes) of Naive and efficient implementations for Class (1) instances.

Instances	Number of edges	Maximum Memory Usage (Mbyte)								
		Naive	A_1	A_1D_1	A_2	A_2D_2	A_3	A_3D_2	A_2A_3	$A_2A_3D_2$
pr1002	2,972	413	39	16	8	8	10	9	8	7
rl1323	3,950	734	66	23	11	10	15	13	11	9
d1655	4,890	1,188	99	30	14	11	17	14	13	10
rl1889	5,631	1,424	130	46	18	14	28	21	16	12
u2152	6,312	X	161	41	17	14	26	20	16	13
pr2392	7,125	X	204	66	21	17	35	26	19	15
pcb3038	9,101	X	328	76	25	21	36	27	22	19
fl3795	11,326	X	504	132	58	42	66	63	40	23
fnl4461	13,359	X	698	158	39	32	66	43	33	26
rl5934	17,770	X	1,226	358	67	51	155	86	50	35
pla7397	21,865	X	1,850	436	123	85	238	144	83	44
usa13509	40,503	X	X	1,534	220	153	498	271	149	79
brd14051	42,128	X	X	1,600	215	149	580	283	149	82
d15112	45,310	X	X	1,795	227	156	508	256	156	86
d18512	55,510	X	X	X	284	198	706	328	194	106
pla33810	101,367	X	X	X	814	508	X	876	507	198

5.2 Comparison with previous works. Hicks proposes a straightforward implementation *rat* and two memory friendly implementations *comprat* and *memrat* of ST Procedure [13]. The implementations are tested using instances of Class (1) on a SGI Power Challenge with 6×194 MHz processors, 1Gbytes of physical memory, and 1Gbytes of swap space. To compare our results with Hicks', we quote some data of [13] in Table 4. An M in the table indicates that the implementation runs out of 2Gbyte memory for that instance. From the table, *rat* runs out of 2Gbyte memory for instances of rl1889 (5,631 edges) and larger. Naive of this paper can solve rl1889 but runs out of 2Gbyte memory for instances of u2152 (6,312 edges) and larger. This confirms that straightforward implementations of ST Procedure are memory consuming. The memory used by *memrat* for Instance brd14051 (the largest one reported in [13]) is about 600Mbytes. For the same instance, A_2D_2 uses about 1/4 and $A_2A_3D_2$ uses about 1/8 of the memory of *memrat*. Implementation A_1 is faster by a factor of $200 \sim 500$, Implementation A_2D_2 is faster by a factor of $100 \sim 400$, and Implementation $A_2A_3D_2$ is faster by a factor of $100 \sim 200$ than *comprat* and *memrat* for large instances. Notice that the CPU used in [13] has frequency 194MHz and the CPU used in this paper has frequency 3.06GHz, so we need to keep in mind this difference of speed when we compare the running time.

5.3 Computing branch decompositions. Seymour and Thomas give an algorithm, which is known as *edge contraction method*, for computing an optimal branch decomposition of a planar graph [22]. The contraction of an edge e in a graph G is to remove e from G, identify the two end vertices of e by a new vertex, and make all edges incident to e incident to the new vertex. We denote by G/e the graph obtained by contracting e in G. Given a 2-connected planar graph G, the algorithm of Seymour and Thomas computes an optimal branch decomposition of G by a sequence of edge contractions of the medial graph $M(G)$ of G as follows: First the carvingwidth cw of $M(G)$ is computed by ST Procedure. An edge e of $M(G)$ is *contractible* if the carvingwidth of $M(G)/e$ is at most cw and $M(G)/e$ is 2-connected. Next, a contractible edge e of $M(G)$ is found by ST Procedure and $M(G)$ is contracted to graph $M(G)/e$. The contraction is repeated on $M(G)/e$ until the graph becomes one with three vertices. A carving decomposition of $M(G)$ with width at most cw is constructed based on the sequence of edge contractions. Finally, the branch decomposition of G is obtained from the carving decomposition of $M(G)$. It is proved in [22] that for any 2-connected planar graph there is a contractible edge and for a 2-connected planar graph G, $M(G)$ is 2-connected. To check if an edge is contractible, ST Procedure is used to test if $M(G)/e$ has carvingwidth at most cw. In the worst case, all edges may be checked to find a contractible one and for a graph of n vertices, the algorithm of Seymour and Thomas may call ST Procedure $O(n)$ times for one contraction and $O(n^2)$ times in total. So the time complex-

Table 3: Computation time and required memory of Naive and Implementations A_1, A_2D_2, and $A_2A_3D_2$.

Class	Instances	Number of edges	bw	Itr	Time (seconds)				Memory (Mbyte)			
					Naive	A_1	A_2D_2	$A_2A_3D_2$	Naive	A_1	A_2D_2	$A_2A_3D_2$
(2)	rand3050	5,032	9	4	283	32.5	34.2	61.5	1,179	180	32	22
	rand6000	10,261	12	2	X	95.5	101	147	X	724	92	41
	rand8700	15,090	14	3	X	292	370	849	X	1,559	160	71
	rand11500	20,279	13	2	X	X	557	2,002	X	X	269	120
	rand33000	60,398	20	2	X	X	5,633	8,540	X	X	1,472	624
	rand54000	100,037	22	2	X	X	X	21,417	X	X	X	1,382
(3)	PI1180	2,022	7	5	23	6.8	9.1	10.4	196	32	8	7
	PI2995	5,043	7	6	156	58.7	93.3	96.5	1,034	178	14	13
	PI5940	10,016	7	8	X	289	522	563	X	683	28	26
	PI8950	15,097	10	9	X	907	1,771	2,089	X	1,541	48	39
	PI11974	20,071	9	9	X	X	3,646	3,702	X	X	46	44
	PI37730	70,022	6	10	X	X	49,136	49,180	X	X	188	160

ity of the algorithm is $O(n^4)$.

We call a contractibility test on an edge a *positive one* if the edge is tested contractible, otherwise a *negative one*. Gu and Tamaki give an algorithm which uses a better strategy to find positive tests [12]. When a negative test is obtained on an edge then the edge will not be tested again unless a necessary condition for that edge to be contractible is satisfied. By this improvement, the algorithm of Gu and Tamaki avoids the repeated negative tests on a same edge, calls ST Procedure $O(n)$ times, and has time complexity $O(n^3)$ for computing an optimal branch decomposition of a planar graph.

We test the algorithm of Seymour and Thomas and the algorithm of Gu and Tamaki for instances in three classes using a number of heuristics to select edges for testing the contractibility. The instances of Class (2) are generated by LEDA function *SEGMENT_INTERSECTION* [2]. Implementation A_1, the most time efficient one, is used as ST Procedure. Both the algorithms have the minimum number of negative calls and running time when the round robin edge selection heuristic is used. The computer used for testing has an AMD Athlon(tm) 64 X2 Dual Core Processor 4600+ (2.4GHz) and has a similar performance with the PC for testing ST Procedure. Table 5 gives the number of calls of ST Procedure and computation time for the algorithms with the round robin edge selection heuristic. The data in the table show that optimal branch decompositions of planar graphs of a few thousands edges can be computed in a practical time. For most instances tested, repeated negative tests are not observed on any edge in the algorithm of Seymour and Thomas. So the advantage of the algorithm of Gu and Tamaki is not shown by those instances when the round robin edge selection heuristic is used. On some other edge selection heuristic, more repeated negative tests are observed in the algorithm of Seymour and Thomas. In this case, the algorithm of Gu and Tamaki has much less negative calls and runs faster than the algorithm of Seymour and Thomas. We omit the details here.

6 Concluding remarks.

We tested our implementations on instances of size up to one hundred thousand edges. The results of this paper show that the branchwidth of those instances can be computed within a reasonable time and memory size. This suggests that the required memory may not be a bottleneck for computing branchwidth and optimal branch decompositions of planar graphs in practice. Our implementations require $O(n^2)$ bytes memory, although the constant behind the Big-Oh may be small. For the instances of n edges (n vertices in the input medial graphs), the memory required for face data could be as large as $n^2/8$ bytes. So they may not be able to solve extremely large instances with a few hundred thousands or more edges within a practical memory size. One approach to solve such instances is to have external memory implementations of ST Procedure. Another approach is to perform re-calculations for face data as well. How to bound the time complexity for re-calculating face data would be a key for this approach.

We have an upper bound $O(n^3)$ on the time complexity of the implementations with re-calculations for

Table 4: Computation time (in seconds) of *rat*, *comprat*, and *memrat* quoted from Table 1 of [13].

Instances	Number of edges	*bw*	*Itr*	Computation time (in seconds).		
				rat	*comprat*	*memrat*
pr1002	2,972	21	2	338	448	562
rl1323	3,950	22	3	876	1,519	1,590
d1655	4,890	29	3	1,318	1,608	2,206
rl1889	5,631	22	3	M	3,931	4,012
u2152	6,312	31	4	M	3,207	4,704
pr2392	7,125	29	3	M	3,813	5,167
pcb3038	9,101	40	4	M	13,817	15,865
fl3795	11,326	25	3	M	18,469	17,142
fnl4461	13,359	48	4	M	35,933	51,305
rl5934	17,770	41	3	M	73,468	66,461
pla7397	21,865	33	2	M	65,197	53,564
usa13509	40,503	63	1/2	M	M	413,861
brd14051	42,128	68	3	M	M	594,468

edge data. Let p be the maximum number of edges kept in those implementations. This bound is true for any $p \geq 1$. In general, a larger p results in a faster running time of the implementations. It is interesting to prove a better upper bound related to p, say $O(n^2(n/p))$, on the time complexity of those implementations.

We have used the eccentricity-based heuristic to get an initial guess for the starting value of k. This heuristic runs fast and gives a guess very close to the carvingwidth for the geometric instances. However, the guess is poor for instances generated by the PIGALE library. The heuristic of [23] can also be used for the guess but it has a performance very similar to the eccentricity-based one: good for geometric instances but poor for the PIGALE instances. The poor guess for the PIGALE instances results in a large number of iterations in the computation, which may become a bottleneck for computing the branchwidth for large instances in this class. To reduce the number of iterations, it is interesting to develop a performance guaranteed and yet efficient heuristic for computing a good approximation of the carvingwidth.

For large instances, computing optimal branch decompositions is still time consuming by the edge contraction method. Divide-and-conquer is a more efficient approach in practice for computing branch decompositions. In this approach, roughly speaking, we partition the medial graph of an instance into two (or more) subgraphs such that the size of the cut set between the two subgraphs is bounded by the carvingwidth of the medial graph. Then we use ST Procedure to test if each subgraph has carvingwidth at most that of the medial graph. If both answers are positive (such a partition is called valid), we recursively find the carving decomposition of each subgraph. A heuristic based on this divide-and-conquer approach is reported in [14]. When the heuristic can not find a valid partition in a recursive step, it uses the edge-contraction method to make a progress. In the worst case, the heuristic may become the edge-contraction algorithm and have time complexity $O(n^4)$. It is worth to explore efficient heuristics to find a valid partition in every recursive step.

Acknowledgment

The authors thank Dr. I.V. Hicks for providing the test instances of Class (1). The work was partially supported by the NSERC Research Grant of Canada and the Japan Society for the Promotion of Science (JSPS) Grant-In-Aid for Scientific Research. The authors also thank anonymous reviewers for their constructive comments.

References

[1] Public Implementation of a Graph Algorithm Library and Editor, 2007. http://pigale.sourceforge.net/.

[2] The LEDA User Manual, Algorithmic Solutions, Version 4.2.1, 2007. http://www.mpi-inf.mpg.de/LEDA/MANUAL/MANUAL.html.

[3] S. Arnborg, D.G. Corneil, and A. Proskurowski. Complexity of finding embedding in a k-tree. *SIAM J. on Discrete Mathematics*, 8:277–284, 1987.

[4] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

Table 5: Number of calls for ST Procedure and computation time (in seconds) for branch decompositions.

Class	Instances	Number of edges	*bw*	Seymour-Thomas Algorithm				Gu-Tamaki Algorithm			
				Negative calls	Positive calls	Total calls	Time	Negative calls	Positive calls	Total calls	Time
(1)	pr1002	2,972	21	102	2,969	3,071	2,651	102	2,969	3,071	2,667
	rl1323	3,950	22	136	3,947	4,083	6,789	136	3,947	4,083	6,879
	rl1889	5,631	22	**181**	5,628	5,809	28,883	**178**	5,628	5,806	29,096
	u2152	6,312	31	192	6,309	6,501	25,493	192	6,309	6,501	26,092
	pr2392	7,125	29	271	7,122	7,393	44,856	271	7,122	7,393	45,728
(2)	rand1408	2,637	23	**172**	2,634	2,806	2,268	**168**	2,634	2,802	2,263
	rand2243	4,046	31	158	3,820	3,978	5,955	158	3,820	3,978	6,002
	rand2825	5,219	36	**203**	4,908	5,111	17,257	**200**	4,908	5,108	17,281
	rand 3216	5,964	38	**237**	5,642	5,879	22,159	**236**	5,642	5,878	22,133
	rand3792	7,079	41	**263**	6,712	6,975	36,391	**262**	6,712	6,979	36,494
	PI1277	2,128	9	**119**	2,125	2,244	1,557	**101**	2,125	2,226	1,563
	PI2009	3,369	7	**95**	3,366	3,461	8,061	**90**	3,366	3,456	8,127
	PI2968	5,031	6	162	5,028	5,190	26,052	162	5,028	5,190	26,230
	PI3586	6,080	8	**177**	6,077	6,254	48,570	**176**	6,077	6,253	49,108
	PI4112	6,922	7	132	6,919	7,051	69,792	132	6,919	7,051	70,220

[5] Z. Bian, Q.P. Gu, M. Marzban, H. Tamaki, and Y. Yoshitake. Empirical study on branchwidth and branch decomposition of planar graphs. Technical Report, SFU-CMPT-TR 2007-22, School of Computing Science, Simon Fraser University, 2007.

[6] H.L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[7] H.L. Bodlaender. A linear time algorithm for finding tree-decomposition of small treewidth. *SIAM J. on Computing*, 25:1305–1317, 1996.

[8] H.L. Bodlaender. Treewidth: characterizations, applications, and computations. In *Proc. of 32nd Workshop on Graph Theoretical Concepts in Computer Science (WG2006), LNCS 4271*, pages 1–14, 2006.

[9] H.L. Bodlaender and D. Thilikos. Constructive linear time algorithm for branchwidth. In *Proc. of 24th International Colloquium on Automata, Languages, and Programming, ICALP'97*, pages 627–637, 1997.

[10] F. Dorn, E. Penninkx, H. Bodlaender, and F.V. Fomin. Efficient exact algorithms for planar graphs: exploiting sphere cut branch decompositions. In *Proc. of the 13th Annual European Symposium on Algorithms (ESA05), LNCS 3669*, pages 95–106, 2005.

[11] F.V. Fomin and D.M. Thilikos. Dominating sets in planar graphs: branch-width and exponential speed-up. *SIAM Journal on Computing*, 36(2):281–309, 2006.

[12] Q.P. Gu and H. Tamaki. Optimal branch decomposition of planar graphs in $O(n^3)$ time. In *Proc. of 32nd International Colloquium on Automata, Languages, and Programming, ICALP2005, LNCS3580*, pages 373–384, 2005.

[13] I.V. Hicks. Planar branch decompositions I: The ratcatcher. *INFORMS Journal on Computing*, 17(4):402–412, 2005.

[14] I.V. Hicks. Planar branch decompositions II: The cycle method. *INFORMS Journal on Computing*, 17(4):413–421, 2005.

[15] I.V. Hicks, A.M.C.A. Koster, and E. Kolotoğlu. Branch and tree decomposition techniques for discrete optimization. In *TutORials in Operation Research: INFORMS–New Orleans 2005*, pages 1–29, 2005.

[16] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, New York, 1999.

[17] G. Reinelt. TSPLIB-A traveling salesman library. *ORSA J. on Computing*, 3:376–384, 1991.

[18] N. Robertson and P.D. Seymour. Graph minors I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, 35:39–61, 1983.

[19] N. Robertson and P.D. Seymour. Graph minors II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[20] N. Robertson and P.D. Seymour. Graph minors X. Obstructions to tree decomposition. *J. of Combinatorial Theory, Series B*, 52:153–190, 1991.

[21] G. Schaeffer. Random sampling of large planar maps and convex polyhedra. In *Proc. of the 31st Annual ACM Symposium on the Theory of Computing (STOC'99)*, pages 760–769, 1999.

[22] P.D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[23] H. Tamaki. A linear time heuristic for the branch-decomposition of planar graphs. In *Proc. of ESA2003*, pages 765–775, 2003.

[24] D.B. West. *Introduction to Graph Theory*. Prentice Hall Inc., Upper Saddle River, NJ, 1996.

Workshop on Analytic Algorithmics and Combinatorics

On the Convergence of Upper Bound Techniques for the Average Length of Longest Common Subsequences

George S. Lueker*

Abstract

It has long been known [2] that the average length of the longest common subsequence of two random strings of length n over an alphabet of size k is asymptotic to $\gamma_k n$ for some constant γ_k depending on k. The value of these constants remains unknown, and a number of papers have proved upper and lower bounds on them. In particular, in [6] we used a modification of methods of [3, 4] for determining lower and upper bounds on γ_k, combined with large computer computations, to obtain improved bounds on γ_2. The method of [6] involved a parameter h; empirically, increasing h increased the computation time but gave better upper bounds. Here we show, for arbitrary k, a sufficient condition for a parameterized method to produce a sequence of upper bounds approaching the true value of γ_k, and show that a generalization of the method of [6] meets this condition for all $k \geq 2$. While [3, 4] do not explicitly discuss how to parameterize their method, which is based on a concept they call domination, to trade off the tightness of the bound vs. the amount of computation, we discuss a very natural parameterization of their method; for the case of alphabet size $k = 2$ we conjecture but do not prove that it also meets the sufficient condition and hence also yields a sequence of bounds that converges to the correct value of γ_2. For $k > 2$, it does not meet our sufficient condition. Thus we leave open the question of whether some method based on the undominated collations of [3, 4] gives bounds converging to the correct value for any $k \geq 2$.

1 Introduction

Let L_{ij} be the random variable giving the expected length of the longest common subsequence (lcs) of two random sequences of length i and j, where each character in the sequences is chosen uniformly and independently from an alphabet of size k. It has long been known [2] that, by a superadditivity argument,

$$\mathsf{E}[L_{nn}] \sim \gamma_k n,$$

for some constant γ_k (depending on k). Recently the breakthrough paper [5] showed that

$$\lim_{k\to\infty} \sqrt{k}\gamma_k = 2,$$

establishing the Sankoff-Mainville conjecture. Nonetheless to my knowledge the value of the γ_k remains unknown for all $k \geq 2$; a number of papers, including [2], have proved upper and lower bounds on it. See [3, Section 4.2] for a brief history of this research. [3, 4] established the best-known (at that time) upper bound on γ_2. ([3] contains many other results as well.) They used a method based on a concept they called domination. They note that extensions of their methods could easily give stronger bounds with a reasonable amount of computation, but suggest that this would probably lead to only limited improvements, and state that the question of whether their method can give bounds approaching the true value remains open [4, p. 456]. In [6] we used a modification of their methods, combined with large computer computations, to obtain improved bounds on γ_2. In particular, for upper bounds we used a concept called canonicity instead of domination. The method involved a parameter h; empirically, increasing h increased the computation time but gave better upper bounds. In this paper we give a sufficient condition for a parameterized method to produce a sequence of bounds approach the true value of γ_k, and show that a generalization of the method of [6] meets this condition for all $k \geq 2$. We conjecture that a suitably parameterized version of the approach used in [3, 4], based on the concept of domination, would meet this condition for the case $k = 2$, and thus yield upper bounds converging to the true value of γ_2; for $k \geq 3$ we show that this version does not meet the condition. Thus we leave open the question of whether a method based the concept of domination used in [3, 4] can produce a sequence of bounds converging to the true value of γ_k, for any alphabet size $k \geq 2$.

We note that it is not surprising that some sequence b_h of upper bounds on γ_k, where the time to compute b_h increases with h, can be shown to converge to γ_k. [1] showed how to compute explicit values of n_0 and C

*Department of Computer Science, University of California, Irvine, Irvine, CA 92697-3435; lueker@ics.uci.edu.

such that

$$n \geq n_0 \implies \gamma_k n \geq \mathsf{E}[L_{nn}] \geq \gamma_k n - C\sqrt{n \log n}.$$

Hence for large n

$$\gamma_k \leq \mathsf{E}[L_{nn}]/n + C\sqrt{n^{-1} \log n}. \tag{1.1}$$

In [1] results from simulations with random numbers were used to estimate the expected lcs length and give bounds that held with 95% confidence, but in principle one could simply evaluate $\mathsf{E}[L_{nn}]$ for successive values of n by exhaustive enumeration to obtain from (1.1) a sequence of bounds guaranteed to converge to γ_k. Although this would give an error bound that converges more rapidly than the bound proven below in Theorem 1.4, it is of interest to investigate the convergence of the methods that have actually produced the best bounds.

We begin by discussing the results of [3, 4]. This discussion differs a bit from that of [3, 4], and uses a framework sufficiently general to cover the modification in [6] as well.

Throughout this paper we let the alphabet of the strings be $\Sigma = \{0, 1, \ldots, k-1\}$, with $k > 1$, and let ϵ denote the empty string. Let a string pair be a pair of strings, say $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$; we denote the string pair by $P = \binom{a_1 a_2 \cdots a_i}{b_1 b_2 \cdots b_j}$. Call $a_1 a_2 \cdots a_i$ the *top* string and $b_1 b_2 \cdots b_j$ the *bottom* string, and say P is of *size* $s(P) = i + j$. Let $lcs(P)$ be the length of the longest common subsequence of the top and bottom strings of P. Call the string pair $\binom{a_1 a_2 \cdots a_i}{b_1 b_2 \cdots b_j}$ a *match pair* if $a_i = b_j$. Say that this match pair *ends with an essential match* if any longest common subsequence must match a_i and b_j, i.e., if both

$$lcs\binom{a_1 a_2 \cdots a_{i-1}}{b_1 b_2 \cdots b_j} < lcs\binom{a_1 a_2 \cdots a_i}{b_1 b_2 \cdots b_j}$$

and

$$lcs\binom{a_1 a_2 \cdots a_i}{b_1 b_2 \cdots b_{j-1}} < lcs\binom{a_1 a_2 \cdots a_i}{b_1 b_2 \cdots b_j}.$$

A *minimal match pair* is a match pair P that ends on an essential match and has $lcs(P) = 1$. Call a string pair P a *null pair* if $lcs(P) = 0$, i.e., if the top and bottom strings of P have no characters in common; the simplest null pair is $\binom{\epsilon}{\epsilon}$.

Let $G(n, \ell)$ be the number of string pairs of size n that have an lcs of length at least ℓ. As [3, 4] point out, the following easily-proven observation has been the basis of a number of published bounds:

THEOREM 1.1. [2] *If for some alphabet size k, and for some $y \in (0, 1)$, we have*

$$G(2n, \lfloor yn \rfloor) = o\big(k^{2n}\big)$$

as $n \to \infty$, then $\gamma_k \leq y$.

It will be convenient to consider lists of string pairs. Say that the *length* of such a list L is the number of elements (i.e., the number of string pairs) in the list, and that the *size* of L, written $s(L)$, is the total number of characters in all of the string pairs in L. Following [3, 4] say that an arbitrary list $\binom{x_1}{y_1}\binom{x_2}{y_2}\cdots\binom{x_\ell}{y_\ell}$ of string pairs *generates* the string pair $\binom{x_1 x_2 \cdots x_\ell}{y_1 y_2 \cdots y_\ell}$; we will write this as

$$generate\left(\binom{x_1}{y_1}\binom{x_2}{y_2}\cdots\binom{x_\ell}{y_\ell}\right) = \binom{x_1 x_2 \cdots x_\ell}{y_1 y_2 \cdots y_\ell}. \tag{1.2}$$

Note that we do not use the terms size and length interchangeably; for example, the list of string pairs

$$\binom{230}{10}\binom{012}{2}\binom{13}{23}$$

has length 3 and size 13. Also note that the size of a list of string pairs is always the same as the size of the string pair it generates. We will use $\|$ to indicate concatenation of lists of string pairs; when an argument of $\|$ is a single string pair, we will promote it to the list with that one pair as its only element so we can for example write $L \parallel \binom{0}{0}$ to denote the list obtained by appending the pair $\binom{0}{0}$ to the list L.

Now consider a finite-state machine $\mathcal{M}$ that takes as input lists of minimal match pairs; for each such list it accepts or rejects. (This goes slightly outside the usual definition of finite-state machine since the input alphabet, namely, the set of minimal match pairs, has infinite cardinality, but this will cause no problems, and in practice it will be easy to restrict our attention to a finite subset of the minimal match pairs.) We will always assume that once $\mathcal{M}$ reaches a rejecting state it will remain in that state; thus it can accept a list only if it accepts every prefix of that list. In view of this we can assume without loss of generality that there is only one rejecting state, which we denote by $\emptyset$. We will also assume that all states of $\mathcal{M}$ are reachable from the start state, since any states that were not reachable from the start state could simply be removed. Let $\mathcal{S}$ be the set of accepting states in $\mathcal{M}$; thus the entire set of states is $\mathcal{S} \cup \emptyset$.

If S is some set of objects, and each $x \in S$ has a nonnegative integer size $s(x)$, the generating function for S is the function $\mathcal{G}(S)$ that maps z to

$$\sum_{x \in S} z^{s(x)}.$$

Suppose S and S' are two sets of objects on which a nonnegative size function s is defined, and let $\times$ denote

the Cartesian product; extend s to $S \times S'$ by letting $s(x, x') = s(x) + s(x')$. It is well-known that

$$\mathcal{G}(S \times S') = \mathcal{G}(S) \cdot \mathcal{G}(S'); \tag{1.3}$$

we will use this fact below. Let $g_\ell(z)$ be the generating function for the set of string pairs P with $lcs(P) \geq \ell$, i.e.,

$$g_\ell(z) = \sum_{n=0}^{\infty} G(n, \ell) z^n. \tag{1.4}$$

Let $f_\ell(z)$ be the generating function for the set of lists of ℓ minimal match pairs that are accepted by $\mathcal{M}$.

Definition 1. *Say that $\mathcal{M}$ covers Σ^* if for every string pair P with $lcs(P) = \ell$ there is a list L of ℓ minimal match pairs and a null pair p such that $\mathcal{M}$ accepts L, and $L \parallel p$ generates P.*

Theorem 1.2. [3, 4] *Suppose that $\mathcal{M}$ covers Σ^* and let $f_\ell(z)$ be the generating function for the set of lists of minimal match pairs of length ℓ accepted by $\mathcal{M}$. Suppose that for some real constants z and K, with*

$$z \in \big(0, k^{-1}\big) \quad \text{and} \quad \lambda(z) \in (0, 1) \tag{1.5}$$

we have

$$f_\ell(z) \leq K\big(\lambda(z)\big)^\ell. \tag{1.6}$$

Then

$$\gamma_k \leq \frac{2\log(kz)}{\log \lambda(z)}. \tag{1.7}$$

Proof. Let $n(z)$ be the generating function for the set of null pairs. One easily sees that $n(z)$ converges for $z \in \big(0, k^{-1}\big)$ (since we can bound the number of null pairs of size n by the number of string pairs of size n, which is $(n+1)k^n$). Then by (1.3) and the fact that $\mathcal{M}$ covers Σ^*, we have

$$g_\ell(z) \leq \sum_{i=\ell}^{\infty} f_\ell(z) n(z). \tag{1.8}$$

Now pick any

$$y > \frac{2\log(kz)}{\log \lambda(z)}; \tag{1.9}$$

note that then

$$y > 0 \quad \text{and} \quad \lambda(z)^y < (kz)^2. \tag{1.10}$$

From (1.4), (1.6), and (1.8), for any ℓ we have

$$\begin{aligned} G(2n, \ell) &\leq \frac{g_\ell(z)}{z^{2n}} \\ &\leq \sum_{i=\ell}^{\infty} n(z) \frac{K\big(\lambda(z)\big)^i}{z^{2n}} \\ &\leq \frac{K n(z) \big(\lambda(z)\big)^\ell}{\big(1 - \lambda(z)\big) z^{2n}}. \end{aligned}$$

Now K, $n(z)$, and $\lambda(z)$ are all constants so using (1.10) it follows that

$$\begin{aligned} G(2n, \lfloor yn \rfloor) &= O\left(\frac{\big(\lambda(z)\big)^{\lfloor yn \rfloor}}{z^{2n}}\right) \\ &= o\Big(\frac{(kz)^{2n}}{z^{2n}}\Big) = o\big(k^{2n}\big). \end{aligned}$$

Thus by Theorem 1.1 any y satisfying (1.9) is an upper bound on γ_k, so the right-hand side of (1.9) is an upper bound on γ_k. ∎

One can bound the generating function $f_\ell(z)$ as follows. Let q_0 be the starting state of the machine. Let δ be the transition function; thus if the machine is in state q and reads minimal match pair p, it moves to state $q' = \delta(q, p)$. Let M be the set of all minimal match pairs. Let $f_\ell(z, q)$ be the generating function for the lists of ℓ minimal match pairs that are accepted by $\mathcal{M}$ assuming it starts in state q. Then

$$f_0(z, q) = \begin{cases} 0 & \text{if } q = \emptyset \\ 1 & \text{if } q \in \mathcal{S} \end{cases}$$

and for $\ell > 0$ we have

$$f_\ell(z, q) = \sum_{p \in M} z^{s(p)} f_{\ell-1}\big(z, \delta(q, p)\big). \tag{1.11}$$

Since M has infinite cardinality, this is not directly amenable to computation, but we can effectively make M finite by treating some sets of minimal match pairs as equivalent. For example, [3, 4] design a machine that treats all minimal match pairs of the form $\binom{0^+01}{1}$ in the same way.

Let $\vec{f}_\ell(z)$ denote the vector, indexed by the elements of $\mathcal{S}$, in which the qth component is $f_\ell(z, q)$. Define

$$\mathbf{equal}(q, q') = \begin{cases} 1 & \text{if } q = q' \\ 0 & \text{otherwise.} \end{cases}$$

Then the recurrence (1.11) can be expressed as

$$\vec{f}_\ell(z) = A \vec{f}_{\ell-1}(z),$$

where A is the matrix $\{a_{q,q'}\}$ in which

$$a_{q,q'} = \sum_{p \in M} z^{s(p)} \mathbf{equal}\big(q', \delta(q, p)\big).$$

Thus finding the value $\lambda(z)$ is essentially the problem of finding the largest eigenvalue of A. [3, 4] used Mathematica to assist with the calculations. In [6] we used numerical iteration of (1.11) to compute bounds on $\lambda(z)$; for this approach another property of the machines is useful.

DEFINITION 2. *$\mathcal{M}$ is regular*[1] *if it has the following two properties:*

a) for every $q \in \mathcal{S}$ the start state is reachable from q, and

b) there exists some state $q \in \mathcal{S}$ and some minimal match pair $p \in M$, such that if $\mathcal{M}$ is in state q and reads p, it remains in state q.

Now we turn to the results of the present paper.

THEOREM 1.3. *If $\mathcal{M}$ is regular, then for each constant $z \in (0, k^{-1})$ there exists $\lambda > 0$, $0 \le \rho < 1$, and a vector $\vec{v}_\lambda$ (indexed by states in $\mathcal{S}$), with all components positive, such that*

$$f_\ell(z, q) = \lambda^\ell v_\lambda(q) + O(\rho^\ell \lambda^\ell). \tag{1.12}$$

Proof. Since $\mathcal{M}$ is regular the matrix A is irreducible and aperiodic; thus by [8, Theorem 1.4] it is primitive,[2] so by the Perron-Frobenius Theorem for primitive matrices [8, Theorem 1.1] (see also [7]), it has a positive real eigenvalue λ of multiplicity one with a corresponding eigenvector in which all components are positive, and all other eigenvalues are smaller in magnitude than λ; the conclusion follows immediately. ∎

COROLLARY 1.1. *If $\mathcal{M}$ is regular, then for each positive z there exists a λ such that for all $q \in \mathcal{S}$,*

$$\lim_{\ell \to \infty} \frac{f_\ell(z, q)}{f_{\ell-1}(z, q)} = \lambda.$$

By devising better and better machines $\mathcal{M}$ one can attempt to produce better and better bounds. We now describe a sufficient condition for a sequence $\mathcal{M}_h$, $h = 1, 2, \ldots$, of machines to produce bounds that approach the true value of γ_k.

DEFINITION 3. *Say a sequence $\mathcal{M}_h$, $h = 1, 2, \ldots$, of machines efficiently covers Σ^* if it covers Σ^* and additionally has the following two properties:*

a) For any two lists L and L' of minimal match pairs, if $\mathcal{M}$ accepts $L \parallel L'$ then it also accepts L and L'.

b) There exists no string pair P of size bounded by h such that two distinct lists L and L' of minimal match pairs, each of which is accepted by $\mathcal{M}_h$, both generate P.

The next section proves the following.

[1] This is closely related to the definition of regular used for computing lower bounds in [3, Definition 3.5].

[2] A square matrix A with no negative entries is *primitive* if for some positive integer k all entries in A^k are positive [8].

THEOREM 1.4. *Suppose that the sequence $\mathcal{M}_h$, $h = 1, 2, \ldots$, efficiently covers Σ^*, and that each machine in the sequence is regular. Then as $h \to \infty$ the sequence of upper bounds on γ_k produced by $\mathcal{M}_h$ approaches γ_k. In particular, the upper bound produced by $\mathcal{M}_h$ is bounded by*

$$\gamma_k + O\left(\left(\frac{\log h}{h}\right)^{1/3}\right),$$

where the hidden constants are allowed to depend on k.

2 Proof of Theorem 1.4

Throughout this section we assume that the conditions of Theorem 1.4 hold and that we are dealing with a fixed alphabet $\Sigma = \{0, 1, \ldots, k-1\}$, where $k \ge 2$. Dependence on k will not always be made explicit; in particular we will allow hidden constants in asymptotic notation to depend on k, and simply use γ instead of γ_k.

Define $F(n, \ell, h)$ to be the number of lists, with length ℓ and size n, of minimal match pairs that are accepted by $\mathcal{M}_h$, and let $f_{\ell,h}(z)$ denote the corresponding generating function, i.e.,

$$f_{\ell,h}(z) = \sum_{n=0}^{\infty} F(n, \ell, h) z^n. \tag{2.13}$$

Since this is the most complicated proof in the paper, we begin by giving a very rough sketch of the proof. To get arbitrarily close bounds from Theorem 1.2 we will need to show that

$$f_{\ell,h}(z) \le K\big(\lambda(z)\big)^\ell$$

for some $\lambda(z)$ close to $(kz)^{2/\gamma}$. Although we are interested in the behavior of $f_{\ell,h}$ when $\ell \gg h$, we first analyze the behavior for h larger than ℓ (in particular when h is substantially larger than $2\ell/\gamma$), and then show how to use this to obtain bounds for $\ell \gg h$; a similar approach was used in [5]. For n of size up to roughly $2\ell/\gamma$ we use part (b) of Definition 3 to bound $F(n, \ell, h)$ by the number $G(n, \ell)$ of string pairs of size n with an lcs length of at least ℓ. Since this is roughly comparable to k^n when n is near $2\ell/\gamma$, the contribution to the sum on the right of (2.13) when n is near $2\ell/\gamma$ is very roughly $(kz)^{2\ell/\gamma}$, as desired. For n significantly smaller than $2\ell/\gamma$ we then use a concentration inequality to show that the contribution to the sum is small, since it corresponds to strings having an lcs much longer than expected. Finally for n significantly larger than ℓ we use the fact that $f(n, \ell, h)$ is substantially lower than k^n (since many characters can only be chosen from $k-1$ rather than k choices), to show that the contribution to the sum is again small. We then use a subadditivity result that follows from part (a) of Definition 3 to extend the bound to the range $\ell \gg h$.

We now give the detailed proof. It is well-known [2] that by superadditivity we have $\mathsf{E}[L_{nn}] \le \gamma n$, from which it follows readily that

$$\mathsf{E}[L_{ij}] \le \frac{1}{2}\gamma(i+j) \tag{2.14}$$

We have

$$\Pr\{|L_{ij} - \mathsf{E}[L_{ij}]| \ge t\} \le 2e^{-t^2/(2(i+j))} \tag{2.15}$$

by a standard application of Azuma's inequality and the method of bounded differences; see [1] and [9, §1.3] for proofs of deviation bounds for the lcs by this method. Recall that let $G(n,\ell)$ is the number of string pairs of size n having an lcs length of at least ℓ. Considering the various possibilities for the lengths of the top and bottom strings, and using (2.14) and (2.15), we have

$$\begin{aligned} G(n,\ell) &\le \sum_{i=0}^{n} k^n \Pr\{L_{i,n-i} \ge \ell\} \\ &= \sum_{i=0}^{n} k^n \Pr\{L_{i,n-i} - \mathsf{E}[L_{i,n-i}] \\ &\qquad\qquad \ge \ell - \mathsf{E}[L_{i,n-i}]\} \\ &\le \sum_{i=0}^{n} k^n \Pr\{L_{i,n-i} - \mathsf{E}[L_{i,n-i}] \\ &\qquad\qquad \ge \ell - \gamma n/2\} \\ &\le 2(n+1)k^n e^{-(\ell-\gamma n/2)^2/(2n)}. \end{aligned} \tag{2.16}$$

Lemma 2.1. *There exists a constant $c \ge 2$, depending only on k, such that the following holds. Choose any positive integer ℓ and real x so that*

$$x \in [0, \ell/\gamma], \tag{2.17}$$

and

$$n_0 = 2\ell/\gamma - x \tag{2.18}$$

is a positive integer. Then for any real z with

$$z \in (0, k^{-1}) \tag{2.19}$$

and integer h with

$$h \ge c\,\ell \tag{2.20}$$

we have

$$\begin{aligned} f_{\ell,h}(z) &= \sum_{n=0}^{\infty} F(n,\ell,h) z^n \\ &\le 2n_0^2 e^{-\gamma^3 x^2/(16\ell)} + \frac{n_0+1}{(1-kz)^2}(kz)^{n_0} + 4k/2^{\ell}. \end{aligned} \tag{2.21}$$

Proof. From (2.17) and (2.18) we have

$$n_0 \ge \ell/\gamma. \tag{2.22}$$

Note that any list of ℓ minimal match pairs maps (under *generate*) to a pair of strings with lcs length at least ℓ; moreover, by (2.20) and part (b) of Definition 3 we know that this map is one-to-one if we restrict the domain to minimal match pair lists of size at most $c\,\ell$ that are accepted by $\mathcal{M}_h$. Thus we have

$$n \le c\,\ell \implies F(n,\ell,h) \le G(n,\ell). \tag{2.23}$$

We estimate the sum in (2.21) by dividing the range of summation into three intervals, namely, $0 \le n < n_0$, $n_0 \le n < c\,\ell$ and $c\,\ell \le n$. (Note that (2.18) and (2.20) guarantee that if we pick c large enough we will have $n_0 \le h$.)

Now using (2.23) and then (2.16) gives

$$\begin{aligned} \sum_{n=0}^{n_0-1} F(n,\ell,h) z^n &\le \sum_{n=0}^{n_0-1} G(n,\ell) z^n \\ &\le \sum_{n=0}^{n_0-1} 2(n+1)\,k^n\, e^{-(\ell-\gamma n/2)^2/(2n)} z^n. \end{aligned} \tag{2.24}$$

When $n < n_0 = 2\ell/\gamma - x$ we have $\gamma n/2 < \ell - \gamma x/2$ so

$$\frac{(\ell-\gamma n/2)^2}{2n} \ge \frac{(\gamma x/2)^2}{2\cdot 2\ell/\gamma} = \frac{\gamma^3 x^2}{16\ell}, \tag{2.25}$$

so using (2.24) and then (2.19) we have

$$\begin{aligned} \sum_{n=0}^{n_0-1} F(n,\ell,h) z^n &\le 2e^{-\gamma^3 x^2/(16\ell)} \sum_{n=0}^{n_0-1} (n+1)(kz)^n \\ &\le 2e^{-\gamma^3 x^2/(16\ell)} \sum_{n=0}^{n_0-1} (n+1) \\ &= n_0(n_0+1)e^{-\gamma^3 x^2/(16\ell)} \\ &\le 2n_0^2 e^{-\gamma^3 x^2/(16\ell)}, \end{aligned} \tag{2.26}$$

where in the last step we used the fact that n_0 is a positive integer.

For the range $n_0 \le n < c\,\ell$ we note that, from (2.23), $F(n,\ell,h)$ is certainly bounded by the number of string pairs of size n, i.e., $(n+1)k^n$, so

$$\begin{aligned} \sum_{n=n_0}^{c\,\ell-1} F(n,\ell,h) z^n &\le \sum_{n=n_0}^{c\,\ell-1} (n+1)(kz)^n \\ &\le \sum_{n=n_0}^{\infty} (n+1)(kz)^n \\ &= \frac{n_0+1-kzn_0}{(1-kz)^2}(kz)^{n_0} \\ &\le \frac{n_0+1}{(1-kz)^2}(kz)^{n_0}. \end{aligned} \tag{2.27}$$

Finally, for the range $n \geq c\,\ell$ we cannot use (2.23), so we will bound $F(n,\ell,h)$ by the total number of sequences of ℓ minimal match pairs of size n; this number is bounded by $\binom{n}{2\ell}$ (for choosing which of the n characters to match) times k^ℓ (for choosing the values of the matched characters) times $(k-1)^{n-2\ell}$ (for choosing the values of the unmatched characters), to obtain

$$\begin{aligned}\sum_{n=c\,\ell}^{\infty} F(n,\ell,h)z^n &\leq \sum_{n=c\,\ell}^{\infty} \binom{n}{2\ell} k^\ell (k-1)^{n-2\ell} z^n \\ &\leq \sum_{n=c\,\ell}^{\infty} \frac{n^{2\ell}}{(2\ell)!}(k-1)^{n-2\ell}k^\ell k^{-n} \\ &= \sum_{n=c\,\ell}^{\infty} \frac{n^{2\ell}}{(2\ell)!}\left(\frac{1}{k-1}\right)^\ell \left(\frac{k-1}{k}\right)^{n-\ell}\end{aligned}$$

where in the penultimate step we used (2.19). The ratio of successive terms in the sum on the right is bounded by

$$\begin{aligned}\left(1+\frac{1}{c\,\ell}\right)^{2\ell}\frac{k-1}{k} &\leq \left(e^{1/(c\,\ell)}\right)^{2\ell} e^{-1/k} \\ &= \exp\left(\frac{2}{c}-\frac{1}{k}\right) \leq \exp\left(-\frac{1}{3k}\right)\end{aligned} \tag{2.28}$$

provided we pick $c \geq 3k$. Thus using the facts that $e^{-1/(3k)} \leq 1 - 1/(4k)$ for $k \geq 2$, and that $(2\ell)! \geq (2\ell/e)^{2\ell}$, we have

$$\begin{aligned}&\sum_{n=c\,\ell}^{\infty} F(n,\ell,h)z^n \\ &\quad\leq \frac{1}{1-e^{-1/(3k)}}\frac{(c\,\ell)^{2\ell}}{(2\ell)!}\left(\frac{1}{k-1}\right)^\ell\left(\frac{k-1}{k}\right)^{c\,\ell-\ell} \\ &\quad\leq 4k\frac{(c\,\ell)^{2\ell}}{(2\ell/e)^{2\ell}}\left(\frac{1}{k-1}\right)^\ell\left(\frac{k-1}{k}\right)^{c\,\ell-\ell} \\ &\quad= 4k\left(\frac{e^2c^2}{4(k-1)}\Big(\frac{k-1}{k}\Big)^{c-1}\right)^\ell \\ &\quad\leq 4k/2^\ell, \qquad (2.29)\end{aligned}$$

where the last step holds providing that we pick c large enough (depending only on k) so that

$$\frac{e^2c^2}{4(k-1)}\Big(\frac{k-1}{k}\Big)^{c-1} \leq \frac{1}{2}.$$

Combining (2.26), (2.27), and (2.29) gives (2.21). ∎

For the remainder of this section we let c denote the constant whose existence is guaranteed by Lemma 2.1. We now further constrain the parameter z to appropriately balance the terms in the sum (2.21) and then refine the estimate.

LEMMA 2.2. *Let h and z vary with ℓ so that*

$$h \geq c\,\ell \tag{2.30}$$

and for some positive constant c' (depending only on k), we have

$$-\Big(\frac{\log\ell}{\ell}\Big)^{2/3} \leq \log(kz) \leq -c'\Big(\frac{\log\ell}{\ell}\Big)^{2/3}. \tag{2.31}$$

Then as ℓ approaches ∞ we have

$$\log f_{\ell,h}(z) \leq \frac{2\ell}{\gamma}\log(kz) + O(\log\ell). \tag{2.32}$$

Proof. Note that (2.31) implies $1-kz = \Omega(\ell^{-1})$. Choose x so that

$$0 \leq x - 6\,k^2\,\ell^{2/3}\log^{1/3}\ell \leq 1 \tag{2.33}$$

and

$$n_0 = 2\ell/\gamma - x$$

is a positive integer. Note that then from (2.31) we have $(kz)^x = \ell^{O(1)}$. Then for large ℓ the conditions of Lemma 2.1 apply so

$$\begin{aligned}f_{\ell,h}(z) &\leq 2n_0^2 e^{-\gamma^3x^2/(16\ell)} + \frac{n_0+1}{(1-kz)^2}(kz)^{n_0} + 4k/2^\ell \\ &= O(\ell^2)e^{-36k^4\gamma^3\ell^{4/3}\log^{2/3}\ell/(16\ell)} \\ &\quad + O(\ell^3)(kz)^{2\ell/\gamma-x} + 4k/2^\ell \\ &= O(\ell^2)e^{-(9/4)k^4\gamma^3\ell^{1/3}\log^{2/3}\ell} \\ &\quad + \ell^{O(1)}(kz)^{2\ell/\gamma} + 4k/2^\ell. \qquad (2.34)\end{aligned}$$

The log of the first term on the right of (2.34) is asymptotic to $-\frac{9}{4}k^4\gamma^3\,\ell^{1/3}\log^{2/3}\ell$ while (from (2.31)) the log of the second term is bounded below by

$$-\frac{2\ell}{\gamma}\left(\frac{\log\ell}{\ell}\right)^{2/3} + O(\log\ell) \sim -\frac{2}{\gamma}\ell^{1/3}\log^{2/3}\ell,$$

so one easily sees (using the naive bound $\gamma \geq k^{-1}$) that the middle term on the right of (2.34) dominates the other two terms for large ℓ. Hence

$$f_{\ell,h}(z) \leq \ell^{O(1)}(kz)^{2\ell/\gamma} \tag{2.35}$$

and thus (2.32) holds. ∎

A direct application of Lemma 2.2 will not suffice for our needs, since we want to investigate the behavior of $f_{\ell,h}$ for fixed h as $\ell \to \infty$. The fact that $\mathcal{M}_h$ satisfies part (a) of Definition 3 will yield a subadditivity property of $\log f_{\ell,h}$ that can be used to overcome this problem. (Subadditivity of a related generating function was used in the classic paper [1].)

LEMMA 2.3. $\log f_{\ell+\ell',h}(z) \le \log f_{\ell,h}(z) + \log f_{\ell',h}(z)$.

Proof. Consider any two sequences $p_1, p_2, \ldots, p_\ell$ and $p'_1, p'_2, \ldots, p'_{\ell'}$ of minimal match pairs. From part (a) of Definition 3 we know that M_h can accept $p_1, p_2, \ldots, p_\ell, p'_1, p'_2, \ldots, p'_{\ell'}$ only if it accepts $p_1, p_2, \ldots, p_\ell$ and it accepts $p'_1, p'_2, \ldots, p'_{\ell'}$. Considering all of the ways that n characters could be split between $p_1, p_2, \ldots, p_\ell$ and $p'_1, p'_2, \ldots, p'_{\ell'}$, we see that

$$F(n, \ell+\ell', h) \le \sum_{i=0}^{n} F(i, \ell, h) F(n-i, \ell', h),$$

so

$$\begin{aligned} f_{\ell+\ell',h}(z) &= \sum_{n=0}^{\infty} F(n, \ell+\ell', h) z^n \\ &\le \sum_{n=0}^{\infty} \sum_{i=0}^{n} F(i, \ell, h) F(n-i, \ell', h) z^n \\ &= f_{\ell,h}(z) f_{\ell',h}(z), \end{aligned}$$

from which the Lemma follows immediately. ∎

The following trivial technical lemma will be useful in the proof of Theorem 1.4.

LEMMA 2.4. *For large enough h (depending only on k) the following will hold. Let ℓ be an integer with*

$$\ell \ge h \tag{2.36}$$

and let

$$d = \left\lceil \frac{2c\,\ell}{h} \right\rceil \quad \text{and} \quad \bar{\ell} = \frac{\ell}{d}. \tag{2.37}$$

Then if ℓ' is $\left\lfloor \bar{\ell} \right\rfloor$ or $\left\lceil \bar{\ell} \right\rceil$ we have

$$\frac{h}{3c} \le \ell' \le \frac{h}{c} \tag{2.38}$$

and

$$\frac{\log \ell'}{\ell'} \ge \frac{\log h}{h} \ge \frac{\log \ell'}{3c\,\ell'}. \tag{2.39}$$

Proof. We have

$$d = \left\lceil \frac{2c\,\ell}{h} \right\rceil \le \frac{2c\,\ell}{h} + 1,$$

so

$$\begin{aligned} \left\lfloor \bar{\ell} \right\rfloor &= \left\lfloor \frac{\ell}{d} \right\rfloor \ge \frac{\ell - (d-1)}{d} \\ &\ge \frac{\ell - \frac{2c\,\ell}{h}}{\frac{2c\,\ell}{h} + 1} = \frac{\ell h - 2c\,\ell}{2c\,\ell + h} \ge \frac{h}{3c} \end{aligned} \tag{2.40}$$

where the last step holds for large h since we chose $c \ge 2$ (in the statement of Lemma 2.1). This establishes the left inequality in (2.38). Also, since $d \ge 2c\,\ell/h$ we have

$$\left\lceil \bar{\ell} \right\rceil = \left\lceil \frac{\ell}{d} \right\rceil \le \frac{\ell}{2c\,\ell/h} + 1 = \frac{h}{2c} + 1 \le \frac{h}{c}$$

for large h, establishing the right inequality in (2.38).

Since $\log \ell'/\ell'$ is decreasing in ℓ' for $\ell' \ge e$ we have from (2.38), for large enough h,

$$\frac{\log \ell'}{\ell'} \le \frac{\log(h/(3c))}{h/(3c)} \le 3c \frac{\log h}{h}$$

and

$$\frac{\log \ell'}{\ell'} \ge \frac{\log(h/c)}{h/c} = c\frac{\log(h/c)}{h} \ge \frac{c}{2} \cdot \frac{\log h}{h} \ge \frac{\log h}{h},$$

establishing (2.39). ∎

We can now complete this section by giving the proof of Theorem 1.4.

Proof. Assume ℓ is an integer with

$$\ell \ge h \tag{2.41}$$

and let

$$d = \left\lceil \frac{2c\,\ell}{h} \right\rceil \quad \text{and} \quad \bar{\ell} = \frac{\ell}{d}. \tag{2.42}$$

Decompose ℓ as

$$\ell = \sum_{i=1}^{d} \ell_i$$

where each ℓ_i is $\left\lfloor \bar{\ell} \right\rfloor$ or $\left\lceil \bar{\ell} \right\rceil$. By Lemma 2.3 we have

$$\log f_{\ell,h}(z) \le \sum_{i=1}^{d} \log f_{\ell_i,h}(z). \tag{2.43}$$

Define z by

$$\log(kz) = -\left(\frac{\log h}{h}\right)^{2/3}. \tag{2.44}$$

Then for large h, using Lemma 2.4 we can apply Lemma 2.2 (with the ℓ_i, which have value $\left\lfloor \bar{\ell} \right\rfloor$ or $\left\lceil \bar{\ell} \right\rceil$, playing the role of ℓ in that Lemma) to the right of (2.43) to conclude that

$$\log f_{\ell,h}(z) \le \frac{2\ell}{\gamma} \log(kz) + O(d \log \bar{\ell}).$$

Thus the condition (1.6) holds with

$$\begin{aligned}\log\lambda(z) &= \frac{2}{\gamma}\log(kz) + O\left(\frac{d}{\ell}\log\bar{\ell}\right)\\ &= \frac{2}{\gamma}\log(kz) + O(\bar{\ell}^{-1}\log\bar{\ell})\\ &= \frac{2}{\gamma}\log(kz) + O(h^{-1}\log h),\end{aligned}$$

(2.45)

so the bound obtained from an application of Theorem 1.2 is

$$\begin{aligned}&\frac{2\log(kz)}{(2/\gamma)\log(kz) + O(h^{-1}\log h)}\\ &\quad= \gamma + O\left(\frac{h^{-1}\log h}{|\log(kz)|}\right)\\ &\quad= \gamma + O\left(\frac{h^{-1}\log h}{h^{-2/3}\log^{2/3} h}\right)\\ &\qquad\text{by (2.44)}\\ &\quad= \gamma + O\left(\left(\frac{\log h}{h}\right)^{1/3}\right),\end{aligned}$$

as desired. ■

3 Bounds based on domination

[3, 4] define a *collation of order* ℓ to be a list of ℓ match pairs followed by one more string pair, possibly $\binom{\epsilon}{\epsilon}$. Each collation has a key, which we can define as follows: Let s_i be the total size of the first i string pairs; then the key is $(s_\ell, s_\ell - 1, \ldots, s_1)$. They say that one collation generating a given string pair P *dominates* another collation of the same order generating P if it has a lexicographically smaller key. They further define a machine $\mathcal{M}$ that rejects only when it can determine that the match pairs read so far could not be the ending of an undominated collation. (Actually their machine reads the list of minimal match pairs from right to left, instead of left to right as in this paper, but this does not affect our analysis.) Clearly every string pair P has an undominated collation of order $\ell = lcs(P)$, and they prove that each of first ℓ string pairs in this collation is minimal. Thus it follows that $\mathcal{M}$ covers Σ^*, so Theorem 1.2 can be used to give bounds on γ_k. They use this to obtain the best known upper bounds on γ_2 as of their writing, and leave as an open question whether their method can yield a sequence of upper bounds converging to γ_2 [4, p. 456].

They do not explicitly give a method for obtaining a sequence of machines yielding improved bounds, but here we discuss a very natural extension of their method, and conjecture that for alphabet size $k = 2$ it does meet the conditions of Theorem 1.4 and thus does yield arbitrarily close upper bounds.

We (trivially) extend their definition of domination to lists of minimal match pairs by saying that a list L of ℓ minimal match pairs is undominated if the collation of order ℓ obtained by appending $\binom{\epsilon}{\epsilon}$ to L is undominated. Construct $\mathcal{M}_h$ as follows. Enumerate all lists of minimal match pairs of size at most h that are dominated, but have no contiguous proper sublist dominated; call this $\mathcal{D}_h$. Now let $\mathcal{M}_h$ be the machine that reads in lists of minimal match pairs and rejects whenever it finds one of the elements of $\mathcal{D}_h$ appearing as a contiguous sublist of its input. It is easy to see that this can be done by a finite state machine, and that this machine will never reject an undominated collation, so by Theorem 1.2 it gives a valid bound for arbitrary k. We will assume in this section that the machine is minimized so it has no distinct equivalent states.

Next we show that this approach meets the conditions of Theorem 1.3. The following lemma will be useful; let $\binom{0}{0}^\ell$ denote a list consisting of ℓ copies of $\binom{0}{0}$.

LEMMA 3.1. *If $\mathcal{M}_h$ rejects $L \parallel \binom{0}{0}^{\lceil h/2\rceil} \parallel L'$, then it must also reject L or L'.*

Proof. It is not hard to see that if a list L of minimal match pairs is undominated, both the list $\binom{0}{0} \parallel L$ and the list $L \parallel \binom{0}{0}$ must also be undominated. Thus $\mathcal{D}_h$ contains no lists beginning or ending with $\binom{0}{0}$. If $\mathcal{M}_h$ rejects $L \parallel \binom{0}{0}^{\lceil h/2\rceil} \parallel L'$ it must be that some list D in $\mathcal{D}_h$ appears as a sublist of $L \parallel \binom{0}{0}^{\lceil h/2\rceil} \parallel L'$. Since D has size at most h it has length at most $h/2$. Since it cannot begin or end with $\binom{0}{0}$, it must then be a sublist of either L or L', proving the lemma. ■

From this lemma (and the fact that the finite state machine is minimized), it follows that whenever the machine has not yet rejected and then sees a sequence of $\lceil h/2\rceil$ copies of $\binom{0}{0}$, it will return to the starting state. Also, if the machine is in the start state and reads $\binom{0}{0}$ it will remain in the start state. Hence the machine is regular as required by Theorem 1.3.

The sequence $\mathcal{M}_h$ trivially meets condition (a) of Definition 3, but at first it might seem that any method based on undominated collations could not meet condition (b): It is easy to give a string pair that has more than two undominated collations. For example, $\binom{01}{10}$ can be written as $\binom{01}{1}\binom{\epsilon}{0}$ or as $\binom{0}{10}\binom{1}{\epsilon}$. This does not give a counterexample to the condition (a) however, since the pair does not end on an essential match. If a list of L minimal match pairs is undominated then *generate*(L) must end on an essential match. We conjecture that no string pair $\binom{a_1a_2\cdots a_n}{b_1b_2\cdots b_m}$ over an

alphabet of size $k = 2$, that ends on an essential match, is generated by two distinct undominated lists of minimal match pairs. If this conjecture can be proved, it would establish that the sequence $\mathcal{M}_h$ based on domination efficiently covers $\{0,1\}$ and thus gives bounds coming arbitrarily close to γ_2.

One easily establishes that this conjecture does not hold for any alphabets of size larger than 2. An easy counterexample is the string pair $\binom{012}{102}$, which has both $\binom{01}{1}\binom{2}{02}$ and $\binom{0}{10}\binom{12}{2}$ as undominated lists of minimal match pairs. A more interesting example is provided by the string pair $\binom{10120}{0210}$, which has both $\binom{10}{0}\binom{1}{21}\binom{20}{0}$ and $\binom{10}{0}\binom{12}{2}\binom{0}{10}$ as undominated lists of minimal match pairs. Thus we cannot apply Theorem 1.4 to the sequence $\mathcal{M}_h$ when the alphabet size is more than 2. Of course, the fact that this particular theorem does not apply leaves open the possibility that $\mathcal{M}_h$, or perhaps some other construction based on undominated collations, produces arbitrarily good approximations for an alphabet of size greater than 2.

4 Bounds based on canonicity

We begin by briefly reviewing the upper bound approach used in [6]. (There it was used only for the case $k = 2$, but here we state it for the case of general alphabet size.)

The construction of [6] is based on the well-known dynamic programming approach for computing longest common subsequences: Given a string pair $\binom{a_1 a_2 \cdots a_n}{b_1 b_2 \cdots b_m}$ we let $lcs[i,j]$ be the length of the longest common subsequence of $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$ and note that for positive i and j we have the recurrence

$$lcs[i,j] = \max \begin{cases} lcs[i-1,j] \\ lcs[i,j-1] \\ lcs[i-1,j-1]+1 \quad (\text{if } a_i = b_j) \end{cases} . \tag{4.46}$$

This computation is shown for the strings 011010 and 1010101 in Figure 1.

It is well-known that we can recover a longest common subsequence by backtracking through the table; when there is more than one longest common subsequence the order in which we consider the options as we backtrack determines the particular longest common subsequence we find. [6] defines the *canonical* longest common subsequence to be the one we obtain by considering options in the order in which they appear in (4.46), i.e.,

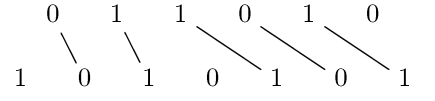
a) if $lcs[i,j] = lcs[i-1,j]$ move up from (i,j) to $(i-1,j)$,

b) else if $lcs[i,j] = lcs[i,j-1]$ move left from (i,j) to $(i,j-1)$,

a) The LCS dynamic programming table for $\binom{011010}{1010101}$ and the canonical backtracking.

		j	0	1	2	3	4	5	6	7
		b_j		1	0	1	0	1	0	1
i	a_i									
0			0	0	0	0	0	0	0	0
1	0		0	0	1	1	1	1	1	1
2	1		0	1	1	2	2	2	2	2
3	1		0	1	1	2	2	3	3	3
4	0		0	1	2	2	3	3	4	4
5	1		0	1	2	3	3	4	4	5
6	0		0	1	2	3	4	4	5	5

b) The canonical lcs is 01101, with the matching



The canonical decomposition is

$$\binom{0}{10}\binom{1}{1}\binom{1}{01}\binom{0}{0}\binom{1}{1}\binom{0}{\epsilon}.$$

Figure 1: Illustration of the well-known dynamic programming solution for the longest common subsequence problem, the definition of the canonical longest common subsequence, and the definition of a canonical decomposition. In part (a), the top two rows give the indices j and the characters b_j, and the left two columns give the indices i and the characters a_i. The remaining entries give the lcs values computed during the dynamic programming solution; the entry indexed by i and j is the length of the lcs of $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$. The unique canonical lcs for $a_1 a_2 \ldots a_6$ and $b_1 b_2 \ldots b_7$ is 01101, corresponding to the path in the diagram. Thus the canonical decomposition for these two strings is as shown in part (b).

c) else move diagonally up and to the left from (i, j) to $(i-1, j-1)$. In this case it must be that $lcs[i-1, j-1] = lcs[i-1, j] = lcs[i, j-1] = lcs[i, j] - 1$, so $a_i = b_j$. We match these two characters.

One easily sees the following.

LEMMA 4.1. [6] *Any pair of strings has exactly one canonical longest common subsequence.*

Define the *canonical path* to be the sequence of entries in the lcs table followed by the canonical backtracking, and the *canonical decomposition* of a string pair to be the list of string pairs we obtain by cutting after each match in the canonical lcs. These definitions are illustrated in Figure 1.

The machine $\mathcal{M}_h$ is designed to attempt to reject lists L of minimal match pairs that are not the canonical decomposition of generate(L). Since the machine must be finite, it cannot remember the entire lcs table or even the entire string read so far. Instead, it has states of the form (x, y, dx, dy) where x and y record the last h characters of the top and bottom input strings (respectively), and dx and dy record the last h differences in the last column and bottom row (respectively) of the lcs table. Thus x and y are strings in Σ^h and dx and dy are bit strings in $\{0,1\}^h$. There is also a rejecting state $\emptyset$. The starting state is $q_0 = (0^h, 0^h, 1^h, 1^h)$. For any minimal match pair p we have $\delta(\emptyset, p) = \emptyset$. For $q \in \mathcal{S}$, if the top or bottom string of p is longer than h we remove characters on the left until only h remain; this can be justified by an argument similar to that in the extended version of [6], and we omit the details. Then we apply the procedure delta of Figure 2 to compute $\delta(q, p)$. Lines 7–13 of this procedure fill in the new portion of the lcs table using (4.46); whenever the machine cannot reconstruct the value stored in a needed lcs entry from the state, it uses a lower bound corresponding to one of the first two cases in (4.46). This machine covers Σ^* (recall Definition 1) in the case $k = 2$ [6], and also for $k > 2$ by essentially the same argument. The key idea is that since values in the table not on the canonical path are lower bounds, they can only be less attractive choices during the canonical backtracking. Thus by Theorem 1.2 it can be used to find upper bounds on γ_k for arbitrary alphabet size.

We now show that $\mathcal{M}_h$ is regular, and that the sequence $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \ldots$ efficiently covers Σ^*. We begin by establishing some properties of the transition function δ of the finite state machine. Call the portion of the table *lcs* generated during the procedure of Figure 2 the limited-history table. The following two simple observations will be useful.

```
1. def procedure delta(q, p):
2.   (x, y, dx, dy) = q
3.   (t, b) = p
4.   lcs[h, h] = h
5.   decodeVertical((h, h), dx); decodeHorizontal((h, h), dy)
6.   a = x || t; b = y || b
7.   for i = 1 to length(a):
8.     for j = 1 to length(b):
9.       if (i > h) or (j > h):
10.        if i == 0: lcs[i, j] = lcs[i, j − 1]
11.        else if (j == 0): lcs[i, j] = lcs[i − 1, j]
12.        else if (a_i == b_j): lcs[i, j] = lcs[i − 1, j − 1] + 1
13.        else: lcs[i, j] = max(lcs[i, j − 1], lcs[i − 1, j])
14.  (i, j) = (length(a), length(b))
15.  while ((i ≥ h) and (j ≥ h)):
16.    if (lcs[i, j] == lcs[i − 1, j])): i−−        # try moving up
17.    else if (lcs[i, j] == lcs[i, j − 1]): j−−    # try moving left
18.    else: i−−; j−−                               # make a match
19.    if ((i == h) and (j == h)): # canonicity test succeeds
20.      last = (length(a), length(b))
21.      return (suffix(a, h), suffix(b, h),
           encodeVertical(last), encodeHorizontal(last))
22.  return ∅    # canonicity test fails
```

Figure 2: The transition function $\delta(q, p)$ for the machine $\mathcal{M}_h$ based on canonicity. The following functions are used:

length(a) returns the length of the string a.

suffix(a, h) returns the string consisting of the last h characters of a.

encodeVertical((i, j)) returns a bit vector of length h whose successive entries are the successive differences between the table entries $lcs[i-h, j]$, $lcs[i-h+1, j]$, $lcs[i-h+2, j]$, ..., $lcs[i, j]$.

encodeHorizontal((i, j)) performs the operation analogous to encodeVertical, but for horizontal differences. It returns a bit vector of length h whose successive entries are the successive differences between the table entries $lcs[i, j-h]$, $lcs[i, j-h+1]$, $lcs[i, j-h+2]$, ..., $lcs[i, j]$.

decodeVertical($(i, j), dx$) performs an operation inverse to encodeVertical; it makes the successive differences of the table entries $lcs[i-h, j]$, $lcs[i-h+1, j]$, $lcs[i-h+2, j]$, ..., $lcs[i, j]$ be the successive bits in the bit string dx, leaving $lcs[i, j]$ unchanged.

decodeHorizontal($(i, j), dy$) performs the operation analogous to decodeVertical, but for horizontal differences.

Also, in this figure only, || denotes concatenation of strings rather than concatenation of lists of string pairs.

LEMMA 4.2. *As we move one step down, to the right, or diagonally down and to the right, among locations for which the values in the limited-history table are computed, the table values either remain the same or increase by* 1.

LEMMA 4.3. *The result returned by the procedure delta of Figure 2 is remains the same if we*

a) initialize $lcs[h,h]$ *to an arbitrary integer instead of* h *in line 4, or*

b) translate all indices into the table lcs by a constant amount.

Now let $\mathcal{S}$ be the set of accepting states of the finite state machine with history h, so the entire set of states is $\mathcal{S} \cup \{\emptyset\}$. The next lemma asserts that if the machine is in any accepting state q and reads a match pair of the form $\binom{c}{c}$, it will remain in an accepting state.

LEMMA 4.4. *Let* c *be any character in* Σ *and let* $p = \binom{c}{c}$. *Then for any* $q \in \mathcal{S}$, *we have* $\delta(q,p) \in \mathcal{S}$.

Proof. Since $q \in \mathcal{S}$ we must have

$$lcs[h,h-1] = lcs[h-1,h] = h-1, \tag{4.47}$$

since otherwise q would have failed the canonicity test. The new characters read corresponding to the match pair p will be a_{h+1} and b_{h+1}; since these match we will have $lcs[h+1,h+1] = h+1$. The canonical backtracking test accepts if we move from $(h+1,h+1)$ to (h,h), which can fail to happen only if $lcs[h+1,h]$ or $lcs[h,h+1]$ is also equal to $h+1$. But this and (4.47) would contradict Lemma 4.2. ■

Let $\mathcal{S}_0$ be the set of states $\langle x,y,dx,dy\rangle \in \mathcal{S}$ in which dx and dy are both 1^h. Note that the start state of M_h is an element of $\mathcal{S}_0$.

LEMMA 4.5. *Let* c *be any character in* Σ *and let* $p = \binom{c}{c}$. *Then for any* $q \in \mathcal{S}_0$, *we have* $\delta(q,p) \in \mathcal{S}_0$. *In particular, if* q *is the start state* $(0^h,0^h,1^h,1^h)$, *then* $\delta\big(q,\binom{0}{0}\big) = q$.

Proof. The proof is illustrated in Figure 3; the portion of the figure above and to the left of the medium-thickness line shows the contents of the table reconstructed from q. The entry in position (h,h) is h, and then the remaining entries shown inside the box bounded by the thick and medium line are determined by the fact that dx and dy are each 1^h.

Let $\delta(q,p)$ be $q' = (x',y',dx',dy')$. The portion of the lcs table corresponding to q' is enclosed in the thin line. Following the procedure in Figure 2 produces the

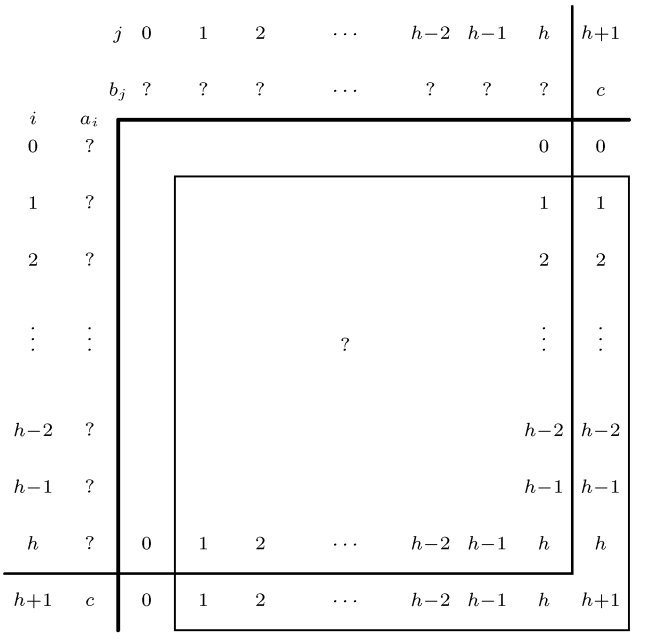


Figure 3: Illustration for the proof of Lemma 4.5.

new entries shown. (The value at $lcs[h+1,h+1]$ is $h+1$ since a_{h+1} matches b_{h+1}; note that otherwise the values of the new entries do not depend on whether or not any of the characters match.) Thus dx' and dy' will also consist of strings of h one bits. ■

LEMMA 4.6. *Let* $q \in \mathcal{S}$ *be any accepting state. Assume that starting in state* q, *the machine reads* h *copies of* $\binom{1}{1}$ *followed by* h *copies of* $\binom{0}{0}$. *Then the new state of the machine will be the start state* $(0^h,0^h,1^h,1^h)$.

Proof. The proof is illustrated in Figure 4; here Lemma 4.3 is used to show multiple calls to delta in a single table. The portion of the figure above and to the left of the thin line shows the state of the table after starting in state q and then reading in h copies of $\binom{1}{1}$. By Lemma 4.4 we know that this state is in $\mathcal{S}$, so we must have

$$lcs[h-1,h] < h \quad \text{and} \quad lcs[h,h-1] < h.$$

Thus all of the entries to the left of or above $lcs[h,h]$ must be less than h. Note that at this point all of the characters stored in the state are ones.

Now consider what happens when we read in the h copies of $\binom{0}{0}$, which is illustrated in the remainder of the figure. All of the new entries $lcs[i,j]$ where $j < h$ will continue to be less than h since the third case of (4.46) will never be used, so in particular $lcs[2h,h-1]$ is less than h. On the other hand, as we step down the main diagonal from $lcs[h,h]$ to $lcs[2h,2h]$, the entries increase

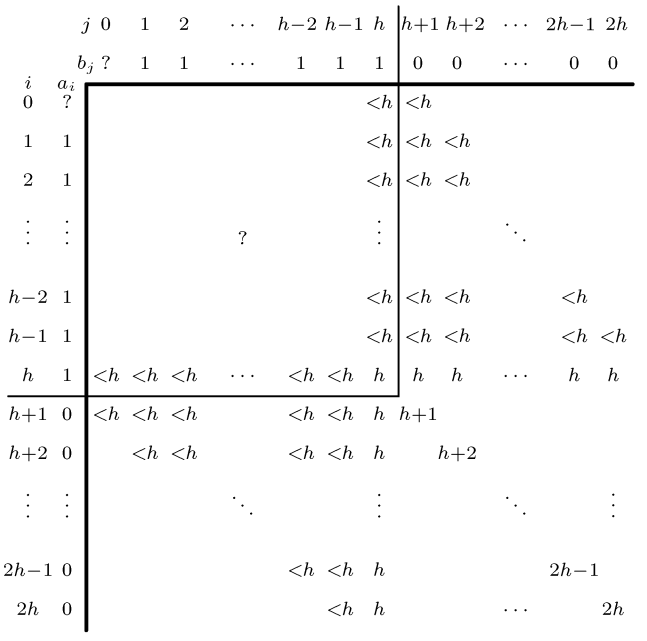


Figure 4: Illustration for the proof of Lemma 4.6.

by 1 since at each step we match a 0 with a 0, so we will have $lcs[2h, 2h] = 2h$. But then by Lemma 4.2 we must have $lcs[2h, j] = j$ for $h \le j \le 2h$. Similarly we have $lcs[i, 2h] = i$ for $h \le i \le 2h$. Thus encoding the bottom-right portion of the table yields the state asserted in the lemma. ■

LEMMA 4.7. *Let $P = p_1, p_2, \ldots, p_m$ be any sequence of minimal match pairs, and let $q \in \mathcal{S}$ and $q' \in \mathcal{S}_0$. Then if the machine starting in state q accepts P, the machine starting in state q' must also accept P.*

Proof. First consider how the limited-history lcs table evolves if we start in $q' = \langle x', y', dx', dy' \rangle$ and then read in the successive minimal match pairs of P; this is illustrated for a specific example of a P and a $q' \in S_0$ in Figure 5. Let lcs' denote the entries in this table, so $lcs'[h, h] = h$. Note that since dy' is a string of h 1-bits, decoding dy' yields $lcs'[h, j] = j$ for $0 \le j \le h$. It follows by inspection of the algorithm of Figure 2 and an easy induction that for any entry $lcs'[i, j]$ that we compute we have

$$(4.48) \qquad i \ge h \text{ and } j \le h \implies lcs'[i, j] = j.$$

(In Figure 5 these entries are shown in a slanted font.) To see this, note that in those cases where we simply copy an entry from the one above the inductive step is trivial; on the other hand, in the cases where we compute an entry $lcs'[i, j]$ from the recurrence (4.46), by the inductive hypothesis we have $lcs'[i-1, j-1] =$

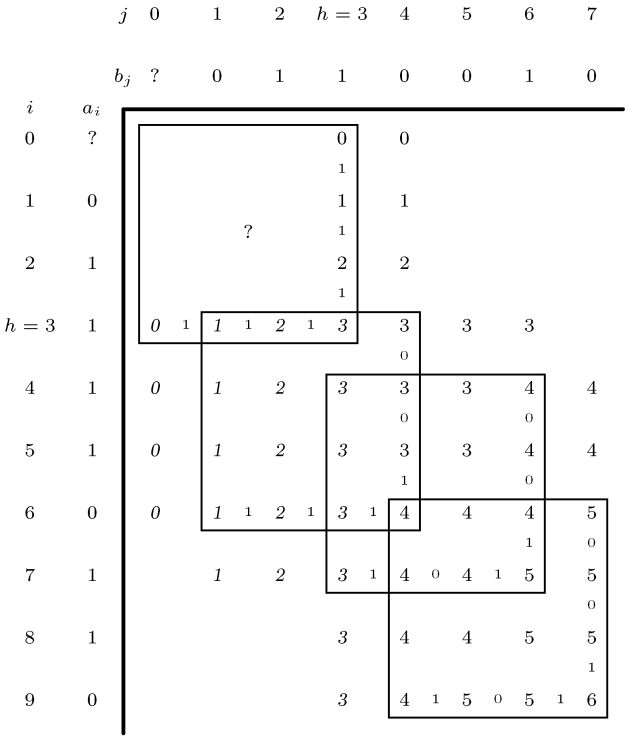


Figure 5: Illustration for the proof of Lemma 4.7, part (a). We start in state $q' = \langle 011, 011, 111, 111 \rangle$, read the input $P = \binom{110}{0}\binom{1}{01}\binom{10}{0}$, and end in state $\langle 110, 010, 001, 101 \rangle$. The portions of the lcs table corresponding to each successive state are outlined by squares with thin lines. The small integers between certain lcs table entries show the differences between these entries.

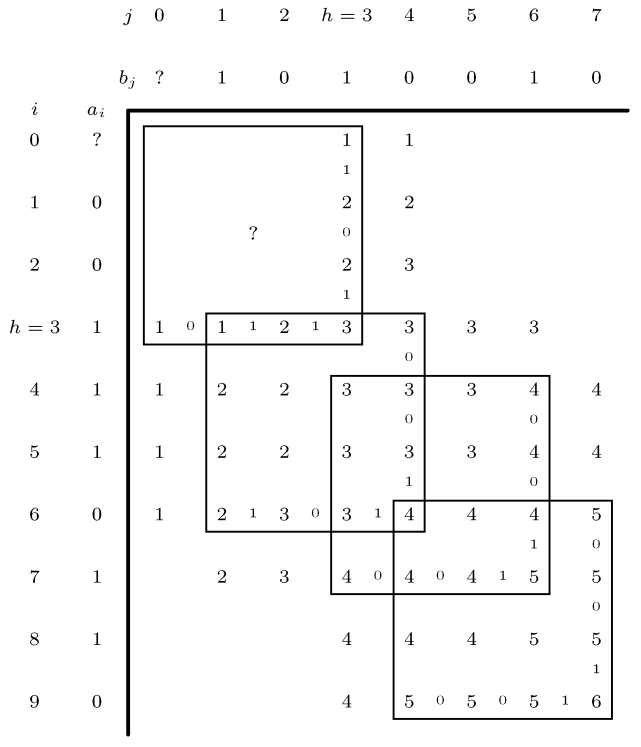


Figure 6: Illustration for the proof of Lemma 4.7, part (b). We start in state $q = \langle 001, 101, 101, 011\rangle$, read the input $P = \binom{110}{0}\binom{1}{01}\binom{10}{0}$, and end in state $\langle 110, 010, 001, 001\rangle$.

$lcs'[i, j-1] = j-1$ and $lcs'[i-1, j] = j$, so we set $lcs'[i, j]$ to j regardless of whether the relevant characters match. By a symmetric argument, for entries we compute we have

$$(4.49) \qquad i \le h \text{ and } j \ge h \implies lcs'[i, j] = i.$$

From (4.48) and (4.49) we conclude that, in particular,

$$(4.50) \qquad (i \ge h \text{ and } j = h) \text{ or } (i = h \text{ and } j \ge h) \implies lcs'[i, j] = h.$$

Next consider how the limited-history lcs table evolves if we start in $q = \langle x, y, dx, dy\rangle$ and then read in the successive minimal match pairs of P, letting lcs denote the entries in this table. (This is illustrated for a specific example of q, and for the same P used in Figure 5, in Figure 6, but one need not examine all of the details in Figure 6 to verify the proof.) From Lemma 4.2 it follows immediately that, for the table entries that we compute,

$$(4.51) \qquad (i \ge h \text{ and } j = h) \text{ or } (i = h \text{ and } j \ge h) \implies lcs[i, j] \ge h;$$

Combining this with (4.50) gives, for the table entries that we compute,

$$(4.52) \qquad (i \ge h \text{ and } j = h) \text{ or } (i = h \text{ and } j \ge h) \implies lcs[i, j] \ge lcs'[i, j];$$

All computed entries $lcs[i, j]$ for which $i \ge h$ and $j \ge h$ can be expressed as nondecreasing functions of the set of values bounded in (4.52), with no further dependence on $a_1a_2 \cdots a_h$ and $b_1b_2 \cdots b_h$, so we have

$$(4.53) \qquad i \ge h \text{ and } j \ge h \implies lcs[i, j] \ge lcs'[i, j].$$

Let p be the canonical backtracking path for the case in which the machine starts in state q. Since the machine starting in state q accepts P, the inequality in (4.53) must be tight along p. Thus the entries lying off p are never more attractive for the table starting in state q' than they were for the table starting in state q, so we must also accept P when we start in state q'. ■

(Note that it follows that all states in $\mathcal{S}_0$ are equivalent, but [6] did not take advantage of this in its calculations.)

From Lemma 4.6 we see that each $\mathcal{M}_h$ satisfies part (a) of Definition 2, and from Lemma 4.5 we see that it satisfies part (b). Thus it is regular, so Theorem 1.3 applies.

From Lemma 4.7 the sequence $\mathcal{M}_h$ satisfies part (a) of Definition 3. As long as the total size of the match pairs read does not exceed h, the machine models the process of computing the lcs table exactly, so by Lemma 4.1 we see that part (b) holds. Thus the sequence $\mathcal{M}_h$ efficiently covers Σ^*, so we have, by Theorem 1.4,

THEOREM 4.1. *For arbitrary alphabet size k, the sequence $\mathcal{M}_h$ based on canonicity is regular and efficiently covers Σ and thus gives bounds coming arbitrarily close to γ_k.*

5 Conclusion

This paper leaves open the question of whether an upper bound method based on domination [3, 4] can produce bounds on γ_k approaching the true value, but we conjecture that it does for an alphabet size of $k = 2$. This paper shows that upper bound methods based on canonicity can produce upper bounds on γ_k approaching the true value, for arbitrary alphabet size k. These methods tend to use a lot of time to achieve small improvements in known bounds, especially when k is large. It would be desirable to find methods to more quickly bound γ_k, particularly for large k.

6 Acknowledgements

We thank one of the anonymous referees of the journal submission of [6], and David Eppstein, for asking questions that encouraged us to pursue the issues addressed in this paper. We also thank Padhraic Smyth for bringing the book [8] to our attention.

References

[1] K. S. Alexander. The rate of convergence of the mean length of the longest common subsequence. *Annals of Applied Probability*, 4(4):1074–1082, 1994.

[2] V. Chvátal and D. Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12:306–315, 1975.

[3] V. Dančík. *Expected Length of Longest Common Subsequences*. PhD thesis, Department of Computer Science, University of Warwick, Sept. 1994.

[4] V. Dančík and M. Paterson. Upper bounds for the expected length of a longest common subsequence of two binary sequences. *Random Structures & Algorithms*, pages 449–458, 1995.

[5] M. Kiwi, M. Loebl, and J. Matoušek. Expected length of the longest common subsequence for large alphabets. *Advances in Mathematics*, 197(2):480–498, 2005.

[6] G. S. Lueker. Improved bounds on the average length of longest common subsequences. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 130–131, 2003. An extended version can be found at http://www.ics.uci.edu/∼lueker/papers/lcs/. Submitted for journal publication.

[7] C. R. MacCluer. The many proofs and applications of Perron's theorem. *SIAM Review*, 42(3):487–498, 2000.

[8] E. Seneta. *Non-Negative Matrices and Markov Chains*. Springer Series in Statistics. Springer, New York, 2nd edition, 2006.

[9] J. M. Steele. *Probability Theory and Combinatorial Optimization*. CMBS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1997.

Markovian embeddings of general random strings

Manuel E. Lladser*†

Abstract

Let $\mathcal{A}$ be a finite set and X a sequence of $\mathcal{A}$-valued random variables. We do not assume any particular correlation structure between these random variables; in particular, X may be a non-Markovian sequence. An adapted embedding of X is a sequence of the form $R(X_1)$, $R(X_1, X_2)$, $R(X_1, X_2, X_3)$, etc where R is a transformation defined over finite length sequences. In this extended abstract we characterize a wide class of adapted embeddings of X that result in a first-order homogeneous Markov chain. We show that any transformation R has a unique coarsest refinement R' in this class such that $R'(X_1)$, $R'(X_1, X_2)$, $R'(X_1, X_2, X_3)$, etc is Markovian. (By refinement we mean that $R'(u) = R'(v)$ implies $R(u) = R(v)$, and by coarsest refinement we mean that R' is a deterministic function of any other refinement of R in our class of transformations.) We propose a specific embedding that we denote as R^X which is particularly amenable for analyzing the occurrence of patterns described by regular expressions in X. A toy example of a non-Markovian sequence of 0's and 1's is analyzed thoroughly: discrete asymptotic distributions are established for the number of occurrences of a certain regular pattern in $X_1, ..., X_n$ as $n \to \infty$ whereas a Gaussian asymptotic distribution is shown to apply for another regular pattern.

1 Introduction

Imagine a gambling scenario where two players name different patterns of heads and tails and flip a coin until one of the patterns is observed. The difficulty in determining which pattern will be the first to be observed or how often will each pattern be observed when the coin is tossed infinitely often depends greatly on the complexity of the patterns involved as well as the probabilistic model of coin-tossing.

In a more general context we may consider a finite non-empty set $\mathcal{A}$ and an infinite sequence of $\mathcal{A}$-valued random variables $X = (X_n)_{n\geq 1}$.

We call $\mathcal{A}$ the *alphabet* and refer to its elements as *characters*. We use the script $\mathcal{A}^*$ to denote the set of all finite length sequences of elements in $\mathcal{A}$. We refer to elements in $\mathcal{A}^*$ as *words* or sometimes *strings*. In our context a *pattern* is any non-empty set $\mathcal{L} \subset \mathcal{A}^*$.

The *frequency statistics* associated with a pattern $\mathcal{L}$ are the random variables

$$S_n^{\mathcal{L}} := \sum_{i=1}^{n} [\![X_1 \cdots X_i \in \mathcal{L}]\!],$$

with $n \geq 1$. Above $X_1 \cdots X_i$ is a shortcut notation for the random word $(X_1, \ldots, X_i)$. Furthermore, $[\![\cdot]\!]$ denotes the Iverson's bracket i.e. a quantity defined to be 1 whenever the statement within the brackets is true, and 0 otherwise. Equivalently, $[\![X_1 \cdots X_i \in \mathcal{L}]\!]$ is the indicator function of the event $[X_1 \cdots X_i \in \mathcal{L}]$.

To fix some ideas about the above definition consider a nonempty set $\mathcal{W} \subset \mathcal{A}^*$. If $\mathcal{W}$ is *suffix reduced* i.e. no word in $\mathcal{W}$ is a proper suffix of another word in $\mathcal{W}$ and $\mathcal{L} = \mathcal{A}^*\mathcal{W}$ i.e. $x \in \mathcal{L}$ if and only if x has a suffix in $\mathcal{W}$ then $S_n^{\mathcal{L}}$ corresponds to the number of substrings of $X_1 \cdots X_n$ that belong to $\mathcal{W}$.

Much of the efforts undertaken in the literature for studying patterns in random strings have been about determining the exact and/or asymptotic distribution of the frequency statistics of one or more patterns. Various techniques have been utilized for this effect for different types of patterns as well as probabilistic models for X. These have included *martingale methods* [15, 21], *combinatorial methods* [12, 2], *renewal arguments* and *formal language recursions* [23, 10], *large deviations* [22], and *symbolic dynamics* [5, 6], among many others.

Perhaps the most commonly used technique for analyzing patterns in random strings is the *(finite) Markov chain embedding technique* (MCET). It originated in [11, 3] as a technique for analyzing patterns described by a finite set of words in the context of memoryless sequences. The case of first-order homogeneous Markov sequences was provided in [4]. An important extension of the technique to consider general *Markovian models* i.e. when X is a k-th order homogeneous Markov chain and *regular patterns* i.e. sets of words described by regular expressions is due to [19] and the follow up work in [18]. Minimality considerations about the automata needed for analyzing the frequency statistics of regular patterns under Markovian models were addressed in [16].

Broadly speaking, the Markov chain embedding

*The University of Colorado, Department of Applied Mathematics, PO Box 526 UCB, Boulder, CO 80309-0526, The United States

†e-mail: manuel.lladser@colorado.edu

technique consists in transforming X into a first-order homogeneous Markov chain that is informative of a pattern of interest. In [17] the embedding of X into a *deterministic finite automaton* (DFA) $G = (V, \mathcal{A}, f, q, T)$ that recognizes a regular pattern $\mathcal{L}$ is defined as the infinite sequence of V-valued random variables [1]

$$X_n^G := f(q, X_1 \cdots X_n). \tag{1.1}$$

We refer to $X^G = (X_n^G)_{n\geq 1}$ as the *embedding of X into G*. If X^G is a first-order homogeneous Markov chain then $S_n^{\mathcal{L}}$ has the same distribution as the number of visits that X^G makes to T in the first n steps. In this case, *transfer matrix methods* of the type implemented in [19, 18] can be used to determine the generating function associated with the frequency statistics of the regular pattern $\mathcal{L}$.

In order for the MCET to work the embedding X^G must be Markovian. This requires a level of compatibility between the distribution of X and the transition function of G. To fix ideas consider a first-order homogeneous Markov chain X with state space $\{a, b\}$ and the regular languages $\mathcal{L}_1 = \{a, b\}^*\{ba\}$ and $\mathcal{L}_2 = \{a, b\}^*\{abba\}$. If AC is the Aho-Corasick automaton [1] associated with the keywords 'ba' and '$abba$' (see Figure 1) then X^{AC} is known to be a first-order homogeneous Markov chain because AC conveys a memory length of order one [16] i.e. each state accessible from the initial state is informative of the last character processed by the automaton. In particular, if

$$\begin{aligned} P[X_1 = a] &= \mu; \\ P[X_1 = b] &= (1-\mu); \\ P[X_{n+1} = a \mid X_n = a] &= p; \\ P[X_{n+1} = b \mid X_n = a] &= (1-p); \\ P[X_{n+1} = a \mid X_n = b] &= q; \\ P[X_{n+1} = b \mid X_n = b] &= (1-q); \end{aligned}$$

then

$$P[S_n^{\mathcal{L}_1} = k_1, S_n^{\mathcal{L}_2} = k_2] = [x^{k_1} y^{k_2}]\, \mu^T \cdot P_{x,y}^{n-1} \cdot \mathbf{1} \tag{1.2}$$

where

$$\mu^T := \begin{bmatrix} \mu & (1-\mu) & 0 & 0 & 0 & 0 \end{bmatrix};$$

$$P_{x,y} := \begin{bmatrix} p & 0 & (1-p) & 0 & 0 & 0 \\ 0 & (1-q) & 0 & x\cdot q & 0 & 0 \\ 0 & 0 & 0 & x\cdot q & (1-q) & 0 \\ p & 0 & (1-p) & 0 & 0 & 0 \\ 0 & (1-q) & 0 & 0 & 0 & xy\cdot q \\ p & 0 & (1-p) & 0 & 0 & 0 \end{bmatrix};$$

$$\mathbf{1}^T := \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

[1] V denotes the (finite) state space of G, $f : V \times \mathcal{A}^* \to V$ its transition function, $q \in V$ its initial state, and $T \subset V$ its set of terminal states.

The row-vector μ^T corresponds to the initial distribution of X^{AC} according to the labels assigned in Figure 1. $P_{x,y}$ is the probability transition matrix of X^{AC} but after multiplying by the (marker) variable x each column associated with a state in AC that contributes to an occurrence of the keyword 'ba'. Similarly the variable y marks matches with the keyword '$abba$'. In particular, using (1.2) we obtain that the moment generating function associated with the random vector $(S_n^{\mathcal{L}_1}, S_n^{\mathcal{L}_2})$ is given by the formula

$$\sum_{k_1,k_2\geq 0, n\geq 1} P[S_n^{\mathcal{L}_1} = k_1, S_n^{\mathcal{L}_2} = k_2]\, x^{k_1}\, y^{k_2}\, z^n$$

$$= z \cdot \mu^T \cdot (\mathbb{I} - z \cdot P_{x,y})^{-1} \cdot \mathbf{1},$$

where $\mathbb{I}$ is the 6×6 identity matrix.

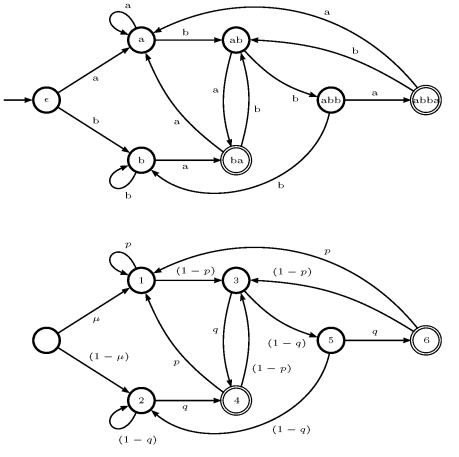


Figure 1: Top, Aho-Corasick automaton AC that recognizes the regular language $\{a, b\}^*\{ba, abba\}$. ϵ is the initial state. ba and $abba$ are terminal states. Transitions into state ba correspond to matches with keyword 'ba' that do not contribute to a match with the keyword '$abba$'. Transitions into state $abba$ correspond to matches with 'ba' and '$abba$'. Bottom, embedding of a general first-order homogeneous Markov chain X with state space $\{a, b\}$ into AC.

The automaton AC also conveys a memory length of order 2; in particular, the embedding of any second-order homogeneous Markov chain X into G is also Markovian [16]. On the contrary, for a general third-order homogeneous Markov chain X over $\{a, b\}$, the Aho-Corasick automaton AC in Figure 1 may not be suitable for a Markovian embedding: starting from the initial state it is possible to reach state a using the text '$aaaaa$' but also '$babaa$', in particular, state a is not informative of the last three characters fed into the automaton. This nuisance can be resolved by

duplicating states until each state becomes informative of the last three characters fed into the automaton [19, 18], or by considering the product between AC and a de Bruijn like graph [16].

In this extended abstract we develop some theoretical bases for a general MCET to consider arbitrary probabilistic models and patterns. By arbitrary probabilistic models we mean possibly non-Markovian models. By arbitrary patterns we mean possibly non-regular ones. The starting point of our approach is the observation that the embedding in (1.1) may be extrapolated as a sequence of the form $R(X_1)$, $R(X_1X_2)$, $R(X_1X_2X_3)$, etc where R is a transformation defined over $\mathcal{A}^*$ and taking values in a certain space $\mathcal{C}$. In compact form we write X^R to refer to the transformed sequence i.e.

$$X_n^R := R(X_1 \cdots X_n).$$

Two natural questions in this context are:

(a) *what conditions are necessary and sufficient in order for X^R to be a first-order homogeneous Markov chain for a possibly non-Markovian sequence X?*, and

(b) given a possibly non-regular pattern $\mathcal{L}$, *is there an R such that X^R is a Markov chain informative of the frequency statistics of $\mathcal{L}$ but also with the 'least' number of states?*

Observe that the actual range of R is not really relevant for answering the above questions: any set in one-to-one correspondence with it may be regarded as the range without affecting the Markovianity of X^R nor the number of states that this chain could potentially visit. In particular, the level curves of R are the most natural choice for the range of the transformation. Equivalently, since the level curves of R form a partition of $\mathcal{A}^*$, we may think of R as an equivalence relation defined over $\mathcal{A}^*$. Henceforth $c \in R$ will mean that c is an equivalence class of R and, for $x \in \mathcal{A}^*$, $R(x)$ will denote the unique equivalence class that contains x. (This notation allows to think of R simultaneously as an equivalence relation and as a transformation.) In particular, $R(x) = R(y)$ is equivalent to having $x\,R\,y$ i.e. that x and y are in the same equivalence class of R. Furthermore, $x \in c$ is equivalent to having $R(x) = c$.

To fix some ideas we consider some examples. If R_1 is the relation defined as $x\,R_1\,y$ if and only if $x = y$ then $R_1(x) = \{x\}$. We refer to this relation as the *identity relation*. Observe that the equivalence classes of R_1 correspond to the level curves of any one-to-one transformation defined over $\mathcal{A}^*$. On the other hand, if R_2 is such that $x\,R_2\,y$ for all $x, y \in \mathcal{A}^*$ then $R_2(x) = \mathcal{A}^*$. We call R_2 the *coarsest relation*. The equivalence classes of R_2 correspond to the level curves of any constant transformation defined over $\mathcal{A}^*$.

1.1 Outline of the paper. In §2 we introduce a class of equivalence relations which are guarantee to produce Markovian embeddings. A characterization of the equivalence relations in this class is provided in Theorem 2.1. Our result resembles the characterization of strong lumpability (also called strong state space aggregation) for Markov chains [14], but in the more general context of possibly non-Markovian sequences. Theorem 2.2 asserts that it is always possible to refine the equivalence classes of a given relation so as to obtain an equivalence relation in the class of transformations characterized by Theorem 2.1, but with the least possible number of equivalence classes. Our result touches bases with the algorithm proposed in [20] and implemented in [8] for finding the optimal strongly lumpable refinement of a partition of the state space of a Markov chain. At the end of §2 we introduce an equivalence relation whose equivalence classes are defined in terms of the distribution of X. Some key properties of this relation are presented in Theorem 2.3. (The proofs of our results in §2 are omitted from this extended abstract and will be provided in the final version of the paper.) In §3 we review briefly some basic limit theorems for Markov chains and see how the results of §2 fit for analyzing the frequency statistics of general patterns in general random strings. The end of §3 is devoted to a case study of a non-Markovian sequence and the frequency statistics of two regular patterns. The study reveals new phenomena which has not been previously observed, even in the context of probabilistic dynamical models [6].

2 Main definitions and results

For an integer $n > 0$, $\mathcal{A}^n$ denotes the set of all sequences of n elements in $\mathcal{A}$. We define $\mathcal{A}^0 = \{\epsilon\}$, where $\epsilon \notin \mathcal{A}$ is by definition the *empty word*. We also define $|x| = n$ whenever $x \in \mathcal{A}^n$. We call $|x|$ the *length of* x. Observe that $x = \epsilon$ if and only if $|x| = 0$. Define $\mathcal{A}^* := \cup_{n\geq 0}\mathcal{A}^n$.

In what follows, $X = (X_n)_{n\geq 1}$ is a given infinite sequence of $\mathcal{A}$-valued random variables defined on a common probability space $(\Omega, \mathcal{F}, P)$. For $x \in \mathcal{A}^*$, $|x| > 0$, we use the notation $[X = x...]$ as a shortcut of the event $[X_1 \cdots X_{|x|} = x]$. We define $[X = \epsilon...] = \Omega$. The *support of* X is the set

$$\mathrm{supp}(X) := \{x \in \mathcal{A}^* : P[X = x...] > 0\}.$$

Observe that the above set is nonempty because $\epsilon \in \mathrm{supp}(X)$. Furthermore, $X_1 \cdots X_n \in \mathrm{supp}(X)$ almost surely for all $n \geq 1$.

Henceforth R will denote an equivalence relation

defined over $\mathrm{supp}(X)$. For technical purposes we extend any such relation to an equivalence relation over $\mathcal{A}^*$ by defining

$$R(x) := \mathcal{A}^* \setminus \mathrm{supp}(X), \quad x \notin \mathrm{supp}(X).$$

DEFINITION 2.1. We will say that X *is embedable w.r.t.* R provided that for all $x, y \in \mathrm{supp}(X)$ and $c \in R$, if $R(x) = R(y)$ then

$$\sum_{\alpha \in \mathcal{A}: R(x\alpha)=c} P[X = x\alpha... \mid X = x...] = \sum_{\alpha \in \mathcal{A}: R(y\alpha)=c} P[X = y\alpha... \mid X = y...].$$

Observe that the left- and right-hand side above are identically zero when $c \cap \mathrm{supp}(X) = \emptyset$. As a result, to check embedability we may always assume that $c \subset \mathrm{supp}(X)$.

It is direct to check that X is embedable w.r.t. the restriction to $\mathrm{supp}(X)$ of the identity relation and the coarsest relation. In particular there exist equivalence relations w.r.t. which X is embedable. Observe however that none of these equivalence relations takes into account the actual distribution of X in their definition. An important instance of such a relation is provided in section §2.3.

2.1 Characterization of embedability. Before stating our first result we need introduce some notation. In what follows, the script μ is reserved to denote a probability measure defined over $\mathrm{supp}(X)$. Since this last set is countable, we will write $\mu(x)$ instead of $\mu(\{x\})$. Furthermore, we use the notation $X^\mu = (X_n^\mu)_{n\geq 0}$ to refer to the unique stochastic process in $\mathrm{supp}(X)$ such that

$$X_0^\mu \stackrel{d}{=} \mu, \tag{2.3}$$

and for all $n \geq 1$ and $x \in \mathrm{supp}(X)$ such that $\mu(x) > 0$

$$X_{|x|+1} \cdots X_{|x|+n} \mid [X_1 \cdots X_{|x|} = x] \stackrel{d}{=} X_1^\mu \cdots X_n^\mu \mid [X_0^\mu = x]. \tag{2.4}$$

Due to condition (2.3), X_0^μ is a $\mathrm{supp}(X)$-valued random variable. Condition (2.4) is equivalent to requesting that μ-almost surely for all x, the conditional distribution of the stochastic process $(X_n^\mu)_{n\geq 1}$ given that $X_0^\mu = x$ is the same as the conditional distribution of $(X_{n+|x|})_{n\geq 1}$ given the event $[X = x...]$. In particular, for all $n \geq 1$, X_n^μ is an $\mathcal{A}$-valued random variable.

THEOREM 2.1. *X is embedable w.r.t. R if and only if, for all μ, the stochastic process $Y = (Y_n)_{n\geq 0}$, with $Y_n := R(X_0^\mu \cdots X_n^\mu)$, is a first-order homogeneous Markov chain with probability transitions that do not depend upon μ.*

Observe that if $\mu = \delta_\epsilon$ (the probability mass function at ϵ) then $(X_n^\mu)_{n\geq 1}$ has the same distribution as X. In particular, the following result is an immediate consequence of the above theorem.

COROLLARY 2.1. *If X is embedable w.r.t. R then the stochastic process X^R is a first-order homogeneous Markov chain.*

2.2 Coarsest embedable refinement. For a given equivalence relation R defined over $\mathrm{supp}(X)$ the process X may or may not be embedable w.r.t. R. However the following result asserts that it is always possible to refine the equivalence classes of R so as to obtain an equivalence relation with the 'least' number of equivalence classes w.r.t. which X is embedable.

THEOREM 2.2. *For each equivalence relation R defined over supp(X), there exists a unique coarsest refinement R' of R w.r.t. which X is embedable.*

2.3 Markov relation induced by X. An equivalence relation R is said to be *right-invariant* if for all $x, y \in \mathcal{A}^*$ the condition $R(x) = R(y)$ implies that $R(x\alpha) = R(y\alpha)$, for all $\alpha \in \mathcal{A}$. In particular, for a right-invariant relation R it applies that

$$R(x) = R(y) \Longleftrightarrow (\forall z \in \mathcal{A}^*) : R(xz) = R(yz).$$

The identity relation as well as the coarsest-relation are trivial examples of right-invariant relations w.r.t. which X is embedable. Another example of such a relation but that takes into account the actual distribution of X is provided by the following definition.

DEFINITION 2.2. The *Markov relation induced by* X is the equivalence relation R^X over $\mathrm{supp}(X)$ defined as follows: xR^Xy if and only if

$$(\forall z \in \mathcal{A}^*) : P[X = xz... \mid X = x...] = P[X = yz... \mid X = y...].$$

We extend R^X to an equivalence relation to the whole of $\mathcal{A}^*$ by setting

$$R^X(x) := \mathcal{A}^* \setminus \mathrm{supp}(X), \quad x \notin \mathrm{supp}(X).$$

THEOREM 2.3. *R^X is a right-invariant relation. Furthermore, X is embedable w.r.t. any right-invariant equivalence relation that is a refinement of R^X; in particular, X is embedable w.r.t. R^X.*

3 Frequency statistics of patterns

There are two ways in which the results of the previous section fit nicely for analyzing the frequency statistics of a pattern $\mathcal{L}$ in the infinite sequence X. We may first induce in $\mathrm{supp}(X)$ the equivalence relation

$$R := \{\mathrm{supp}(X) \cap \mathcal{L}, \mathrm{supp}(X) \setminus \mathcal{L}\}.$$

The equivalence classes of the above relation correspond to the indicator function of $\mathcal{L}$ when restricted to the support of X. In particular, there is no warranty that the embedding X^R is Markovian at all. However, according to Theorem 2.2 and Corollary 2.1, $Y = (Y_n)_{n\geq 1}$ with $Y_n := R'(X_1 \cdots X_n)$ is a first-order homogeneous Markov chain with the least number of equivalence classes among all equivalence relations w.r.t. which X is embedable (see Figure 2). In particular, if we define

$$T := \{c \in R' : c \subset \mathrm{supp}(X) \cap \mathcal{L}\}$$

then

$$S_n^{\mathcal{L}} \stackrel{d}{=} \sum_{i=1}^{n} [\![Y_i \in T]\!], \tag{3.5}$$

i.e. the distribution of $S_n^{\mathcal{L}}$ corresponds to the number of visits that Y makes to the states in T in the first n steps. If Y is irreducible and positive recurrent and π denotes the unique stationary distribution of this chain then the strong law for additive functionals of Markov chains [9] implies that

$$\lim_{n\to\infty} \frac{S_n^{\mathcal{L}}}{n} = \sum_{c\in T} \pi(c), \tag{3.6}$$

almost surely, regardless of the initial distribution of Y. If in addition to the above conditions there exists c_0 such that

$$\sigma_0^2 := E\left(\left\{ \sum_{i=0}^{\tau_0 - 1} [\![Y_i \in T]\!] - \tau_0 \cdot \sum_{c\in T} \pi(c) \right\}^2 \middle| Y_0 = c_0 \right)$$

is strictly positive and finite, where

$$\tau_0 := \min\{n \geq 1 : Y_n = c_0\},$$

then the central limit theorem for additive functionals of Markov chains [9] implies that there exists $\sigma > 0$ such that

$$\lim_{n\to\infty} \frac{S_n^{\mathcal{L}} - n \cdot \sum_{c\in T} \pi(c)}{\sqrt{n}} \stackrel{d}{=} \sigma \cdot W, \tag{3.7}$$

where W is a standard Normal random variable [2]. We remark that the condition $E(\tau_0^2 \mid Y_0 = c_0) < +\infty$ is sufficient to have $\sigma_0^2 < +\infty$.

[2] τ_0 is usually called the *first-return time of Y to c_0*.

In most situations of interest R' will consist of a countable number of equivalence classes and the embedding $X^{R'}$ will not be analytically tractable. A slight improvement in this direction may be attained by letting the alphabet play a more direct role in the embedding. For this we induce in $\mathcal{A}^*$ the *Mihill-Nerode equivalence relation* defined as

$$x\, R_{\mathcal{L}}\, y \iff (\forall z \in \mathcal{L}) : [xz \in \mathcal{L} \Leftrightarrow yz \in \mathcal{L}].$$

The relation $R_{\mathcal{L}}$ is clearly right-invariant. (If $\mathcal{L}$ is a regular pattern then $R_{\mathcal{L}}$ has a finite number of equivalence classes due to the Mihill-Nerode Theorem [13].) In particular, if R is any right-invariant refinement of R^X then so is the product between R and $R_{\mathcal{L}}$ i.e. the equivalence relation $R_{\mathcal{L}}^X$ defined over $\mathcal{A}^*$ as follows

$$x\, R_{\mathcal{L}}^X\, y \iff [x\, R\, y \text{ and } x\, R_{\mathcal{L}}\, y].$$

Observe that if $c_1 \in R$ and $c_2 \in R_{\mathcal{L}}$ then $(c_1 \cap c_2)$ is an equivalence class of $R_{\mathcal{L}}^X$ provided that the intersection is not empty. Conversely, any equivalence class of $R_{\mathcal{L}}^X$ is of the form $(c_1 \cap c_2)$ for a unique $c_1 \in R$ and $c_2 \in R_{\mathcal{L}}$. Due to this the equivalence classes of $R_{\mathcal{L}}^X$ are in one-to-one correspondence with ordered-pairs of the form $(c_1, c_2) \in R \times R_{\mathcal{L}}$ such that $(c_1 \cap c_2) \neq \emptyset$. This property motivates the terminology of product relation.

Since $R_{\mathcal{L}}^X$ is a right-invariant refinement of R^X, it follows from Theorem 2.3 and Corollay 2.1 that the embedding $Y_n = R_{\mathcal{L}}^X(X_1 \cdots X_n)$ can be regarded as a first-order homogeneous Markov chain with state space $R \times R_{\mathcal{L}}$ (see Figure 3). Identity (3.5) will also apply for this embedding but with

$$T := \{(c_1, c_2) \in R \times R_{\mathcal{L}} : c_2 \subset \mathrm{supp}(X) \cap \mathcal{L}\},$$

and so will (3.6) and (3.7) provided that the necessary technical conditions are verified. To make explicit the probability transitions of this chain consider the transition function $f : R \times R_{\mathcal{L}} \times \mathcal{A} \to R \times R_{\mathcal{L}}$ defined as

$$f(c_1, c_2, \alpha) := (R(x\alpha), R_{\mathcal{L}}(y\alpha)),$$

provided that $x \in c_1$ and $y \in c_2$. (The above definition does not depend upon the selection of x or y because R and $R_{\mathcal{L}}$ are right-invariant.) The probability transitions of Y are then easily found to be

$$P[Y_{n+1} = (c_1', c_2') \mid Y_n = (c_1, c_2)] = \sum_{\alpha\in\mathcal{A}: f(c_1,c_2,\alpha) = (c_1', c_2')} P[X = x\alpha... \mid X = x...],$$

provided that $x \in c_1 \cap \mathrm{supp}(X)$. (The probabilities in the summation above do not depend on the selection of x because R is a refinement of R^X.)

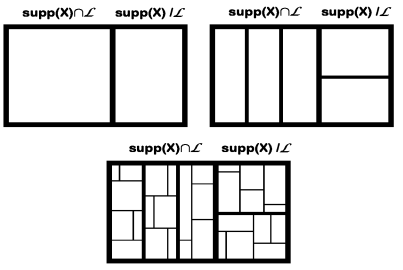


Figure 2: Top-left, schematic representation of partition R induced in $\mathrm{supp}(X)$ by a possibly non-regular pattern $\mathcal{L}$. Top-right, schematic representation of what could be the coarsest refinement R' of R w.r.t. which X is embedable. (R' may have a countable number of equivalence classes.) Bottom, schematic representation of a general refinement of R w.r.t. which X is embedable.

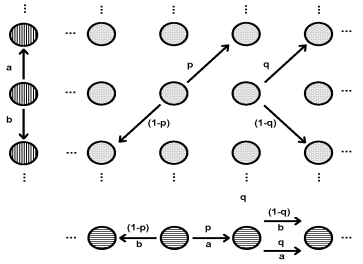


Figure 3: Most-left vertical axis, possible labeled transitions associated with one of the equivalence classes of $R_{\mathcal{L}}$, with $\mathcal{A} = \{a, b\}$. Each circle represents an equivalence class of the right-invariant relation $R_{\mathcal{L}}$. Bottom axis, possible probability and labeled transitions associated with two different states of X^R for a right-invariant refinement R of R^X. Here $0 \le p, q \le 1$ and each circle represents an equivalence class of R. Middle-grid, state space of the embedding of X through the product relation $R^X_{\mathcal{L}}$ between R and $R_{\mathcal{L}}$. The probability transitions associated with two of the states is displayed in accordance with the information available for R and $R_{\mathcal{L}}$.

3.1 A toy non-Markovian model. Let $0 < p < 1/2$ be a given parameter. Consider a sequence $X = (X_n)_{n\ge 1}$ of $\{0,1\}$-valued random variables such that

$$X_{n+1} \stackrel{d}{=} \begin{cases} \text{Bernoulli}(p) & , \ \frac{1}{n}\sum_{i=1}^{n} X_i > \frac{1}{2}; \\ \text{Bernoulli}(1/2) & , \ \frac{1}{n}\sum_{i=1}^{n} X_i = \frac{1}{2}; \\ \text{Bernoulli}(1-p) & , \ \frac{1}{n}\sum_{i=1}^{n} X_i < \frac{1}{2}. \end{cases} \tag{3.8}$$

The binary process X self-regulates the cumulative averages $\sum_{i=1}^{n} X_i/n$ so as to keep them tied to a 50% value. In particular, X evolves in a non-Markovian way. Consider the regular patterns

$$\begin{aligned} \mathcal{L}_1 &= \{0,1\}^*\{1\}, \\ \mathcal{L}_2 &= \{0\}^*\{1\}\{0\}^*(\{1\}\{0\}^*\{1\}\{0\}^*)^*. \end{aligned}$$

Both of these patterns are *primitive* i.e. each is recognized by a DFA whose associated digraph is irreducible and aperiodic. In particular, for binary Markovian sequences [19, 18] and more generally for sequences produced by nice probabilistic dynamical sources [6] the frequency statistics of $\mathcal{L}_1$ and $\mathcal{L}_2$ ought to be asymptotically Normal. However, as the following result shows, non-Gaussian asymptotic distributions may be also encountered for primitive patterns under more general non-Markovian models.

PROPOSITION 3.1. *If $0 < p < 1/2$ then*

$$\pi(n) := \begin{cases} \frac{1-2p}{2(1-p)} & , \ n = 0; \\ \frac{1-2p}{4p(1-p)} \left(\frac{p}{1-p}\right)^{|n|} & , \ n \neq 0; \end{cases} \tag{3.9}$$

satisfies that

$$\sum_{n=0 (\mathrm{mod}\ 2)} \pi(n) = \sum_{n=1 (\mathrm{mod}\ 2)} \pi(n) = \frac{1}{2}.$$

Furthermore, the following applies.

(a) If U and V are $\mathbb{Z}$-valued random variables such that $P[U = n] = 2\pi(n)$, for $n = 0 (\mathrm{mod}\ 2)$, and $P[V = n] = 2\pi(n)$, for $n = 1 (\mathrm{mod}\ 2)$, then

$$\lim_{\substack{n\to\infty \\ n=0(\mathrm{mod}\ 2)}} 2n \cdot \left\{ \frac{S_n^{\mathcal{L}_1}}{n} - \frac{1}{2} \right\} \stackrel{d}{=} U;$$

$$\lim_{\substack{n\to\infty \\ n=1(\mathrm{mod}\ 2)}} 2n \cdot \left\{ \frac{S_n^{\mathcal{L}_1}}{n} - \frac{1}{2} \right\} \stackrel{d}{=} V.$$

(b) If W is a standard Normal random variable then there exists $\sigma > 0$ such that

$$\lim_{n\to\infty} \sqrt{n} \cdot \left\{ \frac{S_n^{\mathcal{L}_2}}{n} - \frac{1}{2} \right\} \stackrel{d}{=} \sigma \cdot W.$$

Observe that $\mathrm{supp}(X) = \{0,1\}^*$. To show the proposition consider the transformation $R : \{0,1\}^* \to \mathbb{Z}$ defined as

$$\begin{aligned} R(x) &:= 2\left\{ \sum_{i=1}^{|x|} x_i - \frac{|x|}{2} \right\}; \\ &= \sum_{i=1}^{|x|} x_i - \sum_{i=1}^{|x|} (1 - x_i). \end{aligned} \tag{3.10}$$

Above it is understood that $R(\epsilon) = 0$. In particular, $R(xy) = R(x) + R(y)$, for all $x, y \in \{0,1\}^*$, and hence the level curves of R define a right-invariant equivalence relation on $\{0,1\}^*$. On the other hand, for $x, z \in \{0,1\}^*$ with $|z| > 0$ it applies from (3.8) that

$$P[X = xz... \mid X = x...] = p^{M(x,z)} \cdot (1-p)^{N(x,z)} \cdot \left(\frac{1}{2}\right)^{|z|-M(x,z)-N(x,z)},$$

where

$$\begin{aligned} M(x,z) &:= \sum_{i=1}^{|z|} [\![z_i = 0]\!] \cdot [\![R(z_1 \cdots z_{i-1}) < -R(x)]\!] \\ &\quad + \sum_{i=1}^{|z|} [\![z_i = 1]\!] \cdot [\![R(z_1 \cdots z_{i-1}) > -R(x)]\!]; \\ N(x,z) &:= \sum_{i=1}^{|z|} [\![z_i = 0]\!] \cdot [\![R(z_1 \cdots z_{i-1}) > -R(x)]\!] \\ &\quad + \sum_{i=1}^{|z|} [\![z_i = 1]\!] \cdot [\![R(z_1 \cdots z_{i-1}) < -R(x)]\!]; \end{aligned}$$

where it is understood that $z_1 \cdots z_0 = \epsilon$. From the above identity it is immediate that if $R(x) = R(y)$ then $R^X(x) = R^X(y)$. In particular, since R is right-invariant, Theorem 2.3 and Corollary 2.1 imply that X^R is a first-order homogeneous Markov chain with state space $\mathbb{Z}$. To obtain the probability transitions of X^R notice that

$$X^R_{n+1} = X^R_n + R(X_{n+1}).$$

Since X^R_n and $(\sum_{i=1}^n X_i - n/2)$ have the same sign, (3.8) implies that

$$X^R_{n+1} - X^R_n = \begin{cases} (+1) & \text{w.p. } p \text{ if } X^R_n > 0; \\ (-1) & \text{w.p. } (1-p) \text{ if } X^R_n > 0; \\ (+1) & \text{w.p. } 1/2 \text{ if } X^R_n = 0; \\ (-1) & \text{w.p. } 1/2 \text{ if } X^R_n = 0; \\ (+1) & \text{w.p. } (1-p) \text{ if } X^R_n < 0; \\ (-1) & \text{w.p. } p \text{ if } X^R_n < 0. \end{cases}$$

The process X^R is recurrent and has period 2. It is positive recurrent because $p \cdot 1 + (1-p) \cdot (-1) < 0$ and $(1-p) \cdot 1 + p \cdot (-1) > 0$. In particular, X^R has a unique stationary distribution π supported over $\mathbb{Z}$. Indeed, since X^R is a birth-death chain we may use the detailed balance condition [9] to obtain the explicit formula in (3.9). Since

$$S^{\mathcal{L}_1}_n = \sum_{i=1}^n X_i,$$

part (a) in the proposition is direct from (3.9) and (3.10), and the well-known result on convergence in distribution of a Markov chain to its stationary distribution for the periodic case [9].

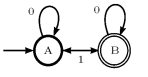


Figure 4: Deterministic finite automaton with initial state A and terminal state B that recognizes the binary regular language $\{0\}^*\{1\}\{0\}^*(\{1\}\{0\}^*\{1\}\{0\}^*)^*$.

To show part (b) consider the DFA displayed in Figure 4. This automaton is the minimal automaton that recognizes $\mathcal{L}_2$. In particular, the equivalence classes of $R_{\mathcal{L}_2}$ are in one-to-one correspondence with the states of this automaton [13]. Furthermore, since $R^X_{\mathcal{L}_2}$ is a right-invariant refinement of R^X, $Y_n = R^X_{\mathcal{L}_2}(X_1 \cdots X_n)$ is a first-order homogeneous Markov chain. Without loss of generality we may assume that $Y = (Y_n)_{n \geq 1}$ has $\mathbb{Z} \times \{A, B\}$ as its state space. Y is irreducible and has period 4. It is also positive recurrent because if we define

$$\pi(n, A) := \pi(n, B) := \frac{\pi(n)}{2} \tag{3.11}$$

then

$$\begin{aligned} &\sum_{(m,i) \in \mathbb{Z} \times \{A,B\}} \pi(m,i) \cdot P[Y_2 = (n, A) \mid Y_1 = (m,i)] \\ &= \sum_{(m,i) = (n-1,B),(n+1,A)} \pi(m,i) \cdot P[Y_2 = (n,A) \mid Y_1 = (m,i)], \\ &= \frac{1}{2} \sum_{m = n-1, n+1} \pi(m) \cdot P[X^R_2 = n \mid X^R_1 = m], \\ &= \frac{1}{2} \sum_{m \in \mathbb{Z}} \pi(m) \cdot P[X^R_2 = n \mid X^R_1 = m], \\ &= \frac{\pi(n)}{2}, \\ &= \pi(n, A). \end{aligned}$$

Similarly, we obtain that

$$\sum_{(m,i) \in \mathbb{Z} \times \{A,B\}} \pi(m,i) \cdot P[Y_2 = (n,B) \mid Y_1 = (m,i)] = \pi(n,B),$$

which shows that (3.11) is the stationary distribution of Y. In particular,

$$\lim_{n \to \infty} \frac{S^{\mathcal{L}_2}_n}{n} = \sum_{n \in \mathbb{Z}} \pi(n,B) = \frac{1}{2}, \tag{3.12}$$

almost surely. On the other hand, observe that if $Y_0 = (0, A)$ then $Y_n = (0, A)$ if and only if $X_n^R = 0$ and n is divisible by four. Let τ denote the first-return time of Y to $(0, A)$ conditioned on having $Y_0 = (0, A)$. Using the Chomsky and Schützenberger method [7], the moment generating function of τ may be determined explicitly to find that its radius of convergence is given by $1/\sqrt{4p(1-p)}$. Since this last quantity is strictly greater than one for $0 < p < 1/2$, the tail distribution of τ decays exponentially fast and therefore τ must have a finite second-moment. Part (b) in the proposition is now a direct consequence of (3.7) and (3.12).

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[2] F. Bassino, J. Clément, J. Fayolle, and P. Nicodème. Counting occurrences for a finite set of words: an inclusion-exclusion approach. 2007. Proceedings of the 2007 Conference on Analysis of Algorithms.

[3] E. A. Bender and F. Kochman. The distribution of subword counts is usually normal. *Eur. J. Comb.*, 14(4):265–275, 1993.

[4] J. D. Biggins and C. Cannings. Markov renewal processes, counters and repeated sequences in Markov chains. *Adv. Appl. Prob.*, 19:521–545, 1987.

[5] J. Bourdon and B. Vallée. Generalized pattern matching statistics. In *Colloquium on Mathematics and Computer Science : Algorithms and Trees*, Trends in Mathematics, pages 249–265. Birkhauser, 2002.

[6] J. Bourdon and B. Vallée. Pattern matching statistics on correlated sources. In *Proc. of the 7th Latin American Symposium on Theoretical Informatics (LATIN'06)*, pages 224–237, Valdivia, Chile, 2006.

[7] N. Chomsky and M. P. Schützenberger. The algebraic theory of context-free languages. *Computer Programming and Formal Languages*, pages 118–161, 1963.

[8] S. Derisavi, H. Hermanns, and W. H. Sanders. Optimal state-space lumping in markov chains. *Inf. Process. Lett.*, 87(6):309–315, 2003.

[9] R. Durrett. *Probability: theory and examples.* Duxbury Press, third edition, 2004.

[10] P. Flajolet, W. Szpankowski, and B. Vallée. Hidden word statistics. *J. ACM*, 53(1):147–183, 2006.

[11] H. U. Gerber and S.-Y. R. Li. The occurrence of sequence patterns in repeated experiments and hitting times in a Markov chain. *Stochastic Processes and their Applications*, 11(1):101–108, 1981.

[12] L. J. Guibas and A. M. Odlyzko. Periods in strings. *J. Comb. Theory, Ser. A*, 30(1):19–42, 1981.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[14] J. Kemeny and J. L. Snell. *Finite Markov Chains.* Van Nostrand Reihnhold LTD., 1960.

[15] S.-Y. R. Li. A martingale approach to the study of occurrence of sequence patterns in repeated experiments. *The Annals of Probability*, 8(6):1171–1176, 1980.

[16] M. Lladser. Minimal Markov chain embeddings of pattern problems. 2006. Proceedings of the 2007 Information Theory and Applications Workshop, University of California, San Diego.

[17] M. Lladser, M. D. Betterton, and R. Knight. Multiple pattern matching: A Markov chain approach. *Journal of Mathematical Biology*, 56:51–92, 2008.

[18] P. Nicodème. Regexpcount, a symbolic package for counting problems on regular expressions and words. *Fundamenta Informaticae*, 56(1-2):71–88, 2003.

[19] P. Nicodème, B. Salvy, and P. Flajolet. Motif statistics. *Theoretical Computer Science*, 287(2):593–617, 2002.

[20] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 1987.

[21] V.I. Pozdnyakov and M. Kulldorff. Waiting times for patterns and a method of gambling teams. *American Mathematical Monthly*, 113(2):134–143, 2006.

[22] M. Régnier and A. Denise. Rare events and conditional events on random strings. *DMTCS*, 6(2):191–214, 2004.

[23] M. Régnier and W. Szpankowski. On pattern frequency occurrences in a Markovian sequence. *Algorithmica*, 22(4):631–649, 1998.

Nearly Tight Bounds on the Encoding Length of the Burrows-Wheeler Transform

Ankur Gupta* Roberto Grossi† Jeffrey Scott Vitter‡

Abstract

In this paper, we present a nearly tight analysis of the encoding length of the Burrows-Wheeler Transform (BWT) that is motivated by the text indexing setting. For a text T of n symbols drawn from an alphabet Σ, our encoding scheme achieves bounds in terms of the hth-order empirical entropy H_h of the text, and takes linear time for encoding and decoding. We also describe a lower bound on the encoding length of the BWT that constructs an infinite (non-trivial) class of texts that are among the hardest to compress using the BWT. We then show that our upper bound encoding length is *nearly tight* with this lower bound for the class of texts we described.

In designing our BWT encoding and its lower bound, we also address the *t-subset problem*; here, the goal is to store a subset of t items drawn from a universe $[1..n]$ using just $\lg \binom{n}{t} + O(1)$ bits of space. A number of solutions to this basic problem are known, however encoding or decoding usually requires either $O(t)$ operations on large integers [Knu05, Rus05] or $O(n)$ operations. We provide a novel approach to reduce the encoding/decoding time to just $O(t)$ operations on *small* integers (of size $O(\lg n)$ bits), without increasing the space required.

1 Introduction

The Burrows-Wheeler transform [BW94] (BWT) has had a profound impact on a myriad of algorithmic fields in Computer Science. The BWT preprocesses an input text T by a reversible transformation—the result is then easily compressible by other simpler encoding methods. The BWT highlights (among other things) that a series of encoders may be more effective than a one-pass compression algorithm. The BWT has shown remarkable evidence to that effect, especially in practice: it is at the heart of the `bzip2` algorithm, which has become a mainstream data compression tool.

The BWT has also revolutionized the field of text indexing [NM06]. Using the BWT, one can build a "compressed text index", a class of data structures that support powerful substring searches and occupy roughly the same space as that required by the best compressors. These results [GV05, FM05] have dispelled the notion that an efficient full-text index must use space superlinear in the text length. Clearly, the BWT is a powerful compression tool that manages to organize data in a searchable way—but where is the magic?

The BWT has been analyzed extensively since its original introduction in 1994, especially in the information-theory community [WMF94, EVKV02]. These results apply to a wide range of statistical models for generating a text T, including high-order Markov sources, tree sources, and finite-state machine (FSM) sources. To the best of our knowledge, the best known encoding of the BWT appears in [BB04], achieving a nearly optimal BWT encoding length that encodes in $O(n)$ time, but decoding takes longer.

In a text indexing setting, recent theoretical results [Man01, FGMS05, KLV06] have shed light on the success of the BWT and present some limits on its compressibility. For a text T of n symbols drawn from an alphabet Σ ($|\Sigma| = \sigma$), these results achieve bounds in terms of the modified hth-order empirical entropy H_h^* of the text. However, little attention is paid to reducing the cost for encoding the hth order count statistics, since $\sigma^h = o(n/\log \sigma)$ is a reasonable assumption. Even in such cases, the costs for encoding the *empirical statistical model* for T may dwarf the leading entropy term. (More precise comparisons follow in Section 3.) In this paper, we deeply consider the problem of storing the empirical statistical model, without any assumption on the size of σ and h. This investigation may lead to a clear

*Department of Computer Science and Software Engineering, Butler University, Indianapolis, IN 46208 (`agupta@butler.edu`). Some work was originally done at Purdue University. Support was provided in part by the Army Research Office (ARO) through grant DAAD20–03–1–0321 and by the National Science Foundation through research grant ISS–0415097.

†Dipartimento di Informatica, Università di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa (`grossi@di.unipi.it`). Support was provided in part by Italian Ministry of Education, University and Research (MIUR).

‡Department of Computer Science, Purdue University, West Lafayette, IN 47097–2066 (`agupta@cs.purdue.edu`, `jsv@purdue.edu`). Support was provided in part by the Army Research Office (ARO) through grant DAAD20–03–1–0321 and by the National Science Foundation through research grant ISS–0415097.

understanding of the best way to encode the BWT.

We present a nearly tight analysis on the encoding length of the BWT. Our encoding scheme can be encoded or decoded in linear time, is strongly motivated by current text indexing techniques, and we present our results to validate the current direction of text indexes. In particular, we present an encoding scheme for the BWT that can be encoded or decoded in $O(n)$ time and requires $nH_h + \min\{g'_h \lg(n/g'_h + 1), H^*_h n + \lg n + g''_h\}$ bits of space, where $g'_h = O(|\Sigma|^{h+1})$ and $g''_h = O(|\Sigma|^{h+1} \lg |\Sigma|^{h+1})$ do not depend on the text length n, while $H^*_h \geq H_h$ is the modified hth-order empirical entropy of T. Notice that we have bounded the encoding of the empirical model cost (everything but nH_h in the above bound) by the *entropy of the underlying text* T. In other words, encoding the empirical statistics does not seem to be a trivial cost!

To show that our encoding is nearly tight with the lower bound encoding length of the BWT, we describe a class of texts, called *δ-resilient texts*, for which our encoding achieves nearly tight bounds. One could view the class of δ-resilient texts as one of the hardest to compress using the BWT. A consequence of this result is the observation that one can, without fear of incurring significant overhead, separate the encoding of the underlying data from the representation of the empirical statistical model. In this paper, we explain how this separation naturally occurs in the text indexing setting.

We also present a novel subset encoding scheme that addresses some of the weaknesses of well-known results from combinatorial enumeration [Knu05, Rus05]. Our approach applies a quasi-arithmetic coder to a set of empirical probability estimates generated by a random sampling technique [Vit84] to avoid operations on large integers. In particular, our subset encoding for a set of t items out of a universe of size n can be decoded using $O(t)$ operations on small integers (of size $O(\lg n)$ bits).

2 Preliminaries

We formulate our analysis of the space complexity in terms of the high-order empirical entropy of a text T of n symbols drawn from alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. In this section, we discuss entropy from both an empirical probability model and a finite set model. We also review the BWT and its connection to suffix arrays.

2.1 Empirical Probabilistic Entropy We provide terminology for the analysis and explore empirical probability models. For each symbol $y \in \Sigma$, let n^y be the number of its occurrences in text T. With symbol y, we associate its empirical probability, $\text{Prob}[y] = n^y/n$, of occurring in T. (By definition, $n = \sum_{y\in\Sigma} n^y$, so the empirical probability is well defined.) Following Shannon's definition of entropy [Sha48], the *0th-order empirical entropy* is $H_0 = H_0(T) = \sum_{y\in\Sigma} -\text{Prob}[y] \times \lg \text{Prob}[y]$. Since $nH_0 \leq n \lg \sigma$, the previous expression simply states that an efficient variable-length coding of text T would encode each symbol y based upon its frequency in T rather than simply $\lg \sigma$ bits. The number of bits assigned for encoding an occurrence of y would be $-\lg \text{Prob}[y] = \lg(n/n^y)$.

We can generalize the definition to higher-order empirical entropy, so as to capture the dependence of symbols upon their context, made up of the h previous symbols in the text.[1] For a given h, we consider all possible h-symbol sequences x that appear in the text. (They are a subset of Σ^h, the set of all possible h-symbol sequences over the alphabet Σ. The last $h-1$ incomplete sequences are easily stored within the bounds we give in this paper; we defer these technical details until the full paper.) We denote the number of occurrences in the text of a particular context x by n^x, with $n = \sum_{x\in\Sigma^h} n^x$ as before, and we let $n^{x,y}$ denote the number of occurrences in the text of the concatenated sequence yx (meaning that y precedes x).

Manzini [Man01] gives a definition of the *hth-order empirical entropy* H_h in terms of H_0. For any given context x, let w_x be the concatenation of the symbols y that appear in the text immediately before context x. We denote its length by $|w_x|$ and its 0th-order empirical entropy by $H_0(w_x)$, thus defining H_h as

$$H_h = \frac{1}{n} \sum_{x\in\Sigma^h} |w_x| H_0(w_x). \tag{2.1}$$

An important observation to note is that $H_{h+1} \leq H_h \leq \lg \sigma$ for any integer $h \geq 0$. Hence, the previous expression states that a better variable-length coding of text T would encode each symbol y based upon the joint and conditional empirical frequency for any context x of y.

One potential difficulty with the definition of H_h is that the inner terms of the summation could equal 0 (or an arbitrarily small constant), which can be misleading when considering the encoding length of a text T. (One relatively trivial case is when the text contains n equal symbols, as no symbol needs to be "predicted".) Manzini introduced *modified high-order empirical entropy* H^*_h to address this point and capture the constraint that the encoding of the text must contain at least $\lg n$ bits for coding its length n. Using a modified $H^*_0 = H^*_0(T) = \max\{H_0, (1 +$

[1] The standard definition of conditional probability for text documents considers the symbol y immediately *after* the sequence x. It makes no difference, since we could use this definition on the reversed text as discussed in [FGMS05].

$\lfloor \lg n \rfloor)/n\}$ to make the change, Manzini defines the H_h^* as

$$H_h^* = H_h^*(T) = \frac{1}{n} \min_{P_h} \sum_{x \in P_h} |w_x| H_0^*(w_x). \tag{2.2}$$

Here, we let P_h be a *prefix cover*, namely, a set of substrings having length at most h such that every string from Σ^h has a unique prefix in P_h.

Immediate consequences of this encoding-motivated entropy measure are that $H_h^* \geq H_h$ and $nH_h^* \geq \lg n$, but nH_h can be a small constant. Let the *optimal prefix cover* P_h^* be the prefix cover that minimizes H_h^* in (2.2). Thus, (2.2) can be equivalently stated as $H_h^* = \frac{1}{n} \sum_{x \in P_h^*} |w_x| H_0^*(w_x)$.[2] The empirical probabilities that are employed in the definition of the high-order empirical entropy can be obtained from the number of occurrences $n^{x,y}$, where $\sum_{x \in P_h^*, y \in \Sigma} n^{x,y} = n$. Indeed, $n^y = \sum_{x \in P_h^*} n^{x,y}$ and $n^x = \sum_{y \in \Sigma} n^{x,y}$. This discussion motivates the following definition, which will guide us through our high-order entropy analysis.

Definition 1. The *empirical statistical model* for a text T is composed of two parts stored using $M(T, \Sigma, h)$ bits:

i. The partition of Σ^h induced by the contexts of the prefix cover P_h^*.
ii. The sequence of non-negative integers, $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$, where $x \in P_n^*$. (Recall that $n^{x,y}$ is the number of occurrences of yx as a substring of T.)

We denote the number of bits used to store the information in parts (i)–(ii) by $M(T, \Sigma, h)$, as n increases.

2.2 Finite Set Entropy We provide a new definition of high-order empirical entropy H_h', based on the finite set model rather than on conditional probabilities, to avoid dealing with empirical probabilities explicitly. We show that our new definition is $H_h - O(|P_h^*| \lg n) \leq H_h' \leq H_h \leq H_h^*$, so that we can provide bounds in terms of H_h'.

For ease of exposition, we "number" the lexicographically ordered contexts x as $1 \leq x \leq \sigma^h$. Let the *multinomial* coefficient $\binom{n}{m_1, m_2, \ldots, m_p} = \frac{n!}{m_1!\, m_2! \cdots m_p!}$ represent the number of partitions of n items into p subsets of size $m_1, m_2, \ldots, m_p$. In this paper, we define $0! = 1$. (Note that $n = m_1 + m_2 + \cdots + m_p$.) When $m_1 = t$ and $m_2 = n - t$, we get precisely the binomial coefficient $\binom{n}{t}$. We define

$$H_0' = H_0'(T) = \frac{1}{n} \lg \binom{n}{n^1, n^2, \ldots, n^\sigma}, \tag{2.3}$$

which counts the number of possible partitions of n items into σ unique buckets, i.e. the alphabet size. We use the optimal prefix cover P_h^* in (2.2) to define *alternative high-order empirical entropy*[3]

$$H_h' = H_h'(T) = \frac{1}{n} \sum_{x \in P_h^*} \lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}}. \tag{2.4}$$

Theorem 2.1. *For any text T and context length $h \geq 0$, we have $H_h' \leq H_h$.*

Proof. It suffices to show that $nH_0' \leq nH_0$ for all alphabets Σ, since then $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} \leq |w_x| H_0(w_x)$.

The bound $nH_0' \leq nH_0$ trivially holds when $\sigma = 1$. We first prove this bound for an alphabet Σ of $\sigma = 2$ symbols. Let t and $n - t$ denote the number of occurrences of the two symbols in T. We want to show that $nH_0' = \lg \binom{n}{t} \leq nH_0 = t \lg(n/t) + (n-t) \lg(n/(n-t))$ by (2.3). The claim is true by inspection when $n \leq 4$ or $t = 0, 1, n-1$. Let $n > 4$ and $2 \leq t \leq n-2$. We apply Stirling's double inequality [Fel68] to obtain

$$\frac{n^n \sqrt{2\pi n}}{e^{n - \frac{1}{12n+1}}} < n! < \frac{n^n \sqrt{2\pi n}}{e^{n - \frac{1}{12n}}}. \tag{2.5}$$

Taking logarithms and applying the inequality to $\lg \binom{n}{t} = \lg(n!) - \lg(t!) - \lg((n-t)!)$, we have

$$nH_0' = \lg \binom{n}{t} < nH_0 - \frac{1}{2} \lg \frac{t(n-t)}{n}$$
$$- \lg e \left[\frac{1}{12t+1} + \frac{1}{12(n-t)+1} - \frac{1}{12n} \right] - \lg \sqrt{2\pi}.$$

Since $t(n-t) \geq n$ and $1/(12t+1) + 1/(12(n-t)+1) \geq 1/(12n)$ by our assumptions on n and t, it follows that $nH_0' \leq nH_0$, proving the result when $\sigma = 2$.

Next, we show the claimed bound for the general alphabet ($\sigma \geq 2$ and $h = 0$) and by using induction on the alphabet size (with the base case $\sigma = 2$ as detailed before). We write

$$\lg \binom{n}{n^1, n^2, \ldots, n^\sigma} = \lg \left[\binom{n - n^\sigma}{n^1, n^2, \ldots, n^{\sigma-1}} \times \binom{n}{n^\sigma} \right]. \tag{2.6}$$

We use induction for the right-hand side of (2.6) to get

$$\lg \binom{n - n^\sigma}{n^1, n^2, \ldots, n^{\sigma-1}} \leq \sum_{y=1}^{\sigma-1} n^y \lg \frac{n - n^\sigma}{n^y}, \tag{2.7}$$

$$\lg \binom{n}{n^\sigma} \leq n^\sigma \lg \frac{n}{n^\sigma} + (n - n^\sigma) \lg \frac{n}{n - n^\sigma}. \tag{2.8}$$

[2] A minor technical note: h now refers to the length of the longest substring in P_h^*, since no larger value of h can yield a more succinct entropy measure.

[3] Actually, it can be defined for any prefix cover P_h, including $P_h = \Sigma^h$.

Original	*Sorted*			*Mappings*		
Q	*F*		*L*	*i*	*LF*(*i*)	Φ(*i*)
mississippi#	i	ppi#missis	s	1	8	7
#mississippi	i	ssippi#mis	s	2	9	10
i#mississipp	i	ssissippi#	m	3	5	11
pi#mississip	i	#mississip	p	4	6	12
ppi#mississi	m	ississippi	#	5	12	3
ippi#mississ	p	i#mississi	p	6	7	4
sippi#missis	p	pi#mississ	i	7	1	6
ssippi#missi	s	ippi#missi	s	8	10	1
issippi#miss	s	issippi#mi	s	9	11	2
sissippi#mis	s	sippi#miss	i	10	2	8
ssissippi#mi	s	sissippi#m	i	11	3	9
ississippi#m	#	mississipp	i	12	4	5

Table 1: Matrix Q for the BWT containing the cyclic shifts of text $T =$ `mississippi#` (column '*Original*'). Sorting of the rows of Q, in which the first (F) and last (L) symbols in each row are separated (column '*Sorted*'). Functions LF and Φ for each row of the sorted Q (column '*Mappings*').

Summing (2.7) and (2.8), we obtain $\sum_{y=1}^{\sigma} n^y \lg \frac{n}{n^y} = nH_0$, thus proving the claim for any alphabet size σ.

COROLLARY 2.1. *For any given text T and context length $h \geq 0$, we have $H_h - O((1/n)|P_h^*| \lg n) \leq H'_h$.*

The above discussion justifies our use of H'_h in later analysis, but we continue to state bounds in terms of H_h since it represents more standard notation. The key point is that we can derive equations in terms of multinomial coefficients without worrying about the empirical probability of symbols appearing in text T.

2.3 The BWT and Suffix Arrays We now give a short description of the BWT in order to explain its salient features. Consider the text $T =$ `mississippi#` in the example shown in Table 1, where `i` < `m` < `p` < `s` < `#` and `#` is an end-of-text symbol. The BWT forms a conceptual matrix Q whose rows are the cyclic (forward) shifts of the text in sorted order and stores the last column $L =$ `ssmp#pissiii` written as a contiguous string. Note that L is an invertible permutation of the symbols in T. In particular, $LF(i) = j$ in Table 1 indicates for any symbol $L[i]$, the corresponding position j in F where $L[i]$ appears. For instance, $LF(8) = 10$ since $L[8] =$ `s` occurs in position 10 of F (as the third `s` among the four appearing consecutively in F).

Using L and LF, we can recreate the text T in reverse order by starting at the last position n (corresponding to `#mississippi`), writing its value from F, and following the LF function to the next value of F. Continuing the example from before, we follow the pointers from $LF(n)$: $LF(12) = 4$, $F[4] =$ `i`; $LF(4) = 6$, $F[6] =$ `p`; $LF(6) = 7$, $F[7] =$ `p`; and so on. In other words, the LF function gives the position in F of the preceding symbol from the original text T. Thus one could store L and recreate T, since we can obtain F by sorting L and the LF function can be derived by inspection. Note that L is compressible using 0th-order compressors, boosting them to attain high-order entropy [FGMS05]. Now, we connect the BWT with L.

We introduce the neighbor function Φ, which is used to represent the compressed suffix array (CSA) in [GV05]. The Φ function can be thought of as the FL mapping, and can represent the BWT much like the LF mapping. The neighbor function Φ is the inverse of the LF mapping, in other words, $\Phi = LF^{-1}$.

We partition the Φ (LF) mapping into Σ-*lists*. Given a symbol $y \in \Sigma$, the list y is the set of positions in L such that for any position p in list y, $L[p] = y$. The fundamental property of these Σ lists is that each list is an *increasing* series of positions. The concatenation of the lists y for $y = 1, 2, \ldots, \sigma$ yields the Φ function (as shown in Table 1). Thus, the value of $\Phi(i)$ is just the ith nonempty entry in the concatenation of the lists, and belongs to some list y.

3 A Nearly Space-Optimal Burrows-Wheeler Transform

We now show our major upper bound result; we describe a nearly optimal analysis of the compressibility of the Burrows-Wheeler transform with respect to high-order empirical entropy, exploiting the properties of the BWT illustrated in Section 2.3.

Let P_h^* be the optimal prefix cover as defined in Section 2, and let $n^{x,y}$ be the corresponding values in equation (2.4), where $x \in P_h^*$ and $y \in \Sigma$. (See also Definition 1.) We denote by $|P_h^*| \leq \sigma^h$ the number of contexts in P_h^*. We describe our main upper bound result in the following theorem.

THEOREM 3.1. (NEARLY SPACE-OPTIMAL BWT) *The Burrows-Wheeler transform for a text T of n symbols drawn from an alphabet Σ can be compressed using $nH_h + M(T, \Sigma, h)$ bits for the best choice of context length h and prefix cover P_h^*, where the number of bits required for encoding the empirical statistical model for P_h^* is $M(T, \Sigma, h) \leq \min\{g'_h \lg(1 + n/g'_h), H_h^* n + \lg n + g''_h\}$, where $g'_h = O(\sigma^{h+1})$ and $g''_h = O(\sigma^{h+1} \lg \sigma^{h+1})$ do not depend on the text length n.*

Using Theorem 3.1, we can compare our analysis with the best bounds from previous work. When compared to the additive term of $O(n \lg \lg n / \lg n)$ in the analysis of the Lempel-Ziv method in [KM99], we obtain an $O(\log n)$ additive term for $\sigma = O(1)$ and $h = O(1)$, giving strong evidence why the

BWT is better than the Lempel-Ziv method. Since $M(T,\Sigma,h) \leq g'_h \lg(n/g'_h + 1)$, our bound becomes $nH_h + O(\lg n)$ when $h = O(1)$ and $\sigma = O(1)$, thus exponentially reducing the additive term of n of the H_h-based analysis in [FGMS05].

In this case, our bound closes the gap in the analysis of the BWT, since it matches the lower bound of $nH_h + \Omega(\lg\lg n)$ up to lower-order terms. The latter comes from the lower bound of $nH_0^* + \Omega(\lg\lg n)$ bits, holding for a large family of compressors (not necessarily related to BWT), as shown in [FGMS05]; the only (reasonable) requirement is that any such compressor must produce a codeword for the text length n when it is fed with an input text consisting of the same symbol repeated n times. Since $H_h \leq H_0^*$, we easily derive the lower bound of $nH_h + \Omega(\lg\lg n)$ bits, but a lower bound of $nH_h + \Omega(\lg n)$ probably exists since $nH_0^* \geq \lg n$ while nH_h can be zero.

For the modified hth-order empirical entropy, we show that Theorem 3.1 can be upper bounded by $n(H_h + H_h^*) + \lg n + g''_h$ bits. Since $H_h \leq H_h^*$, our bound is strictly smaller than $2.5nH_h^* + \lg n + g_h$ bits in [FGMS05], apart from the lower-order terms. Actually, our bound is definitively smaller in some cases. For example, while a bound of the form $nH_h^* + \lg n + g_h$ bits is not always possible [Man01], there are an infinite number of texts for which $nH_h = 0$ while $nH_h^* \neq 0$. In these cases, our bound is $nH_h^* + \lg n + g''_h$ bits.

We devote the rest of Sections 3–5 to the proof of Theorem 3.1. We describe our analysis for an *arbitrary* prefix cover P_h, so it also holds for the optimal prefix cover P_h^* as in equation (2.4). We make use of the basic idea from [GGV03] to partition each list y further into sublists $\langle x, y\rangle$ by contexts $x \in P_h$. Intuitively, sublist $\langle x, y\rangle$ stores the positions such that the symbols that follow $F[y]$ are x. For context length $h = 1$, if we continue the example in Table 1, we break the Σ lists by context (in lexicographical order `i`, `m`, `p`, `s`, and `#`, and numbered from 1 up to $|P_h|$). The list for $y =$ `i` is $\langle 7, 10, 11, 12\rangle$, and is broken into sublist $\langle 7\rangle$ for context $x =$ `p`, sublist $\langle 10, 11\rangle$ for context $x =$ `s`, and sublist $\langle 12\rangle$ for $x =$ `#`.

For any context $x \in P_h$, if we encode its sublists using nearly $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}}$ bits, we automatically achieve the hth-order empirical entropy as seen in equation (2.4). For example, context $x =$ `i` should be represented with nearly $\lg \binom{4}{1,1,2}$ bits, since two sublists contain one entry each and one sublist contains two entries. The empirical statistical model should record the partition induced by P_h, record which sublists are empty, and encode the lengths of all nonempty sublists using $M(T,\Sigma,h)$ bits.

Encoding: We run the boosting algorithm from [FGMS05] on the BWT to find the optimal value of context order h and the optimal prefix cover P_h^* using the cost of $nH'_h + M(T,\Sigma,h)$ according to equation (2.4). Once we know h and set $P_h = P_h^*$, we can cleanly separate the contexts and encode the Φ function. Thus, we follow the two steps below:

1. We encode the empirical statistical model given in Definition 1.
2. For each context $x \in P_h$, we separately encode the sublists $\langle x, y\rangle$ for $y \in \Sigma$ to capture high-order entropy. Each of these sublists is a subset of the integers in the range $[1, n^x]$. These sublists form a *partition* of the integers in the interval $[1, n^x]$.

The storage for step 1 is $M(T,\Sigma,h)$, the number of bits required for encoding the model. (See Definition 1.) The storage required for step 2 should use nearly $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}}$ bits per context x, and should not exceed a total of nH_h bits plus lower-order terms, once we determine P_h^*, as stated in Theorem 3.1.

Decoding: We retrieve the empirical statistical model encoded in step 1 above, which tells us which sublists are nonempty and their lengths. (Note that the values of n and n^x can be obtained from these lengths.) Next, we retrieve the sublists encoded in step 2 since we know their lengths. At this point, we have recovered the Φ function and we can decode the BWT.

4 Encoding Sublists in High-Order Entropy

In this section, we discuss how to encode the sublists in step 2. This is the first part of the proof of Theorem 3.1.

One simple method for encoding the sublists would be to simply encode each sublist $\langle x, y\rangle$ as a subset of $t = n^{x,y}$ items out of a universe of $n' = n^x$ items. We can use *t-subset encoding*, requiring the information-theoretic minimum of $\lceil \lg \binom{n'}{t} \rceil$ bits, with $O(t)$ operations on large integers, according to [Knu05, Rus05]. All the t-subsets are enumerated in some canonical order (e.g. lexicographic order) and the subset occupying rank r in this order is encoded by the value of r itself, which requires $\lceil \lg \binom{n'}{t} \rceil$ bits. In this way, the t-subset encoding can also be seen as the compressed representation of an *implicit bitvector* of length n': If the subset contains $1 \leq s_1 < s_2 < \cdots < s_t \leq n'$, the s_ith entry in the bitvector is **1**, for $1 \leq i \leq t$; the remaining $n' - t$ bits are **0**s. Thus, we can use subset rank (and unrank) primitives for encoding (and decoding) sublist $\langle x, y\rangle$ as a sequence r of $\lceil \lg \binom{n^x}{n^{x,y}} \rceil$ bits.

Instead, we will encode sublists one context at a time. In other words, we encode the sublists $\langle x, 1\rangle, \langle x, 2\rangle, \ldots, \langle x, \sigma\rangle$ at once. We encode each context x by encoding the string w_x (from Section 2.1), which consists of the symbols y that precede x, concatenated together in BWT order. One

simple method to encode w_x that we can use is a quasi-arithmetic coder from [HV94] (Theorem 1), requiring $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} + 2$ bits of space.

LEMMA 4.1. (QUASI-ARITHMETIC CODER [HV94]) *Suppose we know the values of n^x and $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$ for each context x. We can encode all contexts using one quasi-arithmetic coder for each context x taking just $nH_h + O(\sigma^h)$ bits of space. Decoding any context requires $O(n^x)$ operations on integers of size $O(\sigma)$.*

4.1 The Wavelet Tree In this section, we detail an alternate method of encoding each context x, motivated by applications to text indexing. We review the *wavelet tree* data structure, which is a binary tree structure that reduces the compression of a string from alphabet Σ to the compression of σ binary strings. We direct the reader to [GGV03] for more details and examples.

We will store one wavelet tree data structure for each context x, built on the string w_x, as used earlier. Recall that w_x is a string with alphabet Σ of length n^x. Each leaf represents one of the t^x nonempty sublists (equivalently, one of the t^x distinct symbols appearing in w_x) from context x. We implicitly consider each left branch to be associated with a **0** and each right branch to be associated with a **1**. Each internal node u is a t-subset data structure with the elements in its left subtree stored as **0**, and the elements in its right subtree stored as **1**. For instance, the root node conceptually stores a single bitvector B of length n^x represented by a t-subset data structure. If $B[p] = \mathbf{0}$, then the pth symbol in w_x is one of $1, \ldots, \sigma/2$. $B[p] = \mathbf{1}$ otherwise.

The key observation is to note that each of the $t^x - 1$ internal nodes represents elements relative to its subtrees. To decode any particular sublist in a wavelet tree, a query would only need to access $O(\lg t^x)$ internal nodes in a balanced wavelet tree. In particular, to recover the entries of any sublist $\langle x, y \rangle$, we start from the leaf corresponding to sublist $\langle x, y \rangle$ and examine the t-subsets in its ancestors.

LEMMA 4.2. (WAVELET TREE COMPRESSION) *Suppose we know the values of n^x and $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$ for each context x. We can encode all contexts using one wavelet tree for each context x taking just $nH_h + O(\sigma^{h+1})$ bits of space.*

One problem with our current implementation of the wavelet tree is its use of subset encoding using t-subsets, requiring $O(t)$ operations on large integers [Knu05, Rus05]. To solve this problem, we introduce our subset encoder in Section 4.2, which is a data structure of independent interest. It will replace the t-subsets in the wavelet tree, without adding any additional space.

4.2 Subset Encoding With Small Integers In this section, we describe a technique for subset encoding, storing a set S of t items out of a universe of size n such that it can be encoded or decoded using $O(t)$ operations on small integers (of size $O(\lg n)$ bits). As we described in Section 4, we can think of a t-subset as a succinct way to store an implicit bitvector. We could encode this bitvector using arithmetic (or quasi-arithmetic) coding, but encoding/decoding would require $O(n)$ operations.

Another approach is to encode the *gaps* between the items $s_1, s_2, \ldots, s_t$ that appear in the set S. (The ith gap is formally $s_i - s_{i-1}$, where $s_0 = 1$.) To encode the items, we associate a probability distribution for the different gap values, and encode each gap according to its probability using any of a number of techniques (say, for instance, the quasi-arithmetic coder from [HV94]). Using this method, the items are decoded *sequentially* using $O(t)$ operations.

We will generate the gaps sequentially. For this section, we redefine t to be the number of items left to encode out of a remaining universe of size n. In other words, the values of n and t will scale as we sequentially generate gaps. (As described in [Vit84], this won't be a problem.) We define X to be the random variable that determines the length of the next gap value to be generated. Note that the range of X is the set of integers in the interval $0 \leq x \leq n-t$. We will restrict gaps to a length of at most n/t and aggregate the probabilities of larger gaps into a single escape gap. If we need to encode an escape gap of length $g > n/t$, we reset $n = n - g - 1$ and continue processing. In other words, the range for X is $0 \leq x \leq n/t$.

One approach is to generate the gap X using the exact probabilities $f(x)$. The probability distribution function (pdf) for $f(x)$ is

$$f(x) = \begin{cases} \alpha_1 \frac{t}{n} \frac{(n-x-1)^{\underline{t-1}}}{(n-1)^{\underline{t-1}}} & \text{if } 0 \leq x < n/t; \\ \alpha_2 \frac{(n-n/t-1)^{\underline{t}}}{n^{\underline{t}}} & \text{if } x = n/t; \\ 0 & \text{otherwise,} \end{cases}$$

where we use the notation $a^{\underline{b}}$ to denote the *falling power* $a(a-1)\ldots(a-b+1) = a!/(a-b)!$. The constants α_1, α_2 are normalization factors so that $f(x)$ sums up to 1. Generating gaps according to this pdf requires large integer computations. Instead, we use a probability estimate $g(x)$ from [Vit84] that is easy to compute using the built-in logarithm functions. We define $g(x)$ as

$$g(x) = \begin{cases} \beta_1 \frac{t}{n}(1 - \frac{x}{n})^{t-1} & \text{if } 0 \leq x < n/t; \\ \beta_2 e^{-1} & \text{if } x = n/t; \\ 0 & \text{otherwise,} \end{cases}$$

where β_1 and β_2 are normalization factors so that $g(x)$ sums up to 1. Here, X can be generated

quickly with only one uniform or exponential random variable [Vit84]. We will use this probability estimate in the quasi-arithmetic coder to generate the gaps using only $O(t)$ operations on *small* integers (of size $O(\lg n)$ bits). However, we may spend additional bits to encode each gap, since our probabilities are only estimates for pdf $f(x)$. We address this point in the following lemma, showing that the worst-case difference between the encodings is quite small. The complete proof of this result is somewhat more technical and involves two probability estimates; we defer these details until the full paper.

LEMMA 4.3. *The number of extra bits needed to encode a gap $X = x$ using a probability estimate $g(x)$ instead of $f(x)$ is at most $O(1/n)$. Summed over all gaps, the total encoding cost is at most $O(1)$ bits of additional space using an arithmetic coder.*

Proof. We provide a sketch of the proof in this abstract. The details will be in the full version of the paper.

We look at the worst case where we encode a gap $X = x$ using the probability estimate $g(x)$ rather than $f(x)$. The extra bits needed to encode any gap is $\lg(1/g(x)) - \lg(1/f(x)) = \lg(f(x)/g(x))$. We write

$$\begin{aligned}\lg(f(x)/g(x)) &\leq \lg\left(\frac{n-x-1}{(n-1)(1-x/n)}\right)^{t-1} \\ &= \lg\left(1+\frac{x}{n(n-x-1)}\right)^{1-t}.\end{aligned}$$

The worst-case ratio of $f(x)/g(x)$ occurs when $x = n/t$. Substituting and using simple algebra, we arrive at the result when $t \leq \sqrt{(}n)$. In the full version of the paper, we describe a second estimate with the same result when $t > \sqrt{(}n)$, thus showing the result.

We also need several other properties so that we can use a quasi-arithmetic coder; we defer this discussion until the full version of the paper. Putting together the details, we arrive at Theorem 4.1, which describes our subset-encoding scheme.

THEOREM 4.1. (SMALL INTEGER SUBSETS) *Suppose each of the t items is drawn from $[1..n]$, where we already know t and n (and do not need to encode them). Then, there exists an encoding of a subset S of t items drawn from $[1..n]$ that requires $\lg\binom{n}{t} + O(1)$ bits of space and can be encoded or decoded in $O(t)$ operations on small integers, of size $O(\lg n)$ bits.*

5 The Empirical Statistical Model

In this section, we present the second part of the proof of Theorem 3.1. In Section 3, we described a method of storing the BWT using $nH_h + M(T, \Sigma, h)$ bits. Our scheme was divided into two components: the encoding of a series of small disjoint subtexts, one for each context x, and the encoding of the length of each subtext, together with the statistics of each subtext. We did not analyze the cost required to store this empirical statistical information. We briefly recap now:

- For each context x, the storage for step 2 uses fewer than $\lg\binom{n^x}{n^{x,1},n^{x,2},\ldots,n^{x,\sigma}} + t^x$ bits by Lemma 4.1 and 4.2. We use equation (2.4) and Theorem 2.1 to bound the above term by $nH'_h + |P^*_h|\sigma$ for all contexts $x \in P^*_h$ in the worst case. Since $|P^*_h| \leq \sigma^h$, we bound the space required to store the BWT by $nH_h + \sigma^{h+1}$ bits.
- To decode step 2, we need to know the number of symbols of each type stored in each subtext for context x. Collectively, this information maintains the empirical statistical model used to achieve hth order entropy. We call its encoding length $M(T, \Sigma, h)$ (in bits), and we are interested in discovering how succinctly this information can be stored. Thus, the storage for step 1 is $M(T, \Sigma, h)$ bits.[4]

Our storage of the BWT requires $nH_h + \sigma^{h+1} + M(T, \Sigma, h)$ bits, and bounding the quantity $M(T, \Sigma, h)$ may help in understanding the compressible nature of the BWT. We will devote the rest of this section to developing two bounds for the storage of the empirical statistical model. One benefit of pursuing bounds in this framework is that it simplifies the burden of analysis: namely, it translates the overhead costs of the BWT into the cost for encoding the integer lengths $n^{x,y}$.

5.1 Definitions and a Simple Bound In this section, we describe a simple encoding for the empirical statistical model, which takes $M(T, \Sigma, h)$ bits to encode. Recall from Definition 1 in Section 2.1 that the empirical statistical model encodes two items: the partition of Σ^h induced by the optimal prefix cover P^*_h, and the sequence of lengths $n^{x,y}$ of the sublists. The partition is easily stored using a bitvector of length σ^h (or a subset encoding of the partition using $\lceil \lg\binom{\sigma^h}{|P^*_h|} \rceil \leq \sigma^h$ bits). To store the sequence of lengths $n^{x,y}$, we simply store the concatenation of the gamma codes for each length $n^{x,y}$ and bound its length.

We briefly review Elias' gamma and delta codes [Eli75] before detailing the proof. The gamma code for a positive integer ℓ represents ℓ in two parts: the first encodes $1 + \lfloor \lg \ell \rfloor$ in unary, followed by the value of $\ell - 2^{\lfloor \lg \ell \rfloor}$ encoded in binary, for a total of $1 + 2\lfloor \lg \ell \rfloor$ bits. For example, the gamma codes for $\ell = 1, 2, 3, 4, 5, \ldots$ are

[4] In this section, we will show that $M(T, \Sigma, h) \geq \sigma^{h+1}$.

1, **01 0**, **01 1**, **001 00**, **001 01**, . . ., respectively. The delta code requires fewer bits asymptotically by encoding $1 + \lfloor \lg \ell \rfloor$ via the gamma code rather than in unary. For example, the delta codes for $\ell = 1, 2, 3, 4, 5, \ldots$ are **1**, **010 0**, **010 1**, **011 00**, **011 01**, . . ., and require $1 + \lfloor \lg \ell \rfloor + 2\lfloor \lg \lg 2\ell \rfloor$ bits. Now, we describe a simple upper bound on encoding the empirical statistical model.

LEMMA 5.1. *The empirical statistical model requires* $M(T, \Sigma, h) = O\left(\sigma^{h+1} \lg(1 + n/\sigma^{h+1})\right)$ *bits.*

Proof. In this encoding, we represent the lengths using the gamma code. We obtain a bitvector Z by concatenating the gamma codes for $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$ for $x = 1, 2, \ldots, |P_h^*|$. The bitvector Z contains $O(\sum_{x \in P_h^*, y \in \Sigma} \lg n^{x,y})$ bits; this space is maximized when all lengths $n^{x,y}$ are equal to $\Theta(n/(|P_h^*| \times \sigma) + 1)$ by Jensen's inequality [CT91]. Since $|P_h^*| \leq \sigma^h$, we bound the total space by $O\big((|P_h^*| \times \sigma) \lg\big(1 + n/(|P_h^*| \times \sigma)\big)\big) = O\big(\sigma^{h+1} \lg(1 + n/\sigma^{h+1})\big)$ bits. We do not need to encode n as it can be recovered from the sum of the sublists lengths.

The result of Lemma 5.1 is interesting, but it carries a dependance on n, unlike the bounds in related work, which are related only to σ and h [FGMS05]. In Section 5.2, we show an alternate analysis that remedies this problem and relates the encoding costs to the modified entropy nH_h^*, as defined in Section 2.1.

5.2 Nearly Tight Upper Bound on $M(T, \Sigma, h)$

In this section, we describe a nearly tight upper bound for encoding the empirical statistical model. As we described in Section 5.1, we can easily store the partition of Σ^h induced by the optimal prefix cover P_h^* using at most σ^h bits. We provide a new analysis for storing the sequence of lengths $n^{x,y}$ in Theorem 5.1.

THEOREM 5.1. *The empirical statistical model requires at most* $M(T, \Sigma, h) \leq nH_h^* + \lg n + O\left(\sigma^{h+1} \lg \sigma^{h+1}\right)$ *bits of space.*

The results of Theorem 5.1 highlight a remarkable property of the Burrows-Wheeler Transform, namely that maintaining the statistics of the text requires *more* space than the actual encoding of the information.

To prove Theorem 5.1, we have to encode the sequence of sublist lengths $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$, where $x = 1, 2, \ldots, |P_h^*|$ and $\sum_{x \in P_h^*, y \in \Sigma} n^{x,y} = n$. We use the following encoding scheme for each context x:

- If context x contains a single nonempty sublist y, we use σ bits to mark the yth sublist as nonempty. Then, we store the length $n^{x,y} = n^x$.
- If context x contains two or more nonempty sublists, we again use σ bits to mark the nonempty sublists. To describe the rest of the method, let $n'_1 = n^x$ and $n'_j = n^x - \sum_{i=1}^{j-1} n^{x,i}$ be a scaled universe where $j \geq 2$. We use σ bits for context x, one bit per sublist. The bit for sublist y is **1** if and only if $n^{x,y} > n'_y/2$; in this case, we set $t_y = n'_y - n^{x,y}$. Otherwise, we set the bit for sublist y to **0** and set $t_y = n^{x,y}$. Notice that $t_y \leq n'_y/2$ in both cases. Now, we encode t using its delta code. Given n'_y and t_y, we can recover the value of $n^{x,y}$ as expected.

LEMMA 5.2. *We can encode the sublist lengths* $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$ *for any context* x *with two or more nonempty sublists using at most* $\gamma n^x H_0^*(w_x) + O(\sigma)$ *bits, where* $0 < \gamma < 1/2$ *is a constant.*

Proof. Our scheme requires 2σ bits to store auxiliary information. Now we bound the total size of encoding the values $t_1, t_2, \ldots, t_\sigma$ using the delta code for each nonempty sublist. Our approach is to amortize the cost of writing the delta code of t_y with the encoding of its associated sublist y. We introduce some terminology to clarify the proof. For any arbitrarily fixed constant γ with $0 < \gamma < 1/2$, let $t_\gamma > 0$ be constants such that for any integer $t > t_\gamma$, $\lg t + 2 \lg \lg(2t) + 1 < \gamma(2t - \lg t - 1)$.

Then, for nonempty sublists y with $t_y \leq t_\gamma$, the delta code for t_y will take $O(\lg t_\gamma) = O(1)$ bits of space. Summing these costs for all such sublists, we would require at most $O(\sigma \lg t_\gamma) = O(\sigma)$ bits for context x.

For nonempty sublists y with $t_y > t_\gamma$, we use at most $\gamma(2t_y - \lg t_y - 1)$ bits to write the delta code of t_y using the observation above.

Now, we will use the fact that $t_y \leq n'_y/2$ for each sublist y in our scheme to bound the encoding length of sublist y, and then amortize accordingly. In general, for any $1 < t < n/2$, $\lg \binom{n}{t} \geq \lg \binom{2t}{t} \geq \lg(2^{2t}/2t) = 2t - \lg t - 1$. Since each sublist y with $t_y > t_\gamma$ satisfies this condition by the construction of our scheme, we can bound the delta code of t_y by $\gamma \lg \binom{n'_y}{t_y}$ bits. Summing over all such sublists for context x, we would require at most $\gamma \lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} + \sigma = \gamma n^x H_0^*(w_x) + \sigma$ bits using the analysis from Section 4.1, thus proving the lemma.

The above scheme requires us to store the length n^x of each context x, since the sum of the t_y values we store may be less than n^x. (For the case with a single nonempty context, n^x is the size of the only sublist.) For example, suppose for some context x, $n^x = 20$, $n^{x,1} = 11$, $n^{x,2} = 3$, $n^{x,3} = 5$, $n^{x,4} = 1$. According to our scheme, we would store the t_y values 9, 3, 1, and 1, which sum up to $14 < 20$. To determine the value of $n^{x,1}$, we must therefore compute $n^x - t$; thus, we must know the value of n^x.

The storage of n^x is a subtle but important point, and it is a key component in understanding the *lower bound* on encoding length for the BWT, which we discuss more in Section 6. In Lemma 5.3, we describe a technique to store the sequence of lengths n^x for $x = 1, 2, \ldots, |P_h^*|$ using $\lg n^x$ bits, plus some small additional costs. We use this result to give a simple bound on the space of our encoding scheme in Lemma 5.4.

LEMMA 5.3. *The sequence of lengths n^x for $x = 1, 2, \ldots, |P_h^*|$ can be stored using $\sum_{x \in P_h^*} \lg n^x + \lg n + O(\sigma^{h+1} \lg \sigma^{h+1})$ bits of space.*

Proof. For each context x with n^x entries, we encode its length n^x in binary using $b(x) = \lfloor \lg n^x \rfloor + 1$ bits. These $b(x)$ bits do not permit a decoding of n^x by themselves, since they are not prefix codes. We describe how to fix this problem. We permute the contexts x so that they are sorted by their $b(x)$ values. Now, contexts requiring the same number of bits $b(x)$ to store their lengths are contiguous. In other words, we know that for any two consecutive contexts x and x' in the sorted order, either $b(x) = b(x')$ or $b(x) < b(x')$. What remains is the storage of the positions where $b(x) < b(x')$. We store this information using $|P_h^*|$ bits.

To remember which lengths $b(x)$ actually occur, we observe that the number of distinct lengths is at most $\lg n + 1$, since $1 \leq b(x) \leq \lg n + 1$. We store a bitvector of length $\lg n + 1$ bits to keep track of this information. Finally, we store the permutation to restore the original order of the contexts using $O(\lg |P_h^*|!) = O(\sigma^{h+1} \lg \sigma^{h+1})$ bits, thus proving the lemma.

LEMMA 5.4. *The empirical statistical model requires at most $(1+\gamma)nH_h^* + \lg n + O(\sigma^{h+1} \lg \sigma^{h+1})$ for all contexts x, where $0 < \gamma < 1/2$ is a constant.*

Proof. Using the definition of modified hth-order empirical entropy in equation (2.2), we bound the first term in Lemma 5.3 by $\sum_{x \in P_h^*} \lg n^x \leq nH_h^*$. According to our scheme, storing the length n^x along with σ bits is sufficient to encode any context x with a single nonempty sublist. For the remaining contexts, we apply Lemma 5.2 to achieve the desired result.

We can further improve our bound by amortizing the cost of storing the length n^x for context x with the encoding of its sublists. The technique is reminiscent of the one we used in Lemma 5.2. We change our encoding scheme as follows.

- If context x contains a single nonempty sublist y, we use σ bits to mark the yth sublist as nonempty. Then, we store the length $n^{x,y} = n^x$.
- If context x contains two or more nonempty sublists, we use the scheme below.
- Let t_γ be defined as in Lemma 5.2. For any arbitrarily fixed constant γ with $0 < \gamma < 1/2$, let $n_\gamma > 0$ be a constant such that for any integer $n > n_\gamma$ and $t > t_\gamma$ with $t \leq n/2$, $\binom{n}{t} \geq \frac{n(n-1)\ldots(n-\lceil 1/\gamma \rceil)}{(\lceil 1/\gamma \rceil + 1)!} > n^{\lceil 1/\gamma \rceil}$.
- Instead of encoding $n^{x,1}$ as the first sublist length for context x, we use σ bits to indicate that we encode n^{x,y_b} first, where $t_b = \min\{n^{x,y_b}, n^x - n^{x,y_b}\}$ satisfies the condition $\binom{n^x}{t_b} > (n^x)^{\gamma^{-1}}$. If no such y_b exists, we encode $n^{x,1}$ as before.

LEMMA 5.5. *We can encode the sublist lengths $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$ along the context length n^x for any context x with two or more nonempty sublists using at most $n^x H_0^*(w_x) + O(\sigma)$ bits.*

Proof. The cost for encoding the sublist lengths is analyzed using Lemma 5.2. We focus on bounding the cost for $\lg n^x$. If any sublist y_b satisfies the constraint in our scheme, we know that $\lg n^x < \gamma \lg \binom{n^x}{t_b}$, which is the same upper bound on the number of bits required to encode t_b. Thus, encoding both n^{x,y_b} and n^x will take $2\gamma \lg \binom{n^x}{t_b} + O(\sigma)$ bits of space. The encoding size for the rest of the new sequence remains the same as we observed in Lemma 5.2, thus we require at most $2\gamma n^x H_0^*(w_x) + O(\sigma)$ bits. Since $\gamma < 1/2$, this shows the bound for contexts x that satisfy the constraint.

If no sublist satisfied the constraint, then we know that each $t_i \leq t_\gamma (1 \leq i \leq \sigma)$ so the delta code for each t_i takes $O(\lg t_\gamma) = O(1)$ bits, which take at most $O(\sigma)$ bits overall. Then, the $\lg n^x$ bits for encoding n^x can be bounded by $n^x H_0^*(w_x)$ as in Lemma 5.4, since $n^x H_0^*(w_x) \geq \lg n^x$. This case will contribute at most $n^x H_0^*(w_x) + O(\sigma)$ bits to the bound, thus proving the bound.

Combining Lemma 5.5 with our encoding for the singleton context, we prove Theorem 5.1 and Theorem 3.1.

6 Nearly Tight Bounds for the BWT

Manzini conjectures that the BWT cannot be compressed to just $nH_h^* + \lg n + g_h$ bits of space, where $g_h = O(\sigma^{h+1} \lg \sigma)$. However, in Section 5.2, we provide an analysis that gives an upper bound of $n(H_h + H_h^*) + \lg n + g_h''$ bits, where $g_h'' = O(\sigma^{h+1} \lg \sigma^{h+1})$. Since there are an infinite number of strings where $nH_h = 0$ but $nH_h^* \neq 0$, our bound is $nH_h + M(T, \Sigma, h) \leq nH_h^* + \lg n + g_h''$ in these cases, matching Manzini's conjectured lower bound (but not for all strings).

In this section, we will explore other classes of strings that help establish a non-trivial lower bound on the compressibility of the BWT. Surprisingly, the encoding of the BWT requires an amount of space very close to our encoding length for the upper bound. In

particular, we will prove the following theorem, which shows that our upper bound analysis is nearly tight.

THEOREM 6.1. *For any chosen positive constants $\delta \leq 1$ and $k > \lceil 1/\delta \rceil = d$, there exists an infinite family of strings such that for any string of length n in the family, its encoding length $nH_h + M(T, \Sigma, h)$ satisfies the following two relations:*

(6.9)
$$nH_h^* + \frac{k-1}{k}\delta n H_h - O(\text{poly}(kd)) \leq nH_h + M(T, \Sigma, h)$$

$$nH_h + M(T, \Sigma, h) \leq nH_h^* + \delta n H_h + \lg n + g_h''. \quad (6.10)$$

When $\delta > 1$, we use Theorem 3.1 as the upper bound for (6.10). To prove inequality (6.10), we give a tighter analysis of the space-intensive part of the encoding scheme from Section 5. To capture the primary challenge from Section 5, we define a *δ-resilient* text. Let BWT(T) denote the result of applying the BWT to the text T. For any given constant δ such that $0 < \delta \leq 1$, the text T is *δ-resilient* if the optimal partition induced by P_h^* for BWT(T) satisfies $\max_{y \in \Sigma}\{n^{x,y}\} \leq n^x - \lceil 1/\delta \rceil$ for every context $x \in P_h^*$. In other words, no partition x of BWT(T) induced by P_h^* contains more than $n^x - \lceil 1/\delta \rceil$ identical symbols. We define $d = \lceil 1/\delta \rceil$. Now, we apply Theorem 5.1 to δ-resilient texts and achieve the following lemma.

LEMMA 6.1. *For any constant δ with $0 < \delta \leq 1$ and any δ-resilient text T of n symbols over Σ, we have $nH_h + M(T, \Sigma, h) \leq n(\delta H_h + H_h^*) + \lg n + g_h''$.*

To prove our lower bound inequality (6.9) from Theorem 6.1, we describe a construction scheme that takes user-defined parameters and creates a δ-resilient text T of length n. We will then count the total number of δ-resilient texts that our construction scheme generates, and use a combinatorial argument to bound the space required to distinguish between these texts.

6.1 Constructing δ-resilient Texts In this section, we describe how to construct δ-resilient texts using a generalized construction scheme; then, we will use the resulting class of texts to prove inequality (6.9) of Theorem 6.1. First, we define some terminology that will help clarify the discussion. Let $d = \lceil 1/\delta \rceil$, where $0 < \delta \leq 1$ is a constant. Let T_s be a support text of length n_s composed of an alphabet $\Sigma = \{\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_k, \mathtt{b}, \mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_k, \#\}$, where $k = O(\text{polylg}(n)) > d$ is a fixed positive integer. We assume that $\mathtt{a}_i < \mathtt{a}_{i+1} < \mathtt{b} < \mathtt{c}_j < \mathtt{c}_{j+1} < \#$ for all i and j. We define T_s as

$$T_s = \underbrace{(\mathtt{a}_1\mathtt{c}_1)^{\ell_1}}_{r_1}\ \underbrace{(\mathtt{a}_2\mathtt{c}_2)^{\ell_2}}_{r_2}\ \ldots\ \underbrace{(\mathtt{a}_k\mathtt{c}_k)^{\ell_k}}_{r_k},$$

where each length parameter $\ell_i \geq d$. We define a run r_i as the sequence of ℓ_i substrings of the form $\mathtt{a}_i\mathtt{b}^*\mathtt{c}_i$. In T_s, $\mathtt{b}$ never appears. The length of the support string T_s is $n_s = 2\sum_{i=1}^{k} \ell_i$. We now prove the following lemma.

LEMMA 6.2. *The* BWT(T_s) *is*

$$\text{BWT}(T_s) = \overbrace{\underbrace{\mathtt{c}_k(\mathtt{c}_1)^{\ell_1-1}}_{P_1}\ \underbrace{\mathtt{c}_1(\mathtt{c}_2)^{\ell_2-1}}_{P_2}\ \ldots\ \underbrace{\mathtt{c}_{k-1}(\mathtt{c}_k)^{\ell_k-1}}_{P_k}}^{B_1}$$
$$\overbrace{\underbrace{(\mathtt{a}_1)^{\ell_1}}_{Q_1}\ \underbrace{(\mathtt{a}_2)^{\ell_2}}_{Q_2}\ \ldots\ \underbrace{(\mathtt{a}_k)^{\ell_k}}_{Q_k}}^{B_2},$$

where $B_1 = P_1P_2\ldots P_k$ as the first block of the BWT *transform, and $B_2 = Q_1Q_2\ldots Q_k$ as the second block. Here, P_i refers to the positions of the* BWT *corresponding to strings that start with symbol $\mathtt{a}_i$, and Q_i refers to positions of the* BWT *corresponding to strings that start with symbol $\mathtt{c}_i$.*

Proof. Consider the strings in the BWT matrix M, sorted in lexicographical order. According to the rank of symbols in alphabet Σ, all strings beginning with $\mathtt{a}_i$ will precede strings before $\mathtt{a}_{i+1}$. Similarly, strings beginning with $\mathtt{c}_i$ will precede strings beginning with $\mathtt{c}_{i+1}$. Finally, all strings beginning with $\mathtt{a}_i$ will precede strings beginning with $\mathtt{c}_1$. Also, there are exactly ℓ_i strings that begin with $\mathtt{a}_i$ and $\mathtt{c}_i$. We now focus on the strings that begin with $\mathtt{c}_i$.

Each string beginning with $\mathtt{c}_i$ has the symbol $\mathtt{a}_i$ preceding it (or equivalently, at the end of the string) in all cases. Thus, the part of the BWT corresponding to strings beginning with $\mathtt{c}_i$ is $(\mathtt{a}_i)^{\ell_i}$. Collectively, we call this block B_2.

Each string beginning with $\mathtt{a}_i$ has the symbol $\mathtt{c}_i$ preceding it (or at the end of the string, since it's cyclic), except the string corresponding to the first $\mathtt{a}_i$ in run r_i. This string is lexicographically the first string among all of the strings beginning with $\mathtt{a}_i$ and is preceded by $\mathtt{c}_{i-1}$ or $\mathtt{c}_k$ if $i = 1$. Thus, the part of the BWT corresponding to strings beginning with $\mathtt{a}_i$ is $\mathtt{c}_{i-1}(\mathtt{c}_i)^{\ell_i-1}$. If $i = 1$, $\mathtt{c}_{i-1}$ is replaced with $\mathtt{c}_k$. Collectively, we call this block B_1.

Thus, the lemma is proved.

Now, we introduce $d = \lceil 1/\delta \rceil$ partition vectors $v_i = \langle v_i[1], v_i[2], \ldots v_i[k] \rangle$ which will generate a δ-resilient property for B_2; B_1 remains unchanged, but will implicitly encode the length of the corresponding portions of B_2. We augment T_s as follows: for each entry of v_i for all i, we replace the $v_i[j]$th occurrence of the string $\mathtt{a}_j\mathtt{c}_j$ with $\mathtt{a}_j\mathtt{b}\mathtt{c}_j$. We will make d such replacements in each of the k partitions. We call this augmented text T_s', of length $n_s' = n_s + dk$.

LEMMA 6.3. *The* BWT(T'_s) *is*

$$\text{BWT}(T'_s) = \overbrace{P'_1P'_2\dots P'_k}^{B_1}\ \overbrace{(\mathtt{a}_1)^d(\mathtt{a}_2)^d\dots(\mathtt{a}_k)^d}^{A}\ \overbrace{Q'_1Q'_2\dots Q'_k}^{B_2},$$

where P'_i is composed of symbols preceding strings that start with $\mathtt{a}_i$, A is composed of symbols preceding strings that start with b*, and Q'_i is composed of symbols preceding strings that start with $\mathtt{c}_i$.*

Proof. This proof is similar to Lemma 6.2, where each string in P_i precedes strings in P_{i+1}. Here, all strings in P'_i precede strings in P'_{i+1}, strings in Q'_i precede strings in Q'_{i+1}, and strings in P'_i precede strings beginning with b (called A) and strings in A precede strings in Q'_1.

Then, P'_i is a string of length ℓ_i similar to P_i, but the single occurrence of $\mathtt{c}_{i-1}$ (or $\mathtt{c}_k$ if $i = 1$) could be in any of the ℓ_i positions. Also, Q'_i is a string of length ℓ_i where d positions contain the symbol b, and all others are $\mathtt{a}_i$. Block A consists of exactly d occurrences of each $\mathtt{a}_i$ sorted in lexicographical order, since all d strings beginning with $\mathtt{bc}_i$ precede all strings beginning with $\mathtt{bc}_{i+1}$, thus finishing the proof.

A simple verification will show that blocks A and B_2 are δ-resilient portions of T'_s. Furthermore, block A is deterministic once the parameters d and k have been chosen; block B_1 encodes the length of each Q'_i. To have a fully δ-resilient text, we want B_1 to have the same property, so we generate the string $T = T'_s(T_s)^{d-1}\#$. This will include $d-1$ occurrences of a different symbol inside each P'_1. Note that $|T| = n = dn_s + dk + 1$.

LEMMA 6.4. *The* BWT(T) *is*

$$\text{BWT}(T) = \overbrace{P''_1P''_2\dots P''_k}^{B_1}\ A\ \overbrace{Q''_1Q''_2\dots Q''_k}^{B_2}\ \mathtt{c}_k,$$

where P''_i is composed of symbols preceding strings that start with $\mathtt{a}_i$, A is composed of symbols preceding strings that start with b*, and Q''_i is composed of symbols preceding strings that start with $\mathtt{c}_i$.*

Proof. The strings P''_i and Q''_i are of length $d\ell_i$. Similar to the arguments in Lemma 6.3, P''_1 consists of the symbol $\mathtt{c}_1$ in all but $d\ell_1 - d$ positions; one position contains # and the other $d-1$ positions contain $\mathtt{c}_k$. P''_i consists of the symbol $\mathtt{c}_i$ in all but $d\ell_i - d$ positions; the other d positions contain $\mathtt{c}_{i-1}$. Each Q''_i is similar to the previous case, except its length is now $d\ell_i$. Q''_i still contains only d occurrences of b.

Finally, the last $\mathtt{c}_k$ is the symbol preceding # in the text, which is lexicographically the largest symbol, and therefore the last string represented in the BWT, thus finishing the proof.

6.2 Encoding a δ-resilient Text In this section, we analyze the space required to store a δ-resilient text T. Since B_1 and A are deterministic once d and k are chosen, we focus only on the encoding cost of B_2. First, we prove the following lemma.

LEMMA 6.5. *For any set of p objects, at least half of them will take at least $\lg p - 1$ bits to encode so that the objects can be distinguished from one another.*

Proof. Since one can distinguish at most 2^j objects from one another using j bits, the most succinct encoding would greedily store two objects using one bit each, four objects using two bits each, and so on. Thus, we need to make sure that $\sum_i^j 2^i \geq p$. Thus, $j + 1 \geq \lg p$, and the lemma follows.

Let Λ be the set of all possible choices λ of length parameters $\ell_1, \ell_2, \dots, \ell_k$ used to generate δ-resilient texts in Section 6.1. By construction, $|\Lambda| = \binom{n_s/2-dk+k-1}{k-1}$. For a given choice λ of parameters, we choose d positions in each partition Q''_i that will contain a b. However, we are only choosing from the first ℓ_i positions for each run r_i (i.e. the positions that correspond to the entries in T'_s). Once these positions are chosen, we perform the steps described in our construction scheme. Since the BWT is a reversible transform, we have $\binom{\ell_i}{d}$ possible partitions Q''_i and our construction scheme generates one of

$$X = \sum_{\lambda\in\Lambda} \binom{\ell_1}{d}\binom{\ell_2}{d}\dots\binom{\ell_k}{d}$$

different texts. We let an adversary encode the X texts in any way he wishes. Then, we use Lemma 6.5 to consider only half of these texts, namely the ones that take at least $\lg X - 1$ bits to encode. Now we analyze the quantity $\lg X - 1$.

To help analyze $\lg X - 1$, we divide Λ into two sets Y and Z of equal cardinality, such that for any texts $y \in Y$ and $z \in Z$, the product $p(y) \leq p(z)$, where $p(T) = \prod_1^k \binom{\ell_i}{d}$. In words, Y contains the texts T where $p(T)$ is smaller, and Z contains the ones where $p(T)$ is larger. We take a single arbitrary text S from set Y and determine which choice λ^* of length parameters ℓ_i was used. We separate the k terms corresponding to λ^* from $\lg X - 1$ and analyze their cost separately. The terms are $\sum_1^k \lg \binom{\ell_i}{d} = nH'_1(S)$, by our definition of finite set empirical entropy. Since $nH'_h(S) \leq nH'_1(S)$, the contribution of this part of $\lg X - 1$ is at least $nH'_h(S)$ bits. We translate this into a bound in terms of nH^*_h using the following lemma.

LEMMA 6.6. *For a δ-resilient text, $nH_h^* - \Theta(k \lg d) \leq nH_h'$.*

Proof. It suffices to show that $nH_0^* - \Theta(\lg d) \leq nH_0'$ for each partition Q_i'' in a δ-resilient text, since there are at most $2k+1$ partitions. We apply Stirling's double inequality to the expression $\lg \binom{\ell_i}{d}$ and find that

$$\begin{aligned} \lg \binom{l_i}{d} &> \ell_i H_0 + \frac{1}{2} \lg \frac{\ell_i}{d(\ell_i - d)} - O(1) \\ &> \ell_i H_0 + \frac{1}{2} \lg \frac{1}{d} - O(1), \end{aligned}$$

thus proving the lemma.

Thus, the total contribution of the part of $\lg X - 1$ corresponding to the text S is at least $nH_h^*(S) - \Theta(k \lg d)$ bits. Now we bound the term X to figure out the entire cost of encoding the string S. We will lower bound X by the sum for just the set Z and obtain

$$\begin{aligned} X &\geq \sum_{z \in Z} \prod_{1}^{k} \binom{\ell_i}{d} \\ &\geq \sum_{z \in Z} p(S) \\ &= \frac{1}{2} \binom{n_s/2 - dk + k - 1}{k-1} p(S). \end{aligned}$$

Taking logs, we require $nH_h^*(S) + \lg \binom{n_s/2-dk+k-1}{k-1} - 1$ bits of space.

To finish the proof, we analyze the contribution of the term $\lg \binom{n_s/2-dk+k-1}{k-1}$. For ease of notation, let $g = n_s/2 - dk + k - 1$. We want to show that $(k-1) \lg g/(k-1) \leq \lg \binom{g}{k-1}$. The claim is true by inspection when $g \leq 4$ or $k-1$ is $0, 1$, or $g-1$. For the remainder of the cases, we apply Stirling's inequality as in Theorem 2.1 to verify the claim. Now, $(k-1) \lg g/(k-1) \geq (k-1) \lg n_s/2 - (k-1) \lg(dk) - (k-1) \lg k$. Thus, the contribution of this part of $\lg X - 1$ is at least $(k-1) \lg n - \Theta(k \lg(dk))$ bits, proving inequality (6.9) and Theorem 6.1 for any arbitrary δ-resilient text S.

References

[BB04] D. Baron and Y. Bresler. An $o(n)$ semipredictive universal encoder via the bwt. *IEEE Transactions on Information Theory*, 50:928–937, 2004.

[BW94] M. Burrows and D.J. Wheeler. A block sorting data compression algorithm. Technical report, Digital Systems Research Center, 1994.

[CT91] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, 1991.

[Eli75] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21:194–203, 1975.

[EVKV02] Michelle Effros, Karthik Visweswariah, Sanjeev R. Kulkarni, and Sergio Verdu. Universal lossless source coding with the burrows-wheeler transform. *IEEE Transactions on Information Theory*, 48(5):1061–1081, 2002.

[Fel68] William Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, New York, 3rd edition, 1968.

[FGMS05] Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Gabriella Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005. (Also in CPM 2003, ACM-SIAM SODA 2004.).

[FM05] Paolo Ferragina and Giovanni Manzini. On compressing and indexing data. *Journal of the ACM*, 52(4):552–581, 2005. (Also in IEEE FOCS 2000.).

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, January 2003.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[HV94] Paul G. Howard and Jeffrey Scott Vitter. Arithmetic coding for data compression. *Proceedings of the IEEE*, 82(6), June 1994.

[KLV06] Haim Kaplan, Shir Landau, and Elad Verbin. A simpler analysis of burrows-wheeler based compression. pages 282–293, 2006.

[KM99] S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with lempel-ziv algorithms. *SIAM J. Comput.*, 29(3):893–911, 1999.

[Knu05] Donald E. Knuth. *Combinatorial Algorithms*, volume 4 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, 2005. In preparation.

[Man01] Giovanni Manzini. An analysis of the Burrows — Wheeler transform. *Journal of the ACM*, 48(3):407–430, May 2001.

[NM06] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. 2006. To appear.

[Rus05] Frank Ruskey. *Combinatorial Generation*. 2005. In preparation.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, July 1948.

[Vit84] Jeffrey Scott Vitter. Faster methods for random sampling. *Communications of the ACM*, 27(7):703–718, July 1984.

[WMF94] Marcelo J. Weinberger, Neri Merhav, and Meir Feder. Optimal sequential probability assignment for individual sequences. *IEEE Transactions on Information Theory*, 40:384–396, 1994.

Bloom maps

David Talbot* John Talbot†

Abstract

We consider the problem of succinctly encoding a static map to support approximate queries. We derive upper and lower bounds on the space requirements in terms of the error rate and the entropy of the distribution of values over keys: our bounds differ by a small constant factor. For the upper bound we introduce a novel data structure, the *Bloom map*, generalising the Bloom filter to this problem. The lower bound follows from an information theoretic argument.

1 Introduction

The ability to query a map to retrieve a *value* given a *key* is fundamental in computer science. As the universe from which keys are drawn grows in size, information theoretic lower bounds imply that any data structure supporting error-free queries of a map requires unbounded space per key. However, if we are willing to accept errors, constant space per key is sufficient.

For example, in information retrieval we may wish to query the frequencies (values) of word sequences (keys) in documents. *A priori* these sequences are drawn from a universe that is exponential in the length of a sequence. Returning an incorrect value for a small proportion of queries may be acceptable, if this enables us to support queries over a far larger data set.

Consider a map consisting of n key/value pairs $M = \{(x_1, v(x_1)), (x_2, v(x_2)), \ldots, (x_n, v(x_n))\}$, where the keys $X = \{x_1, x_2, \ldots, x_n\}$ are drawn from a large universe U and each value $v(x)$ is drawn from a fixed set of possible values $V = \{v_1, v_2, \ldots, v_b\}$. Suppose further that the distribution of values over keys is given by $\vec{p} = (p_1, p_2, \ldots, p_b)$. Thus if $X_i = \{x \in X \mid v(x) = v_i\}$ then $|X_i| = p_i n$.

We consider the problem of constructing a space-efficient data structure supporting queries on M. For any key $x \in U$ the data structure should return the associated value $v(x)$ if $x \in X$, otherwise (i.e. if $x \in U \backslash X$) it should return $\perp \notin V$.

Using an information theoretic argument, we derive lower bounds on the space required to solve this problem when errors are allowed. These lower bounds are in terms of the error rate and the entropy of the distribution of values over keys $H(\vec{p})$.

We introduce the *Bloom map*, a data structure generalising the Bloom filter [1] to the approximate map problem. The space requirements of this data structure are within a $\log e$ factor of the lower bound. To be precise for an error rate of ϵ the Bloom map uses $\log e(\log 1/\epsilon + H(\vec{p}))$ bits per key.

To our knowledge, this paper is the first to make use of the distribution of values over keys to analyse the approximate map problem. In particular, the Bloom map is the first data structure to take advantage of this distribution to save space by using variable length codes for distinct values. In the many practical settings where distributions with low entropy are encountered we expect the Bloom map to be of significant interest.

The main prior work on the approximate map problem is the Bloomier filter introduced by Chazelle et al. [3]. To store key/value pairs with values drawn from a range of size b with false positive probability ϵ, the Bloomier filter requires $\alpha(\log 1/\epsilon + \log b)$ bits per key effectively using a fixed width encoding for any value in the range. It always returns the correct value for any $x \in X$. The Bloomier filter uses a perfect hash function introduced earlier by Czech et al. [5] whose analysis implies that the optimal α is approximately 1.23. A simple calculation shows that in many cases the Bloom map will use less space. In fact it is also straightforward to extend the Bloom map to make use of the same family of perfect hash functions thereby reducing its space requirements to $1.23(\log 1/\epsilon + H(\vec{p}))$.

An extension of the Bloom filter suitable for this problem when values are geometrically distributed has also recently been proposed by Talbot and Osborne [7].

In the next section we give a complete statement of the problem and prove lower bounds on the space requirements of any data structure supporting approximate queries of a static map with bounded errors (our most general result is Theorem 2.2). In section 3 we introduce the Simple Bloom map, a data structure supporting approximate queries that has near-optimal space requirements. In section 4 we present more computationally efficient versions of the Bloom map.

*School of Informatics, University of Edinburgh. This author is supported by a grant from Google Research. Email: d.r.talbot@sms.ed.ac.uk.

†Department of Mathematics, University College London. Email: talbot@math.ucl.ac.uk. This author is a Royal Society University Research Fellow.

2 Problem statement and lower bounds

Consider a map of n key/value pairs $M = \{(x_1, v(x_1)), (x_2, v(x_2)), \ldots, (x_n, v(x_n))\}$, where the keys $X = \{x_1, x_2, \ldots, x_n\}$ are drawn from a large universe U of size u and each value $v(x)$ is drawn from a fixed set of possible values $V = \{v_1, v_2, \ldots, v_b\}$. Suppose further that the distribution of values over keys is given by $\vec{p} = (p_1, p_2, \ldots, p_b)$, where $\sum_{i=1}^{b} p_i = 1$ and $\min_{i \in [b]} p_i > 0$. Thus if $X_i = \{x \in X \mid v(x) = v_i\}$ then $|X_i| = p_i n$. We call such a collection M of key/value pairs a *$\vec{p}$-map*.

We consider the problem of constructing a space-efficient data structure supporting queries on a static $\vec{p}$-map M. For any key $x \in U$ the data structure should return the associated value $v(x)$ if $x \in X$, otherwise it should return $\perp \notin V$. We will be interested in the case when n is large, $u \gg n$ and $b, \vec{p} = (p_1, p_2, \ldots, p_b)$ are constant.

Given u, n, b and $\vec{p} = (p_1, p_2, \ldots, p_b)$ the total number of distinct $\vec{p}$-maps is

$$\binom{u}{n}\binom{n}{p_1 n, p_2 n, \ldots, p_b n}.$$

By Stirling's formula the multinomial coefficient is $2^{nH(\vec{p})+O(\log n)}$, where $H(\vec{p}) = -\sum_{i=1}^{b} p_i \log p_i$ is the entropy of $\vec{p}$. (Logarithms here and elsewhere are base two.) If we use m-bit binary strings to distinguish between all $\vec{p}$-maps without errors we require

$$2^m \geq \binom{u}{n}\binom{n}{p_1 n, p_2 n, \ldots, p_b n}.$$

Hence we need $m \geq n(\log u - \log n + H(\vec{p}) + o(1))$ bits. For n large and $u \gg n$ this is prohibitive: in particular we require more than a constant number of bits per key. Hence we are obliged to consider lossy data structures.

There are three distinct types of error that we will consider:

- *(False positives)* $x \in U \backslash X$ is incorrectly assigned a value $v_i \in V$;
- *(False negatives)* $x \in X_i$ is incorrectly assigned the value $\perp$;
- *(Misassignments)* $x \in X_i$ is incorrectly assigned a value $v \in V \backslash \{v_i\}$.

Let s be a binary string supporting queries by keys $x \in U$, i.e. $s : U \to V \cup \{\perp\}$. Suppose that we use s to encode a $\vec{p}$-map M with key set X. We wish to bound the proportion of keys on which s returns an incorrect value. For $i \in [b]$ we define

$$\begin{aligned} f^+(s) &= \frac{|\{x \in U \backslash X \mid s(x) \neq \perp\}|}{|U \backslash X|}, \\ f_i^*(s) &= \frac{|\{x \in X_i \mid s(x) \in V \backslash \{v_i\}\}|}{|X_i|}, \\ f_i^-(s) &= \frac{|\{x \in X_i \mid s(x) = \perp\}|}{|X_i|}. \end{aligned}$$

Thus $f^+(s)$ is the proportion of false positives returned on $U \backslash X$, $f_i^*(s)$ is the proportion of misassigned values on X_i and $f_i^-(s)$ is the proportion of false negatives returned on X_i.

Given constants $\epsilon^+ > 0$ and $\epsilon^*, \epsilon^- \geq 0$ we will say that s *$(\epsilon^+, \epsilon^*, \epsilon^-)$-encodes M* if it satisfies: $f^+(s) \leq \epsilon^+$ and, for all $i \in [b]$, $f_i^*(s) \leq \epsilon^*$ and $f_i^-(s) \leq \epsilon^-$. (We will assume throughout that $\max\{\epsilon^+, \epsilon^*, \epsilon^-\} < 1/8$.)

If the only errors we allow are false positives then we have an $(\epsilon^+, 0, 0)$-encoding data structure. (An example of such a data structure is the Bloomier filter [3]). Theorem 2.1 gives lower bounds on the space requirements of such a data structure. (The proof follows a counting argument generalising the argument applied to the approximate set membership problem by Carter et al. [2].)

THEOREM 2.1. *The average number of bits required per key in any data structure that $(\epsilon^+, 0, 0)$-encodes all $\vec{p}$-maps is at least*

$$\log 1/\epsilon^+ + H(\vec{p}) + o(1).$$

Proof. Suppose that the m-bit string s $(\epsilon^+, 0, 0)$-encodes some particular $\vec{p}$-map M with key set X. For $i \in [b]$ let $A_i^{(s)} = \{x \in U \mid s(x) = v_i\}$, $a_i^{(s)} = |A_i^{(s)}|$ and define $q_i^{(s)}$ by $a_i^{(s)} = p_i n + \epsilon^+(u - n)q_i^{(s)}$. Since $X_i = \{x \in X \mid v(x) = v_i\}$ has size $p_i n$ and s always answers correctly on X_i we have $q_i^{(s)} \geq 0$.

The proportion of $x \in U \backslash X$ for which $s(x) \neq \perp$ is $\sum_{i=1}^{b} \epsilon^+ q_i^{(s)}$. Since $f^+(s) \leq \epsilon^+$, this implies that $\sum_{i=1}^{b} q_i^{(s)} \leq 1$.

If N is any $\vec{p}$-map with key set Y that is also $(\epsilon^+, 0, 0)$-encoded by s then, since s correctly answers all queries on keys in Y, we have $Y_i = \{y \in Y \mid v(y) = v_i\} \subseteq A_i^{(s)}$, for all $i \in [b]$. Hence, since $|Y_i| = p_i n$, s can $(\epsilon^+, 0, 0)$-encode at most the following number of distinct $\vec{p}$-maps

$$\prod_{i=1}^{b} \binom{a_i^{(s)}}{p_i n} = \prod_{i=1}^{b} \binom{p_i n + \epsilon^+(u-n)q_i^{(s)}}{p_i n}.$$

Choosing $q_1, q_2, \ldots, q_b \geq 0$ to maximise this expression,

subject to $\sum_{i=1}^{b} q_i \leq 1$, we must have

$$2^m \prod_{i=1}^{b} \binom{p_i n + \epsilon^+(u-n)q_i}{p_i n} \geq \binom{u}{n} \binom{n}{p_1 n, p_2 n, \ldots, p_b n}.$$

Using the fact that $\frac{(a-b)^b}{b!} \leq \binom{a}{b} \leq \frac{a^b}{b!}$ and taking logarithms we require

$$m + \sum_{i=1}^{b} p_i n \log(p_i n + \epsilon^+(u-n)q_i) \geq n \log(u-n).$$

Dividing by n, recalling that $\sum_{i=1}^{b} p_i = 1$ and rearranging we obtain

$$\frac{m}{n} \geq \log 1/\epsilon^+ + \sum_{i=1}^{b} p_i \log 1/q_i + \log\left(1 - \frac{n}{u}\right) - \sum_{i=1}^{b} p_i \log\left(1 + \frac{n(p_i - \epsilon^+ q_i)}{\epsilon^+ q_i u}\right).$$

Our assumption that $u \gg n$ (which is equivalent to $n/u = o(1)$) together with the fact that $\log(1+\alpha) = O(\alpha)$ for α small implies that the last two terms are $o(1)$. Hence the average number of bits required per key satisfies

$$\frac{m}{n} \geq \log 1/\epsilon^+ + \sum_{i=1}^{b} p_i \log 1/q_i + o(1).$$

Gibbs' inequality implies that the sum is minimised when $q_i = p_i$ for all $i \in [b]$, the result follows. □

This calculation can be extended to the case when errors are also allowed on keys in the set X.

THEOREM 2.2. *The average number of bits required per key in any data structure that $(\epsilon^+, \epsilon^*, \epsilon^-)$-encodes all $\vec{p}$-maps is at least*

$$(1-\epsilon^-)\log 1/\epsilon^+ + (1-\epsilon^- - \epsilon^*)H(\vec{p}) - H(\epsilon^-, \epsilon^*, 1-\epsilon^- - \epsilon^*) + o(1).$$

Proof. The basic idea behind the proof of this result is the same as that of Theorem 2.1, however the details are somewhat more involved. (See Appendix.) □

The Bloom map, which we introduce in Section 3, is $(\epsilon, \epsilon, 0)$-encoding. To enable us to evaluate how far its space requirements are from optimal we give the following simple corollary.

COROLLARY 2.1. *The average number of bits required per key in any data structure that $(\epsilon, \epsilon, 0)$-encodes all $\vec{p}$-maps is at least*

$$(1-\epsilon)(\log 1/\epsilon + H(\vec{p}) - (\epsilon + \epsilon^2)) + o(1).$$

Proof. Substitute $\epsilon^+ = \epsilon^* = \epsilon$ and $\epsilon^- = 0$ into Theorem 2.2 and use $\log(1-\epsilon) \geq -(\epsilon + \epsilon^2)$. □

3 The Simple Bloom map

Let M be a $\vec{p}$-map with key set X. Thus, for $i \in [b]$, $X_i = \{x \in X \mid v(x) = v_i\}$ has size $p_i n$. Our first succinct data structure supporting queries for M is the Simple Bloom map. This is constructed by simply storing the values directly in a Bloom filter.

Let B be an array of size m that is initially empty. For each $i \in [b]$ we choose $k_i \geq 1$ independent random hash functions $h_{i,j} : U \to [m]$ (we will explain how to set $k_1, k_2, \ldots, k_b$ optimally below). To store the key/value pair (x, v_i) we compute $h_{i,j}(x)$ for each $j \in [k_i]$ and set the bits $B[h_{i,j}(x)]$ to one. To query B with a key $x \in U$ we compute $h_{i,j}(x)$ for each $i \in [b], j \in [k_i]$ and set

$$\mathsf{qval}(x) = \left\{ i \in [b] \mid \wedge_{j=1}^{k_i} B[h_{i,j}(x)] = 1 \right\}.$$

If $\mathsf{qval}(x) = \emptyset$ we return $\perp$ otherwise we return v_c, where $c = \max \mathsf{qval}(x)$. Note that if $(x, v_i) \in M$ then $i \in \mathsf{qval}(x)$ and so $\perp$ is never returned when querying x, i.e. there are no false negatives. However both false positives and misassignments can occur.

Let $t = n \sum_{i=1}^{b} p_i k_i$ be the total number of hashes performed during the creation of B. Let ρ be the proportion of bits that remain zero in B.

By a simple martingale argument, identical to that given by Mitzenmacher [6] for the Bloom filter, ρ is extremely close to its expected value if $t = O(m)$. (To be precise: if Y_j is the expected number of bits that remain zero in the Simple Bloom map B, conditioned on the first j hashes then $Y_0 = \mathbf{E}[\rho m]$ while $Y_t = \rho m$. The Y_j form a martingale with $|Y_{j+1} - Y_j| \leq 1$. Azuma's inequality now implies that for any $\lambda > 0$ we have

$$\mathbf{Pr}\left\{ \rho < \mathbf{E}[\rho] - \frac{\lambda\sqrt{t}}{m} \right\} < e^{-\lambda^2/2}.$$

Hence if $t = O(m)$ then ρ is extremely unlikely to be much smaller than its expected value. Note that this argument also implies that the same is true for the more efficient Bloom maps described in Section 4.)

Let $\hat{\rho} = \mathbf{E}[\rho] = (1 - 1/m)^t$. By the previous remark we may assume that $\rho \geq \hat{\rho}$.

If $f^+(B)$ is the false positive probability of B, i.e. the probability that B returns $v \neq \perp$ for a fixed

$x \in U \backslash X$, then

$$\begin{aligned} f^+(B) &= \mathbf{Pr}\{\mathsf{qval}(x) \neq \emptyset\} \\ &\leq \sum_{i=1}^{b} \mathbf{Pr}\{i \in \mathsf{qval}(x)\} \\ &= \sum_{i=1}^{b} (1-\rho)^{k_i} \\ &\leq \sum_{i=1}^{b} (1-\hat{\rho})^{k_i}. \end{aligned}$$

If $f_i^*(B)$ is the misassignment probability for B over keys in X_i, i.e. the probability that B returns $v \in V \backslash \{v_i\}$ for a fixed $x \in X_i$, then

$$\begin{aligned} f_i^*(B) &= \mathbf{Pr}\{\max \mathsf{qval}(x) > i\} \\ &\leq \sum_{j=i+1}^{b} \mathbf{Pr}\{j \in \mathsf{qval}(x)\} \\ &= \sum_{j=i+1}^{b} (1-\rho)^{k_j} \\ &\leq \sum_{i=1}^{b} (1-\hat{\rho})^{k_i}. \end{aligned}$$

Hence in order to minimise $f^+(B)$ and $f_i^*(B)$ we consider the constrained optimisation problem: minimise $\sum_{i=1}^{b} (1-\hat{\rho})^{k_i}$ subject to $\sum_{i=1}^{b} p_i k_i = t/n$.

A standard application of Lagrange multipliers yields the solution

$$k_i = \frac{t}{n} + \frac{H(\vec{p}) + \log p_i}{\log(1-\hat{\rho})}.$$

For this choice of the k_i we have

$$\sum_{i=1}^{b} (1-\hat{\rho})^{k_i} = (1-\hat{\rho})^{t/n} 2^{H(\vec{p})} \sum_{i=1}^{b} p_i = 2^{H(\vec{p})} (1-\hat{\rho})^{t/n}.$$

Now

$$\begin{aligned} 2^{H(\vec{p})}(1-\hat{\rho})^{t/n} &= 2^{H(\vec{p})} \left(1 - \left(1 - \frac{1}{m}\right)^t\right)^{t/n} \\ &= 2^{H(\vec{p})} \exp\left(\frac{\ln \hat{\rho} \ln(1-\hat{\rho})}{n \ln(1-1/m)}\right). \end{aligned}$$

This last expression (without the factor $2^{H(\vec{p})}$) is familiar from the standard Bloom filter error analysis: by symmetry it is minimised at $\hat{\rho} = 1/2$. Using the approximation $\hat{\rho} \approx e^{-t/m}$ we require $t = m \ln 2$. (Note that as for the standard Bloom filter the expected proportion of bits set in B is $1/2$.)

Thus to guarantee $f^+(B) \leq \epsilon$ and $f_i^*(B) \leq \epsilon$ for all $i \in [b]$ it is sufficient to take

$$m = n \log e (\log 1/\epsilon + H(\vec{p}))$$

and

$$k_i = \log 1/\epsilon + \log 1/p_i \quad \text{for } i \in [b].$$

(As with the standard Bloom filter, the k_i must be integers, for simplicity we will ignore this.)

Having given lower bounds on the space required by $(\epsilon, \epsilon, 0)$-encoding data structures in Corollary 2.1 we would like to claim that the Simple Bloom map B is $(\epsilon, \epsilon, 0)$-encoding. This is not quite true: the *expected* proportion of false positives and misassignments is at most ϵ but this does not guarantee that B is $(\epsilon, \epsilon, 0)$-encoding. (This is no different from the often overlooked fact that for an ordinary Bloom filter with false positive probability ϵ the *proportion* of keys in $U \backslash X$ for which the filter returns a false positive may be larger than ϵ.) However the events "B returns a false positive on query x", $x \in U \backslash X$, are independent and have probability at most $f^+(B)$. Hence a simple application of Hoeffding's bound for the tail of the binomial distribution shows that with high probability the proportion of false positives is at most $f^+(B) + O(1/\sqrt{u-n})$. Applying the same bound to the proportion of misassignments, we have an *essentially* $(\epsilon, \epsilon, 0)$-encoding data structure. (Note that a similar argument implies that the same is true of the more efficient Bloom maps described in Section 4.)

THEOREM 3.1. *The Simple Bloom map $(\epsilon, \epsilon, 0)$-encodes all $\vec{p}$-maps and uses $\log e(\log 1/\epsilon + H(\vec{p}))$ bits per key.*

Note that by Corollary 2.1 the space requirements of the Simple Bloom map are essentially a factor $(1-\epsilon)^{-1} \log e$ from optimal, for $\epsilon \leq 0.01$ this is less than 1.46.

We remark that an $(\epsilon, \epsilon, \epsilon)$-encoding data structure can be created from the Simple Bloom map by simply discarding $\epsilon p_i n$ keys from X_i for each $i \in [b]$. The amount of memory saved is $\epsilon n \log e(\log 1/\epsilon + H(\vec{p}))$ (cf. Theorem 2.2).

Although the Simple Bloom map is succinct it suffers from two obvious drawbacks if b is not small: the number of hashes/bit probes performed during a query and the number of independent hash functions required is $O(b \log(b/\epsilon))$. In section 4 we explain how to overcome these problems by "reusing" hash functions and using an optimal binary search tree.

4 Efficient Bloom maps

Let M be a $\vec{p}$-map that we wish to store. Sort the list of probabilities of keys so that $p_1 \geq p_2 \geq \cdots \geq p_b$. Construct an optimal alphabetic binary tree $T(\vec{p})$ for

$\vec{p}$ with leaves labelled $v_1, v_2, \ldots, v_b$ (by for example the Garsia–Wachs algorithm, see Knuth [4] page 446). The label of a leaf w is denoted by $\mathsf{val}(w)$. Note that $T(\vec{p})$ is a full binary tree, i.e. every node is either a leaf or has exactly two children.

For any binary tree T let $r(T)$ denote its root and T_L, T_R denote its left and right subtrees respectively. For any node w let P_w denote the set of nodes on the path in T from the root to w and let $l_w = |P_w| - 1$ be the depth of w. For $d \geq 0$ let T_d be the set of nodes in T at depth d.

We number the nodes in $T(\vec{p})$ from left to right at each level, starting at the root and going down. We call these numbers *offsets*. (So the root has offset 0, its left child has offset 1 and its right child has offset 2 etc.) Note that all nodes have distinct offsets. The offset of a node w is denoted $\mathsf{off}(w)$. To each node $w \in T(\vec{p})$ we also associate an integer k_w. We will specify choices for the k_w later, we first impose two simple conditions: $k_w \geq 1$ for all nodes and $k_w \geq \log 1/\epsilon$ for all leaves. Set (4.1)

$$m = \log e \left(\sum_{i=1}^{b} p_i n \sum_{w \in P_{v_i}} k_w \right), \quad k = \max_{i \in [b]} \sum_{w \in P_{v_i}} k_w.$$

We now impose a third condition on the k_w: they are chosen so that $m \leq 2n \log e \log(b/\epsilon)$.

Let $h_1, h_2, \ldots, h_k$ be independent random hash functions, $h_j : U \to [m]$. (We will refer to these as the *base* hash functions.) For a node w let $s_w = \sum_{u \in P_w \setminus \{w\}} k_w$. We associate k_w hash functions with w: $h_{w,1}, h_{w,2}, \ldots, h_{w,k_w}$, where $h_{w,j} : U \to [m]$ is defined by $h_{w,j}(x) = h_{s_w+j}(x) + \mathsf{off}(w) \mod m$.

The Bloom map B is an array of size m that is initially empty (all bits are zero). To store a key/value pair (x, v_i) we use the algorithm $\mathsf{Store}(x, v_i, T(\vec{p}), B)$ (see Figure 1). This does the following: for each node w in the path P_{v_i}, starting from the root, it evaluates the associated k_w hashes at x and sets the corresponding bits in B. Note that (ignoring offsets) the hash functions used while storing (x, v_i) are $h_1, h_2, \ldots, h_{t_i}$, where $t_i = \sum_{w \in P_{v_i}} k_w$. Hence the bits which are set in B by $\mathsf{Store}(x, v_i, T(\vec{p}), B)$ are chosen independently and uniformly at random. Moreover, since each key is stored with at most one value, the entire process of storing the $\vec{p}$-map in B is equivalent to setting $t = n \sum_{i=1}^{b} p_i t_i$ independently chosen random bits in B.

To query B with a key $x \in U$ we use the algorithm $\mathsf{Query}(x, T(\vec{p}), B)$. This calls $\mathsf{Findval}(x, v, T(\vec{p}), B)$ with v initialised to $\perp$ and returns the value of v when $\mathsf{Findval}(x, v, T(\vec{p}), B)$ terminates (see Figure 1). Starting with $T(\vec{p})$, $\mathsf{Findval}$ evaluates the hash functions associated with the root of the current tree, returning false if it finds a zero bit in B, otherwise it continues down the tree, first looking at the right subtree and then, if this fails, looking at the left subtree. If it reaches a leaf at which the corresponding bits in B are all set then v is assigned the value associated with this leaf and it returns true, otherwise the value of v will remain equal to $\perp$.

```
Store(x, v_i, T, B)
for d = 0 to l_{v_i}
  w ← P_{v_i} ∩ T_d
  for j = 1 to k_w
    B[h_{w,j}(x)] ← 1

Query(x, T, B)
v ← ⊥
Findval(x, v, T, B)
return v

Findval(x, v, T, B)
w ← r(T)
for j = 1 to k_w
  if B[h_{w,j}(x)] = 0 then return false
if w is a leaf then
  v ← val(w); return true
if Findval(x, v, T_R, B) then return true
return Findval(x, v, T_L, B)
```

Figure 1: Storing and querying keys in a Bloom map

By our choice of $m = t \log e$ the expected proportion of bits that remain zero in B (once we have stored the $\vec{p}$-map M) is $1/2$ and with high probability the actual proportion, which we denote by ρ, is very close to this. For simplicity we will assume that $\rho \geq 1/2$.

We now consider the probability of errors. To simplify our analysis we assume that any leaf v_i is at depth $\log 1/p_i$ (since $T(\vec{p})$ is an optimal alphabetic binary tree this is almost true). For $x \in U$ and $i \in [b]$ define

$$\mathcal{H}_i(x) = \{h_{w,l}(x) \mid w \in P_{v_i}, l \in [k_w]\}$$

and

$$\mathsf{qval}(x) = \left\{ i \in [b] \mid \wedge_{h \in \mathcal{H}_i(x)} B[h] = 1 \right\}.$$

Thus $i \in \mathsf{qval}(x)$ iff all of the bits in B indexed by the hash functions on the path P_{v_i} evaluated at x are set. If $\mathsf{qval}(x) = \emptyset$ then Query returns $\perp$, otherwise, since $\mathsf{Findval}$ always explores right subtrees first, it returns v_c, where $c = \max \mathsf{qval}(x)$. If $x \in X_i$ then $i \in \mathsf{qval}(x)$ and so no false negatives can occur. False positives and misassignments are possible, we consider the case of false positives first.

If $x \in U \setminus X$ then for fixed $i \in [b]$ the bits in $\mathcal{H}_i(x)$ are simply independent random choices from $[m]$. This is because if $t_i = \sum_{w \in P_{v_i}} k_w$ then the hash functions we evaluate are simply offsets, modulo m, of the first t_i

of our base hash functions. By our assumptions that: $k_w \geq 1$ for all nodes; $k_{v_i} \geq \log 1/\epsilon$ and v_i is at depth $\log 1/p_i$, we have $t_i \geq -\log \epsilon p_i$. Since $\rho \geq 1/2$ the false positive probability satisfies

$$\begin{aligned} f^+(B) &= \mathbf{Pr}\{\mathsf{qval}(x) \neq \emptyset\} \\ &\leq \sum_{i=1}^{b} \mathbf{Pr}\{i \in \mathsf{qval}(x)\} \\ &\leq \sum_{i=1}^{b} (1-\rho)^{t_i} \\ &\leq \sum_{i=1}^{b} \frac{1}{2^{t_i}} \leq \epsilon. \end{aligned}$$

Calculating the probability of a misassignment when B is queried with $x \in X_i$ is more involved. Note that if an incorrect value $v_j \neq v_i$ is returned for $x \in X_i$ then $j > i$. For $i < j$ and $x \in X_i$ let $P_{i,j} = P_{v_j} \backslash P_{v_i}$ be the part of the path P_{v_j} that is disjoint from the path P_{v_i} and let $\mathcal{H}_{i,j}(x) = \{h_{w,l}(x) \mid w \in P_{i,j}, l \in [k_w]\}$. The misassignment probability satisfies

$$\begin{aligned} f_i^*(B) &= \mathbf{Pr}\{\max \mathsf{qval}(x) > i\} \\ &\leq \sum_{j=i+1}^{b} \mathbf{Pr}\{j \in \mathsf{qval}(x)\} \\ &= \sum_{j=i+1}^{b} \mathbf{Pr}\left\{\wedge_{h \in \mathcal{H}_{i,j}(x)} B[h] = 1\right\}. \end{aligned} \tag{4.2}$$

To bound this probability we consider the following: suppose that rather than storing all of the key/value pairs from M in B we had instead stored all of them *except* (x, v_i). Let B' denote the resulting m-bit array. Let $t_{i,j} = |\mathcal{H}_{i,j}(x)|$. Since (x, v_i) has not been stored in B' we have (by the same argument as used for $f^+(B)$) that

$$\mathbf{Pr}\left\{\wedge_{h \in \mathcal{H}_{i,j}(x)} B'[h] = 1\right\} \leq \frac{1}{2^{t_{i,j}}}. \tag{4.3}$$

If all of the bits in B indexed by elements in $\mathcal{H}_{i,j}$ are set then either they are all set in B' or there must be at least one bit in $\mathcal{H}_{i,j}$ that is only set once (x, v_i) is stored. The later case can only occur if $\mathcal{H}_i \cap \mathcal{H}_{i,j} \neq \emptyset$. Hence

$$\begin{aligned} \mathbf{Pr}\left\{\wedge_{h \in \mathcal{H}_{i,j}(x)} B[h] = 1\right\} \leq \mathbf{Pr}\left\{\wedge_{h \in \mathcal{H}_{i,j}(x)} B'[h] = 1\right\} \\ + \mathbf{Pr}\{\mathcal{H}_i \cap \mathcal{H}_{i,j} \neq \emptyset\}. \end{aligned} \tag{4.4}$$

If $\hat{h}_1 \in \mathcal{H}_i(x)$ and $\hat{h}_2 \in \mathcal{H}_{i,j}(x)$ then $\mathbf{Pr}\{\hat{h}_1 = \hat{h}_2\}$ is either $1/m$ or 0, since $\hat{h}_1$ and $\hat{h}_2$ either use different base hash functions (and so are independent and random in $[m]$) or they use the same base hash function with different offsets and hence are distinct.

Recall that $k = \max_{i \in [b]} \sum_{w \in P_{v_i}} k_w$. If $c \in [b]$ satisfies $k = \sum_{w \in P_{v_c}} k_w$ then, by (4.1), $m \geq np_c k \log e$. Moreover the k_w were chosen so that $n \leq m \leq 2n \log e \log(b/\epsilon)$. Hence

$$\begin{aligned} \mathbf{Pr}\{\mathcal{H}_i \cap \mathcal{H}_{i,j} \neq \emptyset\} &\leq \frac{|\mathcal{H}_i| \cdot |\mathcal{H}_{i,j}|}{m} \\ &\leq \frac{k^2}{m} \\ &\leq \left(\frac{m}{np_c \log e}\right)^2 \frac{1}{m} \\ &\leq \left(\frac{2 \log b/\epsilon}{p_c}\right)^2 \frac{1}{n} = O\left(\frac{1}{n}\right), \end{aligned} \tag{4.5}$$

where the final equality uses our assumption that $\vec{p}$, b and ϵ are constant.

Combining (4.2), (4.3), (4.4) and (4.5) we obtain

$$f_i^*(B) \leq \sum_{j=i+1}^{b} \frac{1}{2^{t_{i,j}}} + O\left(\frac{1}{n}\right) \approx \sum_{j=i+1}^{b} \frac{1}{2^{t_{i,j}}}, \tag{4.6}$$

where $t_{i,j} = \sum_{w \in P_{i,j}} k_w$. Thus to ensure $f_i^*(B) \leq \epsilon$ we choose the k_w so that $\sum_{j=i+1}^{b} 2^{-t_{i,j}} \leq \epsilon$. There are various ways in which this can be done and exactly how we choose the k_w will effect not only $f_i^*(B)$ but also the memory required to store the Bloom map and the amount of work we expect to do when querying it. Since different space/time trade-offs may be of interest in different applications we define two special types of Bloom map: Standard and Fast.

- *(Standard)* $k_w = 1$ for all internal nodes (i.e. all non-leaf nodes), $k_{v_i} = \log 1/\epsilon + \log(H_b - 1) + 1$ for all leaves (where $H_b = \sum_{l=1}^{b} 1/l$ is the bth Harmonic number).
- *(Fast)* $k_w = 2$ for all internal nodes, $k_{v_i} = \log 1/\epsilon + 2$ for all leaves.

THEOREM 4.1. *The Standard and Fast Bloom maps are both $(\epsilon, \epsilon, 0)$-encoding for $\vec{p}$-maps. The average number of bits required per key is:*

- *(Standard):* $\log e(\log 1/\epsilon + H(\vec{p}) + \log(H_b - 1) + 1)$.
- *(Fast):* $\log e(\log 1/\epsilon + 2H(\vec{p}) + 2)$.

If $x \in U \backslash X$ then the expected number of bit probes performed during Query$(x, T(\vec{p}), B)$ *is at most: (Standard) $H(\vec{p}) + 2$; (Fast) 3.*

If $x \in X_i$ then the expected number of bit probes performed during Query$(x, T(\vec{p}), B)$ *is at most: (Standard)* $O((\log b)^2) + \log 1/p_i + \log 1/\epsilon$*; (Fast)* $3\log(b-i+1) + 2\log 1/p_i + \log 1/\epsilon + 2$.

The Standard Bloom map uses little more than a factor $(1-\epsilon)^{-1}\log e$ extra bits per key than the lower bound of Corollary 2.1. (In addition to the factor of $(1-\epsilon)^{-1}\log e$ it uses at most an extra $1 + \log\log b$ bits per key, since $H_b < \log b$.) The Fast Bloom map uses slightly more space but has the advantage of using significantly fewer bit probes when querying keys: in particular we expect to perform at most 3 bit probes on $x \in U\backslash X$. In any case the Fast Bloom map uses less than 2.9 times as much memory per key as the lower bound and if $H(\vec{p})$ is small compared to $\log 1/\epsilon$ this factor will be much closer to 1.46.

We note that other choices for the k_w are possible and depending on the application may be desirable.

Proof of Theorem 4.1. We first show that both Bloom maps are $(\epsilon, \epsilon, 0)$-encoding. We know already that $f^+(B) \le \epsilon$ so we consider $f_i^*(B)$. We require the following simple lemma.

LEMMA 4.1. *Let T be a full binary tree with leaves $v_1, v_2, \ldots, v_b$ at depths $l_1 \le l_2 \le \cdots \le l_b$.*

(a) If $1 \le i < j \le b$ then the number of nodes in $P_{v_j}\backslash P_{v_i}$ is at least $\log\left(\sum_{k=i}^{j} 2^{l_j - l_k}\right)$.

(b) If T_d is the set of nodes in T at depth d then

$$\sum_{d=0}^{l_b} \frac{|T_d|}{2^d} \le 1 + \sum_{i=1}^{b} \frac{l_i}{2^{l_i}}.$$

(c) The number of left branches on the path P_{v_i} is at most $\log(b-i+1)$.

Proof. First note that if T is a perfect binary tree (i.e. a full binary tree with all leaves at the same depth) then the number of nodes on $P_{i,j}$, (where $P_{i,j}$ is the part of the path from the root to v_j that is disjoint from the path to v_i), is at least $\log(j-i+1)$.

Now extend the tree T to a tree T' by replacing each leaf $v_k \in \{v_i, v_{i+1}, \ldots, v_{j-1}\}$ by a perfect binary tree of depth $l_j - l_k$. By our previous remark the number of nodes on $P_{i,j}$ is at least $\log s$, where s is the number of leaves lying strictly between v_{i-1} and v_{j+1} in T'. Since $s = \sum_{k=i}^{j} 2^{l_j - l_k}$ part (a) now follows.

For (b) note that if we define $l_0 = 0$ then

$$\begin{aligned}\sum_{d=0}^{l_b} \frac{|T_d|}{2^d} &\le 1 + \sum_{i=1}^{b} (l_i - l_{i-1})\left(1 - \sum_{j=1}^{i} \frac{1}{2^{l_j}}\right) \\ &= 1 + \sum_{i=1}^{b} \frac{l_i}{2^{l_i}}.\end{aligned}$$

For (c) note that if the path from the root to v_i has left branches at depths $d_1, d_2, \ldots, d_t$ then the number of leaves to the right of v_i is at least $\sum_{j=1}^{t} 2^{l_i - (d_j+1)}$ (this is because T is full). Since all of the depths of the left branches are distinct and at most $l_i - 1$, the number of leaves to the right of v_i is at least $\sum_{j=0}^{t-1} 2^j = 2^t - 1$. However the number of leaves to the right of v_i is $b - i$ and so $t \le \log(b-i+1)$. □

Lemma 4.1 (a), together with our assumption that v_k is at depth $\log 1/p_k$ in $T(\vec{p})$ and the fact that $p_1 \ge p_2 \ge \cdots \ge p_b$ implies that the number of internal nodes on $P_{i,j}$ is at least $\log\left(\sum_{j=i}^{k} p_k/p_j\right) - 1$. Let a be the common value of k_w for all internal nodes. By (4.6) we have

$$\begin{aligned}f_i^*(B) &\le \sum_{j=i+1}^{b} \frac{1}{2^{t_{i,j}}} \\ &\le \sum_{j=i+1}^{b} \left(\frac{p_j}{\sum_{k=i}^{j} p_k}\right)^a \frac{1}{2^{k_{v_j} - a}} \\ &\le \sum_{j=i+1}^{b} \frac{1}{(j-i+1)^a 2^{k_{v_j} - a}},\end{aligned}$$

where the last inequality follows from the fact that $p_j \le p_k$ for all $i \le k \le j$. In the case of the Standard Bloom map we have $a = 1$ and $k_{v_j} = \log 1/\epsilon + \log(H_b - 1) + 1$, hence $f_i^*(B) \le \epsilon$. For the Fast Bloom map $a = 2$, $k_{v_j} = \log 1/\epsilon + 2$ and $\sum_{k=1}^{\infty} 1/l^2 = \pi^2/6$ imply that

$$f_i^*(B) \le \sum_{l=2}^{b-i+1} \frac{\epsilon}{l^2} \le \epsilon\left(\frac{\pi^2}{6} - 1\right) < \epsilon.$$

Hence both Bloom maps are $(\epsilon, \epsilon, 0)$-encoding.

Now consider how much work we expect to do when querying B. We measure this in terms of the expected number of bit probes performed. (Note that as described each bit probe performed by Findval involves the evaluation of a hash function, this need not be the case. The use of offsets ensures that we never need to evaluate more than $k = \max_{i \in [b]} \sum_{i \in P_{v_i}} k_w$ base hash functions, different offsets can then be added as required.) We consider the cases $x \in X$, $x \in U\backslash X$ separately.

Let negbp denote the expected number of bit probes performed by $\mathsf{Query}(x, T(\vec{p}), B)$ for $x \in U \backslash X$. The easiest case is the Fast Bloom map, in which every internal node w has $k_w = 2$. Let negbp(T) be the expected number of bit probes performed by Findval in a tree T. We wish to find negbp = negbp$(T(\vec{p}))$. Starting from the root of $T(\vec{p})$ we have

$$\text{negbp}(T(\vec{p})) \leq 1 + \frac{1}{2} + \frac{1}{4}(\text{negbp}(T_L(\vec{p})) + \text{negbp}(T_R(\vec{p}))),$$

since if b_1, b_2 are the first two bit probes then $\mathbf{Pr}\{b_1 = 0\} = \rho \geq 1/2$ and $\mathbf{Pr}\{b_1 = b_2 = 1\} = (1-\rho)^2 \leq 1/4$. Iterating and using the fact that all nodes in $T(\vec{p})$ have at least two associated bit probes we find

$$\text{negbp} \leq \frac{3}{2} \sum_{j=0}^{\infty} \frac{1}{2^j} = 3.$$

In the Standard Bloom map $k_w = 1$ for every internal node, hence if w is at depth l_w then the probability that the bit probe associated with w is evaluated during $\mathsf{Query}(x, T(\vec{p}), B)$, is at most 2^{-l_w}. Moreover for a leaf v_i at depth $\log 1/p_i$ the probability that Findval performs more than one bit probe at v_i is at most $p_i/2$ and in this case we expect to perform at most two extra bit probes at the leaf. Hence if $T_d(\vec{p})$ is the set of nodes in $T(\vec{p})$ at depth d then the expected number of bit probes performed during $\mathsf{Query}(x, T(\vec{p}), B)$ is at most

$$\text{negbp} \leq 2 \sum_{i=1}^{b} \frac{p_i}{2} + \sum_{d=0}^{\infty} \frac{|T_d(\vec{p})|}{2^d} = 1 + \sum_{d=0}^{\infty} \frac{|T_d(\vec{p})|}{2^d}.$$

By Lemma 4.1 (b) this is at most $H(\vec{p}) + 2$.

Finally we calculate the expected number of bit probes performed by $\mathsf{Query}(x, T(\vec{p}), B)$, for $x \in X_i$, which we denote by posbp(i). This will be the number of bits set during $\mathsf{Store}(x, v_i, T(\vec{p}), B)$, plus the expected number of bit probes performed by Findval in the "false subtrees" it explores, where a false subtree is any maximal subtree disjoint from the path P_{v_i}. The number of false subtrees is simply the number of left branches in the path P_{v_i}, since at each such branch Findval first explores the right (false) subtree. By Lemma 4.1 (c) the number of false subtrees is at most $\log(b - i + 1)$. To simplify our analysis we will assume that the bit probes in false subtrees are independent and random. By a similar argument to that used during the calculation of the misassignment probability above this is essentially true.

For the Fast Bloom map we expect to perform at most three bit probes in each false subtree. Since the number of false subtrees in $T(\vec{p})$ is at most $\log(b-i+1)$ the expected number of bit probes performed in false subtrees is at most $3\log(b-i+1)$. Since the number of bits set by $\mathsf{Store}(x, v_i, T(\vec{p}), B)$ is $2\log 1/p_i + \log 1/\epsilon + 2$ we have

$$\text{posbp}(i) \leq 3\log(b-i+1) + 2\log 1/p_i + \log 1/\epsilon + 2.$$

Now consider the Standard Bloom map. Any false subtree is a full binary tree with $z \leq b - i$ leaves and hence corresponds to an optimal binary search tree for some probability distribution $q = (q_1, q_2, \ldots, q_z)$. Since $H(q) \leq \log z \leq \log(b-i)$ the expected number of bit probes performed in any false subtree is at most $\log(b-i)+2$. The number of bits set by $\mathsf{Store}(x, v_i, T(\vec{p}), B)$ is $\log 1/p_i + \log 1/\epsilon + \log(H_b - 1) + 1$. Hence

$$\text{posbp}(i) = O((\log b)^2) + \log 1/p_i + \log 1/\epsilon.$$

This completes the proof of Theorem 4.1. □

Appendix

Proof of Theorem 2.2. Let M be a fixed $\vec{p}$-map with key set X. Suppose that M is $(\epsilon^+, \epsilon^*, \epsilon^-)$-encoded by the m-bit string s. For $i \in [b]$ let $a_i^{(s)} = |\{x \in U \mid s(x) = v_i\}|$ and let $w_i^{(s)} = |\{x \in U \backslash X \mid s(x) = v_i\}|$. Define $q_i^{(s)} \geq 0$ by $w_i^{(s)} = \epsilon^+(u-n)q_i^{(s)}$. So $a_i^{(s)} \leq n + \epsilon^+(u-n)q_i^{(s)}$. Since $f^+(s) \leq \epsilon^+$ we have

$$\sum_{i=1}^{b} q_i^{(s)} \leq 1. \tag{4.7}$$

We now need to consider how many distinct $\vec{p}$-maps $N = \{(y_1, v(y_1)), (y_2, v(y_2)), \ldots, (y_n, v(y_n))\}$ can be $(\epsilon^+, \epsilon^*, \epsilon^-)$-encoded by the string s. Let Y be the key set of N and for $i \in [b]$ let $Y_i = \{y \in Y \mid v(y) = v_i\}$, so $|Y_i| = p_i n$. For $0 \leq j \leq b$ let $y_{i,j} = |\{y \in Y_i \mid s(y) = v_j\}|$.

Since $f_i^-(s) \leq \epsilon^-$, $f_i^*(s) \leq \epsilon^*$ and s returns a value from $V \cup \{\perp\}$ for each element in Y_i we have the following three constraints on the $y_{i,j}$

(4.8)

$$y_{i,0} \leq \epsilon^- p_i n, \quad \sum_{j \in [b] \backslash \{i\}} y_{i,j} \leq \epsilon^* p_i n, \quad \sum_{j=0}^{b} y_{i,j} = p_i n.$$

We can now bound the number of choices for the $y_{i,j}$. Since $\sum_{j=0}^{b} y_{i,j} = p_i n$, and each $y_{i,j} \geq 0$ the number of choices for the $y_{i,j}$ is at most $\binom{p_i n + b}{b}$ (the number of ways of choosing $b+1$ non-negative integers whose sum is $p_i n$).

For a particular choice of the $y_{i,j}$ the number of choices for the keys in Y_i is at most

$$\binom{u}{y_{i,0}} \prod_{j=1}^{b} \binom{n + \epsilon^+(u-n)q_j^{(s)}}{y_{i,j}}. \tag{4.9}$$

(This is because any particular choice for the keys in Y_i is given by choosing $y_{i,0}$ keys on which s returns $\perp$ and then choosing $y_{i,j}$ keys on which s returns v_j, for each $j \in [b]$.)

Let $y'_{i,0}, y'_{i,1}, \ldots y'_{i,b}$ be chosen to maximise (4.9) subject to (4.8). The number of choices for the keys in Y_i is at most

$$\binom{p_i n + b}{b} \binom{u}{y'_{i,0}} \prod_{j=1}^{b} \binom{n + \epsilon^+(u-n)q_j^{(s)}}{y'_{i,j}}.$$

Hence the total number of $\vec{p}$-maps which can be $(\epsilon^+, \epsilon^*, \epsilon^-)$-encoded by the string s is at most

$$\prod_{i=1}^{b} \left(\binom{p_i n + b}{b} \binom{u}{y'_{i,0}} \prod_{j=1}^{b} \binom{n + \epsilon^+(u-n)q_j^{(s)}}{y'_{i,j}} \right).$$

Letting $q_1, q_2, \ldots, q_b \geq 0$ be chosen to maximise this expression subject to $\sum_{j=1}^{b} q_j \leq 1$ we require

$$2^m \prod_{i=1}^{b} \left(\binom{p_i n + b}{b} \binom{u}{y'_{i,0}} \prod_{j=1}^{b} \binom{n + \epsilon^+(u-n)q_j}{y'_{i,j}} \right) \geq \binom{u}{n} \binom{n}{p_1 n, \ldots, p_b n}.$$

Using $\frac{(a-b)^b}{b!} \leq \binom{a}{b} \leq \frac{a^b}{b!} \leq a^b$ we require

$$2^m \prod_{i=1}^{b} \left((p_i n + b)^b \binom{p_i n}{y'_{i,0}, y'_{i,1}, \ldots, y'_{i,b}} u^{y'_{i,0}} \times \prod_{j=1}^{b} (\epsilon^+ q_j u)^{y'_{i,j}} \left(1 + \frac{n(1-\epsilon^+ q_j)}{\epsilon^+ q_j u}\right)^{y'_{i,j}} \right) \geq u^n \left(1 - \frac{n}{u}\right)^n.$$

Taking logarithms; using Stirling's formula to approximate the multinomial coefficient and using

$$\sum_{i=1}^{b} \sum_{j=0}^{b} y'_{i,j} = \sum_{i=1}^{b} p_i n = n$$

we obtain

$$m \geq -\sum_{i=1}^{b} b \log(p_i n + b) - \sum_{i=1}^{b} p_i n H\left(\frac{y'_{i,0}}{p_i n}, \ldots, \frac{y'_{i,b}}{p_i n}\right) + O(\log n) + \sum_{i=1}^{b} \sum_{j=1}^{b} y'_{i,j} \log\left(\frac{1}{\epsilon^+ q_j}\right) - \sum_{i=1}^{b} \sum_{j=1}^{b} y'_{i,j} \log\left(1 + \frac{n(1-\epsilon^+ q_j)}{\epsilon^+ q_j u}\right) + n \log\left(1 - \frac{n}{u}\right).$$

Defining $r_{i,j} = y'_{i,j}/p_i n$; noting that the first sum in the previous inequality is $O(\log n)$ and using $\log(1+\alpha) = O(\alpha)$ for α small we obtain

$$m \geq n \sum_{i=1}^{b} p_i \left(\sum_{j=1}^{b} r_{i,j} \log(r_{i,j}/\epsilon^+ q_j) + r_{i,0} \log r_{i,0} \right) + O(\log n) + O\left(\frac{n^2}{u}\right).$$

Dividing by n and using $u \gg n$ we find that the average number of bits required per key is at least

$$\frac{m}{n} \geq \sum_{i=1}^{b} p_i (1 - r_{i,0}) \log 1/\epsilon^+ + \sum_{i=1}^{b} p_i \left(r_{i,0} \log r_{i,0} + r_{i,i} \log r_{i,i} + r_{i,i} \log 1/q_i\right) + \sum_{i=1}^{b} \sum_{j \in [b] \setminus \{i\}} p_i r_{i,j} \log r_{i,j} + \sum_{i=1}^{b} \sum_{j \in [b] \setminus \{i\}} p_i r_{i,j} \log 1/q_j + o(1). \tag{4.10}$$

Defining $t_{i,j} = r_{i,j}/(1 - r_{i,0} - r_{i,i})$ and $\vec{t_i} = (t_{i,1}, \ldots, t_{i,i-1}, t_{i,i+1}, \ldots, t_{i,b})$ we have

$$\sum_{i=1}^{b} \sum_{j \in [b] \setminus \{i\}} p_i r_{i,j} \log r_{i,j} = \sum_{i=1}^{b} p_i (1 - r_{i,0} - r_{i,i})(\log(1 - r_{i,0} - r_{i,i}) - H(\vec{t_i})). \tag{4.11}$$

Defining $u_{i,j} = q_j/(1-q_i)$ and applying Gibbs' inequality we obtain

$$\sum_{i=1}^{b} \sum_{j \in [b] \setminus \{i\}} p_i r_{i,j} \log 1/q_j = \sum_{i=1}^{b} p_i (1 - r_{i,0} - r_{i,i}) \left(\log \frac{1}{1-q_i} + \sum_{j \in [b] \setminus \{i\}} t_{i,j} \log 1/u_{i,j} \right) \geq \sum_{i=1}^{b} p_i (1 - r_{i,0} - r_{i,i}) \left(\log 1/(1-q_i) + H(\vec{t_i}) \right) \tag{4.12}$$

Substituting (4.11) and (4.12) into (4.10) yields

$$\begin{aligned}\frac{m}{n} \geq & \sum_{i=1}^{b} p_i(1-r_{i,0}) \log 1/\epsilon^+ \\ & + \sum_{i=1}^{b} p_i \left(r_{i,0} \log r_{i,0} + r_{i,i} \log r_{i,i} + r_{i,i} \log 1/q_i\right) \\ + \sum_{i=1}^{b} & p_i(1-r_{i,0}-r_{i,i}) \left(\log(1-r_{i,0}-r_{i,i}) + \log 1/(1-q_i)\right) \\ & + o(1).\end{aligned}$$

Defining $r_i^* = \sum_{j \in [b] \setminus \{i\}} r_{i,j}$ we have (by (4.8)) that $r_i^* \leq \epsilon^*$. We also have $r_{i,0} \leq \epsilon^-$ and so $r_{i,i} = 1 - r_{i,0} - r_i^* \geq 1 - \epsilon^- - \epsilon^*$. Hence

$$\begin{aligned}\frac{m}{n} \geq & (1-\epsilon^-) \log 1/\epsilon^+ - H(\epsilon^-, \epsilon^*, 1-\epsilon^- - \epsilon^*) \\ & + \sum_{i=1}^{b} p_i \left(r_{i,i} \log 1/q_i + r_i^* \log 1/(1-q_i)\right) + o(1).\end{aligned}$$

Thus

$$\begin{aligned}\frac{m}{n} \geq & (1-\epsilon^-) \log 1/\epsilon^+ - H(\epsilon^-, \epsilon^*, 1-\epsilon^- - \epsilon^*) \\ & + (1-\epsilon^- - \epsilon^*) \sum_{i=1}^{b} p_i \log 1/q_i \\ & + \sum_{i=1}^{b} p_i r_i^* \log 1/(1-q_i) + o(1).\end{aligned} \tag{4.13}$$

Finally applying Gibbs' inequality and noting that the last summation in (4.13) is non-negative yields our desired lower bound on the average number of bits required per key

$$\begin{aligned}\frac{m}{n} \geq & (1-\epsilon^-) \log 1/\epsilon^+ + (1-\epsilon^- - \epsilon^*) H(\vec{p}) \\ & - H(\epsilon^-, \epsilon^*, 1-\epsilon^- - \epsilon^*) + o(1).\end{aligned}$$

□

References

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[2] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 59–65, New York, NY, USA, 1978. ACM Press.

[3] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier filter: an efficient data structure for static support lookup tables. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39, 2004.

[4] Donald E. Knuth. *Volume 3 The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, 1998.

[5] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech. A family of perfect hashing methods. *British Computer Journal*, 39(6):547–554, 1996.

[6] Michael Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.

[7] David Talbot and Miles Osborne. Randomised Language Modelling for Statistical Machine Translation. *ACL'07: Proc. of 45th Annual Meeting of ACL*, pages 512–519, Prague, 2007.

Augmented Graph Models for Small-World Analysis with Geographical Factors

Van Nguyen* Chip Martel†

Abstract

Small-world properties, such as small-diameter and clustering, and the power-law property are widely recognized as common features of large-scale real-world networks. Recent studies also notice two important geographical factors which play a significant role, particularly in Internet related setting. These two are the *distance-bias tendency* (links tend to connect to closer nodes) and the property of *bounded growth* in localities. However, existing formal models for real-world complex networks usually don't fully consider these geographical factors.

We describe a flexible approach using a standard augmented graph model (e.g. Watt and Strogatz's [33], and Kleinberg's [20] models) and present important initial results on a refined model where we focus on the small-diameter characteristic and the above two geographical factors. We start with a general model where an arbitrary initial node-weighted graph H is augmented with additional random links specified by a generic 'distribution rule' τ and the weights of nodes in H. We consider a refined setting where the initial graph H is associated with a growth-bounded metric, and τ has a distance-bias characteristic, specified by parameters as follows. The base graph H has neighborhood growth bounded from both below and above, specified by parameters $\beta_1, \beta_2 > 0$. (These parameters can be thought of as the dimension of the graph, e.g. $\beta_1 = 2$ and $\beta_2 = 3$ for a graph modeling a setting with nodes in both 2D and 3D settings.) That is $2^{\beta_1} \leq \frac{N_u(2r)}{N_u(r)} \leq 2^{\beta_2}$ where $N_u(r)$ is the number of nodes v within metric distance r from u: $d(u,v) \leq r$. When we add random links using distribution τ, this distribution is specified by parameter $\alpha > 0$ such that the probability that a link from u goes to $v \neq u$ is $\propto \frac{1}{d^\alpha(u,v)}$. We show which parameters produce a small-diameter graph and how the diameter changes *depending on the relationship between the distance-bias parameter α and the two bounded growth parameters $\beta_1, \beta_2 > 0$*. In particular, for most connected base graphs, the diameter of our augmented graph is logarithmic if $\alpha \leq \beta_1$, and poly-log if $\beta_2 \leq \alpha < 2\beta_1$, but polynomial if $\alpha > 2\beta_2$. These results also suggest promising implications for applications in designing routing algorithms.

1 Introduction.

A large inter-disciplinary community is now studying characteristics of real-world complex networks. In 1998, Watts and Strogatz [33] produced a graph model for small-world networks, with *small diameter* and *high clustering* coefficient (two nodes with a shared neighbor are likely connected by a link). These small-world properties occur in large-scale networks such as social and biological networks as well as the Internet. The *power-law* feature of vertex degrees has also been widely recognized as a common property. For example, the distribution of vertex degrees in the Internet topology (at both inter-domain & router levels) has a power-law shape [12].

Geographical factors play a significant role in many real world networks, particularly in Internet related settings and for disease spread. Faloutsos et al. [12] observed in the Internet topology that a 'ball' of neighbors within distance R has size approximated by R^β when R is small enough. Note that β varies between locations due to different population density physical characteristics or economic conditions. Researchers also found similar observations in wireless networks and peer-to-peer networks; so, several papers consider *bounded growth* features or similar notions when studying Internet-related problems, e.g. [31, 9, 19, 17, 1]. Recent studies of the Internet's topology also evaluate the role of *distance-bias* where links are more likely to connect closer nodes [34, 24]. [1]

Distance-bias is a key factor in Kleinberg's small-world model[20] that extended Watts and Strogatz's model to explain another important aspect of small-worlds: a simple greedy strategy, that moves to the neighbor which is closest to the destination, finds short paths (reflecting Milgram's mail-forwarding experiment[28]). Kleinberg augments a k-dimensional

*CNRS and University of Paris Diderot - Paris 7, France. E-mail: vanknguyen@gmail.com

†Computer Science Dept., UC-Davis, USA. E-mail: martel@cs.ucdavis.edu. Supported by NSF grant 0520190.

[1] The authors in [34] make a controversial claim of an approximate linearity of distance bias in the link distribution. However, it seems hard to accurately describe this distance-bias distribution.

grid graph with random links, *where a link is more likely to connect closer nodes*. Specifically, each node u has one random link which goes to a node v with probability inversely proportional to $d^{\alpha}(u,v)$, where $d(u,v)$ is the lattice distance between u and v – for brevity, we write $Pr[u \rightarrow v] \propto d^{-\alpha}(u,v)$. Kleinberg shows that his greedy strategy finds short routes if and only if $\alpha = k$, the grid dimension. More recently, Liben-Nowell et al. [26] provided a further insight to how greedy geographical routing in social networks can be efficient. This study of a large online social network (using web-logs) shows that one-third of the friendships are independent of geography, but the other two-thirds exhibit a distance-bias.

However, existing models for small-world properties and power-law degrees don't fully consider geographical factors. Kleinberg's original model is among a few that consider distance-bias, but his grid graphs have a uniform distribution of node degrees and node density (with fixed growth rate $\beta = k$ everywhere), so they are too simple to simulate localities in real-world networks, where skewed distributions are typical. Thus, we want a more general approach to simultaneously simulate several of these mentioned properties.

In this paper, we use this standard augmented graph model that adds random links to a fixed base graph. However we use a more general approach with arbitrary base graphs, and where the number of random links added to a node can vary between nodes. We then refine this general setting to reflect two geographical factors: the *distance-bias tendency* (links tend to favor closer distances) and the property of *bounded growth* in neighborhood expansion and determine parameters of these geographical factors that lead to small diameter graphs (poly-log in the number of nodes).

Specifically, we use an augmented graph specified by a pair (H, τ) where *the base graph* $H = (V, w, E)$ is a graph of n nodes with weights specified by a vector $w = \{w_u, u \in V\}$, initial links in E, and τ is a collection of probability distributions $\{\tau_u, u \in V\}$. For each node $u \in V$ we add w_u random links to E, and each random link goes to a node $v \neq u$ with a probability specified by τ_u: $Pr[u \rightarrow v] = \tau_u(v)$. This model is broad enough to simulate several common features. For example, to simulate a specific power-law degree distribution we simply assign to H a proper weight vector w that reflects this power-law function. We can also use harmonic distributions (for τ), as Kleinberg did, to simulate distance-bias links. Modeling base graphs in an accurate flexible way is, however, challenging: we want a general and versatile approach so our base graphs can have arbitrary degrees and various bounded-growth rates, while still being tractable for diameter analysis and routing algorithms.

Our augmented graph model is an extension of Kleinberg's grid setting that allows non-uniform neighborhoods and distance-bias link distributions. There have been a number of other recent papers with extensions of Kleinberg's original model (see a review in §1.1 below). However we ask different questions and follow a different direction, compared to these papers, which mainly focus on augmenting graphs into navigable small-worlds (where greedy routing can find short paths).

Our main results. We refine the augmented graph model so base graphs are associated with a growth-bounded metric, $d(u,v)$, giving the distance between node-pairs. The base graph is then augmented with distance-bias random links. We use 3 parameters to characterize the model. The base graph has neighborhood growth bounded both below and above specified by parameters $\beta_1, \beta_2 > 0$: $2^{\beta_1} \leq \frac{|N_u(2r)|}{|N_u(r)|} \leq 2^{\beta_2}$ where $N_u(r)$ is the set of nodes within distance r from u. Also, the distribution of random links is specified by parameter $\alpha > 0$: $Pr[u \rightarrow v] \propto d^{-\alpha}(u,v)$. For example, a graph representing sensors in both a 2D setting (e.g. a field) and a 3D setting (a building) would have $\beta_1 = 2$ and $\beta_2 = 3$.

We show when these augmented graphs have small diameter and how the diameter changes *depending on the relationship between the distance-bias parameter and the two bounded growth parameters*. For a connected base graph, the diameter of our augmented graph is logarithmic in n if $\alpha \leq \beta_1$, and poly-log if $\beta_2 \leq \alpha < 2\beta_1$, but polynomial if $\alpha > 2\beta_2$. Table 1 summarizes our results compared with other similar work. Our results also help explain why the Internet graph can be seen as a small-world with low diameter although it is locally growth bounded (assuming the bounded-growth and distance-bias characteristics observed in [12, 34]).

We obtain the logarithmic diameter result for the case $\alpha \leq \beta_1$ by showing, a more general result: we present a general sufficient condition for the general (H, τ) setting so that with high probability (w.h.p.) there emerges a giant component of size $n(1-o(1))$ with a logarithmic diameter.

Our work defines and analyzes more complex models which can simultaneously reflect many features in real-world complex networks. However, to be more practical in future work we need to also consider distance-bias where α can vary in different neighborhoods.

We also use new techniques to extend diameter results in the Erdös and Rènyi's random graph setting (where *arcs* are added with a fixed probability) to our setting where random arcs are added to *nodes*. Our constructions also yield settings with efficient hierarchi-

Existing settings with a distance-bias feature	Parameters	Graph Diameter
A. A typical case of our refined setting:	$\alpha \in [0, \beta_1]$	$O(\log n)$
Connected, growth bounded graphs ($\beta_1 \leq \log \frac{\lvert N_u(2r)\rvert}{\lvert N_u(r)\rvert} \leq \beta_2$)	$\alpha \in [\beta_2, 2\beta_1)$	poly-log
augmented by undirected random links: one for each node u	$\alpha \in (2\beta_2, \infty)$	polynomial
which goes to another v with $Pr[u \to v] \propto d^{-\alpha}(u, v)$	$\alpha \in (\beta_1, \beta_2) \cup [2\beta_1, 2\beta_2]$	open work
B. Kleinberg's small-world model, a special case of A:	$\alpha \in [0, \beta)$	$O(\log n)$ [27]
β-dimension grids as base graphs ($\beta_1 = \beta_2 = \beta$), augmented	$\alpha = \beta$	$O(\log n)$ [27]
by the above distance-bias random links	$\alpha \in (\beta, 2\beta)$	poly-log [29]
(one per node, α-harmonic)	$\alpha \in (2\beta, \infty)$	polynomial [29]
C. Long-Range Percolation Graphs:	$\alpha < \beta$	O(1) [3, 8]
(comparable to Kleinberg's grid setting	$\alpha = \beta$	$\theta(\frac{\log n}{\log \log n})$ [8]
(see §1.1)	$\alpha \in (\beta, 2\beta)$	poly-log [3, 8]
	$\alpha \in (2\beta, \infty)$	polynomial [3, 8]

Table 1: Our refined setting (A) compared with similar settings and results (B and C). While B uses a node-based distribution similar to ours, C and also the Chung-Lu model (see §1.1) use pair-based.

cal routing schemes. Our diameter results present the limits where routing algorithms (with limited network resource) can be improved towards. We also discuss application issues for routing algorithms (§3.1 and §5).

1.1 More related work. Kleinberg's small-world model [20] started the active study of *navigable small-worlds* where decentralized routing using restricted local resource only, e.g. by a greedy strategy, can still find short paths (see [22] for a good survey). Followup work such as [2, 25, 15, 27] provided further insight to Kleinberg's original model and improved certain aspects on routing. More recent work such as [10, 32, 13, 16, 11, 23] mainly focused on when and how certain fundamental graphs can be augmented into a navigable small-world. [2] However, although our paper also uses an augmented graph model, we don't focus on modeling navigability, but instead focus on small diameter graphs and routing with more information. Similarly to Duchon et al. [10], we also consider the base graph H as bounded growth graph, however, we focus on using distance-bias random links. While navigable small-worlds can find useful applications in peer-to-peer networks, our graph models reflect the physical aspects of the Internet-related networks.

Liben-Nowell et al. [26] and Kumar et al. [23] extended Kleinberg's original model to better analyze navigability and greedy routing performance in real-world social networks. They proposed to use rank-based friendship for connecting a set of people who reside on a given set of nodes on a metric space where population densities at nodes are not necessarily uniform. Particularly, the probability that a person u befriends a person v is inversely proportional to the rank of a person v w.r.t. u, which is the number of people who lives closer to u than v. Although it has a distance-bias flavor, when applied to the k-dimensional original Kleinberg's grid setting as a special case (with uniform population density), the rank-based friendship induces the special distance-bias distribution $Pr[u \to v] \propto d^{-k}(u, v)$, where the distance-bias degree α equals the grid dimension k. We instead look at arbitrary α, which in general does not produce navigability except for the special $\alpha = k$. We show when and how (how α coordinates with bounded growth parameters in H) our augmented graphs feature small diameter. We also consider other efficient routing schemes, which are still decentralized, though they use more resource than Kleinberg's greedy routing.

Closest to our diameter results for Kleinberg's setting, is the diameter work on long-range percolation graphs (row C of table 1), which is a random graph setting where the vertices are the nodes of a β-dimension grid and each pair of nodes (u, v) has an undirected edge with probability $\lambda d^{-\alpha}(u, v)$, where α, β, λ are constant parameter. None of these papers analyze growth-bounded settings. This paper can be seen as a generalization for growth-bounded graphs of our previous results with grid-like settings in [29], however we consider a significantly broader landscape and hence, have developed significantly different techniques.

Chung and Lu propose a generalized random graph model [6, 7] with arbitrary given expected degrees.

[2] The aim is to consider a general *small-world model using augmented graphs* [16], and study the possibility of augmenting an arbitrary graph H into a navigable small-world by a properly selected τ (with one link added for each node). See [14] for a recent survey.

They assign weights to the nodes and have an independent random link between any two nodes with a probability proportional to the product of the two node weights (and do not use any distance factor). Thus, the expected degree of each node is specified by its weight. We also use node weights, to specify the number of random links added to a given node, but we use a node-based approach instead.

2 Definitions and building blocks.

This section present initial results on the general model (H, τ) that serve as a basis for obtaining our main results in §3-§4. They are also of interest on their own for the general problem of analyzing the diameter of a random graph. We then introduce in §2.3 our refined model for bounded-growth and distance-bias, and *the main ideas to analyze that model.*

DEFINITION 2.1. *Let $H(V, w, E)$ be an undirected base graph with initial edges $E(H)$ (called local links) and n nodes with weights specified by a vector $w = \{w_u, u \in V\}$ and τ be a collection of probability distributions associated with vertices of H: $\tau = \{\tau_u, u \in V\}$. Given a pair (H, τ) the node-based augmented networks $\mathcal{NAN}(H, \tau)$ are formed from H as follows: add w_u random links to each node $u \in V(H)$, where each such link goes to a node $v \neq u$ with probability $\tau_u(v)$.*

We also write $G = \mathcal{NAN}(H, \tau)$ for G to denote a random graph from the setting $\mathcal{NAN}(H, \tau)$. We only consider *undirected random links* here, but our results in §3 can be extended for directed random links. Intuitively, each node v *generates* certain random links, while some other random links also incident to v are not generated by v but some other nodes. The *weight* of a vertex set $S \subseteq V(H)$ is $\sum_{u \in S} w_u$. For $k > 0$, H is *k-heavy* if all the nodes in H have weight at least k.

EXAMPLE 2.1. *(Kleinberg's grid setting)*
The undirected version of Kleinberg's grid setting ([20]) is a $\mathcal{NAN}(H, \tau)$ setting where H is an $n \times n$ grid with all the nodes having weight 1, and τ describes the inverse square rule, i.e. $\tau_u(v) \propto d^{-2}(u, v)$, where $d(u, v)$ is the lattice distance between u and v.

When working with asymptotic notions we informally say that the size n of H goes to infinity (appendix §A more formally describes our settings: we use an infinite family of base graphs $\mathcal{H}$ with sizes increasing to infinity). We also give in §A an example for modeling *power-law graphs.*

A random variable X is *stochastically greater than* ($\geq_{st}$) a random variable Y, or Y is stochastically less than ($\leq_{st}$) X if $\Pr[X < a] \leq \Pr[Y < a]$ for all $a > 0$. We use $diam(X)$ to denote the diameter of a graph X. We say a random event $E(n)$ occurs with *very high probability* VHP when $\Pr[E(n)] = 1 - O(e^{-n})$.

For our proofs we use the following formal notions of events occurring with very high probability. Let $eNeg(x)$ denote the class of functions which are (exponentially) *negligible* in the following sense. We say a positive function $f(x)$ is $eNeg(x)$ if $\exists c > 0, \hat{c} > 0, \forall x > 0 : 0 \leq f(x) < \hat{c}e^{-cx}$. We write $f(x) = eNeg(x)$ to mean that $f(x)$ belongs in $eNeg(x)$. Given $y = h(x)$, we say $f(x)$ is $eNeg(y)$ or $eNeG(h(x))$ if $\exists c > 0, \hat{c} > 0, \forall x > 0 : 0 \leq f(x) < \hat{c}e^{-ch(x)}$. We also write $f(x) = eNeg(h(x))$ to mean that $f(x)$ belongs in $eNeg(h(x))$. For a constant $a > 0$, it is easy to see that for any constant $k > 0$ if $f(x) = eNeg(x^a)$ then so is $x^k f(x)$.

Given a function $y = h(x))$, we say a random event $E(n)$ occurs with *very high probability* $VHP(h(n))$ when $n \to \infty$ if $1 - \Pr[E(n)] = eNeg(h(n))$, which we may also write as $\Pr[E(n)] = 1 - eNeg(h(n))$.

Lemma 2.1 below is useful and straightforward to prove using Chernoff's bound (see A.1).

LEMMA 2.1. *For any constant $\alpha > 1$ and any positive constant $\beta < 1$, the sum of n identical Bernoulli random variables of expectation $p = p(n)$ is at most αnp with VHP, and is at least βnp with $VHP(np)$.*

2.1 On uniform τ: comparing diameter with traditional random graphs.

We consider the diameter of $\mathcal{NAN}(H, \tau)$ in a simple specific case when τ is uniform: a random link from a node u chooses the target v uniformly from $V - \{u\}$. This can be re-defined as the random graph setting below, which is a variant of ER (Erdös and Rènyi's) random graph $\mathbb{G}(n, p)$.[3]

Random setting $\mathbb{J}$. A random graph $J(n, Z)$ from setting $\mathbb{J}$, is an n node random graph whose edges are formed by adding $Z = Z(n)$ random (undirected) links to each node where a link from a node $u \in V$ is *equally likely* to go to each node $v \neq u \in V$. We also write $J = \mathbb{J}(n, Z)$.

So, $\mathbb{J}$ is $\mathcal{NAN}(H, \tau = uniform)$ where the base graph H has n nodes, no edges and all the nodes have the same weight $Z = Z(n) > 0$. Graphs from $\mathbb{J}(n, Z)$ are similar to those from $\mathbb{G}(n, p)$ with a suitable choice of p (roughly $Z/2n$), but the distributions of edges are slightly different, and there are some dependencies between edges for $\mathbb{J}(n, Z)$. Note that p can be a function of n in both settings. We can re-use some of the classic work on the diameter of ER random graphs in

[3] $\mathbb{G}(n, p)$ is an n-node random graph where each of the possible edges exists with probability p. Graphs can be directed or undirected, but we focus on undirected graphs.

Bollobas's classic text [5]. Lemma 2.2 below follows from results in [5].

LEMMA 2.2. *For the ER random graph $K = \mathbb{G}(n,p)$, suppose that there is a fixed integer d such that $(np)^d/n \to 0$ and $(np)^{d+1}/(n\log n) \to \infty$. Then K has diameter at most $d+4$ with $VHP(np)$.*
(Bollobas [5] showed the diameter concentrates on d and $d+1$ for d a constant or a slow function of n.)

The above applies for example when $np = n^{1/d}/\log n$. We now extend lemma 2.2 to $\mathbb{J}(n,Z)$:

THEOREM 2.1. *Consider a random graph J=$J(n,Z)$ where $Z = Z(n)$ such that there is a fixed integer d: when $n \to \infty$, $Z^d/n \to 0$ and $Z^{d+1}/(n\log n) \to \infty$. Then J has diameter at most $d+4$ with $VHP(Z)$.*

To prove theorem 2.1, the general idea is to compare our graph J to an ER random graph $K = \mathbb{G}(n,p)$ with the same number of vertices n and appropriate link probability, $p = .5Z(n)/n$. We choose this p so K is usually less dense than J, so that roughly $diam(K) \geq_{st} diam(J)$, yet $diam(K) \leq d+4$ with $VHP(Z)$: since $np = .5Z$, the conditions of lemma 2.2 apply, so $diam(K) \leq d+4$ with $VHP(Z)$. [4]

We use a series of comparisons of graph settings, using two intermediate settings as transitions between J and K. (Roughly speaking, we show that
$diam(K) \geq_{st} diam(L) \geq_{st} diam(I) \geq_{st} diam(J)$
where L and I are the mentioned intermediate settings.)

For any two arbitrary random graph settings X and Y using the same number of vertices, we write $X \preceq Y$ if there is an algorithm for generating Y in an *increasing manner*, i.e. non-loop edges are added until at a proper middle point of the process, the current graph is an instance of X. The following observation is straightforward.

Observation A. For any two arbitrary random graph settings X and Y using the same number of vertices, if $X \preceq Y$ then $diam(X)$ is stochastically greater than $diam(Y)$.

Intuitively, we think of a code M to generate Y *increasingly* where M has two parts $M = M_1||M_2$ (i.e. we run M_1 then M_2), and when M_1 finishes we obtain an instance of X. Note that the notion of *increasing* only applies for non-loop links: the codes may remove loops which clearly do not affect the graph diameter. We write $gen(M)$ for the random graph generated by executing M. Thus observation A can use this form: $diam(X) \geq_{st} diam(Y)$ if there exists an increasing M_1, M_2 such that $X = gen(M_1)$ and $Y = gen(M_1||M_2)$.

Proof. [of theorem 2.1] We compare $diam(J)$ with $diam(K)$ where $K = \mathbb{G}(n,p)$ and $p = .5\frac{Z}{n}$. The basic idea is to use observation A in a series of comparisons of graph settings. First, we define two intermediate settings as follows.

Random setting $\mathbb{I}$. A random graph $I = \mathbb{I}(n,m)$ from setting $\mathbb{I}$ is a graph on a set V of n nodes and from each node $u \in V$ we randomly choose a set of $m < n$ distinct nodes $v_1, v_2 \ldots, v_m$ from $V - \{u\}$ then create m undirected links: one link from u to each v_i. Observe that, in contrast to $\mathbb{J}$, we do not allow parallel links (two links from u to the same node).

Random setting $\mathbb{L}$. A random graph $L = \mathbb{L}(n,p)$ from setting $\mathbb{L}$ is a graph on a set V of n nodes and from each node $u \in V$ we create an undirected link to each node $v \in V - \{u\}$ with probability p. Note that in $\mathbb{G}(n,p)$ for each pair of nodes u and v we create a random link between them with probability p, while in $\mathbb{L}(n,p)$, it is a "double consideration" from both u's side and v's side and so, we end up with a double link between u and v with probability p^2. Thus, it is rather obvious that $K \preceq L$.

To compare $diam(J)$ with $diam(K)$ where $K = \mathbb{G}(n,p)$ and $p = .5\frac{Z}{n}$, we choose two intermediate graphs $I = \mathbb{I}(n,.9Z)$ and $L = \mathbb{L}(n,p)$. Our overall plan is to show that $diam(K) \leq diam(L) \leq diam(I) \leq diam(J)$. However as mentioned above, $K \preceq L$ and hence, the remaining job is to compare L with I and I with J.

We now show that $L \preceq I$ with VHP. For M_1 to generate L, each node u generates an undirected link to any node $v \neq u$ with probability p. Let R_u denote the number of random links generated by a node u. We can now add additional random edges to L using a process M_2 to get an instance of $I = \mathbb{I}(n,.9Z)$ as long as $R_u \leq .9Z$ for each node u. In M_2, for each node u where $R_u < .9Z$, we added $(.9Z - R_u)$ distinct random links from u[5]. Let E denote the event that $R_u \leq .9Z$ for all u. For a given node u, R_u is the sum of $n-1$ Bernoulli random variables with expectation $p = .5\frac{Z}{n}$, and hence by lemma 2.1, with $VHP(n)$, R_u is at most $1.8p(n-1) < .9Z$. So, $\Pr[R_u > .9Z] = eNeg(n)$. But $\Pr[\neg E] \leq \sum_{u\in V} \Pr[R_u > .9Z] = n\Pr[R_u > .9Z]$ which is also $eNeg(n)$. Hence, E also occurs with $VHP(n)$ and so, with $VHP(n)$ the code $M_1||M_2$ generates an instance of I. Thus, for any constant $C > 0$, $\Pr[diam(L) \leq C] \leq \Pr[diam(I) \leq C] + \Pr[\neg E] \leq \Pr[diam(I) \leq C] + eNeg(n)$.

Similarly, we almost have $I \preceq J$. This can be proved by showing that in graph setting J, with $VHP(Z)$ a given node u generates at least $.9Z$ distinct random

[4] More exactly, we show that $\Pr[diam(J) \leq d+4]$ is at least $\Pr[diam(K) \leq d+4] - eNeg(Z)$ while $\Pr[diam(K) \leq d+4] = 1 - eNeg(Z)$ and hence, $\Pr[diam(J) \leq d+4] = 1 - eNeg(Z)$.

[5] Randomly choose $(.9Z - R_u)$ distinct nodes from $V - V'$ where V' is the set of nodes that the links u generated in M_1 go to, then create a link between u and each of these.

links from Z attempts. For any $1 \leq k \leq Z$, consider the $(k+1)^{th}$ attempt. The probability that u creates a new random link (to a node which is not picked by the first k attempts) is at least $\frac{n-1-k}{n-1} \geq 1 - \frac{Z}{n-1}$. Choose n large enough such that $1 - \frac{Z}{n-1} > 0.95$ (note that $Z^d/n \to 0$, with $d \geq 1$ when $n \to \infty$). Therefore, the number of distinct random links created by u can be lower bounded by the sum of Z identical Bernoulli random variables of expectation 0.95. Using lemma 2.1, with $VHP(Z)$ a given node u generates at least $0.9Z$ distinct random links. So, for any constant $C > 0$, $\Pr[diam(I) \leq C] \leq \Pr[diam(J) \leq C] + eNeg(Z)$.

Using the 3 graph comparisons above and choosing $C = d + 4$, we have $\Pr[diam(K) \leq d + 4] \leq \Pr[diam(J) \leq d + 4] + eNeg(n) + eNeg(Z) < \Pr[diam(J) \leq d + 4] + eNeg(Z)$ since $n >> Z$. But $diam(J) \leq d + 4$ with $VHP(np)$ or $VHP(Z)$, using lemma 2.2. Thus, $diam(J) \leq d + 4$ with $VHP(Z)$.

2.2 Connecting pieces. To extend our results for arbitrary τ, we partition the random graph G into vertex subsets and consider the diameter of the induced subgraphs. We also *contract* each vertex subset into a super-node and obtain a new graph F of $\hat{n}$ nodes, where for each link between two nodes u and v in different subsets, we place a link between the two corresponding nodes in F. We then bound the diameter of F. Note that a path π in G becomes a super-path π_F in F where all the sub-paths of π that are within a contracted subgraph in G are hidden in a super-node of F. For an instance of G, it is easy to see that $diam(G) \leq x \times y$ if $x = diam(F)$ and y is the largest diameter among any of the induced subgraphs. We focus on the following scenario when the vertex subsets have large enough size and weight such that we can link any two such subsets by $O(1)$ links (so $diam(F) = O(1)$). We use a critical parameter m_τ which is the minimum probability any edge (u, v) is created under distribution τ. To assure that any part of the graph is reached with at least moderate probability, we assume in the lemma below that we have enough random edges to compensate for a small value of m_τ (the weight $z(n)$ used below).

Lemma 2.3. (Connecting lemma) *Consider $G = \mathcal{NAN}(H, \tau)$ with $n = |V|$ and a graph partition of H with parameters $\hat{n}, m$ (both functions of n) such that V is partitioned into $\hat{n}$ disjoint subsets $V_1, V_2, \ldots, V_{\hat{n}}$, each of size at least m. Assume also that H has enough weight so that there are at least $z = z(n)$ independent random links created by each set V_i, where $z\hat{n}mm_\tau = \Omega(\hat{n}^\epsilon)$ for some constant $\epsilon > 0$. Then the super-graph F of the V_i's as single nodes has diameter at most $(\lfloor \frac{1}{\epsilon} \rfloor + 4)$ with $VHP(\hat{n}^\epsilon)$*

As a simple corollary, if there is a function $D = D(n)$ such that all the subgraphs induced by the V_i's in G have diameter at most D then $diam(G) = O(D)$ with $VHP(\hat{n}^\epsilon)$.Lemma 2.3's proof is in §A.2 wherein the basic idea is to show that $diam(F)$ is upper bounded by a constant, $\lfloor \frac{1}{\epsilon} \rfloor + 4$ with $VHP(\hat{n}^\epsilon)$ so the diameter of G is upper bounded by $(\lfloor \frac{1}{\epsilon} \rfloor + 4) \times D$ with $VHP(\hat{n}^\epsilon)$. We use theorem 2.1 by comparing F with $J = \mathbb{J}(\hat{n}, Z)$ in diameter where $Z = .5z\hat{n}mm_\tau$ and using $d = \lfloor \frac{1}{\epsilon} \rfloor$.

2.3 A distance-bias and bounded-growth model.

Basic definitions for metric spaces. The pair (V, d), including a set of nodes V and a distance function $d : V^2 \to \mathbf{R}^+$ is a *metric space*, and d is a *metric*, if for any $u, v, w \in V$, $d(u, v) = 0 \Leftrightarrow u = v$, $d(u, v) = d(v, u)$ (symmetry) and $d(u, v) \leq d(u, w) + d(w, v)$ (triangle inequality). For $r > 0$, a subset $X \subset V$ is an *r-cover* of V if any node $u \in V, u \notin X$ is within distance r from at least one node $v \in X$. A subset $X \subset V$ is an *r-packing* if any two nodes in X are at least distance $2r$ apart. A subset $X \subset V$ is an *r-net* if X is both an r-cover and an $\frac{r}{2}$-packing. It is well known that for any $r > 0$, there exists an r-net for any metric space, which can be constructed greedily.

Consider base graph $H(V, w, E)$ associated with *metric distance* d. We define the ball of center u and radius r, $N_u(r) = \{v \in V | d(u, v) \leq r\}$, the subset of nodes within *distance* r from u. For any subset $B \subset V$, we call $r = min_{u \in V} max_{v \neq u} d(u, v)$ the radius of B, that is there exits a node $u \in V$ such that $N_u(r)$ is the smallest ball which contains B. We also write $r = Radius(B)$. For a node $u \in V$, define $R_u = min\{r : N_u(r) = V\}$, the maximum distance of any node from u.

For $\Delta > 1$, H is (or more precisely, (V, d) is) *growth bounded* (GB) *above* by growth Δ, or is GB^Δ, if

$$\forall u \in V, \forall r, |N_u(2r)| \leq \Delta |N_u(r)|$$

Also, for a constant $\bar{\Delta} > 1$, H is GB *below* by growth degree $\bar{\Delta}$ (w.r.t. d), or is $GB_{\bar{\Delta}}$, if

$\forall u \in V, \forall r \leq R_u/2, |N_u(r)| \geq 2 \Rightarrow \bar{\Delta} |N_u(r)| \leq |N_u(2r)$

Intuitively, the "speed" that nodes from V "come into view", when we expand a ball around $u \in V$, may vary but bounded from above and below, specified by Δ and $\bar{\Delta}$. Similar notions were used in e.g. [31]. For some constant $\beta > 0$, it is easy to see that GB_{2^β} implies $|N_u(r)| = \Omega(r^\beta)$ and also GB^{2^β} implies $|N_u(r)| = O(r^\beta)$ $(\forall u \in V)$ (but the the reverse direction is not true).

Definition 2.2. ($\mathcal{D}ist\mathcal{B}ias$ setting) *For $\alpha, \beta > 0$, a graph $G = \mathcal{NAN}(H, \tau)$ is a $\mathcal{D}ist\mathcal{B}ias(\alpha, \beta)$ setting if H has an associated metric function d such that H is GB_{2^β}* w.r.t *d, and $\tau_u(v) \propto d^{-\alpha}(u, v)$. We use $\mathcal{D}ist\mathcal{B}ias(\alpha)$ for*

$\alpha = \beta$. *Similarly,* $\mathcal{D}ist\mathcal{B}ias(\alpha, \beta_1, \beta_2)$ *is also defined with* H *both* $GB_{2^{\beta_1}}$ *and* $GB^{2^{\beta_2}}$.

EXAMPLE 2.2. *Kleinberg's 2-D grid model (Example 2.1) is* $\mathcal{D}ist\mathcal{B}ias(2)$*; Kleinberg's tree-based setting [21] is* $\mathcal{D}ist\mathcal{B}ias(1)$ *with* $d(u, v)$ *as the size of the minimum subtree containing both* u *and* v.

The connecting lemma is a key tool we repeatedly use later. In §3, we consider the diameter of $\mathcal{D}ist\mathcal{B}ias(\alpha, \beta_1, \beta_2)$ with larger α where the random links strongly favor closer distances. Our analysis is based on a partitioning hierarchy where we recursively partition the graph (into subgraphs then these subgraphs into smaller subgraphs ...) such that each partition follows the pattern in the connecting lemma so that the diameter of a super graph of subgraphs under any such partition is $O(1)$. We then use a probabilistic recurrence argument to upper bound the diameter of our original graph. This probabilistic recurrence is the reason for the extra effort in working with the high probabilities (i.e. why need the VHP notation) in the connecting lemma.

In §4, we consider the setting with smaller α when, given a random link, "any distant corner" of the graph has a fair chance to be reached by this arc. This "fairness" feature helps to show that, with proper conditions, the growth in size of a neighborhood using both local and random links is exponential before reaching a threshold size. We construct several such neighborhoods such that their union includes almost all the nodes of our graph, and use our connecting lemma to show that we can connect these neighborhoods by only $O(1)$ links between any two. Thus, a giant component with logarithmic diameter will emerge with high probability. In fact, this result applies for the general $\mathcal{NAN}(H, \tau)$ under some proper conditions on the weights of H, and applies for the setting $\mathcal{D}ist\mathcal{B}ias$ as a corollary.

3 Strong distance-bias.

We consider the diameter of $\mathcal{D}ist\mathcal{B}ias(\alpha, \beta_1, \beta_2)$ for the case $\alpha \in [\beta_2, 2\beta_1)$ in §3.1, and for other cases in §3.2.

3.1 A moderate distance-bias regime: a fractal perspective.

Here, our analysis method is based on a partitioning hierarchy which reflects a fractal picture: the whole graph is an $O(1)$-diameter super graph of nodes representing subgraphs, and each subgraph itself is a smaller-scale $O(1)$-diameter super graph, and so on, until reaching poly-log size and thus poly-log diameter. Intuitively, one can think of an online map with zoom levels: country, state, county, district, etc. For example, a typical short $s - t$ route consists of $O(1)$ super-links between super 'state' nodes at the highest zoom-out level. However, when we zoom-in on a super 'state' node, we unfold a sub-path of $O(1)$ links between super 'county' nodes. Figure 3.1 traces such a route, unfolding to the highest zoom-in at both the source and destination sites. We need some reasonable conditions on the base graphs to create such a fractal picture. The following Voronoi-type partition is a key to our analysis. Consider a base graph $H(V, E)$ and an associated metric d. Let $T = \{u_i\}_{i=1}^k$ be a subset of nodes in V. *A* T*-partition is a partition of* V *into subsets induced by a node set* T as follows. We partition V into disjoint sets V_i with $V = \cup_{i=1}^k V_i$, by using the u_i's as centers: we put any node $v \in V$ into V_i if u_i (among these k center nodes) is the closest to v by the metric distance (handle ties arbitrarily). Voronoi-type partitions *induced by the nodes of an* r*-net* are very useful in our analysis.

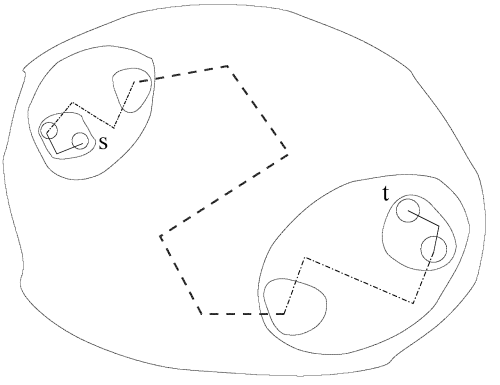


Figure 1: *Tracing a short route in our 'fractal-like map'.* By unfolding super-nodes at different depths, we 'observe' various-depth super-links (dashed) of super-graphs and physical ones (solid) of the actual network. The super-links within a sub-graph create an $O(1)$-length path. Each super-node is a collapsed sub-graph and hides within it part of the actual path, which has poly-log length.

THEOREM 3.1. *For a* $\mathcal{D}ist\mathcal{B}ias(\alpha, \beta_1, \beta_2)$ *setting where* $\beta_2 \leq \alpha < 2\beta_1$*, w.h.p., a graph* $G = \mathcal{NAN}(H, \tau)$ *has diameter* $O(log^{\gamma} n)$ *for a constant* $\gamma = \gamma(\alpha, \beta_1, \beta_2)$ *if the following conditions on the* base graph H *are met*
(i) $\exists$ *constants* $c_1, c_2 \geq 0$ *and a* c_1*-net* T *s.t. any subset* $B \in T$*-partition has at least two nodes and has node weight average* $\frac{w(B)}{|B|} \geq c_2$
(ii) $\exists$ *constant* $c > 0$ *s.t. for any two nodes* $u, v \in V$ *if the smallest ball* B *containing both* u *and* v *has size at least* c*, then the shortest path between* u *and* v *has hop-length (in* H*) at most* B*'s size.*

Note that (i) is a reasonable condition on 'weight density' (on average, nodes have at least c_2 random

links), and (ii) is also reasonable in this intuitive view: the shortest chain of acquaintances between any two persons (who are not too close) is less than the number of people in their local neighborhood. Thus the path doesn't have to go far outside this neighborhood. If all node weights in our graph are at least c for a constant $c > 0$, we call this an $O(1)$-*weight graph*. Thus, $O(1)$-weight meshes easily meet both (i) and (ii) even with a constant fraction of missing links. Below, we sketch the ideas to prove theorem 3.1. Basically, we describe and analyze our partitioning hierarchy then show how our "connecting pieces" argument can be applied recursively, down to the bottom level.

The partitioning hierarchy. To construct our partitioning hierarchy we fix a constant $\xi \in (0,1)$ (to be chosen later) and define a basic partition, ξ-PAR, which is recursively applied to create the hierarchy. Let $R = Radius(V)$. For the whole node set V, this initial partition results in node subsets of radius between $R^\xi/2$ and R^ξ and of roughly the same size. Basically, this ξ-PAR is a special Voronoi-type partition that is induced by the nodes of an R^ξ-net but with this special feature: this ξ-PAR is perfectly textured by the given T-partition, i.e. no subset of T-partition is split between two subsets of this ξ-PAR. Recall that for any subset $B \subset V$, $r = min_{u \in V} max_{v \neq u} d(u,v)$ is the radius of B ($r = Radius(B)$), i.e. there exits a node $u \in V$ such that $N_u(r)$ is the smallest ball which contains B.

To construct such a ξ-PAR of V, we construct an R^ξ-net $S = \{s_0, s_1, s_2 \ldots\}$ then divide V into subsets V_i's around these s_i's as follows: for each given subset B from the T-partition we choose $s_i \in S$ closest to B's center (handle ties arbitrarily) then we place B entirely into V_i (centered by s_i). Each vertex subset of this partition has radius between $R^\xi/2$ and R^ξ by the r-net definition, and has size at least $a = \theta(R^{\xi\beta_1})$ because of (i) and the growth-bounded definition. Now, for each subgraph with size $\geq 2a$, we can split it into smaller subgraphs with size between a and $2a$ such that the obtained subgraphs are still well-textured by the T-partition, and hence have average weight $\geq c_2$. (Instead of using this texturing argument, we can think of the small subsets in the T-partition as single 'unit cells' and do our partitions on these cells).

Define the ξ-*graph* for such a ξ-PAR to be the multi-graph we obtain when we contract the subsets of the ξ-PAR into single super-nodes. The links between any two given nodes in different subsets become the (multiple) links between the two respective super-nodes.

For $L = L(n)$, *a* (ξ, L)-*hierarchy of partitions* is obtained when we ξ-PAR these V_i's into smaller subsets, and recursively repeat this until we reach subsets with size L or smaller. By a (ξ, L)-hierarchy we create a hierarchy of vertex subsets of the base graph H such that each subset has enough average weight (so we will have random links to connect subsets within a level with VHP), subsets shrink in size significantly from level to level (so the height of the hierarchy is small), and the total number of subsets in the hierarchy is linear. Also, the subsets in the same ξ-PAR have similar sizes. (All these properties are formulated in lem. B.1 in §B.)

We now sketch the ideas in proving theorem 3.1. We use a (ξ, L)-hierarchy and by choosing proper values of ξ and L, show that there exists a constant C such that all the ξ-graphs induced by a ξ-PAR in the hierarchy have diameter at most C w.h.p. Thus, a standard (probabilistic) recurrence argument will show that $diam(G) = O(L \log^\gamma n)$ where $\gamma = \log(C+1) \times \log_{\frac{1}{\xi}} 2$. For each ξ-PAR, we apply the Connecting lemma (lem. 2.3) and show that there exists a constant $\epsilon > 0$ (depending on $\xi, \alpha, \beta_1, \beta_2$) such that the ξ-graph induced by this ξ-PAR has diameter at most $C = \lfloor \frac{1}{\epsilon} \rfloor + 4$ with probability $\geq 1 - O(e^{-\hat{n}^\epsilon})$ where $\hat{n}$ is the number of subgraphs, i.e. super-nodes of this ξ-graph (the detailed proof in §B shows why we need $\alpha \in [2\beta_1, \beta_2)$ to find proper ξ). By choosing $L = \log^\lambda n$ for constant λ large enough, we show that the failure probability in each connecting-pieces step is small enough such that the union of all these failure events (at all the ξ-PARs) is still negligible.The full proof is in §B, where we show the above arguments in details, using the above mentioned properties of our partitioning hierarchy.

Using weight density with respect to metric distance. The assumptions on the base graphs can be weakened as follows. We need to guarantee a certain density of weight in the base graph so the subgraphs at the bottom level of our partitioning hierarchy have enough weight for the connecting lemma. Also, if these subgraphs are connected by just the local links then we can upper bound the diameter of each subgraph by its size, and hence poly-log of n. These factors (on weight density and local connectivity) can be realized when we can divide our graphs into small subgraphs that are connected and have enough weight.

A connected subgraph F of H is a *semi-ball* if F can be contained inside a ball B_F so that F's size is at least c times B_F's size for a constant $c \in (0,1)$. For $k = k(n)$, base graph H is k-*dense* w.r.t. metric distance d if there exists a constant $c > 0$ such that for n large enough, there exists a partition V into sets A_i, such that each subgraph induced by A_i is a connected semi-ball that has weight $\geq k$ and average weight $\geq c$. The following theorem is rather straightforward from our analysis above with theorem 3.1.

THEOREM 3.2. *For a* $\mathcal{DistBias}(\alpha, \beta_1, \beta_2)$ *setting where*

$\beta_2 \leq \alpha < 2\beta_1$, using base graphs at least $(\frac{\beta_2}{2\beta_1-\alpha})$-dense, w.h.p. the setting has poly-log diameter.

Furthermore, we can extend the *k-dense* graph notion by allowing the vertex set A_i's to overlap but the subgraphs induced by them are still edge-disjoint (but possibly not vertex-disjoint). Thus we can split a 'super-hub' node into virtual nodes of smaller weight (like chopping up a broccoli). So, *most connected $O(1)$-weight graphs can be augmented into ones with poly-log diameter w.h.p.*

Some application issues in routing and network design. Our analysis above (particularly our partitioning hierarchy), suggests *a new routing strategy* based on the above partitioning hierarchy. In this hierarchy of recursive partitions, a parent block of metric diameter Δ is partitioned into child blocks of metric diameter Δ^μ (for a fixed constant $\mu \in (\frac{\alpha}{2\beta_1}, 1)$). Thus, this hierarchy has only a *poly-log number of levels* and yet, is *far less steep* than a b-ary tree (for constant $b \geq 2$) where nodes near the root can be overloaded and highly congested. This suggests a hierarchical routing strategy which is very efficient for a large class of our *distance-bias* structures (example 3.1 below): a short $s-t$ route within a parent block can be formed by combining sub-routes within a few sub-blocks. In our initial results, this routing scheme uses a *small distributed routing database* (keeping information for short-cut links between blocks) and finds *routes of poly-log expected length.* Because of the space limit we defer the details to the full version.

EXAMPLE 3.1. *(variable growth in wireless networks) Consider a wireless network where the growth rate varies from 2 to 3 (E.g. a sensor network covering several nearby structures: so nodes in a neighborhood grow at nearly a cubic rate at the center of a big block, nearly a square rate on a lawn or in a large one-story building). Using the unit disk model for wireless networks, if we add one random link to each node using distribution τ where $\tau_u(v) \propto d^{-\alpha}(u,v)$ for $\alpha \in (3,4)$, then the conditions in theorem 3.1 or 3.2 are satisfied. Hence, w.h.p. the new graph has poly-log diameter.*

3.2 On strong distance-bias regimes. For larger α, the random links are shorter generally, and when $\alpha > 2\beta_2$, they become too short to produce short-cut paths between subgraphs as before and hence, the graph has diameter polynomial in n if it is connected (if not connected, the diameter is ∞)

THEOREM 3.3. *Consider a $\mathcal{D}ist\mathcal{B}ias(\alpha,\beta_1,\beta_2)$ setting where $\alpha > 2\beta_2$ and the base graph has constant-bounded weights. If the graph is connected then there exists a constant $c > 0$ such that the graph diameter is $\Omega(n^c)$.*

Proof. [Sketch] Consider such a graph $G = \mathcal{NAN}(H,\tau)$ with vertex set V and radius Δ. Note that $\Delta > n^{\frac{1}{\beta_2}}$. Choose a constant $\xi \in (0,1)$ such that $\alpha\xi > 2\beta_2$. Now, we claim the probability that we have a random link with length at least Δ^ξ is negligible and hence, clearly w.h.p. the graph diameter is $\Omega(\Delta^{1-\xi})$, i.e. $\Omega(n^c)$ for $c = \frac{1-\xi}{\beta_2}$. The probability of having a random link between two given nodes at distance $\geq \Delta^\xi$ apart is $O(\Delta^{-\xi\alpha})$; so the probability for having any link of length $\geq \Delta^\xi$ is $\leq n^2 \times O(\Delta^{-\xi\alpha}) = O(\Delta^{2\beta_2-\xi\alpha}) = o(1)$.

On other distance-bias regimes. The cases $\alpha \in (\beta_1, \beta_2)$ and $\alpha \in [2\beta_1, 2\beta_2]$ are mostly open. For the former case we believe that the graph diameter is still poly-log and whether it is logarithmic or not depends on the graph topology.

4 Weak distance-bias.

We show that with reasonable conditions on the weights of the base graphs, $\mathcal{D}ist\mathcal{B}ias(\alpha,\beta_1,\beta_2)$ has logarithmic diameter when $\alpha \leq \beta_1$. The random links are now not too distance-biased so the distribution of a random link from any given node u is fairer to distant targets: the random link from u has a fair chance to escape from a moderate size neighborhood around u. Thus, we show that the growth in size when expanding a neighborhood around u is *exponential before reaching a moderate threshold size.* This is even easier in areas where the growth rate is higher (so worst case is $\beta_2 = \beta_1$).

To show logarithmic diameter: we construct several such neighborhoods in this exponential growth manner that cover most of our vertex set. We then show that any two of these subgraphs (neighborhoods) can be connected by $O(1)$ links using our Connecting lemma. We show that even in the worst case with $\beta_2 = \beta_1$, i.e. we don't care what β_2 is as long as $\alpha \leq \beta_1$. Thus, denote $\beta = \beta_1$ in the rest of this section for simpler notation.

THEOREM 4.1. *For $0 < \alpha \leq \beta$ there exists a constant $q > 0$ such that a $\mathcal{D}ist\mathcal{B}ias(\alpha,\beta)$ setting using a q-heavy n-node base graph almost surely has a giant connected component with size $n(1-o(1))$ and diameter $O(\log n)$.*

Before showing this theorem we consider a general sufficient condition for having a giant component with logarithmic diameter for the more general setting $\mathcal{NAN}(H,\tau)$. Let $S_k(u)$ denote the set of nodes within k links from u (in contrast, $N_r(u)$ is the nodes within a metric distance r from u).

DEFINITION 4.1. *1) (**Expansion property**) Consider a random graph $G = \mathcal{NAN}(H,\tau)$. For constants $\mu,\xi \in (0,1)$, τ has* the (μ,ξ) expansion property *if a random link leaves a vertex set $\mathcal{C}$ of size at most n^μ*

with probability at least ξ (or hits $\mathcal{C}$ with prob. at most $1-\xi$). More formally, for $n = |V|$:

$$\forall u \in V, \forall \mathcal{C} \subset V, |\mathcal{C}| < n^{\mu} : \tau_u(\mathcal{C}) \leq 1 - \xi$$

where $\tau_u(\mathcal{C}) = \sum_{v \in \mathcal{C}} \tau_u(v)$. We also say, τ is (μ,ξ)-expansion.

2) For a constant $\eta > 0$, rule τ is η-fair if $m_\tau = min_{u \neq v}\{\tau_u(v)\}$ is at least $\Omega(n^{-\eta})$.

These properties assure a fair and diverse rule for random links: for any given node u, no small set of vertices takes a dominant role in attracting u's links and also no single node is 'ignored' (it is connected to u with probability at least $m_\tau = \Omega(n^{-\eta})$). Many existing networks feature these properties.

Example 4.1. *Watts and Strogatz's small-world networks are (μ,ξ)-expansion for any constants $\mu, \xi \in (0,1)$, and 1-fair. Kleinberg's grid-based [20] and tree-based [21] small-world models are (μ,$1-\mu-o(1)$)-expansion for any constant $\mu \in (0,1)$, and are $(1+\epsilon)$-fair for any constant $\epsilon > 0$.*

From a heavy enough node set many random links will leave the set, resulting in large sets with small diameter. Particularly, the expansion property produces exponential growth in neighbor sets, $S_k(u)$, until the neighborhood reaches a large enough threshold size. Lemma 4.1 below describes a general sufficient condition on when and how we can construct a neighbor ball with logarithmic diameter. We leave lemma 4.1's proof to appendix §C.

Lemma 4.1. (Expansion lemma) *Given arbitrary constants $\mu, \xi \in (0,1)$, consider a random graph $G = \mathcal{NAN}(H, \tau)$ where τ is (μ,ξ)-expansion and $\exists \epsilon > 0$ s.t $\mathcal{H}$ is $(\frac{1}{\xi} + \epsilon)$-heavy. There exists a constant $c > 0$ such that for any random graph from $\mathcal{F}$ with size n large enough, for any given node u, w.h.p. $|S_{c \log n}(u)| \geq n^{\mu}$.*

Our general setting, conditioned on a few reasonable criteria, creates a broad class of random structures where paths of logarithmic length are typical. The conditions on the expansion property and the weight density allow us to construct large enough neighborhoods with logarithmic diameter. The other fairness condition assures we can connect neighborhoods.

Theorem 4.2. *(Sufficient condition for short paths) Consider $G = \mathcal{NAN}(H, \tau)$ with n nodes. If there exist constants $\mu, \xi \in (0,1)$ such that τ is (μ,ξ)-expansion and $(1+\mu-\epsilon_1)$-fair and $\mathcal{H}$ is $(\frac{1}{\xi} + \epsilon_2)$-heavy for some constants $\epsilon_1, \epsilon_2 > 0$, then G almost surely has a giant connected component with size $n(1-o(1))$ and diameter $O(\log n)$.*

Note that theorem 4.2's conditions are met by all the settings in examples 4.1 and A.1 (in appendix §A) with proper parameters.

Proof. [Sketch] For simplicity, consider the case of connected base graph. We split the weights into two parts: the first part is used to construct subgraphs of size n^{μ} as balls of neighbors (lemma 4.1), then the rest of the weight can be used to connect these subgraphs, using paths of $O(1)$ links between any two subgraphs – now nodes in our super-graph – by using the Connecting lemma. However, a naive approach in constructing these neighbor balls may result in overlapping balls and violate the requirement for independent random links in our super-graph. When the base graph is connected, we use the shortest path distance d (using number of arcs) as a metric and create an L-net $X = \{u_1, u_2, \ldots, u_k\}$ in the metric space (V, d), for an appropriate L. Note that for any connected undirected graph, the shortest path distance is clearly a metric, and there exists an r-net (def. in §2.2) with respect to this distance for any $r > 0$. Now we choose the nodes in X to be centers of our neighbor balls: we partition V into k disjoint subsets $\{A_i\}_{i=1}^{k}$, by putting any node $v \in V$ into A_i if u_i (among these k center nodes) is the closest to v (handle ties arbitrarily). We choose L so that our neighbor balls are large enough. See the full proof in appendix §C.1.

Now, to show theorem 4.1 we just need to show that our graph from $\mathcal{DistBias}(\alpha, \beta)$ meets the conditions of theorem 4.2. However, the main work is to justify the expansion property, which is stated explicitly in lemma 4.2 below.

Lemma 4.2. *Consider the random graph $G = (H, \tau)$ in theorem 4.1. For any $\mu \in (0,1)$ there exists $\xi = \xi(\mu) \in (0,1)$ such that τ is (μ,ξ)-expansion*

We defer lemma 4.2's proof, which is long and tedious, to a fuller version.

4.1 Extension with f-dense graphs.

For $f = f(n) > 0$, a base graph H of n nodes is f-dense if there exists a constant $c > 0$ [6] such that the node set $V(H)$ can be be put into subsets *of size at most f/c and weight at least f* as follows. These vertex subsets are not necessarily disjoint, but their union is V, and they induce *connected* subgraphs which are mutually *edge-disjoint*. So, if we contract these vertex subsets to single super-nodes then the induced graph becomes f-heavy. Note that for each pair of different super-nodes $\hat{u}$ and $\hat{v}$ there is a link between $\hat{u}$ and $\hat{v}$ if there is

[6] More formally instead, one should use an infinite collection of base graphs $\mathcal{H} = \{H_i, i \in I\}$.

a link between $\hat{u}$'s vertex subset and $\hat{v}$'s in H or the two subsets intersect (because a big star hub joins both of them as we will illustrate below). Clearly, if the contracted graph has graph diameter D then H has diameter $O(fD)$.

Note that these subgraphs can overlap but only in vertices. If we restrict the subgraphs to be fully disjoint, we then exclude many graphs from being f-dense (and hence, contractible to f-heavy), e.g. a simple star network where a big hub connects to all other vertices. However, here we can split the star network into several subgraphs overlapping only in the hub node (like chopping a broccoli) to make our f-dense and contracting arguments work. In fact, for any constant $c > 0$, for any $f(n) < cn$, any collection of *c-heavy connected base* graphs is f-dense.

Thus, the above results can be extended by using f-dense graphs instead. For example, we extend theorem 4.1 to theorem 4.3 below.

THEOREM 4.3. *For $0 < \alpha \leq \beta$ there exists a constant $q > 0$ such that a $\mathcal{DB}ias(\alpha, \beta)$ setting using a q-dense n-node base graph almost surely has a giant connected component with size $n(1 - o(1))$ and diameter $O(\log n)$.*

COROLLARY 4.1. *For $0 < \alpha \leq \beta$, a $\mathcal{DB}ias(\alpha, \beta)$ setting using a connected $O(1)$-weight base graph has diameter $O(\log n)$ w.h.p.*

Now, if the Internet topology (e.g. at AS-level) can be simulated by a setting $\mathcal{DB}ias(\alpha, \beta)$, then it is straightforward from theorems 4.3 and 4.2 that: *For the Internet topology (e.g. at AS-level), if $\alpha \approx 1$ (as observed in [34]) and β is between 1 and 2 (as in [12, 34]), then we would expect the Internet topology to have poly-log diameter with VHP.*

5 Conclusion and future work.

Our results present a flexible approach for modeling the common features of real-world networks, based on the standard augmented graph model. We provide strong analysis results for our refined model for large-scale real-world networks with a focus on the geographical factors. In future work, we want to see if we can continue this approach to refine the general (H, τ) model in other ways so we can simultaneously simulate more, or different sets of, common features of complex networks. The set of tools on diameter analysis we produce in this paper can be useful for later work.

On application issues. In §3.1 we discussed an hierarchical routing strategy which is naturally efficient for a large class of our *distance-bias* structures. On-going work considers network constructions for building hybrid ad-hoc networks[18] by adding a wired infrastructure to an ad-hoc wireless network. This hierarchical routing strategy can potentially optimize multiple network factors such as route length, network cable cost and congestion. We have obtained initial results [30] for a special case of the networks in Example 3.1, where we consider *adding long links to* a 2-dimension grid-like network (so $\beta_1 = \beta_2 = 2$). Our network constructions and hierarchical routing strategy also motivates further study of the remaining parameter regimes ($\alpha \in (\beta_1, \beta_2)$ and $\alpha \in (2\beta_1, 2\beta_2)$) as well as new efficient routing strategies for the logarithmic diameter regime ($\alpha < \beta_1$).

Acknowledgment

This work was partly supported by NSF grant 0520190. We would like to thank Pierre Fraigniaud for making a number of helpful comments and suggestions.

References

[1] I. Abraham, D. Malkhi, and O. Dobzinski, "Land: Stretch $(1 + \epsilon)$ locality-aware networks for dhts," in *Proc. of ACM Symp. on Discrete Algorithms (SODA)*, 2004.

[2] L. Barriere, P. Fraigniaud, E. Kranakis, and D. Krizanc, "Efficient routing in networks with long range contacts," in *15th Intr. Symp. on Dist. Comp., DISC'01*, pp. 270–284.

[3] I. Benjamini and N. Berger, "The diameter of a long-range percolation clusters on finite cycles," *Random Structures and Algorithms*, vol. 19, no. 2, pp. 102–111, 2001.

[4] B. Bollobas, "Random graphs," vol. 44, pp. 1–20, 1991.

[5] ——, *Random Graphs*, 2nd ed. Cambridge University Press, 2001, ch. 10.

[6] F. Chung and L. Lu, "Connected components in random graphs with given degree sequences," *Annals of Combinatorics*, vol. 6, pp. 125–145, 2002.

[7] ——, "The average distances in random graphs with given expected degrees," *Internet Mathematics*, vol. 1, pp. 91–114, 2003.

[8] D. Coppersmith, D. Gamarnik, and M. Sviridenko, "The diameter of a long- range percolation graph," *Random Structures and Algorithms*, vol. 21, no. 1, pp. 1–13, 2002.

[9] S. Dolev, E. Kranakis, D. Krizanc, and D. Peleg, "Bubbles: Adaptive routing scheme for high-speed dynamic networks," *SIAM J. Comput.*, vol. 29, no. 3, pp. 804–833, 1999.

[10] P. Duchon, N. Hanusse, E. Lebhar, and N. Schabanel, "Could any graph be turned into a small-world?" *Theor. Comput. Sci.*, vol. 355, no. 1, pp. 96–103, 2006.

[11] ——, "Towards small world emergence," in *18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2006, pp. 225–232.

[12] C. Faloutsos, P. Faloutsos, and M. Faloutsos, "On power-law relationships of the internet topology," in *Proc. of ACM SIGCOMM*, 1999.

[13] P. Fraigniaud, "A new perspective on the small-world phenomenon: Greedy routing in tree-decomposed graphs," in *In 13th Annual European Symposium on Algorithms (ESA)*, 2005.
[14] ——, "Small worlds as navigable augmented networks: Model, analysis, and validation," in *In 15th Annual European Symposium on Algorithms (ESA)*, 2007.
[15] P. Fraigniaud, C. Gavoille, and C. Paul, "Eclecticism shrinks the world," in *Proc. of ACM Symp. on Princ. of Dist. Comp. (PODC)*, 2004, pp. 169–178.
[16] P. Fraigniaud, E. Lebha, and Z. Lotker, "A doubling dimension threshold $\theta(\log\log n)$ for augmented graph nagavibility," in *In 14th Annual European Symposium on Algorithms (ESA)*, 2006.
[17] J. Gao and L. Zhang, "Tradeoffs between stretch factor and load balancing ratio in routing on growth restrict graphs," in *Proc. of ACM Symp. on Princ. of Dist. Comp. (PODC)*, 2004.
[18] A. Helmy, "Small worlds in wireless networks," *IEEE Commun. Lett.*, vol. 7, no. 10, pp. 490–492, Oct. 2003.
[19] D. Karger and M. Ruhl, "Finding nearest neighbors in growth-restricted metrics," in *Proc. of ACM Symp. on Theory of Computing (STOC)*, 2002.
[20] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proc. of ACM Symp. on Theory of Computing (STOC)*, 2000.
[21] ——, "Small-world phenomena and the dynamics of information," in *Neural Info. Proce. Sys. (NIPS)*, 2001.
[22] ——, "Complex networks and decentralized search algorithms," in *Proceedings of the International Congress of Mathematicians (ICM)*, 2006.
[23] R. Kumar, D. Liben-Nowell, and A. Tomkins, "Navigating low-dimensional and hierarchical population networks," in *In 14th Annual European Symposium on Algorithms (ESA)*, 2006, pp. 480–491.
[24] A. Lakhina, J. W. Byers, M. Crovella, and I. Matta, "On the geographic location of internet resources," *IEEE Journal of Selected Areas in Communications*, vol. 21, no. 6, 2003.
[25] E. Lebhar and N. Schabanel, "Almost optimal decentralized routing in long-range contact networks," in *International Colloquium on Automata, Languages and Programming*, 2004, pp. 894–905.
[26] D. Liben-Nowell, J. Novak, R. Kumar, P. Raghavan, and A. Tomkins, "Geographic routing in social networks," *In Proceedings of the National Academy of Sciences*, vol. 102, no. 33, pp. 11 623–11 628, 2005.
[27] C. Martel and V. Nguyen, "Analyzing Kleinberg's (and other) smallworld models," in *Proc. of ACM Symp. on Princ. of Dist. Comp. (PODC)*, 2004.
[28] S. Milgram, "The small world problem," *Psychology Today*, vol. 22, pp. 61–67, 1967.
[29] V. Nguyen and C. Martel, "Analyzing and characterizing small-world graphs," in *Proc. of ACM Symp. on Discrete Algorithms (SODA)*, 2005.
[30] ——, "Designing low cost networks with short routes and low congestion," in *Proc. of IEEE INFOCOM*, 2006.
[31] C. G. Plaxton, R. Rajaraman, and A. W. Richa, "Accessing nearby copies of replicated objects in a distributed environment," in *9th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1997, pp. 311–320.
[32] A. Slivkins, "Distance estimation and object location via rings of neighbors," in *Proc. of ACM Symp. on Princ. of Dist. Comp. (PODC)*, 2005.
[33] D. Watts and S. Strogatz, "Collective dynamics of small-world networks," *Nature*, vol. 393, pp. 440–32, 1998.
[34] S. Yook, H. Jeong, and A. Barabasi, "Modeling the internets large-scale topology," in *Proceedings of the National Academy of Sciences*, vol. 99, 2002, pp. 13 382–13 386.

A Addition to §2.

We also consider the following slightly extended setting, where the number of random links generated by each node has binomial distribution $Bin(w_u, p)$. Intuitively, node u is assigned a fixed number of *seeds* w_u, where each seed gives birth to a random link with probability p. For simplicity, some of our results are presented using the basic setting (definition 2.1) but these can be easily extended for this extended setting, which is necessary in our proof of theorem 4.2. Note that, however, our results using $\mathcal{NAN}(H,\tau)$ can be easily extended for $\mathcal{NAN}(H,\tau,p)$.

Definition A.1. *Given a pair (H,τ) as in definition 2.1 and a constant $p \in (0,1]$, the node-based augmented networks $\mathcal{NAN}(H,\tau,p)$ are the ones augmented from H as follows: to each node $u \in V(H)$ we add $X_u \sim Bin(w_u,p)$ random links, each link goes to a node $v \neq u$ with probability $\tau_u(v)$.*

The *weight* of a vertex set $S \subseteq V(H)$ is $\sum_{u\in S} pW_u$ for $\mathcal{NAN}(H,\tau,p)$.

Example A.1. *(power-law)*
Consider $G = \mathcal{NAN}(H,\tau,p)$ where the base graph H is a set of weighted nodes and $E(H) = \emptyset$, and where $\tau_u(v) \propto w_v$.

It is not hard to see that the expected degree of a node u is $2pw_u$ (roughly, half generated by u itself and the other half from other nodes: in fact, u's degree is $\sim \left(Bin(w_u,p) + Bin(W - w_u, \frac{pw_u}{W-w_u})\right)$ where $W = \sum_{u\in V} w_u$). As with Chung and Lu's model, this setting can simulate a power-law graph when we assign to H an appropriate weight vector.

When working with asymptotic bounds, we use the following family model for a broader context.

Definition A.2. *A family $\mathcal{NAN}(\mathcal{H},\tau)$ is a pair of two infinite collections $\mathcal{H} = \{H^{(i)}, i \in I\}$ and $\tau = \{\tau^{(i)}, i \in$*

I} *where each* $H^{(i)}$ *is a base graph with a weight vector* $w(H^{(i)})$ *and is associated with a distributions* $\tau^{(i)}$ *as in Def.2.1, and there are* $H^{(i)}$*'s with arbitrarily large sizes.*

Also, a family $\mathcal{NAN}(\mathcal{H}, \tau, p)$ *is defined accordingly as in Definition 2.*

Note that we can extract from $\mathcal{H}$ a sequence $\{H^{(i_k)}\}_{k=1}^{\infty}$ such that $\{V(H^{(i_k)})\}_{k=1}^{\infty}$ increases to infinity. We may implicitly switch between the two contexts of constructing random graphs from: 1) a single base graph and, 2) an infinite collection of base graphs. In the later, τ has a broader meaning, as a rule of distribution for generating random links over a collection of base graphs. However, which τ is being used should be clear from the surrounding context. For simplicity, we sometimes mention asymptotic bounds on a random graph $G = \mathcal{NAN}(H, \tau)$ given the number of nodes $n = |V(H)|$ going to infinity, which is a slight abuse of notation (also adopted by many other papers). To be more rigorous we would explicitly use the second context, where we repeatedly pick a pair (H, τ) from a $\mathcal{NAN}$ family with n going to infinity.

A.1 Chernoff's bound.

We use the following version of Chernoff's bound [4].

LEMMA A.1. *Let* $X_1, \ldots, X_n$ *be independent Bernoulli random variables, with* $E(X_i) = p_i$*. Set* $X = \sum_{i=1}^{n} X_i, p = \sum_{i=1}^{n} p_i/n$ *and* $q = 1 - p$*. Then for* $0 < t < q$ *we have*

$$\Pr[X \geq n(p+t)] \leq e^{-2t^2 n}$$

Furthermore, for $0 < \epsilon < 1$ *we have*

$$\Pr[X \leq pn(1-\epsilon)] \leq e^{-\epsilon^2 pn/2}]$$

A.2 Proof of the Connecting Lemma.

Lemma 2.3. *Consider* $G = \mathcal{NAN}(H, \tau)$ *with* $n = |V|$*. Consider a graph partition of* H *with parameters* $\hat{n}, m$ *(both functions of* n*) such that* V *is partitioned into* $\hat{n}$ *disjoint subsets* $V_1, V_2, \ldots, V_{\hat{n}}$ *each of size at least* m*.*

Assume also that H *has enough weight so that there are at least* $z = z(n)$ *independent random links created by each set* V_i*, where* $z\hat{n}mm_\tau = \Omega(\hat{n}^\epsilon)$ *for some constant* $\epsilon > 0$*. Then the super-graph* F *of the* V_i*'s as single nodes has diameter at most* $(\lfloor \frac{1}{\epsilon} \rfloor + 4)$ *with* $VHP(\hat{n}^\epsilon)$

Proof. We show that $diam(F) \leq (\lfloor \frac{1}{\epsilon} \rfloor + 4)$ with $VHP(\hat{n}^\epsilon)$ by comparing F with $J = \mathbb{J}(\hat{n}, Z)$ in diameter where $Z = .5z\hat{n}mm_\tau$. However, we need the following intermediate graph setting.

Random setting $\mathbb{U}$. For $\hat{n}p < 1$, let $U(\hat{n}, z, p)$ be a random graph from setting $\mathbb{U}$ on a set of $\hat{n}$ nodes such that from each node u we generate z independent random links, each goes to any other node $v \neq u$ with uniform probability p or loops back to u with probability $1 - (\hat{n} - 1)p$.

We now consider random graph $U(\hat{n}, z, p_U)$ where $p_U = mm_\tau$. Note that in F, each node also generates z random links where some may be a loop, and for any two given different super nodes $\hat{u}$ and $\hat{v}$, the probability that a random link generated by $\hat{u}$ goes to $\hat{v}$ is at least $p_U = mm_\tau$ since $\hat{v}$ "contains" m vertices in V. Thus, by using an observation below, $U \preceq F$ and hence, $diam(U) \geq_{st} diam(F)$.

Observation B. Let $H = (V, W, E)$ be a base graph. Consider two random link distributions τ and τ' for base graph H such that for any two nodes $u \neq v$ in V, $\tau_u(v) \leq \tau'_u(v)$ and as a result, τ will likely generate more loops than τ' (we say τ is dominated by τ'). Hence, $X \preceq Y$ for random graphs $X = \mathcal{NAN}(H, \tau)$ and $Y = \mathcal{NAN}(H, \tau')$. Observation B is rather straightforward, using our usual argument with a code $M = M_1||M_2$ generating Y while M_1 generating X.

Similarly as with theorem 2.1's proof, we show that we almost have $J \preceq U$. For simplicity, we ignore the loops in both settings. First, note that U can be constructed as follows: we toss z identical coins where $\Pr[\text{"}head\text{"}] = (\hat{n} - 1)p_U$ and for each "head" we randomly pick a node $v \neq u$ (i.e. with probability $\frac{1}{\hat{n}-1}$) and add a link between u and v. This suggests the following code $M_1||M_2$ to construct U. For M_1 we first create a random instance of J on a set of $\hat{n}$ nodes. Now in M_2, for each node u we toss z such above coins and count the number of heads. If H_u, this number of heads, is greater than Z, we then add $H_u - Z$ additional uniform random links from u. Let Bad_{IJ} denote the event if $H_u < Z$ ever occurs for some node u then, code $M_1||M_2$ generates U if Bad_{IJ} does not happen.

We show that $\Pr[Bad_{IJ}] = eNeg(\hat{n}^\epsilon)$. We consider $\Pr[H_u < Z]$ for a given node u. Note that H_u is the sum of z uniform Bernoulli r.v., each with expectation $(\hat{n} - 1)p_U$. So, $E[H_u] = z(\hat{n} - 1)p_U \approx 2Z$. Thus, by lemma 2.1, $H_u > Z$ with $VHP(Z)$. On the other hand, $\Pr[Bad_{IJ}] \leq \sum_{u \in V_U} \Pr[H_u < Z] = \hat{n} \times \Pr[H_u < Z] = \hat{n} \times eNeg(Z)$. Since $Z = \Omega(\hat{n}^\epsilon)$, $\Pr[Bad_{IJ}]$ is $eNeg(\hat{n}^\epsilon)$. Therefore, $diam(J)$ "roughly greater than" $diam(U)$: for any constant $C > 0$, $\Pr[diam(U) \leq C] \geq \Pr[diam(J) \leq C] - eNeg(\hat{n}^\epsilon)$.

Using the above diameter comparisons and theorem 2.1, F has diameter $(\lfloor \frac{1}{\epsilon} \rfloor + 4)$ with $VHP(\hat{n}^\epsilon)$.

B Addition to §3 with full proofs.

LEMMA B.1. *Consider a random graph* $G = \mathcal{NAN}(H, \tau)$ *in a* $\mathcal{DistBias}(\alpha, \beta_1, \beta_2)$ *setting with the associated metric* d*. Consider a* (ξ, L)*-hierarchy constructed from* H

i) For any ξ-PAR in the hierarchy, when a vertex set B with radius Δ is ξ-PARed into subsets $B_1, B_2, \ldots$, each subset B_i has radius $\leq \Delta^{\xi}$, and size within $[a, 2a]$ where $a = \theta(\Delta^{\xi\beta_1})$, and weight $\geq c_2|B_i|$.
ii) The hierarchy (of recursive ξ-PARs) has height $K < log_{\frac{1}{\xi}} 2 \times \log\log n + O(1)$ and size $O(n)$.
iii) Suppose that there exist $\xi \in (0,1)$ and a constant C such that the ξ-graph of each ξ-PAR in the hierarchy has diameter at most C. Then, $diam(G) = O(L\log^{\gamma} n)$ where $\gamma = \log(C+1) \times \log_{\frac{1}{\xi}} 2$.

Proof. i) Straightforward from the definition of our (ξ, L)-hierarchy.
ii) From i), in each ξ-PAR at level i of the hierarchy, the subsets have radius at most R^{ξ^i}, where $R = Radius(V)$. From $R^{\xi^K} \geq 2$, we have $\xi^K \times \log R \geq 1$, then $(\frac{1}{\xi})^K \leq \log R$ and hence, $K \leq \log_{\frac{1}{\xi}} \log R \leq \log_{\frac{1}{\xi}} (\frac{1}{\beta_1} \log n) \leq \log_{\frac{1}{\xi}} 2 \times \log\log n + O(1)$. The size of the hierarchy is the size of a tree where the number of leaves is less than n, so the size is $O(n)$
iii) In the hierarchy, for any vertex set B which is ξ-PARed into subsets $B_1, B_2, \ldots$, the diameter of B, $diam(B)$, is at most $C \times max_i diam(B_i) + C - 1$. Using ii), a standard recursion argument shows that $diam(G) \leq (L+1)C^K - 1 = O(L\log^{\gamma} n)$ where $\gamma = \log C \log_{\frac{1}{\xi}} 2$ and the constant in the big O depends on C, β_1, ξ.

We now use lemma B.1 to show theorem 3.1.
Theorem 3.1. *For a $\mathcal{DistBias}(\alpha, \beta_1, \beta_2)$ setting where $\beta_2 \leq \alpha < 2\beta_1$, w.h.p., the graphs have diameter $O(log^{\gamma} n)$ for a constant $\gamma = \gamma(\alpha, \beta_1, \beta_2)$ if the following conditions on the base graph H are met*
(i)$\exists$ constants $c_1, c_2 \geq 0$ and a c_1-net T s.t. any subset $B \in T$-partition has at least two nodes and has node weight average $\frac{w(B)}{|B|} \geq c_2$
(ii) $\exists$ constant $c > 0$ s.t. for any two nodes $u, v \in V$ if the smallest ball B containing both u and v has size at least c, then the shortest path between u and v has length at most B's size.

Proof. We consider a (ξ, L)-hierarchy and later, by choosing proper values of ξ and L, show that there exists a constant C such that all the ξ-graphs induced by a ξ-PAR in the hierarchy have diameter at most C w.h.p. Consider an arbitrary ξ-PAR where a vertex set B with size n_B and radius Δ is ξ-PARed into subsets $B_1, B_2, \ldots, B_{\hat{n}}$. We apply the Connecting lemma (lem. 2.3) using the considered ξ-PAR as the partition in the Connecting lemma.

From lemma B.1i), size m of the smallest subset is $\theta(\Delta^{\xi\beta_1})$. By the sufficient weight property (lemma B.1i: subgraphs have weight at least a constant multiple of their size), the number of random links generated from each subset is $z = \Omega(\Delta^{\xi\beta_1})$. It is easy to see that the inverse coefficient $C^{-1} = \theta(1)$ if $\alpha > \beta_2$ and $C^{-1} = O(\log n)$ if $\alpha = \beta_2$ (but C^{-1} may be a polynomial of n if $\alpha < \beta_2$; thus, we need $\alpha \geq \beta_2$). On the other hand, note that $\hat{n}m = \Omega(n_B)$ since the subsets have similar sizes (at most double of any other).

So $Z = z\hat{n}mm_{\tau} = \Omega(\Delta^{\xi\beta_1} \times n_B \times \frac{1}{\log n \Delta^{\alpha}}) = \Omega(\frac{\Delta^{(1+\xi)\beta_1}}{\log n \Delta^{\alpha}})$. So, for $\Delta = \Omega(\log n)$ and by choosing $\xi < 1$ large enough such that $(1+\xi)\beta_1 > \alpha$ then choosing constant $\epsilon_1 \in (0, (1+\xi)\beta_1 - \alpha)$, we have $Z = \Omega(\Delta^{\epsilon_1})$. Note that by having $\alpha < 2\beta_1$ we can choose $\xi < 1$ large enough to have $(1+\xi)\beta_1 - \alpha > 0$. Now choose $\epsilon = \frac{\epsilon_1}{\beta_2}$ and note that $\hat{n} < n_B \leq \Delta^{\beta_2}$ then $Z = \Omega(\hat{n}^{\epsilon})$. That is, the Connecting lemma applies and the considered ξ-graph has diameter $C = \lfloor \frac{1}{\epsilon} \rfloor + 4$ with probability at least $1 - eNeg(\hat{n}^{\epsilon})$.

By choosing L $(= \log^{\lambda} n)$ large enough, we show that the failure probability in each connecting-pieces step is small enough that even the union of all such failure events (at all the ξ-PARs) is still negligible. We apply lemma B.1iii) for our (ξ, L)-hierarchy. The lemma only fails to apply if one of the ξ-graphs has diameter greater than C. By lemma B.1ii), the number of ξ-PARs is $O(n)$ so, lemma B.1iii) fails to apply with probability at most $O(n) \times eNeg(\hat{n}^{\epsilon})$. This probability is still $eNeg(\hat{n}^{\epsilon})$ if we choose $L = \log^{\frac{1}{\epsilon}} n$ so that $e^{\hat{n}^{\epsilon}} >> e^{L^{\epsilon}} = n$. Now, with $VHP(\hat{n}^{\epsilon})$ the diameter of our graph G is $O(L\log^{\gamma_1} n)$ where $\gamma_1 = \log C \log_{\frac{1}{\xi}} 2$ by lemma B.1iii). That is $diam(G) = O(log^{\gamma} n)$ where $\gamma = \frac{1}{\epsilon} + \log(4 + \frac{1}{\epsilon}) \log_{\frac{1}{\xi}} 2$ where ξ is to be chosen from $(\frac{\alpha - \beta_1}{\beta_1}, 1)$ and ϵ is to be chosen from $(0, \frac{(1+\xi)\beta_1 - \alpha}{\beta_2})$. (It is a rather tedious task to minimize $\gamma = \gamma(\xi, \epsilon)$ but note that the term $\frac{1}{\epsilon}$ dominates so an easy upper bound is $\frac{2\beta_2}{2\beta_1 - \alpha}$.)

C Addition to §4 with proofs.

Expansion lemma (L.4.1). *Given arbitrary constants $\mu, \xi \in (0,1)$, consider a family $\mathcal{F} = \mathcal{NAN}(\mathcal{H}, \tau, p)$ which has the (μ,ξ)-expansion property. There exists a constant $c > 0$ such that for any random graph from $\mathcal{F}$ with size n large enough, for any given node u, w.h.p. $|S_{c\log n}(u)| \geq n^{\mu}$, if there exists $\epsilon > 0$ s.t $\mathcal{H}$ is:*
a) $(\frac{1}{p\xi} + \epsilon)$-heavy, or
b) $(\frac{1}{p\xi} + \epsilon)$-dense.

Proof. [Sketch] Part a. A random link escapes a vertex set of size n^{μ} with probability $\geq \xi$, so when we consider extending a neighborhood S of size less than n^{μ}, each node has enough weight to generate at least expected $\gamma = p(\frac{1}{p\xi} + \epsilon)\xi = 1 + p\xi\epsilon$ random links going out

of the current neighbor set. So, w.h.p. the new neighbor set has size at least $min\{n^\mu, \gamma|S|\}$. We show that before reaching size n^μ, this process in expanding neighborhoods dominates a standard branching process with growth rate $\gamma > 1$. We omit the details which use a Chernoff-bound argument similar to the one used in [29].

Part b is a corollary of part a), using a contracted graph which is $(\frac{1}{p\xi} + \epsilon)$-heavy.

C.1 Proof of theorem 4.2. Here, we need to use the extended model $G = \mathcal{NAN}(H, \tau, p)$ as defined in §A.

Theorem 4.2. *Consider $G = \mathcal{NAN}(H, \tau, p)$ with n nodes. If there exist constants $\mu, \xi \in (0, 1)$ such that τ is (μ,ξ)-expansion and $(1 + \mu - \epsilon_1)$-fair and $\mathcal{H}$ is $(\frac{1}{p\xi} + \epsilon_2)$-heavy some for constants $\epsilon_1, \epsilon_2 > 0$, then G almost surely has a giant connected component with size $n(1 - o(1))$ and diameter $O(\log n)$.*

Proof. First, consider the connected case. Consider a random graph $G = \mathcal{NAN}(H, \tau, p)$ for connected base graphs $H(V, W, E)$. We now use the intuition that node u is assigned a fixed number of *seeds* W_u, where each seed gives birth to a random link with probability p. Choose $\hat{p} \in (0, 1)$ such that $\frac{1}{p\xi} + \epsilon_2 > \frac{1}{\hat{p}p\xi} + \frac{\epsilon_2}{2}$. As mentioned above, our idea is to split the weights and hence, we generate G in two stages accordingly, where only the first part of weights is in the early stage. In order to split the weights, in the first stage we do this random pre-processing: each seed can be used in this stage with probability $\hat{p}$, otherwise it needs to wait for the second stage (but in both cases, it may also fail to generate a random link with probability p). Let G_1 be the graph obtained after the first state; clearly $G_1 = \mathcal{NAN}(H, \tau, \hat{p}p)$ and is $(\frac{1}{\hat{p}p\xi} + \epsilon_2/2)$-dense.[7]

We now can assume $p = 1$ without loss of generality, and consider $G_1 = \mathcal{NAN}(H, \tau, \hat{p}$ which is $(\frac{1}{\hat{p}\xi} + \epsilon_2/2)$-dense. We partition V into disjoint subsets $\{V_i\}_{i=1}^m$ (where $V = \cup_{i=1}^m)V_i$ as follows. From lemma 4.1, there exists a constant c such that for any node u with high probability, $|S_{c \log n}(u)| \geq n^\mu$. Use metric function d as the shortest path distance (using number of arcs) and let $X = \{u_1, u_2, \ldots, u_k\}$ be a $(2c \log n)$-net in metric space (V, d). We partition V into disjoint subsets $\{A_i\}_{i=1}^k$, $|V| = \sum_{i=1}^k |A_i|$, by putting any node $v \in V$ into A_i if u_i (among these k center nodes in X) is the closest to v (handle ties arbitrarily). Note that, for any A_i, almost surely $|A_i| \geq n^\mu$, and any two nodes in A_i are at most distance $4c \log n$ apart. For any A_i with $|A_i| > n^\mu$, we chop it into pieces of size n^μ and possibly one of size $< n^\mu$. Now our partition $\{V_i\}_{i=1}^m$ is the collection of all the full or chopped-up pieces from the A_i's.

Consider the V_i's which have size n^μ. Clearly, (with high probability) the union, V', of these is at least a constant fraction of V. We now go to the second stage and release all the remaining seeds which we hold off in the first stage (these occupy an expected $(1-\hat{p})$ fraction of the whole). By doing so, clearly, each V_i' has $\Omega(n^\mu)$ fresh weight, which are now used for our 'connecting pieces'. Apply the connecting lemma to $V' = \cup V_i'$, by having: $m = \theta(n^\mu), \hat{n} = \theta(n^{1-\mu}), m_\tau = \Omega(n^{-(1+\mu-\epsilon_1)})$, and hence, $z\hat{n}mm_\tau = \Omega(n^{\epsilon_1}) = \Omega(\hat{n}^{\frac{\epsilon_1}{1-\mu}})$. Thus, with $VHP(\hat{n}^{\frac{\epsilon_1}{1-\mu}})$, i.e. $VHP(n^{\epsilon_1})$, there exists a (giant) connected component with node set V' and diameter $O(\log n)$. Note that the other pieces from $V - V'$ are also linked to this giant component by $O(\log n)$ links; so, if it is connected the whole graph has expected diameter $O(\log n)$.

In the unconnected case ($d(u, v) = \infty$ for some u, v), we can use the $(2c \log n)$-net above, however a similar construction can still be used to construct the A_i in a greedy fashion: start with $X = \emptyset, Y = V$, and choose an arbitrary $u_0 \in Y$ then set $X = X + \{u_0\}, A_0 = N_{u_0}(r), Y = Y - A_0$, and choose an arbitrary $u_1 \in Y$ then set $X = X + \{u_1\}, A_1 = N_{u_1}(r), Y = Y - A_1 \ldots$, and so on $\ldots$ until $Y = \emptyset$. Note that the giant component will include most nodes, except some isolated small component - 'islands' (with size $o(\log n)$). That is, almost all shortest paths have length only $O(\log n)$.

[7] Note to distinguish this two-stage generation of G from the superposition of these two random graphs: $G_1 = \mathcal{NAN}(H, \tau, \hat{p}p)$ and $G_2 = \mathcal{NAN}(H, \tau, (1 - \hat{p})p)$

Exact Analysis of the Recurrence Relations Generalized from the Tower of Hanoi*

Akihiro Matsuura†

Abstract

In this paper, we analyze the recurrence relations generalized from the Tower of Hanoi problem of the form $T(n,\alpha,\beta) = \min_{1\le t\le n}\{\alpha\, T(n-t,\alpha,\beta)+\beta\, S(t,3)\}$, where $S(t,3)=2^t-1$ is the optimal solution for the 3-peg Tower of Hanoi problem. It is shown that when α and β are natural numbers and $\alpha \ge 2$, the sequence of differences of $T(n,\alpha,\beta)$'s, i.e., $T(n,\alpha,\beta)-T(n-1,\alpha,\beta)$, consists of numbers of the form $\beta 2^i\alpha^j$ (i, $j \ge 0$) lined in the increasing order.

1 Introduction.

The Tower of Hanoi puzzle with 3 pegs was invented by Edouard Lucas in 1883 [9]. He also presented 4-peg puzzle in 1889. In 1907, Dudeney reproduced it as "The Reve's Puzzle" [3]. The original problem and its variants have not only been used as an introductory example of recursive algorithms, but have been also studied widely in computational research fields [2],[5],[7],[8],[11]-[14]. Stockmeyer's survey [13] lists more than 200 references, not included articles in psychological journals and textbooks in discrete mathematics. In the simplest case with 3 pegs and n disks, the algorithm of first moving the upper $n-1$ disks to the intermediate peg, then moving the bottom disk to the peg of destination, and finally moving the remaining $n-1$ disks to the destination, is the best possible and the total number of moves is 2^n-1. Somewhat surprisingly, for the general Tower of Hanoi problem with $k\,(\ge 4)$ pegs and n disks, the optimal solution is not known yet. The best upper bound is obtained by the algorithms by Frame [5] and Stewart [11]. Their algorithms are rediscovered many times ([12] lists them). Furthermore, in [8], Klavžar et al. have shown that seven different approaches to the multi-peg Tower of Hanoi problem, which include the ones by Frame and Stewart, are all equivalent. On the other hand, the subexponential lower bound was first proven by Szegedy [14] and it was improved by Chen et al.[2]. Since the upper bound is believed to be optimal, it is called the "presumed optimal" solution.

*Supported in part by MEXT Grant-in-Aid for Scientific Research on Priority Areas "New Horizon in Computing."

†Department of Computers and Systems Engineering, Tokyo Denki University.

The Stewart's recursive algorithm for the 4-peg Tower of Hanoi is written as follows. For $1 \le t \le n$, consider the procedures of first moving the top $n-t$ disks to the intermediate peg using the 4 pegs, moving the remaining t disks to the destination using the available 3 pegs, and then moving the $n-t$ disks to the destination with the 4 pegs. The algorithm chooses the minimum one among them. When the total number of moves is denoted by $S(n,4)$, the recurrence relation is written as

$$S(n,\ 4) = \min_{1\le t\le n}\big\{2\, S(n-t,\ 4) + S(t,3)\big\}.$$

This is solved with the difference

$$S(n,4) - S(n-1,4) = 2^{i-1}$$

for $t_{i-1} < n \le t_i$, where t_i is the triangular number, i.e., $t_i = i(i+1)/2$.

To clarify the combinatorial structures latent in this type of recurrence relation, we investigate the general recurrence relation of the form

$$T(n,\alpha,\beta) = \min_{1\le t\le n}\big\{\alpha T(n-t,\alpha,\beta) + \beta S(t,3)\big\}\ (n \ge 1),$$

$T(0,\alpha,\beta) = 0$, where α and β are arbitrary natural numbers. $S(n,4)$ is then written as $S(n,4) = T(n,2,1)$.

The main contribution of this paper is to exactly solve this relation for all natural numbers α and β. Suppose that $\{a_n\}_{n\ge 1}$ is the integer sequence which consists of numbers of the form $2^i\alpha^j$ $(i,j \ge 0)$ lined in the increasing order. Then for $\alpha \ge 2$, the difference of $T(n,\alpha,\beta)$'s is written using this sequence as

$$T(n,\alpha,\beta) - T(n-1,\alpha,\beta) = \beta a_n.$$

$T(n,\alpha,\beta)$ is then computed by summing up the differences. We note that when $\alpha = 3$, $a_n = 1,\ 2,\ 3,\ 4,\ 6,\ 8,\ 9,\ 12,\ 16,\ 18,\ \cdots$. These numbers are called "3-smooth numbers" and are explored in relation to the distribution of prime numbers [6] and new number representations [1],[4],[10].

The remaining of the paper is organized as follows: In Section 2, we state the main results. In Section 3, the proof of the main theorem is given. Some Tower of Hanoi variants are discussed in Section 4 and concluding remarks are given in Section 5. Finally, Appendix follows.

2 Main Results.

2.1 Linearity of $T(n,\alpha,\beta)$ on β. We first show that $T(n,\alpha,\beta)$ is linear on β.

THEOREM 2.1. *For any natural numbers α and β, $T(n,\alpha,\beta)$ is linear on β. Namely,*

$$T(n,\alpha,\beta) = \beta T(n,\alpha,1)$$

holds.

Proof. By induction on n. When $n = 0$, $T(0,\alpha,\beta) = 0 = \beta T(0,\alpha,1)$. Therefore, the equality holds.

Next, suppose that for $n = k$, the equality holds. By the definition of $T(n,\alpha,\beta)$ and by the assumption of induction,

$$\begin{aligned} & T(k+1,\alpha,\beta) \\ = & \min_{1\le t\le k+1}\{\alpha T(k+1-t,\alpha,\beta) + \beta S(t,3)\} \\ = & \min_{1\le t\le k+1}\{\alpha\beta T(k+1-t,\alpha,1) + \beta S(t,3)\} \\ = & \beta \min_{1\le t\le k+1}\{\alpha T(k+1-t,\alpha,1) + S(t,3)\} \\ = & \beta T(k+1,\alpha,1). \end{aligned}$$

Therefore, the linearity of $T(n,\alpha,\beta)$ also holds for $n = k+1$.

This completes the proof of Theorem 2.1. □

We note that the linearity of $T(n,\alpha,\beta)$ also holds for any real number β.

2.2 Properties of $\Delta T(n,\alpha,1)$ and $T(n,\alpha,1)$. Owing to Theorem 2.1, it is enough to compute $T(n,\alpha,1)$ instead of $T(n,\alpha,\beta)$. We consider the following recurrence relation for $T(n,\alpha,1)$:

$$T(n,\alpha,1) = \min_{1\le t\le n}\{\alpha T(n-t,\alpha,1) + S(t,3)\} \ (n \ge 1),$$

$$T(0,\alpha,1) = 0.$$

Tables 1 and 2 show the values for $T(n,3,1)$ and $T(n,4,1)$ up to $n \le 9$. In the tables, $t_{\min}$ is the value of the argument with which the right-hand side of the recurrence relation takes the minimum and $\Delta T(n,\alpha,1)$ is the differences of $T(n,\alpha,1)$'s.

When $\alpha = 3$, we observe that all the numbers of the sequence $\{2^i 3^j\}_{i,j\ge 0}$ appear in the increasing order as the differences of $T(n,3,1)$'s. When $\alpha = 4$, at some n's, $T(n,4,1)$ takes the minimum at two values of $t_{\min}$, which is essentially different from the case $\alpha = 3$. For clarifying the characteristics of $\Delta T(n,\alpha,1)$'s, we define a set of number sequences as follows. Let p and q be any natural numbers and let $\{a_n\}_{n\ge 1}$ be the sequence of numbers of the form $p^i q^j$ $(i,j \ge 0)$ which are lined in the increasing order. (Note that when $q = p^l$ for some integer l, $p^i q^j$'s such that $p^i q^j = p^{i'} q^{j'}$ and $(i,j) \ne (i',j')$ appear successively.) Then the sequence of differences of $T(n,\alpha,1)$'s is shown to be exactly of this form $\{p^i q^j\}$. Namely, we show the following theorem.

Table 1: The values of $t_{\min}$, $T(n,3,1)$, and $\Delta T(n,3,1)$

n	1	2	3	4	5	6	7	8	9
$t_{\min}$	1	2	2	3	3	4	4	4	5
$T(n,3,1)$	1	3	6	10	16	24	33	45	61
ΔT	1	2	3	4	6	8	9	12	16

Table 2: The values of $t_{\min}$, $T(n,4,1)$, and $\Delta T(n,4,1)$

n	1	2	3	4	5	6	7	8	9
$t_{\min}$	1	2	2,3	3	3,4	4	4,5	4,5	5
$T(n,4,1)$	1	3	7	11	19	27	43	59	75
ΔT	1	2	4	4	8	8	16	16	16

THEOREM 2.2. *Let α be a natural number and let $\{a_n\}_{n\ge 1}$ be the number sequence which consists of numbers of the form $2^i\alpha^j$ $(i,j \ge 0)$ lined in the increasing order. Then for $n \ge 1$, the difference of $T(n,\alpha,1)$'s is written as follows.*

$$T(n,\alpha,1) - T(n-1,\alpha,1) = \begin{cases} 1 & (\alpha = 1) \\ a_n & (\alpha \ge 2) \end{cases}$$

Combining Theorems 2.1 and 2.2 leads to the following corollary.

COROLLARY 2.1. *Under the same condition with Theorem 2.2, $T(n,\alpha,\beta)$ is computed as follows.*

$$T(n,\alpha,\beta) = \begin{cases} \beta n & (\alpha = 1,\ n \ge 0) \\ \beta \sum_{i=1}^{n} a_i & (\alpha \ge 2,\ n \ge 1) \end{cases}$$

3 Proof of Theorem 2.2.

When $\alpha = 1$, for any n, $\min_{1\le t\le n}\{\alpha T(n-t,\alpha,1) + S(t,3)\}$ takes the minimum at $t = 1$ with $T(n,1,1) = T(n-1,1,1) + S(1,3) = T(n-1,1,1) + 1$. Therefore, $T(n,1,1) - T(n-1,1,1) = 1$ holds.

When $\alpha \ge 2$, the proof is divided into the following two cases: When α is not of the form 2^l for any integer $l \ge 1$ (Case 1); and otherwise (Case 2).

Case 1. We proceed by induction on n.

When $n = 0$, since $T(0,\alpha,1) = 0$ and $T(1,\alpha,1) = \alpha T(0,\alpha,1) + S(1,3) = 0 + (2^1 - 1) = 1$, $T(1,\alpha,1) - T(0,\alpha,1) = 1$. On the other hand, $a_1 = 2^0\alpha^0 = 1$. Therefore, $T(1,\alpha,1) - T(0,\alpha,1) = a_1$ holds.

When $n \ge 1$, for $i \ge 0$, let k_i be the integer such that $a_{k_i} = 2^i$. We assume that the following equation

holds up to k_i.

(3.1) $T(n,\alpha,1) - T(n-1,\alpha,1) = a_n \ (1 \le n \le k_i)$

We extend this equation for n's such that $k_i + 1 \le n \le k_{i+1}$. For brevity, define $T_{n,t} := \alpha\, T(n-t,\alpha,1) + S(t,3)$. Then $T(n,\alpha,1) = \min_{1\le t\le n}\{T_{n,t}\}$.

Now we clarify with which argument $T_{n,t}$ is minimized.

LEMMA 3.1. *Under the assumption of the induction, the following statements hold.*

(i) When $k_i \le n \le k_{i+1} - 1$, $T(n,\alpha,1) = \min_{1\le t\le n}\{T_{n,t}\}$ takes the minimum at $t = i+1$.

(ii) When $n = k_{i+1}$, $T(n,\alpha,1) = \min_{1\le t\le n}\{T_{n,t}\}$ takes the minimum at $t = i+2$.

The next lemma on the sequence $\{a_n\} = \{p^i q^j\}$ plays the crucial role to prove Lemma 3.1 and Theorem 2.2.

LEMMA 3.2. *Let p and q be any natural numbers such that $q \ge p \ge 2$ and let $\{a_n\}_{n\ge1}$ be the sequence with numbers of the form $p^i q^j$ $(i, j \ge 0)$ lined in the increasing order. Then the following statements hold.*

(i) When $q \ne p^l$ for any integer l, for any n such that $p^i < a_n < p^{i+1}$, $a_n = q a_{n-(i+1)}$.

(ii) When $q = p^l$ for some integer l, for any i and n such that $a_n = p^i$, $a_{n+1} = q a_{n-i}$.

A proof of Lemma 3.2 is given in Appendix. Throughout this section, Lemma 3.2 is used with parameter $(p,\, q) = (2,\, \alpha)$.

Proof of Lemma 3.1. The difference $T_{n,t+1} - T_{n,t}$ is computed as follows.

$$\begin{aligned} & T_{n,t+1} - T_{n,t} \\ = \ & \{\alpha\, T(n-(t+1),\alpha,1) + S(t+1,3)\} \\ & - \{\alpha\, T(n-t,\alpha,1) + S(t,3)\} \\ = \ & -\alpha\{T(n-t,\alpha,1) - T(n-t-1,\alpha,1)\} \\ & + (2^{t+1}-1) - (2^t - 1) \\ = \ & -\alpha a_{n-t} + 2^t \quad \text{(by Assumption (3.1))}. \end{aligned} \tag{3.2}$$

(i) When $k_i \le n \le k_{i+1} - 1$, we first show that for $t < i+1$, $T_{n,t}$ is monotonically decreasing. At (3.2), when $t < i+1$, both $-a_{n-t}$ and 2^t take the maximums at $t = i$. Therefore,

$$\begin{aligned} T_{n,t+1} - T_{n,t} &\le -\alpha\, a_{n-i} + 2^i \\ &< -\alpha a_{k_i - i} + 2^i \ (\text{since } k_i \le n) \\ &= -a_{k_i+1} + a_{k_i} \ (\text{by Lemma 3.2(i)}) \\ &< 0. \end{aligned}$$

Thus, $T_{n,t}$ is monotonically decreasing when $t < i+1$.

When $t \ge i+1$, both a_{n-t} and 2^t take the minimums at $t = i+1$. Therefore,

$$\begin{aligned} T_{n,t+1} - T_{n,t} &\ge -\alpha a_{n-(i+1)} + 2^{i+1} \\ &\ge -\alpha a_{k_{i+1}-1-(i+1)} + a_{k_{i+1}} \\ &\quad (\text{since } n \le k_{i+1} - 1) \\ &= -a_{k_{i+1}-1} + a_{k_{i+1}} \ (\text{by Lemma 3.2(i)}) \\ &> 0. \end{aligned}$$

Thus, $T_{n,t}$ is monotonically increasing when $t \ge i+1$. Consequently, when $k_i \le n \le k_{i+1} - 1$, $T_{n,t}$ takes the minimum at $t = i+1$.

(ii) When $n = k_{i+1}$, the argument is exactly the same with the case $n = k_i$ in (i). So, $T_{k_{i+1},t}$ takes the minimum at $t = i+2$.

This completes the proof of Lemma 3.1. □

Now we are ready to show Case 1 of Theorem 2.2. It is further divided into two subcases: When $k_i + 1 \le n \le k_{i+1} - 1$ (Case 1-1); and when $n = k_{i+1}$ (Case 1-2).

Case 1-1. By Lemmas 3.1 and 3.2, $T(n,\alpha,1) - T(n-1,\alpha,1)$ is computed for $k_i + 1 \le n \le k_{i+1} - 1$ as follows.

$$\begin{aligned} & T(n,\alpha,1) - T(n-1,\alpha,1) \\ = \ & T_{n,i+1} - T_{n-1,i+1} \\ = \ & \alpha\{T(n-(i+1),\alpha,1) - T(n-1-(i+1),\alpha,1)\} \\ & + S(i+1,3) - S(i+1,3) \\ = \ & \alpha a_{n-(i+1)} \quad \text{(by Assumption (3.1))} \\ = \ & a_n \quad \text{(by Lemma 3.2(i))}. \end{aligned}$$

Thus, Case 1-1 is shown.

Case 1-2. When $n = k_{i+1}$, we should prove $T(k_{i+1},\alpha,1) - T(k_{i+1}-1,\alpha,1) = a_{k_{i+1}}$ $(= 2^{i+1})$. By Lemma 3.1, $T(k_{i+1},\alpha,1)$ and $T(k_{i+1}-1,\alpha,1)$ take the minimums at $t = i+2$ and $t = i+1$, respectively. Therefore,

$$\begin{aligned} & T(k_{i+1},\alpha,1) - T(k_{i+1}-1,\alpha,1) \\ = \ & T_{k_{i+1},i+2} - T_{k_{i+1}-1,i+1} \\ = \ & \alpha\{T(k_{i+1}-(i+2),\alpha,1) - T(k_{i+1}-1-(i+1), \\ & \alpha,1)\} + S(i+2,3) - S(i+1,3) \\ = \ & (2^{i+2}-1) - (2^{i+1}-1) = 2^{i+1}. \end{aligned}$$

Thus, Case 1-2 is shown and the proof for Case 1 is completed.

Case 2. Now $\alpha = 2^l$ for some integer $l \ge 1$. Similarly to Case 1, we proceed by induction on n. For $i \ge 0$, let k_i be the largest index n such that $a_n = 2^i$.

When $n = 0$, the proof is exactly the same with Case 1.

When $n \geq 1$, we assume that the following equation holds up to k_i.

$$T(n,\alpha,1) - T(n-1,\alpha,1) = a_n \quad (1 \leq n \leq k_i)$$

We extend this equation up to $k_i + 1 \leq n \leq k_{i+1}$, i.e., for n's such that $a_n = 2^{i+1}$. Similarly to Lemma 3.1, we clarify with which argument $T_{n,t}$ is minimized.

LEMMA 3.3. *Under the assumption of the induction, the following statements hold.*

(i) When $n = k_i$, $T(n,\alpha,1) = \min_{1\leq t\leq n}\{T_{n,t}\}$ takes the minimum at $t = i+1$.

(ii) When $k_i + 1 \leq n \leq k_{i+1} - 1$, $T(n,\alpha,1) = \min_{1\leq t\leq n}\{T_{n,t}\}$ takes the minimum at $t = i+1,\ i+2$.

(iii) When $n = k_{i+1}$, $T(n,\alpha,1) = \min_{1\leq t\leq n}\{T_{n,t}\}$ takes the minimum at $t = i+2$.

Proof. Similarly to Lemma 3.1, we compute the difference $T_{n,t+1} - T_{n,t} = -\alpha a_{n-t} + 2^t$.

(i) When $n = k_i$, we first show that when $t < i+1$, $T_{n,t}$ is monotonically decreasing. When $t < i+1$, again both $-a_{n-t}$ and 2^t take the maximums at $t = i$. Therefore,

$$\begin{aligned} T_{k_i,t+1} - T_{k_i,t} &\leq -\alpha\, a_{k_i-i} + 2^i \\ &= -a_{k_i+1} + a_{k_i} \text{ (by Lemma 3.2(ii))} \\ &< 0. \end{aligned}$$

Thus, $T_{k_i,t}$ is monotonically decreasing when $t < i+1$.

When $t \geq i+1$, both $-a_{n-t}$ and 2^t take the minimums at $t = i+1$. Therefore,

$$\begin{aligned} T_{k_i,t+1} - T_{k_i,t} &\geq -\alpha a_{k_i-(i+1)} + 2^{i+1} \\ &> -\alpha a_{k_i-i} + 2^{i+1} \\ &= -a_{k_i} + 2^{i+1} \text{ (by Lemma 3.2(ii))} \\ &= -2^i + 2^{i+1} > 0. \end{aligned}$$

Thus, $T_{k_i,t}$ is monotonically increasing when $t \geq i+1$. In all, when $n = k_i$, $T_{k_i,t}$ takes the minimum at $t = i+1$.

(ii) When $k_i + 1 \leq n \leq k_{i+1} - 1$, we note that a_n is equal to 2^{i+1} constantly due to the definition of k_i. When $t < i+1$, both a_{n-t} and 2^t take the maximums at $t = i$. Therefore,

$$\begin{aligned} T_{n,t+1} - T_{n,t} &\leq -\alpha\, a_{n-i} + 2^i \\ &= -a_{n+1} + 2^i \text{ (by Lemma 3.2(ii))} \\ &< -a_{k_i} + 2^i = 0. \end{aligned}$$

Thus, $T_{n,t}$ is monotonically decreasing when $t < i+1$.

When $t = i+1$, $T_{n,t+1} - T_{n,t}$ is computed as

$$\begin{aligned} T_{n,i+2} - T_{n,i+1} &= -\alpha a_{n-(i+1)} + 2^{i+1} \\ &= -\alpha a_{(n-1)-i} + 2^{i+1} \\ &= -a_n + 2^{i+1} \text{ (by Lemma 3.2(ii))} \\ &= 0. \end{aligned}$$

Therefore, $T_{n,i+2} = T_{n,i+1}$ holds.

When $t > i+1$, both $-a_{n-t}$ and 2^t take the minimums at $t = i+2$. Therefore,

$$\begin{aligned} T_{n,t+1} - T_{n,t} &\geq -\alpha a_{n-(i+2)} + 2^{i+2} \\ &> -\alpha a_{n-i} + 2^{i+2} \\ &= -a_{n+1} + 2^{i+2} \text{ (by Lemma 3.2(ii))} \\ &> 0. \end{aligned}$$

Thus, $T_{n,t}$ is monotonically increasing when $t > i+1$. In all, when $k_i + 1 \leq n \leq k_{i+1} - 1$, $T_{k_i,t}$ takes the minimum at $t = i+1,\ i+2$.

(iii) When $n = k_{i+1}$, the proof is the same with case (i).

This completes the proof of Lemma 3.3. □

Now we are ready to prove Case 2 of Theorem 2.2, i.e., $T(n,\alpha,1) - T(n-1,\alpha,1) = a_n$ for $k_i + 1 \leq n \leq k_{i+1}$. In this case, by Lemma 3.3(i), (ii), and (iii), we observe that $T_{n,t}$ takes the minimum at least at $t = i+2$ and $T_{n-1,t}$ takes the minimum at least at $t = i+1$. (For all of the three cases, we choose such common arguments to simplify the computation.) Therefore, for $k_i + 1 \leq n \leq k_{i+1}$,

$$\begin{aligned} & T(n,\alpha,1) - T(n-1,\alpha,1) \\ = \ & T_{n,i+2} - T_{n-1,i+1} \\ = \ & \alpha\{T(n-(i+2),\alpha,1) - T(n-1-(i+1),\alpha,1)\} \\ & +(2^{i+2}-1) - (2^{i+1}-1) \\ = \ & 2^{i+1}. \end{aligned}$$

Therefore, the proof for Case 2 is shown.

This completes the proof of Theorem 2.2. □

4 Tower of Hanoi Variants on Graphs.

One of the motivation for considering the recurrence relations for $T(n,\alpha,\beta)$ is because they appear in some variants of the Tower of Hanoi problem. For example, we consider the Tower of Hanoi problem on the graphs in Fig. 1, where pegs are located on all of the vertices and disks are moved only through the edges. The objective for the graph in Fig. 1(a) (and (b), resp.) is to move all the n disks from A to C (and A to B, resp.). Then these problems admit algorithms with the following recurrence relations, respectively.

$$\begin{aligned} T_1(n,3,2) &= 3\,T_1(n-1,3,2) + 2 \quad (n \geq 2) \\ T_1(0,3,2) &= 0, \quad T_1(1.3.2) = 1 \end{aligned}$$

$$T_2(n,3,1) = \min_{1\leq t\leq n}\{3\,T_2(n-t,3,1) + S(t,3)\}$$

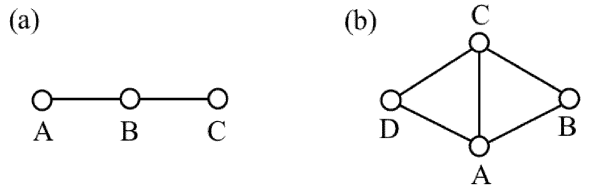


Figure 1: Variants of the Tower of Hanoi problem on graphs.

$$T_2(0,3,1) = 0$$

Therefore, the analysis of the recurrence relations for $T(n,\alpha,\beta)$ could be used for these types of Tower of Hanoi variants.

5 Concluding Remarks.

We made exact analysis of the recurrence relations generalized from the Tower of Hanoi problem. The differences of $T(n,\alpha,\beta)$'s had unexpectedly simple form such as $\{2^i\alpha^j\}$. It has to be noted that the results of this paper are not the one to improve the bounds of the original multi-peg Tower of Hanoi problem, rather, the contribution should lie on clarifying the combinatorial structures in the set of recurrence relations generalized from the Stewart's algorithm. Relations with number theory, especially with smooth numbers and the properties of the sequence $\{p^iq^j\}$ should be further explored.

References

[1] R. Blecksmith, M. McCallum, and J.L. Selfridge, *3-smooth representations of integers*, Amer. Math. Monthly, 105 (Jun.-Jul., 1998), pp. 529–543.

[2] X. Chen and J. Shen, *On the Frame-Stewart conjecture about the Towers of Hanoi*, SIAM J. Comput., 33 (2004), pp. 584–589.

[3] H.E. Dudeney, The Reve's Puzzle, *The Canterbury Puzzles (and Other Curious Problems)*, Thomas Nelson and Sons, Ltd., London, 1907.

[4] P. Erdős, *Problem Q814*, Math. Magazine, 67 (1994), pp. 67, 74.

[5] J.S. Frame, *Solution to advanced problem 3918*, Amer. Math. Monthly, 48 (1941), pp. 216–217.

[6] G.H. Hardy and S. Ramanujan, *On the normal number of prime factors of a number n*, Quart. J. Math. Oxford Ser., 48 (1917), pp. 76–92.

[7] S. Klavžar and U. Milutinović, *Simple explicit formula for the Frame-Stewart numbers*, Ann. Combinat., 6 (2002), pp. 157–167.

[8] S. Klavžar, U. Milutinović, and C. Petr, *On the Frame-Stewart algorithm for the multi-peg Tower of Hanoi problem*, Discr. Math., 120 (2002), pp. 141–157.

[9] E. Lucas, *Récréation Mathématiques*, Vol. III, Gauthier-Villars, Paris, 1893.

[10] N.J.A. Sloane, *The On-Line Encyclopedia of Integer Sequences*, http://www.research.att.com/~njas/sequences, 1996-2005.

[11] B.M. Stewart, *Solution to advanced problem 3918*, Amer. Math. Monthly, 48 (1941), pp. 217–219.

[12] P.K. Stockmeyer, *Variations on the four-post Tower of Hanoi puzzle*, Congress. Numer., 102 (1994), pp. 3–12.

[13] P.K. Stockmeyer, *The Tower of Hanoi: A bibliography*, manuscript. Available via http://www.cs.wm.edu/~pkstoc/h_papers.html, 1997-2005.

[14] M. Szegedy, *In how many steps the k peg version of the Towers of Hanoi game can be solved?*, Proc. of STACS 99, LNCS 1563 (1999), pp. 356-361.

A Proof of Lemma 3.2.

In the proof of Lemma 3.2, we use the following lemma.

LEMMA A.1. *Let p and q be natural numbers such that $q \geq p \geq 2$ and let $\{a_n\}_{n\geq 1}$ be the sequence with numbers of the form p^iq^j ($i,\ j \geq 0$) lined in the increasing order. Then the following statements hold.*

(i) When $q \neq p^l$ for any integer l, for any integer $j \geq 0$, $\left|\{a_n \,|\, q^j < a_n < q^{j+1}\}\right| = \max\{i \,|\, i \in \mathbf{N},\ p^i < q^{j+1}\}$.

(ii) When $q = p^l$ for some integer l, for any integers j and k such that $j \geq 0$ and $0 \leq k \leq l-1$,

$$|\{a_n \,|\, a_n = p^kq^j = p^{jl+k}\}| = j+1.$$

Proof of Lemma A.1. (i) By induction on j. We first define $I_j := \{a_n \,|\, q^j < a_n < q^{j+1}\}$ and $i_j := \max\{i \,|\, i \in \mathbf{N},\ p^i < q^{j+1}\}$.

When $j = 0$, $I_0 = \{a_n \,|\, 1 < a_n < q\} = \{i \,|\, 1 < p^i < q\}$. Therefore, $|I_0| = i_0$.

Next, assume that $|I_j| = i_j$. Since $I_{j+1} = \{a_n \,|\, q^{j+1} < a_n < q^{j+2}\}$ is the union of the two sets $\{qa_n \,|\, a_n \in I_j\}$ and $\{p^i \,|\, i \in \mathbf{N},\ q^{j+1} < p^i < q^{j+2}\}$,

$$\begin{aligned}
|I_{j+1}| &= \left|\{qa_n \,|\, a_n \in I_j\}\right| + \left|\{p^i \,|\, q^{j+1} < a_k < q^{j+2}\}\right| \\
&= |I_j| + \left|\{p^i \,|\, i \in \mathbf{N},\ q^{j+1} < p^i < q^{j+2}\}\right| \\
&= \max\{i \,|\, i \in \mathbf{N},\ p^i < q^{j+1}\} + \\
&\quad \left|\{p^i \,|\, q^{j+1} < p^i < q^{j+2}\}\right| \\
&= \left|\{p^i \,|\, i \in \mathbf{N},\ p^1 < p^i < q^{j+2}\}\right| \\
&= i_{j+1}.
\end{aligned}$$

Therefore, (i) is proven.

(ii) In this case, the numbers in the sequence $\{p^iq^j\}$ are written as

$$\begin{aligned}
p^iq^j &= 1,\ p,\ p^2,\ \cdots,\ p^{l-1}, \\
&\quad p^l,\ p^l,\ p^{l+1},\ p^{l+1},\ \cdots,\ p^{2l-1},\ p^{2l-1}, \\
&\quad \cdots \\
&\quad p^{jl},\ p^{jl},\ \cdots,\ p^{jl+k},\ \cdots,\ p^{(j+1)l-1}, \\
&\quad \cdots.
\end{aligned}$$

For any k such that $0 \le k \le l-1$, since there are $j+1$ ways to compute p^{jl+k} using p and $q = p^l$, p^{jl+k} appears $j+1$ times. This implied that

$$|\{a_n \,|\, a_n = p^k q^j = p^{jl+k}\}| = j+1.$$

This completes the proof of Lemma A.1. □

Proof of Lemma 3.2. (i) By induction on i.

When $i = 1$, since $q \ge p$, there is no a_n such that $p^0 = 1 < a_n < p^1 = p$. Therefore, the equality in (i) holds.

Next, assume that for all n such that $p^{s-1} < a_n < p^s$, $1 \le s \le i$, $a_n = qa_{n-s}$ holds. We show that for any N such that $p^i < a_N < p^{i+1}$, $a_N = qa_{N-(i+1)}$ holds. We divide into two cases: When $a_N = q^j$ for some integer j (Case 1); and otherwise (Case 2).

Case 1. When $a_N = q^j$, by Lemma A.1(i), there exist i a_n's between q^{j-1} and $a_N = q^j$. So, $a_{N-(i+1)} = q^{j-1}$. Therefore, $a_N = qa_{N-(i+1)}$ holds.

Case 2. When $a_N \ne q^j$ for any integer $j \ge 0$, a_N is divisible by p. So, there exists M such that $p^{i-1} < a_M < p^i$ and $a_N = pa_M$. Then by the assumption of the induction, $a_M = qa_{M-i}$. Therefore, $a_N = pa_M = p(qa_{M-i}) = q(pa_{M-i})$. To prove $a_N = qa_{N-(i+1)}$, it is enough to show that $pa_{M-i} = a_{N-(i+1)}$.

By the definition of $\{a_n\}$ and since $p^i \in \{a_n \,|\, a_M < a_n < a_N\}$,

$$\begin{aligned} & \left|\{a_n \,|\, a_M/q < a_n < a_N/q\}\right| \\ = \; & \left|\{a_n \,|\, a_M < a_n < a_N\}\right| - 1 \\ = \; & (N-M-1)-1 \\ = \; & N-M-2. \end{aligned}$$

Using this with $a_M/q = a_{M-i}$, a_N/q is computed as

$$a_N/q = a_{(M-i)+(N-M-2)+1} = a_{N-(i+1)}.$$

Therefore, $a_N = qa_{N-(i+1)}$ holds.

This completes the proof for (i).

(ii) Suppose that $q = p^l$ for some integer $l \ge 1$. For $j \ge 0$ and $0 \le k \le l-1$, let $G_{j,k}$ be the subsequence of $\{a_n\}$ which consists of p^{jl+k}'s. By Lemma A.1(ii), note that $|G_{j,k}| = j+1$. Furthermore, for $j \ge 0$, let G_j be the union of $G_{j,k}$'s for $0 \le k \le l-1$, i.e.,

$$\begin{aligned} G_j &= \bigcup_{k=0}^{l-1} G_{j,k} \\ &= \{p^{jl},\ \cdots,\ p^{jl},\ \cdots,\ p^{jl+k},\ \cdots, p^{jl+k}, \\ &\quad \cdots,\ p^{(j+1)l-1},\ \cdots,\ p^{(j+1)l-1}\}. \end{aligned}$$

We note that in this notation, same numbers are not identified as opposed to the usual definition of a set.

Now, it is enough to show that for any $a_n = p^{jl+k}$ in G_j, $a_{n+1} = qa_{n-(jl+k)}$.

Let $a_n = p^{jl+k}$ be any elements in G_j with $j \ge 0$ and $0 \le k \le l-1$. Suppose that a_n is the tth element (p^{jl+k}) in $G_{j,k}$, where $1 \le t \le j+1$. Then n and $n-(jl+k)$ are explicitly written as follows.

$$\begin{aligned} n &= \sum_{m=1}^{j} |G_m| + (j+1)k + t \\ &= \sum_{m=1}^{j} ml + (j+1)k + t \\ &= \frac{lj(j+1)}{2} + (j+1)k + t. \end{aligned}$$

$$\begin{aligned} n-(jl+k) &= \frac{lj(j+1)}{2} + (j+1)k + t \\ &\quad - (jl+k) \\ &= \frac{l(j-1)j}{2} + jk + t \\ &= \sum_{m=1}^{j-1} |G_m| + jk + t. \end{aligned}$$

Since t is within $1 \le t \le j+1$ and $|G_{j-1,k}| = j$, the place $a_{n-(jl+k)}$ is located in $\{G_j\}$ differs in the following two cases: When $1 \le t \le j$ (Case 1); and when $t = j+1$ (Case 2). We consider each of these cases.

Case 1. When $1 \le t \le j$, $a_{n-(jl+k)}$ is the tth element in $G_{j-1,k}$. So, $a_{n-(jl+k)} = p^{(j-1)l+k}$. Therefore, we obtain

$$qa_{n-(jl+k)} = p^l p^{(j-1)l+k} = p^{jl+k} = a_{n+1}.$$

Case 2. When $t = j+1$, a_n is the last element in $G_{j,k}$. Then, $a_{n-(jl+k)}$ is the first element in $G_{j-1,k+1}$, i.e., $p^{(j-1)l+k+1}$ except for the case $a_n = p^{(j+1)l-1}$ is the last element in G_j. We consider this exceptional case later. Now since $a_{n-(jl+k)} = p^{(j-1)l+k+1}$,

$$qa_{n-(jl+k)} = p^l p^{(j-1)l+k+1} = p^{jl+k+1} = a_{n+1}.$$

Note that the last equality holds because a_n is the last p^{jl+k} in $G_{j,k}$.

We finally consider the exceptional case, that is, when $a_n = p^{(j+1)l-1}$ is the last element in G_j. In this case, $k = l-1$, so $a_{n-(jl+k)} = a_{n+1-(j+1)l}$. Since $|G_j| = (j+1)l$ and a_{n+1} is the first element in G_{j+1}, $a_{n+1-(j+1)l}$ is the first element in G_j, i.e., p^{jl}. Therefore,

$$qa_{n-(jl+k)} = p^l p^{jl} = p^{(j+1)l} = a_{n+1}.$$

This completes the proof of Lemma 3.2. □

Generating Random Derangements*

Conrado Martínez† Alois Panholzer‡ Helmut Prodinger§

Abstract

In this short note, we propose a simple and efficient algorithm to generaterandom *derangements*, that is, permutations without fixed points. We discuss the algorithm correctness and its performance and compare it to other alternatives. We find that the algorithm has expected linear complexity, works in-place with little additional auxiliary memory and qualitatively behaves like the well-known Fisher-Yates shuffle for random permutations or Sattolo's algorithm for random cyclic permutations.

1 Introduction

Derangements are permutations without fixed points, i.e., a permutation $\pi : [1..n] \to [1..n]$ is a derangement if and only if it does not exist a value i, $1 \le i \le n$, such that $\pi(i) = i$. In other words, for the classical representation of a permutation as a set of cycles, a derangement is a permutation which does not contain any singleton cycle.

Derangements were first introduced by Pierre Raymond de Montmort. Their enumeration was also solved by de Montmort and about the same time by Nicholas Bernoulli [1]. Later L. Euler would give an independent proof (see [3, 4]).

The number D_n of derangements of size n is given by

(1.1)
$$D_n = n!\cdot\left[\frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + \frac{(-1)^n}{n!}\right] = \left\lfloor \frac{n!+1}{e} \right\rfloor.$$

The number D_n is often written $!n$; these numbers are also called *subfactorials* [7, 8] and are a particular case of the so-called *rencontres numbers* [15]. Since $!n \approx n!/e$ the probability that a random permutation of size n is a derangement quickly converges to $1/e \approx 0.36788$.

The goal of this short note is to present a new algorithm for the generation of random derangements and provide a precise analysis of its performance. Our baseline for comparison is the combination of an efficient algorithm for the generation of random permutations, like the *Fisher-Yates shuffle* [5, pp. 26–27] (also known as *Knuth shuffle*), together with the rejection method. Since the probability that a random permutation is a derangement is roughly $1/e$, the average number of times that we have to generate a random permutation until we get a derangement is e. If we measure the complexity of our algorithms as the number of random numbers[1] that we need to generate, the algorithm sketched above has average cost $e \cdot n + o(n)$. An additional drawback of the approach using the rejection method is that it needs to check whether the generated permutation is a derangement or not.

2 The Algorithm

Our basis for random derangement generation are the modern implementation of the Fisher-Yates shuffle given by Durstenfeld [2] and later popularized by Knuth [10], and Sattolo's algorithm [16]. Sattolo's algorithm is a very simple modification of the Fisher-Yates shuffle that generates random cyclic permutations, that is, permutations that consist of a single cycle. The performance of this algorithm has been throughly analyzed in [12, 13, 18].

We modify Sattolo's algorithm so that, with appropriate probability, some elements get marked and will not be moved from their position afterwards. Each of these elements is part of a different cycle, but none of these cycles will be of length 1.

The algorithm starts filling the array A with the identity permutation, and with no marked elements. The variable u records the number of unmarked elements in the subarray $A[1..i]$.

Then we start a scanning of the array, from right to left. At iteration i, if $A[i]$ was marked in some previous iteration we just jump to the next position

*The first author was supported by the Spanish Min. of Science and Technology project TIN2006-11345 (ALINEX). The second author was supported by the Austrian Science Foundation FWF, grant S9608-N13. The third author was supported by the South African Science Foundation NRF, grant 2053748.

†Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya. E-08034 Barcelona, Spain. Email: conrado at lsi dot upc dot es.

‡Institut für Diskrete Mathematik und Geometrie, Technische Universität Wien. Wiedner Hauptstraße 8-10/104. 1040 Wien, Austria. Email: Alois.Panholzer at tuwien dot ac dot at.

§Department of Mathematics, University of Stellenbosch. 7602 Stellenbosch, South Africa. Email: hproding at sun dot ac dot za.

[1] More precisely, random numbers of $\mathcal{O}(\log n)$ bits.

$i-1$. Otherwise, we choose at random one unmarked element, say $A[j]$, from $A[1..i-1]$. We use a simple loop to select such an element. Then $A[i]$ and $A[j]$ are swapped. But now, in order to be able to close the cycle to which j belongs, we might decide to mark the position j, so that no new element "joins" the cycle to which j belongs. We do so with some carefully chosen probability: we will show later that, indeed, $(u-1)D_{u-2}/D_u$ is the correct probability that will guarantee that any derangement is produced by the algorithm with the same probability. Then we move to the next element to the left. Since $A[i]$ was not marked, u must be decremented by one; furthermore, if during iteration i we decide to mark position $j < i$, then we must decrement u by an additional unit.

The algorithm just discussed is more formally presented as Algorithm 1.

Algorithm 1 Generation of random derangements.

```
1:  procedure RandomDerangement(n)
2:      for i ← 1 to n do A[i] ← i; mark[i] ← false
3:      i ← n; u ← n
4:      while u ≥ 2 do
5:          if ¬mark[i] then
6:              repeat j ← Random(1, i − 1)
7:              until ¬mark[j]
8:              A[i] ↔ A[j]
9:              p ← Uniform(0, 1)
10:             if p < (u − 1)D_{u−2}/D_u then
11:                 mark[j] ← true; u ← u − 1
12:             u ← u − 1
13:         i ← i − 1
14:     return A
```

2.1 Correctness By construction, Algorithm 1 produces always derangements. The first observation is that the elements in $A[i+1..n]$ never get involved in any further swap from iteration i downwards. Thus if a value $k > i$ has been moved to $A[1..i]$ during some previous iteration it will never be moved back to its original position. In this respect, our algorithm shares this important property with its close relatives Fisher-Yates' and Sattolo's: any given item can move some number of times to the left (i.e., to positions with lower index), but it can move to the right only once. The second observation is that a marked element is never sitting at its original position: we only mark elements once they have been swapped and then they never move again. Last but not least, as in Sattolo's algorithm, a swap involves $A[i]$ and $A[j]$ with $i \neq j$, thus we cannot create singleton cycles (compare to Fisher-Yates' algorithm, where a swap $A[i] \leftrightarrow A[i]$ in iteration i occurs with probability $1/i$).

To conclude the proof of correctness of Algorithm 1, we show that each derangement of size n has probability $1/D_n$ of being produced by the algorithm. The key identity to prove this is the recurrence for D_n:

$$D_n = (n-1)(D_{n-1} + D_{n-2}), \quad n \geq 2,$$

with $D_0 = 1$ and $D_1 = 0$.

We prove this by induction on n. Suppose $u = n = 2$. Then $A[1] = 1$ is swapped with $A[2] = 2$ and since $D_0 = D_2 = 1$, $A[1]$ is marked and we update $u := 0$, so nothing else happens and the algorithm has generated $A = [2, 1]$, the unique derangement of size 2—with probability 1.

For $u = n > 2$, the algorithm will probabilistically decide whether n is part of a cycle of two elements or not. In the former case, which has probability $(u-1)D_{u-2}/D_u$, the algorithm has choosen one of the remaining $u-1$ elements, say $A[j]$, and swaps $A[j]$ and $A[n]$ to construct the 2-cycle. Then position j is marked so that the element sitting there does not get involved in any further step of the algorithm, and we "recursively" generate a random derangement of size $u = n-2$ with the other elements. In the second case, which has probability $1 - \frac{(u-1)D_{u-2}}{D_s}$, we might think as if the algorithm has "reserved" uniformly at random one of the $u-1$ available slots, say j, and then recursively generates a random derangement of size $u = n-1$ with the remaining elements. The swap between $A[j]$ and $A[n]$ corresponds to the insertion of n into the slot we had "reserved" previously.

Thus, the probability that we generate a particular derangement is given either by

$$(u-1)\frac{D_{u-2}}{D_u}\frac{1}{u-1}\frac{1}{D_{u-2}} = \frac{1}{D_u},$$

or by

$$\left(1 - (u-1)\frac{D_{u-1}}{D_u}\right)\frac{1}{u-1}\frac{1}{D_{u-1}} = \frac{1}{D_u},$$

as we wanted to prove.

Algorithm 1 can be seen as a clever implementation of the straightforward random generation based on the recursive method (see for instance [6, 14, 17]). The recursive method needs to generate $2n$ random numbers, but it is only well suited if we represented a permutation as a set of cycles, using some linked data structure to store them.

Our algorithm is equivalent to the one stemming from the recursive method, but we have removed recursion, the derangement is stored in an array and it is generated *in-situ* without need of additional data structures other than the n bits for marks and some table to

store the D_n values (which the pure recursive method needs too). Also, if the elements which we wanted to "derange" were the numbers 1 to n, we could dispense the array of marks altogether; we could change the sign of $A[i]$ at iteration $j > i$ to indicate that it is marked, and multiply it by -1 again at iteration i (since we do not need that mark any longer).

The significant difference of our method with respect to the pure recursive method is that to find non-marked j's we are using a simple generate-and-reject loop (lines 6-7 of Algorithm 1). This loop will perform a single iteration most of the times, but it might generate many unusable values in a given round, so that the cost of our algorithm might be, in principle, very large.

3 Analysis of the performance

We measure the complexity of our algorithms by the number of times that we need to generate random numbers. Under this model, the complexity of Fisher-Yates shuffle and Sattolo's algorithm is obviously n, but the loop in lines 6 to 7 make the analysis of Algorithm 1 not entirely trivial. Additionally, the running time of any of these algorithms is clearly and directly related to the number of random numbers used.

Notice that if $u = 0$ and $i > 1$, all iterations from that point on would simply scan the leftmost i marked elements, had we not stopped the main loop of the algorithm. Hence, Algorithm 1 is equivalent to one where the outer loop stops when $i = 1$. The algorithm generates a uniform random number in $(0, 1)$ at each iteration such that the corresponding $A[i]$ is not marked (line 9 of Algorithm 1). Since the number of marked elements is the number of cycles in the derangement, the expected number of calls is $n - \mathbb{E}[C_n]$, where C_n denotes the number of cycles in a random derangement.

So we concentrate from now on in the number of random integers generated in line 6. Let us denote this number by G, and let G_i denote the number of random numbers generated during iteration i. Then we simply have

$$\mathbb{E}[G] = \sum_{i=2}^{n} \mathbb{E}[G_i]\,,$$

by linearity of expectations. On the other hand, if $A[i]$ were marked we just jump to the next iteration and $G_i = 0$. Let M_i be the indicator random variable such that $M_i = 1$ if $A[i]$ gets marked by the algorithm and $M_i = 0$ otherwise. Hence,

$$\mathbb{E}[G] = \sum_{i=2}^{n} \mathbb{E}[G_i \mid M_i = 0]\,\mathbb{P}[M_i = 0]\,.$$

Let U_i denote the value of u at the beginning of iteration i. If $A[i]$ is not marked then at iteration i we have to choose one of the $U_i - 1$ unmarked elements among the $i - 1$ elements to the left of $A[i]$. The number G_i is clearly geometrically distributed, with $(U_i - 1)/(i - 1)$ the probability of success. So we have

$$\mathbb{E}[G_i \mid M_i = 0] = \mathbb{E}\left[\frac{i-1}{U_i - 1} \mid M_i = 0\right].$$

Looking back again to Algorithm 1 we might say that we decrement u in every iteration and, in addition, we add to u some Δ_i which might be -1 (if some $A[j]$ gets marked), $+1$ (if $A[i]$ was already marked) or 0 (otherwise). That is, the dynamics of U_i is given by $U_i = U_{i+1} - 1 + \Delta_{i+1}$ if $i < n$ and $U_n = n$.

Unwinding the recursion we get

$$U_i = i + \Delta_{i+1} + \cdots + \Delta_n$$

and defining $B_i := -(\Delta_i + \cdots + \Delta_n)$ we finally obtain

$$U_i + B_{i+1} = i.$$

In other words, B_{i+1} denotes the number of marked elements in $A[1..i]$ when our algorithm reaches iteration i. The dynamics of B_i is also simple: $B_i = B_{i+1} - \Delta_i$, with $B_{n+1} = 0$. For this reason, we may also think of B_i as a "balance". Once we introduce B_i we can write $\mathbb{E}[G]$ as

$$\mathbb{E}[G] = \sum_{i=2}^{n} \mathbb{E}\left[\frac{i-1}{i-1-B_{i+1}} \mid M_i = 0\right] \mathbb{P}[M_i = 0]\,.$$

Why do we prefer to write $\mathbb{E}[G]$ in terms of B_i's rather than in terms of S_i's? To answer this question we must consider the canonical cycle representation of permutations. In this form, the cycles are listed in decreasing order of the cycle leaders, and each cycle is listed so that the leader is its first element. In order to understand better some important facts that relate Δ_i and M_i to the cycle decomposition of the generated derangement, it is useful to look at some particular example, such as the one shown in Figure 1. Each row shows the contents of the array at the beginning of the corresponding iteration, the number i of the iteration and the value of the variable u at the start of the iteration. We use a circle to indicate that a particular element is marked, and we show the usual representation of the generated derangement as a set of cycles below.

PROPOSITION 3.1. *For any derangement π of size n, and any i, $1 \le i \le n$, the following properties hold:*

1. *$M_i = 1$ if and only if i is the leader of some cycle in the generated permutation.*

Permutation												i	U_i	Δ_i	B_{i+1}
1	2	3	4	5	6	7	8	9	10	11	12	12	12	−1	0
1	2	3	4	5	6	7	12	9	10	11	8	11	10	0	1
1	2	3	4	11	6	7	12	9	10	5	8	10	9	−1	1
1	2	3	4	11	6	10	12	9	7	5	8	9	7	0	2
9	2	3	4	11	6	10	12	1	7	5	8	8	6	+1	2
9	2	3	4	11	6	10	12	1	7	5	8	7	6	+1	1
9	2	3	4	11	6	10	12	1	7	5	8	6	6	0	0
6	2	3	4	11	9	10	12	1	7	5	8	5	5	−1	0
11	2	3	4	6	9	10	12	1	7	5	8	4	3	0	1
11	2	4	3	6	9	10	12	1	7	5	8	3	2	−1	1
11	4	2	3	6	9	10	12	1	7	5	8	2	0	+1	2
11	4	2	3	6	9	10	12	1	7	5	8	1	0	+1	1

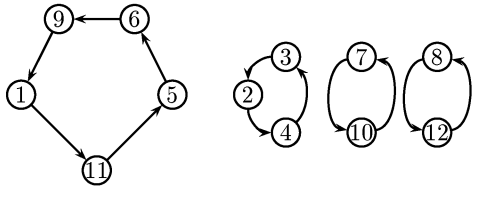


Figure 1: A sample execution of Algorithm 1 and its output.

2. $\Delta_i = 1$ *if* i *is the leader of some cycle in the permutation.*

3. $\Delta_i = -1$ *if* i *is the second smallest element of its cycle.*

4. $\Delta_i = 0$ *if* i *is neither the smallest nor the second smallest element of its cycle.*

5. $B_1 = 0$, $B_2 = 1$.

6. $B_i \leq C_n$, *where* C_n *denotes the number of cycles in a random derangement.*

7. $0 \leq B_{i+1} \leq i$.

8. *If* $M_i = 0$ *then* $B_{i+1} < i - 1$. *Moreover, for any* i, $1 \leq i \leq n$, $B_{i+1} \neq i - 1$.

Proof.

1. Suppose $M_i = 1$, that is, by definition, that $A[i]$ is marked. A moment's thought reveals that there can't be any $j < i$ such that $A[j] = i$. In other words, no element $j < i$ can become part of the cycle to which i belongs—recall that a given item can move several times to the left of the array, but only once to the right—. Conversely, if i is the leader of its cycle, then we have to "close" its cycle at iteration i, not before, not after. Suppose we mark the cycle at some iteration j (necessarily $j \neq i$), then $A[j]$ is swapped to some other place say $\ell < j$, so we mark $A[\ell]$. No element at a position smaller than ℓ can be included in the same cycle, and thus it must hold $\ell = i$. Hence, the element at position i must be marked. When the algorithm reaches position i, it skips the position and the cycle is finally "closed."

2. Trivial. Since $\Delta_i = 1$ if $M_i = 1$, the statement follows.

3. We close the cycle to which j belongs when reaching some marked element $A[j]$, so that j is the leader of that cycle. Position j was marked at some previous iteration k, hence the element u sitting at position k is moved to $A[j]$ and the element v formerly at $A[j]$ moves to position k. The cycle will be "closed" when the main loop reaches position j, but no other element may enter the cycle in the meanwhile. So what we want to proof is that k is part of the cycle, as no other value between j and k can be part of the cycle. And indeed it must be part of the cycle, as the element v that we moved to position k is part of the cycle.

4. By definition, $\Delta_i = 0$ when neither $A[i]$ is marked, nor any other element gets marked. So i is not the leader nor the second smallest element of the cycle, because otherwise at iteration i we would either mark the leader's position or would be at the leader's position.

5. Since 1 is a leader of its cycle, it must be marked when $i = 1$. Therefore the number of unmarked items U_1 in $A[1..1]$ is 0 and hence $B_2 = 1 - U_1 = 1$. As $B_1 = B_2 - \Delta_1$ and $\Delta_1 = 1$, we have $B_1 = 0$.

6. This follows from points 2.-4. of this proposition.

7. This follows directly from the identity $U_i + B_{i+1} = i$ and obvious bounds on U_i.

8. Recall that $U_i + B_{i+1} = i$. Hence $B_{i+1} = i - U_i$. If $U_i \geq 2$ then the statement is trivially true. The situation $U_i = 1$ cannot happen; otherwise, the algorithm would try to construct a derangement of size 1 and that's impossible. Incidentally, this proves that $B_{i+1} \neq i - 1$ for any i. Finally, we have only to show that happens when $U_i = 0$. This means that $A[1], \ldots, A[i]$ are marked and hence $M_j = 1$, for $1 \leq j \leq i$. I.e., $1, 2, \ldots, i$ are leaders of their respective cycles, and in particular $M_i = 1$, which contradicts the hypothesis of the proposition.

Now we can find a crude estimate of $\mathbb{E}[G]$ which is enough for all practical purposes. To begin with,

$$\begin{aligned}
\mathbb{E}[G] &= \sum_{i=2}^{n} \mathbb{E}\left[\frac{i-1}{i-1-B_{i+1}} \mid M_i = 0\right] \mathbb{P}[M_i = 0] \\
&\leq \sum_{i=2}^{n} \mathbb{E}\left[\left|\frac{i-1}{i-1-B_{i+1}}\right| \mid M_i = 0\right] \mathbb{P}[M_i = 0] \\
&\leq \sum_{i=2}^{n} \Big(\mathbb{E}\left[\left|\frac{i-1}{i-1-B_{i+1}}\right| \mid M_i = 0\right] \mathbb{P}[M_i = 0] \\
&\quad + \mathbb{E}\left[\left|\frac{i-1}{i-1-B_{i+1}}\right| \mid M_i \neq 0\right] \mathbb{P}[M_i \neq 0]\Big) \\
&= \sum_{i=2}^{n} \mathbb{E}\left[\left|\frac{i-1}{i-1-B_{i+1}}\right|\right].
\end{aligned}$$

Using the bounds given in Proposition 3.1(6-8) and conditioning in the number of cycles we have

$$\begin{aligned}
\mathbb{E}[G] &\leq \sum_{i=2}^{n} \mathbb{E}\left[\min\left\{i-1, \frac{i-1}{i-1-C_n}\right\}\right] \\
&\leq \sum_{k=1}^{\lfloor n/2 \rfloor} \mathbb{P}[C_n = k] \cdot \left[\sum_{i=1}^{k+1}(i-1) + \sum_{i=k+2}^{n} \frac{i-1}{i-1-k}\right] \\
&\leq \sum_{k=1}^{\lfloor n/2 \rfloor} \mathbb{P}[C_n = k] \cdot \left[\binom{k}{2} + n - k - 1 + kH_{n-1-k}\right] \\
&= \frac{1}{2}\,\mathbb{E}\big[C_n^2\big] + n - 1 - \mathbb{E}[C_n] + \mathcal{O}(\mathbb{E}[C_n \ln(n - C_n)]) \\
&= n + \mathcal{O}(\mathbb{E}\big[C_n^2\big]) + \mathcal{O}(\log n \cdot \mathbb{E}[C_n]),
\end{aligned}$$

where $H_n = \sum_{1 \leq j \leq n}(1/j) \sim \ln n + \mathcal{O}(1)$ denotes the n-th harmonic number.

The required results for the r.v. C_n can be obtained easily via a generating functions approach studying

$$C(z, v) = \sum_{n \geq 0} \mathbb{E}\big[v^{C_n}\big]\, D_n \frac{z^n}{n!}.$$

By using the natural decomposition of a derangement as a product of cycles of length ≥ 2, we get

$$C(z, v) = e^{\left(\log\left(\frac{1}{1-z}\right) - z\right) v} = e^{-vz} \frac{1}{(1-z)^v}.$$

Basic singularity analysis leads then to the following asymptotic expansion of the probability generating function $\mathbb{E}\big[v^{C_n}\big]$, which holds uniformly in a complex neighbourhood of $v = 1$, with an arbitrary $\epsilon > 0$:

$$\begin{aligned}
\mathbb{E}\big[v^{C_n}\big] &= \frac{n!}{D_n}[z^n]C(z, v) \\
&= \frac{e^{1-v}}{(v-1)!} e^{(v-1)\log n} \left(1 + \mathcal{O}(n^{-1+\epsilon})\right).
\end{aligned}$$

A direct application of Hwang's quasi power theorem [9] shows then that the normalized r.v. $\frac{C_n - \mathbb{E}[C_n]}{\sqrt{\mathbb{V}[C_n]}}$ converges in distribution to a standard normal distributed r.v. together with the asymptotic expansions $\mathbb{E}[C_n] = \log n + \mathcal{O}(1)$ and $\mathbb{V}[C_n] = \log n + \mathcal{O}(1)$. It follows from these results that

$$\mathbb{E}[G] \leq n + \mathcal{O}(\log^2 n)$$

Since $\mathbb{E}[G] \geq n - \mathbb{E}[C_n]$, we further have that

$$\mathbb{E}[G] = n + \mathcal{O}(\log^2 n),$$

so that the following theorem holds.

THEOREM 3.1. *The expected total number of random numbers used by Algorithm 1 to generate a random derangement of size n is $2n + \mathcal{O}(\log^2 n)$.*

This compares favorably to the average complexity of random generation based upon a straightforward implementation of the recursive method.

3.2 The number of moves Even though the number of times a particular element is swapped by Algorithm 1 has not direct impact on its performance, its analysis raises interesting mathematical challenges, as in the corresponding analysis for Sattolo's algorithm and Fisher-Yates shuffle [12, 13, 18, 11].

Let us denote by $M_{n,p}$ the number of moves of element p when generating a random derangement of length n and by $M_n := M_{n,U_n}$ the number of moves made by a random element. Here U_n is uniformly distributed on $\{1, 2, \ldots, n\}$.

First we introduce the probability generating functions for the number of moves of specific elements:

$$\varphi_{n,p}(v) := \sum_{m\geq 0} \mathbb{P}[M_{n,p} = m]\, v^m, \qquad n \geq 2, \quad 1 \leq p \leq n.$$

We obtain then the following recurrences for the functions $\varphi_{n,p}(v)$, if we additionally define $\varphi_{n,p}(v) = 0$ if $n \leq 1$ or $p < 1$ or $p > n$:

(3.2)
$$\varphi_{n,n}(v) = \frac{(n-1)D_{n-2}}{D_n} v + \frac{D_{n-1}}{D_n} v \sum_{k=1}^{n-1} \varphi_{n-1,k}(v), n \geq 2,$$

(3.3)
$$\varphi_{n,p}(v) = \frac{D_{n-2}}{D_n}\Big(v + (p-1)\varphi_{n-2,p-1}(v) + (n-1-p)\varphi_{n-2,p}(v)\Big) + \frac{D_{n-1}}{D_n}\big(v + (n-2)\varphi_{n-1,p}(v)\big), \quad 1 \leq p < n.$$

These recurrences are obtained when distinguishing the cases, where element n is contained in a cycle of length 2 (which appears with probability $\frac{(n-1)D_{n-2}}{D_n}$) or in a cycle of length ≥ 3 (which appears with probability $\frac{(n-1)D_{n-1}}{D_n}$).

We will use the recurrences (3.2) and (3.3) to study the r.v. M_n and thus the number of moves of a random element and introduce the probability generating function

$$\psi_n(v) := \sum_{m\geq 0} \mathbb{P}[M_n = m]\, v^m = \frac{1}{n}\sum_{k=1}^{n} \varphi_{n,k}(v)$$

and the abbreviation $\tilde{\psi}_n(v) := n\psi_n(v)$.

Equation (3.2) can then be rewritten as

$$\varphi_{n,n}(v) = \frac{(n-1)D_{n-2}}{D_n} v + \frac{D_{n-1}}{D_n} v\tilde{\psi}_{n-1}(v), \tag{3.4}$$

whereas we obtain by summation for p from 1 up to n from (3.3):

$$\tilde{\psi}_n(v) - \varphi_{n,n}(v) = v + \frac{D_{n-2}(n-1)}{D_n}\tilde{\psi}_{n-2}(v) + \frac{D_{n-1}(n-2)}{D_n}\tilde{\psi}_{n-1}(v). \tag{3.5}$$

After combining (3.4) and (3.5) this gives the following linear recurrence for $\tilde{\psi}_n(v)$:

$$\tilde{\psi}_n(v) = \left(1 + \frac{(n-1)D_{n-2}}{D_n}\right) v + \frac{(n-2+v)D_{n-1}}{D_n}\tilde{\psi}_{n-1}(v) + \frac{(n-1)D_{n-2}}{D_n}\tilde{\psi}_{n-2}(v), \quad n \geq 2, \tag{3.6}$$

with initial values $\tilde{\psi}_0(v) = \tilde{\psi}_1(v) = 0$. To treat this recurrence we introduce the bivariate generating function

$$\psi(z, v) := \sum_{n\geq 2} D_n \tilde{\psi}_n(v)\frac{z^n}{n!} \quad = \sum_{n\geq 2} D_n \psi_n(v)\frac{z^n}{(n-1)!}.$$

Recurrence (3.6) can then be translated into the following first order linear differential equation for $\psi(z, v)$:

$$(1-z)\frac{\partial}{\partial z}\psi(z, v) + (1-z-v)\psi(z, v) = \frac{vz(2-z)e^{-z}}{(1-z)^2}, \tag{3.7}$$

with initial condition $\psi(0, v) = 0$. This differential equation has the following exact solution:

$$\psi(z, v) = e^{-z}\left(1 + \frac{v}{(2-v)(1-z)^2} - \frac{2}{(2-v)(1-z)^v}\right). \tag{3.8}$$

Extracting coefficients leads then from (3.8) to the following exact solution for the probability generating function $\psi_n(v)$ (with $n \geq 2$):

$$\psi_n(v) = \frac{(n-1)!}{D_n}[z^n]\psi(z, v) = \frac{(-1)^n}{nD_n} + \frac{v}{2-v}\left(1 + \frac{1}{n} + \frac{D_{n-1}}{D_n}\right) - \frac{(n-1)!}{D_n}\frac{2}{2-v}\sum_{k=0}^{n}\frac{(-1)^k}{k!}\binom{n-k+v-1}{n-k}. \tag{3.9}$$

Additionally one easily obtains via singularity analysis the asymptotic expansion

$$\psi_n(v) = \frac{v}{2-v} + \mathcal{O}(n^{-1}) + \mathcal{O}(n^{v-2}), \tag{3.10}$$

which holds uniformly for $|v| \leq 2 - \epsilon$ and $\epsilon > 0$. Thus the sequence of probability generating functions $\psi_n(v)$ converges uniformly in a complex neighbourhood of $v = 1$ to $\frac{v}{2-v}$, which is the probability generating function of a geometric r.v. with parameter $\frac{1}{2}$. Therefore M_n converges in distribution to a geometric distribution with parameter $\frac{1}{2}$.

References

[1] P.R. de Montmort. *Essay d'analyse sur les jeux de hazard. Seconde Edition, Revue & augmentée de plusieurs Lettres.* Jacque Quillau, Paris, 1713.

[2] R. Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM*, 7(7):420, July 1964.

[3] L. Euler. Calcul de la probabilité dans le jeu de rencontre. *Memoires de l'Academie des Sciences de Berlin*, (7):255–270, 1753.

[4] L. Euler. Solutio quaestionis curiosae ex doctrina combinationum. *Memoires de l'Academie des Sciences de St. Petersbourg*, 3:57–64, 1811.

[5] R.A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research.* Oliver & Boyd, London, 3rd edition, 1948.

[6] Ph. Flajolet, P. Zimmerman, and B. Van Cutsem. A calculus for the random generation of combinatorial structures. *Theoretical Computer Science*, 132(1-2):1–35, 1994.

[7] I. Goulden and D. Jackson. *Combinatorial Enumerations.* John Wiley & Sons, 1983.

[8] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison–Wesley, 2nd edition, 1994.

[9] Hwang H.-K. On convergence rates in the central limit theorems for combinatorial structures. *European J. Combin.*, 19:329–343, 1998.

[10] D.E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms (volume 2).* Addison-Wesley, 3rd edition, 1997.

[11] G. Louchard, H. Prodinger, and S. Wagner. Joint distributions for movements of elements in Sattolo's and Fisher-Yates' algorithm. Preprint. Available at `http://www.ulb.ac.be/di/mcs/louchard/`, 2007.

[12] H.M. Mahmoud. Mixed distributions in Sattolo's algorithm for cyclic permutations via randomization and derandomization. *J. Appl. Probability*, 40:790–796, 2003.

[13] H. Prodinger. On the analysis of an algorithm to generate a random cyclic permutation. *Ars Combinatoria*, 65:75–78, 2002.

[14] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, 1977.

[15] J. Riordan. *An Introduction to Combinatorial Analysis.* Wiley, New York, 1958.

[16] S. Sattolo. An algorithm to generate a random cyclic permutation. *Information Processing Letters*, 22(6):315–317, 1986.

[17] H. S. Wilf and A. Nijenhuis. *Combinatorial Algorithms.* Academic Press, 2nd edition, 1978.

[18] M. C. Wilson. Probability generating functions for Sattolo's algorithm. *J. Iranian Statistics Soc.*, 3(2):297–308, 2004.

On the Number of Hamilton Cycles in Bounded Degree Graphs

Heidi Gebauer*

Abstract

The main contribution of this paper is a new approach for enumerating Hamilton cycles in bounded degree graphs – deriving thereby extremal bounds.

We describe an algorithm which enumerates all Hamilton cycles of a given 3-regular n-vertex graph in time $O(1.276^n)$, improving on Eppstein's previous bound. The resulting new upper bound of $O(1.276^n)$ for the maximum number of Hamilton cycles in 3-regular n-vertex graphs gets close to the best known lower bound of $\Omega(1.259^n)$. Our method differs from Eppstein's in that he considers in each step a new graph and modifies it, while we fix (at the very beginning) one Hamilton cycle C and then proceed around C, succesively producing partial Hamilton cycles.

Our approach can also be used to show that the number of Hamilton cycles of a 4-regular n-vertex graph is at most $O(18^{n/5}) \leq O(1.783^n)$, which improves a previous bound by Sharir and Welzl. This result is complemented by a lower bound of $48^{n/8} \geq 1.622^n$.

Then we present an algorithm which finds the minimum weight Hamilton cycle of a given 4-regular graph in time $\sqrt{3}^n \cdot \text{poly}(n) = O(1.733^n)$, improving a previous result of Eppstein. This algorithm can be modified to compute the number of Hamilton cycles in the same time bound and to enumerate all Hamilton cycles in time $(\sqrt{3}^n + \text{hc}(G)) \cdot \text{poly}(n)$ with $\text{hc}(G)$ denoting the number of Hamilton cycles of the given graph G. So our upper bound of $O(1.783^n)$ for the number of Hamilton cycles serves also as a time bound for enumeration.

Using similar techniques as in the 3-regular case we establish upper bounds for the number of Hamilton cycles in 5-regular graphs and in graphs of average degree 3, 4, and 5.

1 Introduction

This paper is concerned with finding the min-weight Hamilton cycle, enumerating all Hamilton cycles and bounding the number of Hamilton cycles in bounded degree graphs.

1.1 Known Results

3- and 4-Regular Graphs Eppstein [2] established an algorithm which enumerates all Hamilton cycles of a given 3-regular graph in time $2^{\frac{3n}{8}} \leq 1.297^n$. This value is also the best known upper bound for the number of Hamilton cycles in 3-regular graphs. The corresponding algorithm basically solves the more general problem of listing all Hamilton cycles which contain a given set of *forced* edges. In each step it recursively deletes some edges and marks others as "forced" and then continues with the resulting, new graph. Besides, Eppstein showed a lower bound of $2^{\frac{n}{3}}$ ($2^{\frac{1}{3}} \approx 1.260$) for the number of Hamilton cycles and gave an algorithm which finds the min-weight Hamilton cycle in the same time bound. The latter result has recently been improved by Iwama and Nakashima [4], who showed an upper bound of $O(1.251^n)$. Finally Eppstein established an algorithm which finds the min-weight Hamilton cycle in a given 4-regular graph in time $\left(\frac{27}{4}\right)^{\frac{n}{3}} \leq 1.890^n$.

Planar Graphs Sharir and Welzl [5] proved that a graph of average degree at most 6 has at most 3^n Hamilton cycles, which implies an upper bound of 3^n for the number of Hamilton cycles in planar graphs. However, by using other properties of planar graphs, much better results could be established. The best known upper bound is due to Buchin, Knauer, Kriegel, Schulz and Seidel [1], who showed that a planar graph has at most 2.3404^n Hamilton cycles.

1.2 New Results

3-Regular Graphs One of the main contributions of our work is that we improve Eppstein's upper bound of $2^{\frac{3n}{8}}$ ($2^{\frac{3}{8}} \approx 1.297$) to 1.276^n. The resulting new upper bound of 1.276^n for the maximum number of Hamilton cycles in 3-regular graphs gets close to the corresponding lower bound of $2^{\frac{n}{3}}$ ($2^{\frac{1}{3}} \approx 1.260$) shown by Eppstein. In this extended abstract we will solely show a weaker bound of 1.281^n. The corresponding proof is given in Section 2. The refined result, which can be obtained by performing some fine tuning in this proof, is given in the full version [3]. It is important to note that our method is not a refinement of Eppstein's procedure but a new approach. Whereas Eppstein in each step considers a new graph and recursively modifies it we let the original graph stay as it is (throughout the whole algorithm) – at the beginning we fix *one* Hamilton cycle C and then proceed around C, succesively producing partial Hamilton cycles.

4-Regular Graphs In Section 3 we improve Eppstein's time upper bound of 1.890^n for finding the min-weight Hamilton cycle to $\sqrt{3}^n \cdot \text{poly}(n)$ ($\sqrt{3} \approx 1.732$). Again our approach is rather different from Eppstein's – he uses the concept of forced edges and reduces the problem to the 3-regular case while we enumerate all

*Institute of Theoretical Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland.

paths of length $\frac{n}{2}$ and search, using some algorithmic tricks, the pair whose concatenation forms a cycle with minimum weight. Adapted versions of this algorithm compute the number of Hamilton cycles in the same time bound and enumerate all Hamilton cycles in time $(\sqrt{3}^n + \mathsf{hc}(G)) \cdot \mathsf{poly}(n)$ with $\mathsf{hc}(G)$ denoting the number of Hamilton cycles of the given graph G. Finally we use a similar approach as in the 3-regular case to show that the number of Hamilton cycles is at most $O(1.783^n)$. This improves the upper bound of 2^n obtained by Sharir and Welzl and moreover serves as a time upper bound for enumeration. The last result is complemented by a lower bound of $48^{\frac{n}{8}} \geq 1.622^n$ on the number of Hamilton cycles.

More Results For a given d let $\mathcal{G}_{\text{av}}(d)$ denote the set of graphs on n vertices with average degree d and let $\mathcal{G}_{\text{reg}}(d)$ denote the set of d-regular graphs on n vertices. Using a similar approach as in the 3-regular case we can show the following bounds.

$$\mathsf{hc}(G) \leq \begin{cases} 3 \cdot \left(\frac{2}{\sqrt{3}}\right)^{\frac{n}{6}} \cdot \sqrt{3}^{\frac{n}{2}} \leq 3 \cdot 1.35^n & \text{if } G \in \mathcal{G}_{\text{av}}(3) \\ 3 \cdot \left(\frac{2}{\sqrt{3}}\right)^{\frac{n}{3}} \cdot \sqrt{3}^{n} \leq 3 \cdot 1.82^n & \text{if } G \in \mathcal{G}_{\text{av}}(4) \\ 3 \cdot \left(\frac{2}{\sqrt{3}}\right)^{\frac{n}{3}} \cdot \sqrt{3}^{\frac{3n}{2}} \leq 3 \cdot 2.40^n & \text{if } G \in \mathcal{G}_{\text{av}}(5) \\ 3 \cdot \left(\frac{2}{\sqrt{3}}\right)^{\frac{n}{4}} \cdot \sqrt{3}^{\frac{3n}{2}} \leq 3 \cdot 2.37^n & \text{if } G \in \mathcal{G}_{\text{reg}}(5) \end{cases}$$

2 3-Regular Graphs

THEOREM 2.1. *The Hamilton cycles of a given cubic n-vertex graph can be enumerated in time*

$$1.628^{\frac{n}{2}} \cdot \mathsf{poly}(n) = \mathcal{O}(1.276^n) \tag{2.1}$$

Here we only show the following, weaker result.

THEOREM 2.2. *The Hamilton cycles of a given cubic n-vertex graph G can be enumerated in time*

$$1.64^{\frac{n}{2}} \cdot \mathsf{poly}(n) = \mathcal{O}(1.281^n)$$

The bound in (2.1) can be derived by doing some fine-tuning in the proof of Theorem 2.2. The corresponding steps are presented in the full version [3].

Proof of Theorem 2.2. First we test whether G is Hamiltonian and, if yes, construct *one* Hamilton cycle. This can be done rather quickly due to known algorithms for finding the minimum weight Hamilton cycle, established, for example, by Eppstein [2], and Iwama and Nakashima [4]. Their algorithms have running time $\mathcal{O}(1.260^n)$ and $\mathcal{O}(1.251^n)$, respectively, which, compared to the claimed bound of $\mathcal{O}(1.281^n)$, is neglectable. So we can assume that we are given a Hamilton cycle $C = v_1, \ldots, v_n$.

The basic idea of our proof is the following: First we orient the edges of G in a particular way. Then we consider the following procedure for constructing Hamilton cycles: We walk around C in the direction $v_1, v_2, \ldots, v_n$ and decide for each vertex v_i which of its outgoing edges are included in our Hamilton cycle. It will turn out that there are many vertices where we have no choice, implying that the number of outcomes of our procedure (i.e. Hamilton cycles and attempts where we get stuck) is rather small.

We now give a more formal description of the above. The missing proofs can be found in the full version [3]. Each edge – except for (v_n, v_1) – is directed from the vertex with the lower index to the vertex with the higher index. The edges inside the cycle will be called "diagonals".

DEFINITION 2.1. A vertex is *active* if it is incident to an outgoing diagonal and *passive*, otherwise.

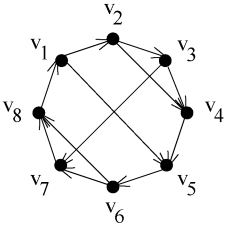


Figure 1: An example for $n = 8$. Here v_1, v_2, v_3, v_6 are active while v_4, v_5, v_7, v_8 are passive.

REMARK 2.1. v_1 and v_2 are active (since they can not have an incoming diagonal) whereas v_{n-1} and v_n are passive (since they can not have an outgoing diagonal).

DEFINITION 2.2. We call a sequence $(v_i, v_{i+1}, \ldots, v_j)$ with $1 \leq i < i+2 \leq j \leq n$, (i) an outward pattern if there is a diagonal (v_i, v_j) and $v_i, v_{i+1} \ldots, v_{j-1}$ are all active, (ii) an inward pattern if there is a diagonal (v_i, v_j) and $v_{i+1}, v_{i+2}, \ldots, v_j$ are all passive.

Figure 2 shows an illustration. It will turn out that

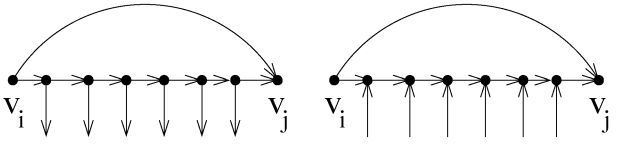


Figure 2: An outward pattern (on the left) and an inward pattern (on the right).

inward patterns have a rather bad influence on the running time of our algorithm whereas outward patterns have a good influence. So the next observation is crucial.

OBSERVATION 2.1. We can assume that the number of outward patterns is at least the number of inward patterns.

This can easily be achieved by possibly reversing the numbering of the vertices, by which inward patterns become outward patterns and vice versa.

We consider the following procedure, which we denote by P_{ham}, for constructing Hamilton cycles.

Procedure P_{ham} First we decide whether or not to select (v_n, v_1). Then we process the vertices $v_1, \ldots, v_{n-1}$ in this order. We refer to the processing of v_i by *round* i. In round i we carefully select some outgoing edges of v_i such that afterwards the following holds.

(i) Each vertex v_j with $j \leq i$ is incident to exactly two selected edges.

(ii) The set of selected edges does not contain a cycle of length smaller than n.

(iii) If v_{i+1} has two incoming edges then at least one of them must be selected.

We call (i) - (iii) the *postconditions (for round i)*. Note that these conditions only filter out selections which can not be completed to a Hamilton cycle. We perform round i as follows.

Case 1: v_i is passive: In this case there is one choice. Indeed, since postcondition (iii) is satisfied after round $i-1$ at least one of the incoming edges of v_i is selected. If both incoming edges are selected then we do not select the outgoing edge of v_i (the only way to fulfill postcondition (i)). Otherwise we select the outgoing edge of v_i (also due to postcondition (i)).

Case 2: v_i is active: In this case there might or might not be two choices.
If the incoming edge of v_i is not selected then we select both of its outgoing edges (the reason is again postcondition (i)). Otherwise we consider two choices: The selection of (v_i, v_{i+1}) and, secondly, the selection of the outgoing diagonal of v_i.

For each of the considered choices we test whether the postconditions of round i are fulfilled. For each choice which passes this test successfully we recursively go to round $i+1$.

We go on until we have performed round $n-1$. At this stage we have decided for each edge whether or not it is selected. We check whether the set S of selected edges forms a Hamilton cycle. If yes, we output S, otherwise we do nothing. Then we backtrack.

DEFINITION 2.3. Let k be a natural number with $k \leq n-1$. Each edge set which can be obtained by performing k rounds of P_{ham} will be called a *choice for* $v_1, \ldots, v_k$.
For the empty sequence ε we let – by a slight abuse of notation – $\mathsf{ch}(\varepsilon)$ denote the set of choices for the very first decision (directly before round 1) and so $\mathsf{ch}(\varepsilon)$ is the set consisting of the empty set and the set containing only the edge (v_n, v_1).

OBSERVATION 2.2. Let C be a choice for $v_1, \ldots, v_k$. By postcondition (i) v_i is then incident to exactly two edges in C for $i \leq k$.

OBSERVATION 2.3. P_{ham} runs in time at most $(|\mathsf{ch}(v_1)| + |\mathsf{ch}(v_1, v_2)| + \ldots + |\mathsf{ch}(v_1, \ldots, v_{n-1})|) \cdot \mathsf{poly}(n)$

This follows directly from the fact that the number of nodes in the recursion tree of P_{ham} is $(|\mathsf{ch}(v_1)| + |\mathsf{ch}(v_1, v_2)| + \ldots + |\mathsf{ch}(v_1, \ldots, v_{n-1})|)$.

Bound on the Number of Choices In the following the indices of vertices are meant modulo n.

DEFINITION 2.4. An active vertex v_i is called *unpleasant* if the diagonal of the previous active vertex v_j points to a vertex in $\{v_{j+2}, \ldots, v_{i-1}\}$. An active vertex which is not unpleasant is called *pleasant*.

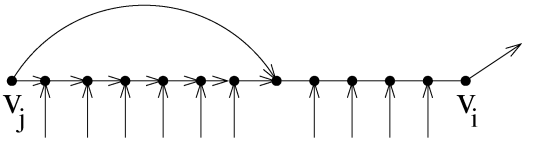


Figure 3: An unpleasant vertex v_i.

REMARK 2.2. v_1 is unpleasant since the diagonal of the last active vertex v_l points to a vertex in $\{v_{l+2}, \ldots, v_n\}$.

Let v_i be any unpleasant vertex and let v_j be its previous active vertex. Then the diagonal of v_j points to a vertex v_k in $\{v_{j+2}, \ldots, v_{i-1}\}$ and so $(v_j, \ldots, v_k)$ is an inward pattern. Conversely, the first active vertex after an inward pattern is unpleasant. Hence the number of unpleasant vertices equals the number of inward patterns. By Observation 2.1 we have

OBSERVATION 2.4. The number of unpleasant vertices is at most the number of outward patterns.

We will partition the sequence $v_1, \ldots, v_{n-1}$ into suitable subsequences and then reduce our original claim to a statement on subsequences. Therefore we extend the notation of a "choice" to subsequences.

DEFINITION 2.5. Let C be a choice for $v_1,\dots,v_s$ and let D be any choice. Then D is *consistent* with C if for each edge e incident to a vertex in $\{v_1,\dots,v_s\}$ it holds: $e\in D$ if and only if $e\in C$.

DEFINITION 2.6. For any subsequence $v_p,\dots,v_q$ of $v_1,\dots,v_{n-1}$ and any choice C for $v_1,\dots v_{p-1}$ we let $\mathsf{ch}_{|C}(v_p,\dots,v_q)$ denote the set of all choices for $v_1,\dots v_q$ which are consistent with C.
The empty subsequence ε is treated as if $q=p-1$ and so $\mathsf{ch}_{|C}(\varepsilon)=\{C\}$.
The elements of $\mathsf{ch}_{|C}(v_p,\dots,v_q)$ will be called *choices for $v_p,\dots,v_q$ consistent with C.*

To split $|\mathsf{ch}(v_1,\dots,v_{n-1})|$ into suitable factors we set

(2.2)
$$\mathsf{maxch}(v_p,\dots,v_q) := \max_{C\in\mathsf{ch}(v_1,\dots,v_{p-1})}(|\mathsf{ch}_{|C}(v_p,\dots,v_q)|)$$

Let $(v_1,\dots,v_{r_1-1}),\dots,(v_{r_k},\dots,v_{n-1})$ be a partition of $v_1,\dots,v_{n-1}$. By induction on r_i we get

(2.3)
$$|\mathsf{ch}(v_1,\dots,v_{n-1})| \le \underbrace{|\mathsf{ch}(\varepsilon)|}_{2}\cdot\mathsf{maxch}(v_1,\dots,v_{r_1-1})\cdot \mathsf{maxch}(v_{r_1},\dots,v_{r_2-1})\cdots\mathsf{maxch}(v_{r_k},\dots,v_{n-1})$$

(The fact that $|\mathsf{ch}(\varepsilon)|=2$ is due to Definition 2.3.) The key-lemma below leads to Theorem 2.2.

LEMMA 2.1. *Let $v_p,\dots,v_q$ be a subsequence of vertices such that v_p is unpleasant and $v_{p+1},\dots,v_q$ are not unpleasant (i.e. pleasant or passive). Let a denote the number of active vertices in $v_p,\dots,v_q$ and let s denote the number of outward patterns fully contained in $v_p,\dots,v_q$. Then for any given choice C for $v_1,\dots,v_{p-1}$ we have*

$$|\mathsf{ch}_{|C}(v_p,\dots,v_q)| \le 1.64^a\cdot\left(\frac{1.64}{2}\right)^{s-1} \tag{2.4}$$

We postpone the proof and concentrate on the consequences of Lemma 2.1. Let u denote the number of unpleasant vertices of G. Since v_1 is unpleasant (see Remark 2.2) we can partition $v_1,\dots,v_{n-1}$ into u subsequences of the form $v_p,\dots v_q$ where v_p is unpleasant and $v_{p+1},\dots,v_q$ are not unpleasant. It can be seen that each outward pattern with starting point in $v_p,\dots,v_q$ is fully contained in $v_p,\dots,v_q$. For $i\le u$ let a_i and s_i denote the number of active vertices and the number of outward patterns in the i'th subsequence, respectively. Besides, let $s:=\sum s_i$ denote the overall number of patterns. By (2.3), (2.2) and (2.4) we have

$$\begin{aligned}|\mathsf{ch}(v_1,\dots,v_{n-1})| &\le 2\cdot\prod_{i=1}^{u}\left(1.64^{a_i}\cdot\left(\frac{1.64}{2}\right)^{s_i-1}\right)\\ &= 2\cdot 1.64^{\sum_{i=1}^{u}a_i}\cdot\left(\frac{1.64}{2}\right)^{s-u}\end{aligned}$$

We have $\sum_{i=1}^{u}a_i=\frac{n}{2}$ (note that $\sum_{i=1}^{u}a_i$ is the overall number of active vertices, which equals the number of diagonals) and, by Observation 2.4, $u\le s$. So $|\mathsf{ch}(v_1,\dots,v_{n-1})|\le 2\cdot 1.64^{\frac{n}{2}}$.
We are almost done with the proof of Theorem 2.2. By Observation 2.3 it suffices to show that $|\mathsf{ch}(v_1,\dots,v_l)|\le 1.64^{\frac{n}{2}}\cdot\mathsf{poly}(n)$ for $l\le n-2$. This can be done similarly to the above (a formal proof is given in the full version).

Proof of Lemma 2.1 From now on by *pattern* we mean an *outward pattern*. We partition the active vertices in $v_p,\dots,v_q$ into disjoint sets $W_1,\dots,W_k$ where k is the number of patterns in $v_p,\dots,v_q$ plus the number of active vertices in $v_p,\dots,v_q$ not belonging to a pattern. Each W_i contains either all active vertices of a pattern or a single active vertex which does not belong to a pattern. Note that $v_p\in W_1$.

DEFINITION 2.7. For $i=1,\dots,k$ let f_i denote the v-index of the first vertex in W_i and let l_i denote the v-index of the last vertex in W_i. We define f_{k+1} as $q+1$ and f_0 as $p-1$.

Note that $W_i=\{v_{f_i},v_{f_i+1},\dots,v_{l_i}\}$ and $f_1=p$. Now we can transform Lemma 2.1 into a more manageable form.

DEFINITION 2.8. For $r=0,\dots,k$ let A_r denote the number of elements of $\mathsf{ch}_{|C}(v_p,\dots,v_{f_{r+1}-1})$ which contain the edge $(v_{f_{r+1}-1},v_{f_{r+1}})$. Similarly, let B_r denote the number of elements of $\mathsf{ch}_{|C}(v_p,\dots,v_{f_{r+1}-1})$ which do not contain the edge $(v_{f_{r+1}-1},v_{f_{r+1}})$.

By Definition 2.6 this definition implies that $\left\{\begin{array}{l}A_0=1,\ B_0=0\\ A_0=0,\ B_0=1\end{array}\right\}$ if (v_{p-1},v_p) $\left\{\begin{array}{l}\in C\\ \notin C\end{array}\right\}$

Note that $A_k+B_k=\mathsf{ch}_{|C}(v_p,\dots,v_q)$. The following two lemmas establish Lemma 2.1.

LEMMA 2.2. *Let $w_i:=|W_i|$. For $r=1,\dots,k$ we have:*

(a) If $w_r=1$ then

$$A_r+B_r \le 2\cdot A_{r-1}+B_{r-1} \tag{2.5}$$
$$A_r \le A_{r-1}+B_{r-1},\quad \text{for}\quad r\le k-1 \tag{2.6}$$

(b) If $w_r \geq 2$ then

$$(2.7) \quad A_r + B_r \leq 2 \cdot F_{w_r} \cdot A_{r-1} + (F_{w_r+1} - 1) \cdot B_{r-1}$$

$$(2.8) \quad A_r \leq F_{w_r+1} \cdot A_{r-1} + F_{w_r} \cdot B_{r-1}, \quad \text{for } r \leq k-1$$

where F_i denotes the i'th Fibonacci number for $i \in \mathbb{N}_0$. (We assume that $F_0 = 0$ and $F_1 = 1$.)

LEMMA 2.3. *Let $w_i := |W_i|$ and let $\tilde{w}_i := |\{j \leq i : w_j \geq 2\}|$. If (2.5) - (2.8) hold then* $A_k + B_k \leq 1.64^{\left(\sum_{i=1}^{k} w_i\right)} \cdot \left(\frac{1.64}{2}\right)^{\tilde{w}_k - 1}$

Lemma 2.2 and Lemma 2.3 imply Lemma 2.1. Indeed, $\sum_{i=1}^{k} w_i = a$ (since $W_1, \ldots, W_k$ is a partition of the active vertices in $v_p, \ldots, v_q$) and $\tilde{w}_k$ equals the number of patterns in $v_p, \ldots, v_q$.

Proof sketch of Lemma 2.3. We proceed by induction. We omit a proof for the base case $k = 1$. For $k \geq 2$ and $1 \leq j \leq k-1$ we obtain by recursion (we omit a formal proof for the first statement)

$$(2.9) \quad A_k + B_k \leq 1.64^{\left(\sum_{i=j+1}^{k} w_i\right)} \cdot \left(\frac{1.64}{2}\right)^{\tilde{w}_k - \tilde{w}_j - 1} \cdot (A_j + B_j)$$

$$(2.10) \quad A_j + B_j \leq 1.64^{\left(\sum_{i=1}^{j} w_i\right)} \cdot \left(\frac{1.64}{2}\right)^{\tilde{w}_j - 1}$$

If $w_j \geq 35$ for some j then we can show that (2.10) remains true even when we multiply the right side with $\frac{1.64}{2}$. If $w_i \leq 34$ for all i then a computer program which considers all possible cases for $w_k, w_{k-1}, \ldots$ up to depth 15 yields that there is a j such that (2.9) remains true even when we multiply the right side with $\frac{1.64}{2}$. In either case we obtain $A_k + B_k \leq 1.64^{\left(\sum_{i=1}^{k} w_i\right)} \cdot \left(\frac{1.64}{2}\right)^{\tilde{w}_k - 1}$, which proves Lemma 2.3.

Proof of Lemma 2.2.

DEFINITION 2.9. By $\mathsf{ch}^{\mathsf{sel}}_{|D}(v_{f_r}, \ldots, v_{f_{r+1}-1})$ we denote the set of elements of $\mathsf{ch}_{|D}(v_{f_r}, \ldots, v_{f_{r+1}-1})$ which contain the edge $(v_{f_{r+1}-1}, v_{f_{r+1}})$.

Let r be an integer with $1 \leq r \leq k$. We distinguish 2 cases

Case (a): $w_r = 1$
Let D be any choice for $v_1, \ldots, v_{f_r-1}$. Our goal is to show that

(i) if $(v_{f_r-1}, v_{f_r}) \notin D$ then $|\mathsf{ch}_{|D}(v_{f_r}, \ldots v_{f_{r+1}-1})| \leq 1$.

(ii) if $(v_{f_r-1}, v_{f_r}) \in D$ then $|\mathsf{ch}_{|D}(v_{f_r}, \ldots v_{f_{r+1}-1})| \leq 2$.

(iii) if $r \leq k-1$ then $|\mathsf{ch}^{\mathsf{sel}}_{|D}(v_{f_r}, \ldots, v_{f_{r+1}-1})| \leq 1$.

(i) and (ii) imply (2.5) whereas (iii) implies (2.6). So it remains to prove (i) - (iii).

PROPOSITION 2.1. Let r be an integer with $1 \leq r \leq k$ and let D be any choice for $v_1, \ldots, v_{f_r-1}$. Then $|\mathsf{ch}_{|D}(v_{f_r}, \ldots, v_{f_{r+1}-1})| \leq |\mathsf{ch}_{|D}(v_{f_r}, \ldots, v_{l_r})|$

This proposition easily follows from the fact that for passive vertices we do not have a choice in P_{ham} (Note that $v_{l_r+1}, \ldots, v_{f_{r+1}-1}$ are passive).

(i) can now be proven straightforward: Since $(v_{f_r-1}, v_{f_r}) \notin D$ we are forced to select both outgoing edges of v_{f_r} in round f_r of P_{ham}. Hence $|\mathsf{ch}_{|D}(v_{f_r})| \leq 1$. Proposition 2.1 then implies (i).

If $(v_{f_r-1}, v_{f_r}) \in D$ then we consider (at most) two possibilities in round f_r: The selection of (v_{f_r}, v_{f_r+1}) versus the selection of the outgoing diagonal of v_{f_r}. Hence $|\mathsf{ch}_{|D}(v_{f_r})| \leq 2$. Proposition 2.1 then implies (ii).

It remains to prove (iii). It suffices to show the next lemma (then we can proceed similarly as in (i)).

LEMMA 2.4. *(v_{f_r}, v_{f_r+1}) either belongs to all elements or to no element of $\mathsf{ch}^{\mathsf{sel}}_{|D}(v_{f_r}, \ldots, v_{f_{r+1}-1})$*

Proof of Lemma 2.4. Let D' be an element of $\mathsf{ch}^{\mathsf{sel}}_{|D}(v_{f_r}, \ldots, v_{f_{r+1}-1})$. It suffices to show that D determines whether (v_{f_r}, v_{f_r+1}) belongs to D'.

OBSERVATION 2.5. By the assumption in Lemma 2.1 and Definition 2.7 v_{f_i} is pleasant for $2 \leq i \leq k$.

By assumption we have that $1 \leq r \leq k-1$, which guarantees that $v_{f_{r+1}}$ is pleasant. Hence the outgoing diagonal of v_{f_r} does not point to a vertex in $\{v_{f_r+2}, \ldots, v_{f_{r+1}-1}\}$. So each incoming diagonal of a vertex in $\{v_{f_r+1}, \ldots, v_{f_{r+1}-1}\}$ has its source in $\{v_1, \ldots, v_{f_r-1}\}$. So for each incoming diagonal e of a vertex in $\{v_{f_r+1}, \ldots, v_{f_{r+1}-1}\}$ it is determined whether e belongs to D'. We set $s := f_{r+1} - f_r$ and

$$(2.11) \quad j := \max_{i \in \{1 \ldots s\}} (i : \text{for } k = 1 \ldots i-1 \text{ the incoming diagonal of } v_{f_r+k} \text{ belongs to } D')$$

If $j < s$ then the incoming diagonal of v_{f_r+j} does not belong to D' (resp. D). Otherwise $v_{f_r+j} = v_{f_{r+1}}$ and the incoming diagonals of $v_{f_r+1}, \ldots, v_{f_{r+1}-1}$ all belong to D'.

PROPOSITION 2.2. (v_{f_r+j-1}, v_{f_r+j}) belongs to D'.

This can be seen by distinguishing the cases where $j = s$ and $j < s$ (and using Observation 2.2).

To finish the proof of Lemma 2.4 consider an $i < j$. The incoming diagonal of v_{f_r+i} is in D' (by choice of j) and v_{f_r+i} is incident to exactly two edges of D' (by Observation 2.2). So

$$(v_{f_r+i-1}, v_{f_r+i}) \in D' \text{ if and only if } (v_{f_r+i}, v_{f_r+i+1}) \notin D' \text{ for } i \text{ with } 1 \leq i \leq j-1$$

Since $(v_{f_r+j-1}, v_{f_r+j}) \in D'$ (by Proposition 2.2) it is determined whether (v_{f_r}, v_{f_r+1}) is in D'. □

Case (b): $w_r \geq 2$
Let m denote w_r. $v_{f_r}, \ldots, v_{f_r+m-1}$ are in the same partition class W_r and so they all belong the same pattern. Figure 4 shows an illustration. Here we

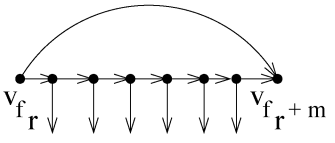


Figure 4: A pattern

only give a proof for (2.7). Let D be a choice for $v_1, \ldots, v_{f_r-1}$. It then suffices to show

$$|\mathsf{ch}_{|D}(v_{f_r}, \ldots, v_{f_{r+1}-1})| \leq \begin{cases} 2 \cdot F_m & \text{if } (v_{f_r-1}, v_{f_r}) \in D \\ F_{m+1} - 1 & \text{if } (v_{f_r-1}, v_{f_r}) \notin D \end{cases}$$

Proof. Here we only show the second claim. The next proposition is stated without proof as well.

Proposition 2.3. Let $v_x, \ldots v_{x+s}$ be a subsequence of $v_1, \ldots v_q$ where $v_x, \ldots v_{x+s}$ are all active and let E be a choice for $v_1, \ldots v_{x-1}$. Then

$$|\mathsf{ch}_{|E}(v_x, \ldots, v_{x+s})| \leq \begin{cases} F_{s+3} & \text{if } (v_{x-1}, v_x) \in E \\ F_{s+2} & \text{if } (v_{x-1}, v_x) \notin E \end{cases}$$

By Proposition 2.3, $|\mathsf{ch}_{|D}(v_{f_r}, \ldots, v_{f_r+m-1})| \leq F_{m+1}$. The proof of Proposition 2.3 yields that the constellation where all edges of the cycle $v_{f_r}, \ldots, v_{f_r+m}, v_{f_r}$ are selected contributes 1 to this bound. (Figure 5 gives an illustration.) However, due to postcondition (ii) of P_{ham} no choice for $v_1, \ldots, v_{f_r+m-1}$ contains a Hamilton cycle of length smaller than n □

Fine Tuning To establish the improved upper bound claimed in Theorem 2.1 we prove the following stronger version of Lemma 2.2.

Lemma 2.5. *Let $w_i := |W_i|$ denote the number of vertices in the partition class W_i. For $r = 1, \ldots, k$ we have:*

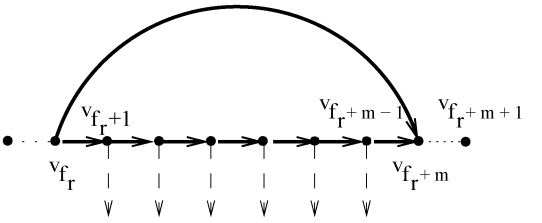


Figure 5: Constellation where all edges of the cycle $v_{f_r}, \ldots, v_{f_r+m-1}, v_{f_r}$ are selected

(a) If $w_r = 1$ then

$$\begin{aligned} A_r + B_r &\leq 2 \cdot A_{r-1} + B_{r-1} \\ A_r &\leq A_{r-1} + B_{r-1}, \quad \text{for } r \leq k-1 \end{aligned}$$

(b) If $w_r \geq 2$ then

$$\begin{aligned} A_r + B_r &\leq 2 \cdot F_{w_r} \cdot A_{r-1} + (F_{w_r+1} - 1) \cdot B_{r-1} \\ A_r &\leq (F_{w_r} + F_{w_r-2}) \cdot A_{r-1} + F_{w_r} \cdot B_{r-1}, \\ & \qquad \text{for } r \leq k-1 \end{aligned}$$

(c) If $w_r = 3$ then additionally one of the following inequalities holds

(c1) $A_r \leq 3 \cdot A_{r-1} + B_{r-1}, \quad \text{for } r \leq k-1$
(c2) $A_r \leq 2 \cdot A_{r-1} + 2 \cdot B_{r-1}, \quad \text{for } r \leq k-1$

This Lemma can be shown by using an improved analysis, some graph-theoretical observations and ad hoc conclusion. The concrete proof is given in the full version.

3 4-Regular Graphs

Finding the Minimum Weight Hamilton Cycle We establish an algorithm MinHC for finding the min-weight Hamilton cycle of a given graph.

Theorem 3.1. *MinHC finds the min-weight Hamilton cycle of a 4-regular n-vertex graph G in time*

$$\sqrt{3}^n \cdot \mathsf{poly}(n), \quad (\sqrt{3} \approx 1.73)$$

With adapted versions of MinHC we can compute the number of Hamilton cycles of G in the same time bound and enumerate all *Hamilton cycles of G in time* $(\sqrt{3}^n + \mathsf{hc}(G)) \cdot \mathsf{poly}(n)$

Proof. Here we only show the first statement and restrict on the case where n is even. A Hamilton cycle in which two given vertices v_i, v_j have distance $\frac{n}{2}$ will be called a (v_i, v_j)-*cycle*. Since the number of vertex pairs is polynomial in n it suffices to establish an algorithm which finds the min-weight (v_1, v_m)-cycle for

given v_1, v_m.
To find the min-weight (v_1, v_m)-cycle we proceed as follows.

Step 1: We enumerate all v_1, v_m-paths of length $\frac{n}{2}$ and then store them in an array called PL. Note that the min-weight (v_1, v_m)-cycle is formed by the concatenation of the two paths p, q which have minimum weight-sum among all internally disjoint pairs of paths stored in PL.

Step 2: For each path p we define the *key of* p to be the sequence of the indices of the inner vertices of p sorted in ascending order. We then sort the elements of PL according to the lexicographical order of their keys. In each sequence (in PL) of paths having the same set of inner points we only keep the path with minimum weight and delete (in PL) all other paths.

Step 3: Each key now occurs at most once in PL. To find the optimal path-pair we then do several rounds: In round i we take the path p stored in the i'th item of PL and search in PL for a path whose set of inner points equals $V(G)\backslash(I(p)\cup\{v_1, v_m\})$ with $I(p)$ denoting the set of inner points of p. If the search was successful then we check whether the concatenation c of the found path and p is of lower weight than the current best cycle. If yes then we store c as current best cycle. Otherwise we do nothing. Finally, the min-weight (v_1, v_m)-cycle is stored as current best cycle.

It remains to show that this algorithm can be performed in time $3^{\frac{n}{2}} \cdot \mathsf{poly}(n)$. For Step 1 it suffices to consider the algorithm where one starts at some vertex and recursively visits all yet untouched neighbours. This algorithm has running time at most $3^{\frac{n}{2}} \cdot \mathsf{poly}(n)$, which implies that the number of elements of PL is bounded by the same expression. For Step 2 it is crucial that an N-element array can be sorted in time $N \cdot \log(N)$. Since in our case $N \leq 3^{\frac{n}{2}} \cdot \mathsf{poly}(n)$ we are done. For Step 3 the fact that finding an element in a sorted N-element array can be done in time $\log(N)$ will do. □

Upper Bound on the Number of Hamilton Cycles

THEOREM 3.2. *Let G be a 4-regular graph on n vertices. Then*
(3.12)

$$\mathsf{hc}(G) \leq 2 \cdot \sqrt{3}^n \cdot \left(\frac{2}{\sqrt{3}}\right)^{\frac{n}{5}}, \quad (\sqrt{3} \cdot \left(\frac{2}{\sqrt{3}}\right)^{\frac{1}{5}} \approx 1.78)$$

Proof idea. As in the 3-regular case we consider a fixed Hamilton cycle and direct the edges accordingly. This time we do not deal with patterns. So there is no "good counterpart" to "compensate" the bad influence of unpleasant vertices. However we will find that there are at most $\frac{n}{5}$ unpleasant vertices. Using a similar approach as in the 3-regular case together with some additional observations we can show (3.12). We note that the last factor in (3.12) is due to unpleasant vertices. To give an explanation for the middle factor: It will turn out that in the critical case half of the vertices have 2 outgoing diagonals and the other half have none. Let v_i be a vertex with 2 outgoing diagonals. Then – independently of whether the incoming edge of v_i is selected – we have 3 possibilities to select some outgoing edges of v_i such that at the end v_i is incident to 2 selected edges. So all in all we have at most $3^{\frac{n}{2}}$ possibilities.

4 Lower Bounds

THEOREM 4.1. *For any $k \geq 3$ and any n divisible by $2k$ there is a k-regular n-vertex graph H with*

(4.13)

$$\mathsf{hc}(H) = \left(\lceil\frac{k-1}{2}\rceil \cdot (k-1)! \cdot (k-2)! \cdot \lfloor\frac{k+1}{2}\rfloor\right)^{\frac{n}{2k}} \geq \left(\frac{k}{e}\right)^n$$

This theorem proves our previously mentioned bound of $48^{\frac{n}{8}} \geq 1.622^n$ for 4-regular graph. The construction we will establish for proving our claim is the best we know for small values of k. The sequence $\sqrt{hc(H)}^{\frac{1}{n}}$ as k tends to ∞ is optimal (as witnessed by Bregman's Theorem).

Proof. We connect $\frac{n}{2k}$ $2k$-vertex gadgets $G_1, \ldots, G_{\frac{n}{2k}}$ in a cyclic order. The figure below shows the construction for $k = 5$. More precisely: A single gadget G_i consists of

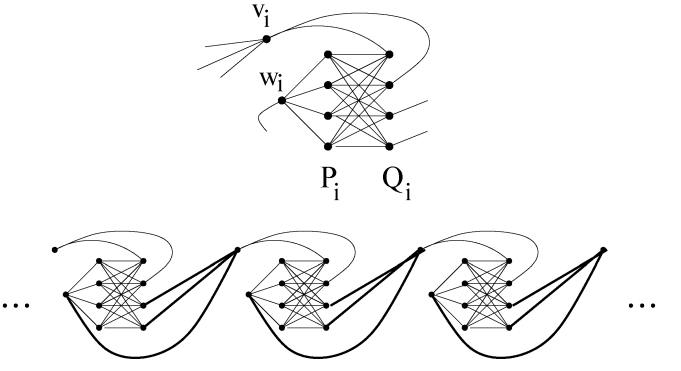


Figure 6: Construction of H for $k = 5$. A gadget (above) and an extract of H (below)

a $K_{k-1,k-1}$ and two extra vertices. Let v_i, w_i denote the extra vertices and P_i, Q_i denote the independent sets

of the $K_{k-1,k-1}$. We connect w_i with all vertices of P_i and we connect v_i with $\lceil \frac{k-1}{2} \rceil$ vertices of Q_i. We then connect v_{i+1} with w_i and all vertices of Q_i which are not adjacent to v_i. It can be shown that the number of v_i, v_{i+1}-paths covering all vertices of G_i equals the base of the second term in (4.13) (a formal proof is given in the full version), implying the "=" part. Some analysis shows the "$\geq$" part. □

Acknowledgment: We would like to thank Tibor Szabó and Emo Welzl for suggesting such a rich and beautiful topic and also for the numerous helpful discussions.

References

[1] K. Buchin, C. Knauer, K. Kriegel, A. Schulz, and R. Seidel, *On the Number of Cycles in Planar Graphs*, 13th Annual International Computing and Combinatorics Conference, Proc. COCOON 2007, to appear

[2] D. Eppstein, *The Traveling Salesman Problem for Cubic Graphs*, Lecture Notes in Computer Science, Volume 2748, Sep 2003, 307-318

[3] H. Gebauer, The full version of *On the Number of Hamilton Cycles in Bounded Degree Graphs*, http://people.inf.ethz.ch/gebauerh/

[4] K. Iwama and T. Nakashima, *An Improved Exact Algorithm for Cubic Graph TSP*, 13th Annual International Computing and Combinatorics Conference, Proc. COCOON 2007, to appear

[5] M.Sharir and E.Welzl, *On the number of crossing-free matchings, (cycles, and partitions)*, Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2006, 860-869

Analysis of the Expected Number of Bit Comparisons Required by Quickselect*

James Allen Fill† Takéhiko Nakama‡

Abstract

When algorithms for sorting and searching are applied to keys that are represented as bit strings, we can quantify the performance of the algorithms not only in terms of the number of key comparisons required by the algorithms but also in terms of the number of bit comparisons. Some of the standard sorting and searching algorithms have been analyzed with respect to key comparisons but not with respect to bit comparisons. In this extended abstract, we investigate the expected number of bit comparisons required by `Quickselect` (also known as `Find`). We develop exact and asymptotic formulae for the expected number of bit comparisons required to find the smallest or largest key by `Quickselect` and show that the expectation is asymptotically linear with respect to the number of keys. Similar results are obtained for the average case. For finding keys of arbitrary rank, we derive an exact formula for the expected number of bit comparisons that (using rational arithmetic) requires only finite summation (rather than such operations as numerical integration) and use it to compute the expectation for each target rank.

1 Introduction and Summary

When an algorithm for sorting or searching is analyzed, the algorithm is usually regarded either as comparing keys pairwise irrespective of the keys' internal structure or as operating on representations (such as bit strings) of keys. In the former case, analyses often quantify the performance of the algorithm in terms of the number of key comparisons required to accomplish the task; `Quickselect` (also known as `Find`) is an example of those algorithms that have been studied from this point of view. In the latter case, if keys are represented as bit strings, then analyses quantify the performance of the algorithm in terms of the number of bits compared until it completes its task. Digital search trees, for example, have been examined from this perspective.

In order to fully quantify the performance of a sorting or searching algorithm and enable comparison between key-based and digital algorithms, it is ideal to analyze the algorithm from both points of view. However, to date, only `Quicksort` has been analyzed with both approaches; see Fill and Janson [3]. Before their study, `Quicksort` had been extensively examined with regard to the number of key comparisons performed by the algorithm (e.g., Knuth [11], Régnier [16], Rösler [17], Knessl and Szpankowski [9], Fill and Janson [2], Neininger and Rüschendorf [15]), but it had not been examined with regard to the number of bit comparisons in sorting keys represented as bit strings. In their study, Fill and Janson assumed that keys are independently and uniformly distributed over (0,1) and that the keys are represented as bit strings. [They also conducted the analysis for a general absolutely continuous distribution over (0,1).] They showed that the expected number of bit comparisons required to sort n keys is asymptotically equivalent to $n(\ln n)(\lg n)$ as compared to the lead-order term of the expected number of *key* comparisons, which is asymptotically $2n \ln n$. We use ln and lg to denote natural and binary logarithms, respectively, and use log when the base does not matter (for example, in remainder estimates).

In this extended abstract, we investigate the expected number of bit comparisons required by `Quickselect`. Hoare [7] introduced this search algorithm, which is treated in most textbooks on algorithms and data structures. `Quickselect` selects the m-th smallest key (we call it the rank-m key) from a set of n distinct keys. (The keys are typically assumed to be distinct, but the algorithm still works—with a minor adjustment—even if they are not distinct.) The algorithm finds the target key in a recursive and random fashion. First, it selects a pivot uniformly at random from n keys. Let k denote the rank of the pivot. If $k = m$, then the algorithm returns the pivot. If $k > m$, then the algorithm recursively operates on the set of keys smaller than the pivot and returns the rank-m key. Similarly, if $k < m$, then the algorithm recursively oper-

*Supported by NSF grant DMS–0406104, and by The Johns Hopkins University's Acheson J. Duncan Fund for the Advancement of Research in Statistics.

†Department of Applied Mathematics and Statistics at The Johns Hopkins University.

‡Department of Applied Mathematics and Statistics at The Johns Hopkins University.

ates on the set of keys larger than the pivot and returns the $(k-m)$-th smallest key from the subset. Although previous studies (e.g., Knuth [10], Mahmoud *et al.* [13], Grübel and U. Rösler [6], Lent and Mahmoud [12], Mahmoud and Smythe [14], Devroye [1], Hwang and Tsai [8]) examined `Quickselect` with regard to key comparisons, this study is the first to analyze the bit complexity of the algorithm.

We suppose that the algorithm is applied to n distinct keys that are represented as bit strings and that the algorithm operates on individual bits in order to find a target key. We also assume that the n keys are uniformly and independently distributed in $(0,1)$. For instance, consider applying `Quickselect` to find the smallest key among three keys k_1, k_2, and k_3 whose binary representations are .01001100..., .00110101..., and .00101010..., respectively. If the algorithm selects k_3 as a pivot, then it compares each of k_1 and k_2 to k_3 in order to determine the rank of k_3. When k_1 and k_3 are compared, the algorithm requires 2 bit comparisons to determine that k_3 is smaller than k_1 because the two keys have the same first digit and differ at the second digit. Similarly, when k_2 and k_3 are compared, the algorithm requires 4 bit comparisons to determine that k_3 is smaller than k_2. After these comparisons, key k_3 has been identified as smallest. Hence the search for the smallest key requires a total of 6 bit comparisons (resulting from the two key comparisons).

We let $\mu(m,n)$ denote the expected number of bit comparisons required to find the rank-m key in a file of n keys by `Quickselect`. By symmetry, $\mu(m,n) = \mu(n+1-m,n)$. First, we develop exact and asymptotic formulae for $\mu(1,n) = \mu(n,n)$, the expected number of bit comparisons required to find the smallest key by `Quickselect`, as summarized in the following theorem.

THEOREM 1.1. *The expected number $\mu(1,n)$ of bit comparisons required by* `Quickselect` *to find the smallest key in a file of n keys that are independently and uniformly distributed in $(0,1)$ has the following exact and asymptotic expressions:*

$$\begin{aligned}\mu(1,n) &= 2n(H_n-1) + 2\sum_{j=2}^{n-1} B_j \frac{n-j+1-\binom{n}{j}}{j(j-1)(1-2^{-j})} \\ &= cn - \frac{1}{\ln 2}(\ln n)^2 - \left(\frac{2}{\ln 2}+1\right)\ln n + O(1),\end{aligned}$$

where H_n and B_j denote harmonic and Bernoulli numbers, respectively, and, with $\chi_k := \frac{2\pi i k}{\ln 2}$ and $\gamma :=$ Euler's constant $\doteq 0.57722$, we define

$$\begin{aligned} c &:= \frac{28}{9} + \frac{17-6\gamma}{9\ln 2} \\ &\quad - \frac{4}{\ln 2}\sum_{k\in\mathbb{Z}\setminus\{0\}} \frac{\zeta(1-\chi_k)\Gamma(1-\chi_k)}{\Gamma(4-\chi_k)(1-\chi_k)} \\ &\doteq 5.27938. \end{aligned} \tag{1.1}$$

The constant c can alternatively be expressed as

$$c = 2\sum_{k=0}^{\infty}\left(1 + 2^{-k}\sum_{j=1}^{2^k}\ln\frac{j}{2^k}\right). \tag{1.2}$$

It is easily seen that the expression (1.1) is real, even though it involves the imaginary numbers χ_k. The asymptotic formula shows that the expected number of bit comparisons is asymptotically linear in n with the lead-order coefficient approximately equal to 5.27938. Hence the expected number of bit comparisons is asymptotically different from that of *key* comparisons required to find the smallest key only by a constant factor (the expectation for key comparisons is asymptotically $2n$). Complex-analytic methods are utilized to obtain the asymptotic formula; in a future paper, it will be shown how the linear lead-order asymptotics $\mu(1,n) \sim cn$ [with c given in the form (1.2)] can be obtained without resort to complex analysis. An outline of the proof of Theorem 1.1 is provided in Section 3.

We also derive exact and asymptotic expressions for the expected number of bit comparisons for the average case. We denote this expectation by $\mu(\bar{m},n)$. In the average case, the parameter m in $\mu(m,n)$ is considered a discrete uniform random variable; hence $\mu(\bar{m},n) = \frac{1}{n}\sum_{m=1}^{n}\mu(m,n)$. The derived asymptotic formula shows that $\mu(\bar{m},n)$ is also asymptotically linear in n; see (4.11). More detailed results for $\mu(\bar{m},n)$ are described in Section 4.

Lastly, in Section 5, we derive an exact expression of $\mu(m,n)$ for each fixed m that is suited for computations. Our preliminary exact formula for $\mu(m,n)$ [shown in (2.7)] entails infinite summation and integration. As a result, it is not a desirable form for numerically computing the expected number of bit comparisons. Hence we establish another exact formula that only requires finite summation and use it to compute $\mu(m,n)$ for $m = 1,\ldots,n$, $n = 2,\ldots,25$. The computation leads to the following conjectures: (i) for fixed n, $\mu(m,n)$ [which of course is symmetric about $(n+1)/2$] increases in m for $m \le (n+1)/2$; and (ii) for fixed m, $\mu(m,n)$ increases in n (asymptotically linearly).

Space limitations on this extended abstract force us to omit a substantial portion of the details of our study. We refer the interested reader to our full-length paper [4].

2 Preliminaries

To investigate the bit complexity of `Quickselect`, we follow the general approach developed by Fill and Janson [3]. Let $U_1, \ldots, U_n$ denote the n keys uniformly and independently distributed on (0, 1), and let $U_{(i)}$ denote the rank-i key. Then, for $1 \le i < j \le n$ (assume $n \ge 2$),

$$P\{U_{(i)} \text{ and } U_{(j)} \text{ are compared}\} = \begin{cases} \dfrac{2}{j-m+1} & \text{if } m \le i \\ \dfrac{2}{j-i+1} & \text{if } i < m < j \\ \dfrac{2}{m-i+1} & \text{if } j \le m. \end{cases} \tag{2.1}$$

To determine the first probability in (2.1), note that $U_{(m)}, \ldots, U_{(j)}$ remain in the same subset until the first time that one of them is chosen as a pivot. Therefore, $U_{(i)}$ and $U_{(j)}$ are compared if and only if the first pivot chosen from $U_{(m)}, \ldots, U_{(j)}$ is either $U_{(i)}$ or $U_{(j)}$. Analogous arguments establish the other two cases.

For $0 < s < t < 1$, it is well known that the joint density function of $U_{(i)}$ and $U_{(j)}$ is given by

$$f_{U_{(i)},U_{(j)}}(s,t) := \binom{n}{i-1, 1, j-i-1, 1, n-j} \times s^{i-1}(t-s)^{j-i-1}(1-t)^{n-j}. \tag{2.2}$$

Clearly, the event that $U_{(i)}$ and $U_{(j)}$ are compared is independent of the random variables $U_{(i)}$ and $U_{(j)}$. Hence, defining

$$P_1(s,t,m,n) := \sum_{m \le i < j \le n} \frac{2}{j-m+1} f_{U_{(i)},U_{(j)}}(s,t), \tag{2.3}$$

$$P_2(s,t,m,n) := \sum_{1 \le i < m < j \le n} \frac{2}{j-i+1} f_{U_{(i)},U_{(j)}}(s,t), \tag{2.4}$$

$$P_3(s,t,m,n) := \sum_{1 \le i < j \le m} \frac{2}{m-i+1} f_{U_{(i)},U_{(j)}}(s,t), \tag{2.5}$$

$$P(s,t,m,n) := P_1(s,t,m,n) + P_2(s,t,m,n) + P_3(s,t,m,n) \tag{2.6}$$

[the sums in (2.3)–(2.5) are double sums over i and j], and letting $\beta(s,t)$ denote the index of the first bit at which the keys s and t differ, we can write the expectation $\mu(m,n)$ of the number of bit comparisons required to find the rank-m key in a file of n keys as

$$\begin{aligned} \mu(m,n) &= \int_0^1 \int_s^1 \beta(s,t) P(s,t,m,n)\, dt\, ds \\ &= \sum_{k=0}^{\infty} \sum_{l=1}^{2^k} \int_{(l-1)2^{-k}}^{(l-\frac{1}{2})2^{-k}} \int_{(l-\frac{1}{2})2^{-k}}^{l2^{-k}} (k+1) \\ &\qquad \times P(s,t,m,n)\, dt\, ds; \end{aligned} \tag{2.7}$$

in this expression, note that k represents the last bit at which s and t agree.

3 Analysis of $\mu(1,n)$

In Section 3.1, we outline a derivation of the exact expression for $\mu(1,n)$ shown in Theorem 1.1; see the full paper [4] for the numerous suppressed details of the various computations. In Section 3.2, we prove the asymptotic result asserted in Theorem 1.1.

3.1 Exact Computation of $\mu(1,n)$ Since the contribution of $P_2(s,t,m,n)$ or $P_3(s,t,m,n)$ to $P(s,t,m,n)$ is zero for $m = 1$, we have $P(s,t,1,n) = P_1(s,t,1,n)$ [see (2.4) through (2.6)]. Let $x := s$, $y := t - s$, $z := 1 - t$. Then

$$\begin{aligned} &P_1(s,t,1,n) \\ &\quad = z^n \sum_{1 \le i < j \le n} \frac{2}{j} \binom{n}{i-1, 1, j-i-1, 1, n-j} \\ &\qquad \times x^{i-1} y^{j-i-1} z^{-j} \\ &\quad = 2 \sum_{j=2}^{n} (-1)^j \binom{n}{j} t^{j-2}. \end{aligned} \tag{3.1}$$

From (2.7) and (3.1),

$$\mu(1,n) = \sum_{j=2}^{n} \frac{(-1)^j \binom{n}{j}}{j-1} \sum_{k=0}^{\infty} (k+1) 2^{-kj} \sum_{l=1}^{2^k} [l^{j-1} - (l-\tfrac{1}{2})^{j-1}]. \tag{3.2}$$

To further transform (3.2), define

$$a_{j,r} = \begin{cases} \dfrac{B_r}{r} \binom{j-1}{r-1} & \text{if } r \ge 2 \\ \frac{1}{2} & \text{if } r = 1 \\ \frac{1}{j} & \text{if } r = 0, \end{cases} \tag{3.3}$$

where B_r denotes the r-th Bernoulli number. Let $S_{n,j} := \sum_{l=1}^{n} l^{j-1}$. Then $S_{n,j} = \sum_{r=0}^{j-1} a_{j,r} n^{j-r}$ (see Knuth [11]), and

$$\begin{aligned}\mu(1,n) &= 2\sum_{j=2}^{n} \frac{(-1)^j \binom{n}{j}}{j-1} \sum_{k=0}^{\infty} (k+1)2^{-kj} \\ &\quad \times \sum_{r=1}^{j-1} a_{j,r} 2^{k(j-r)}(1-2^{-r}) \\ &= 2n(H_n - 1) + 2t_n, \end{aligned} \tag{3.4}$$

where H_n denotes the n-th harmonic number and

$$t_n := \sum_{j=2}^{n-1} \frac{B_j}{j(1-2^{-j})} \left[\frac{n - \binom{n}{j}}{j-1} - 1 \right]. \tag{3.5}$$

3.2 Asymptotic Analysis of $\mu(1,n)$ In order to obtain an asymptotic expression for $\mu(1,n)$, we analyze t_n in (3.4)–(3.5). The following lemma provides an exact expression for t_n that easily leads to an asymptotic expression for $\mu(1,n)$:

LEMMA 3.1. *Let γ denote Euler's constant ($\doteq 0.57722$), and define $\chi_k := \frac{2\pi i k}{\ln 2}$. Then*

$$\begin{aligned} t_n &= -(nH_n - n - 1) + a(n-2) \\ &\quad - \frac{1}{2\ln 2}\left[H_n^2 + H_n^{(2)} - \frac{7}{2}\right] \\ &\quad + \left(\frac{\gamma - 1}{\ln 2} - \frac{1}{2}\right)\left(H_n - \frac{3}{2}\right) \\ &\quad + b - \Sigma_n, \end{aligned}$$

where

$$\begin{aligned} a &:= \frac{14}{9} + \frac{17 - 6\gamma}{18\ln 2} \\ &\quad - \frac{2}{\ln 2} \sum_{k \in \mathbb{Z}\setminus\{0\}} \frac{\zeta(1-\chi_k)\Gamma(1-\chi_k)}{\Gamma(4-\chi_k)(1-\chi_k)}, \\ b &:= \sum_{k \in \mathbb{Z}\setminus\{0\}} \frac{2\zeta(1-\chi_k)\Gamma(-\chi_k)}{(\ln 2)(1-\chi_k)\Gamma(3-\chi_k)}, \\ \Sigma_n &:= \sum_{k \in \mathbb{Z}\setminus\{0\}} \frac{\zeta(1-\chi_k)\Gamma(-\chi_k)\Gamma(n+1)}{(\ln 2)(1-\chi_k)\Gamma(n+1-\chi_k)}, \end{aligned}$$

and $H_n^{(2)}$ denotes the n-th Harmonic number of order 2, i.e., $H_n^{(2)} := \sum_{i=1}^{n} \frac{1}{i^2}$.

The proof of the lemma involves complex-analytic techniques and is rather lengthy, so it is omitted in this extended abstract; see our full-length paper [4]. From (3.4), the exact expression for t_n also provides an alternative exact expression for $\mu(1,n)$.

Using Lemma 3.1, we complete the proof of Theorem 1.1. We know

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + O(n^{-4}), \tag{3.6}$$

$$H_n^{(2)} = \frac{\pi^2}{6} - \frac{1}{n} + \frac{1}{2n^2} + O(n^{-3}). \tag{3.7}$$

Combining (3.6)–(3.7) with (3.4) and Lemma 3.1, we obtain an asymptotic expression for $\mu(1,n)$:

$$\mu(1,n) = 2an - \frac{1}{\ln 2}(\ln n)^2 - \left(\frac{2}{\ln 2} + 1\right)\ln n + O(1). \tag{3.8}$$

The term $O(1)$ in (3.8) has fluctuations of small magnitude due to Σ_n, which is periodic in $\log n$ with amplitude smaller than 0.00110. Thus, as asserted in Theorem 1.1, the asymptotic slope in (3.8) is

$$\begin{aligned} c &= 2a \\ &= \frac{28}{9} + \frac{17 - 6\gamma}{9\ln 2} - \frac{4}{\ln 2} \sum_{k \in \mathbb{Z}\setminus\{0\}} \frac{\zeta(1-\chi_k)\Gamma(1-\chi_k)}{\Gamma(4-\chi_k)(1-\chi_k)}. \end{aligned} \tag{3.9}$$

The alternative expression (1.2) for c is established in a forthcoming revision to our full-length paper [4]; this was also done independently by Grabner and Prodinger [5]. As described in their paper, suitable use of Stirling's formula with bounds allows one to compute c very rapidly to many decimal places.

4 Analysis of the Average Case: $\mu(\bar{m}, n)$

4.1 Exact Computation of $\mu(\bar{m}, n)$ Here we consider the parameter m in $\mu(m,n)$ as a discrete random variable with uniform probability mass function $P\{m = i\} = 1/n$, $i = 1, 2, \ldots, n$, and average over m while the parameter n is fixed. Thus, using the notation defined in Section 2,

$$\mu(\bar{m}, n) = \mu_1(\bar{m}, n) + \mu_2(\bar{m}, n) + \mu_3(\bar{m}, n),$$

where, for $l = 1, 2, 3$,

$$\mu_l(\bar{m}, n) = \int_0^1 \int_s^1 \beta(s,t)\ \frac{1}{n} \sum_{m=1}^{n} P_l(s,t,m,n)\ dt\, ds. \tag{4.1}$$

Here $\mu_1(\bar{m}, n) = \mu_3(\bar{m}, n)$ by an easy symmetric argument we omit, and so

$$\mu(\bar{m}, n) = 2\mu_1(\bar{m}, n) + \mu_2(\bar{m}, n); \tag{4.2}$$

we will compute $\mu_1(\bar{m}, n)$ and $\mu_2(\bar{m}, n)$ exactly in Section 4.1.1.

4.1.1 Exact Computation of $\mu(\bar{m}, n)$ We use the following lemma in order to compute $\mu_1(\bar{m}, n)$ exactly:

LEMMA 4.1.

$$\begin{aligned}
&\int_0^1 \int_s^1 \beta(s,t)\, \frac{1}{n} \sum_{m=2}^{n} P_1(s,t,m,n)\, dt\, ds \\
&= 2\sum_{j=2}^{n-1} \frac{(-1)^j \binom{n-1}{j}}{j(j-1)} + \frac{2}{9}\sum_{j=2}^{n-1} \frac{(-1)^j \binom{n-1}{j}}{j-1} \\
&\quad -2\sum_{j=3}^{n-1} B_j \frac{n-j+1-\binom{n-1}{j-1}}{j(j-1)(j-2)(1-2^{-j})} \\
&\quad -2\sum_{j=2}^{n-1} \frac{(-1)^j \binom{n-1}{j}}{(j+1)j(j-1)(1-2^{-j})}.
\end{aligned}$$

Space limitations on this extended abstract do not allow us to prove this lemma here; we give the proof in our full-length paper [4]. Since

$$\begin{aligned}
\mu_1(\bar{m}, n) &= \frac{1}{n}\mu(1,n) \\
&\quad + \int_0^1 \int_s^1 \beta(s,t)\, \frac{1}{n} \sum_{m=2}^{n} P_1(s,t,m,n)\, dt\, ds,
\end{aligned}$$

it follows from (3.4) and Lemma 4.1 that

$$\begin{aligned}
\mu_1(\bar{m}, n) &= n - 1 - 4\sum_{j=3}^{n} \frac{(-1)^j \binom{n-1}{j-1}}{j(j-1)(j-2)} \\
&\quad + \frac{2}{n}\sum_{j=2}^{n-1} B_j \frac{n-j+1-\binom{n}{j}}{j(j-1)(1-2^{-j})} \\
&\quad + \frac{2}{9}\sum_{j=2}^{n-1} \frac{(-1)^j \binom{n-1}{j}}{j-1} \\
&\quad -2\sum_{j=3}^{n-1} B_j \frac{n-j+1-\binom{n-1}{j-1}}{j(j-1)(j-2)(1-2^{-j})} \\
&\quad -2\sum_{j=2}^{n-1} \frac{(-1)^j \binom{n-1}{j}}{(j+1)j(j-1)(1-2^{-j})}. \qquad (4.3)
\end{aligned}$$

Similarly, after laborious calculations, one can show that

$$\mu_2(\bar{m}, n) = -\frac{4}{n}\sum_{j=2}^{n} \frac{(-1)^j \binom{n}{j}}{j(j-1)[1-2^{-(j-1)}]} + 2(n-1). \qquad (4.4)$$

From (4.2)–(4.4), we obtain

$$\begin{aligned}
&\mu(\bar{m}, n) \\
&= 2(n-1) - 8\sum_{j=3}^{n} \frac{(-1)^j \binom{n-1}{j-1}}{j(j-1)(j-2)} \\
&\quad + \frac{4}{n}\sum_{j=2}^{n-1} B_j \frac{n-j+1-\binom{n}{j}}{j(j-1)(1-2^{-j})} \\
&\quad + \frac{4}{9}\sum_{j=2}^{n-1} \frac{(-1)^j \binom{n-1}{j}}{j-1} \\
&\quad -4\sum_{j=3}^{n-1} B_j \frac{n-j+1-\binom{n-1}{j-1}}{j(j-1)(j-2)(1-2^{-j})} \\
&\quad -4\sum_{j=2}^{n-1} \frac{(-1)^j \binom{n-1}{j}}{(j+1)j(j-1)(1-2^{-j})} \\
&\quad -\frac{4}{n}\sum_{j=2}^{n} \frac{(-1)^j \binom{n}{j}}{j(j-1)[1-2^{-(j-1)}]} + 2(n-1). \qquad (4.5)
\end{aligned}$$

We rewrite or combine some of the terms in (4.5) for the asymptotic analysis of $\mu(\bar{m}, n)$ described in the next section. We define

$$\begin{aligned}
F_1(n) &:= \sum_{j=3}^{n} \frac{(-1)^j \binom{n}{j}}{(j-1)(j-2)}, \\
F_2(n) &:= \sum_{j=2}^{n-1} \frac{B_j}{j(1-2^{-j})} \left[\frac{n-\binom{n}{j}}{j-1} - 1\right], \\
F_3(n) &:= \sum_{j=2}^{n-1} \frac{(-1)^j \binom{n-1}{j}}{j-1} \\
F_4(n) &:= \sum_{j=3}^{n-1} \frac{B_j}{j(j-1)(1-2^{-j})} \left[\frac{n-1-\binom{n-1}{j-1}}{j-2} - 1\right], \\
F_5(n) &:= \sum_{j=3}^{n} \frac{(-1)^j \binom{n}{j}}{j(j-1)(j-2)[1-2^{-(j-1)}]}.
\end{aligned}$$

Then

$$\begin{aligned}
\mu(\bar{m}, n) &= 2(n-1) - \tfrac{8}{n}F_1(n) + \tfrac{4}{n}F_2(n) + \tfrac{4}{9}F_3(n) \\
&\quad -4F_4(n) + \tfrac{8}{n}F_5(n). \qquad (4.6)
\end{aligned}$$

4.2 Asymptotic Analysis of $\mu(\bar{m}, n)$ We derive an asymptotic expression for $\mu(\bar{m}, n)$ shown in (4.6).

Routine arguments show that

$$F_1(n) = -\frac{1}{2}n^2\ln n + \left(\frac{5}{4} - \frac{\gamma}{2}\right)n^2 - n\ln n + \frac{n^2}{2(n-1)} - (\gamma+1)n + O(1), \tag{4.7}$$

$$F_3(n) = n\ln n + (\gamma-1)n - \ln n + O(1), \tag{4.8}$$

$$F_4(n) = \frac{1}{9}n\ln n + \left(\tilde{a} + \frac{1}{9}\gamma - \frac{1}{9}\right)n + \frac{8}{9}\ln n + O(1), \tag{4.9}$$

$$F_5(n) = -\frac{1}{2}n^2\ln n + \frac{3+\ln 2-\gamma}{2}n^2 - \frac{1}{2\ln 2}n(\ln n)^2 + \left(\frac{1}{\ln 2} - \frac{1}{2}\right)n\ln n + O(n), \tag{4.10}$$

where

$$\tilde{a} := \frac{7}{36\ln 2} - \frac{41}{72} - \frac{\gamma}{12\ln 2} - \sum_{k\in Z\setminus\{0\}} \frac{\zeta(1-\chi_k)\Gamma(1-\chi_k)}{(\ln 2)(2-\chi_k)\Gamma(4-\chi_k)}.$$

Since $F_2(n)$ is equal to t_n, which is defined at (3.5) and analyzed in Section 3.2, we already have an asymptotic expression for $F_2(n)$. Therefore, from (4.6)–(4.10), we obtain the following asymptotic formula for $\mu(\bar{m}, n)$:

$$\mu(\bar{m}, n) = 4(1+\ln 2-\tilde{a})n - \frac{4}{\ln 2}(\ln n)^2 + 4\left(\frac{2}{\ln 2} - 1\right)\ln n + O(1). \tag{4.11}$$

The asymptotic slope $4(1+\ln 2-\tilde{a})$ is approximately 8.20731. We have not (yet) sought an alternative form for $\tilde{a}$ like that for c in (1.2).

5 Derivation of a Closed Formula for $\mu(m, n)$

The exact expression for $\mu(m, n)$ obtained in Section 2 [see (2.7)] involves infinite summation and integration. Hence it is not a preferable form for numerically computing the expectation. In this section, we establish another exact expression for $\mu(m, n)$ that only involves finite summation. We also use the formula to compute $\mu(m, n)$ for $m = 1, \ldots, n$, $n = 2, \ldots, 20$.

As described in Section 2, it follows from equations (2.6)–(2.7) that

$$\mu(m,n) = \mu_1(m,n) + \mu_2(m,n) + \mu_3(m,n),$$

where, for $q = 1, 2, 3$,

$$\mu_q(m,n) := \sum_{k=0}^{\infty}\sum_{l=1}^{2^k}\int_{s=(l-1)2^{-k}}^{(l-\frac{1}{2})2^{-k}}\int_{t=(l-\frac{1}{2})2^{-k}}^{l2^{-k}}(k+1) \times P_q(s,t,m,n)\,dt\,ds. \tag{5.1}$$

The same technique can be applied to eliminate the infinite summation and integration from each $\mu_q(m, n)$. We describe the technique for obtaining a closed expression of $\mu_1(m, n)$.

First, we transform $P_1(s, t, m, n)$ shown in (2.3) so that we can eliminate the integration in $\mu_1(m, n)$. Define

$$C_1(i,j) := I\{1 \le m \le i < j \le n\} \times \frac{2}{j-m+1}\binom{n}{i-1, 1, j-i-1, 1, n-j}, \tag{5.2}$$

where $I\{1 \le m \le i < j \le n\}$ is an indicator function that equals 1 if the event in braces holds and 0 otherwise. Then

$$P_1(s,t,m,n) = \sum_{f=m-1}^{n-2}\sum_{h=0}^{n-f-2} s^f t^h C_2(f,h), \tag{5.3}$$

where

$$C_2(f,h) := \sum_{i=m}^{f+1}\sum_{j=f+2}^{f+h+2} C_1(i,j)\binom{j-i-1}{f-i+1} \times \binom{n-j}{h-j+f+2}(-1)^{h-i-j+1}.$$

Thus, from (5.1) and (5.3), we can eliminate the integration in $\mu_1(m, n)$ and express it using polynomials in l:

$$\mu_1(m,n) = \sum_{f=m-1}^{n-2}\sum_{h=0}^{n-f-2} C_3(f,h)\sum_{k=0}^{\infty}(k+1) \times \sum_{l=1}^{2^k} 2^{-k(f+h+2)}[l^{h+1} - (l-\tfrac{1}{2})^{h+1}] \times [(l-\tfrac{1}{2})^{f+1} - (l-1)^{f+1}], \tag{5.4}$$

where

$$C_3(f,h) := \frac{1}{(n+1)(f+1)}C_2(f,h).$$

One can show that

$$\left[l^{h+1} - \left(l - \frac{1}{2}\right)^{h+1}\right]\left[\left(l - \frac{1}{2}\right)^{f+1} - (l-1)^{f+1}\right]$$

$$= \sum_{j=1}^{f+h+1} C_4(f,h,j) l^{j-1}, \tag{5.5}$$

where

$$C_4(f,h,j) := (-1)^{f+h-j+1}\left(\frac{1}{2}\right)^{h-j+2} \times \sum_{j'=0 \bigvee (j-1-h)}^{(j-1)\bigwedge f} \binom{f+1}{j'}\binom{h+1}{j-1-j'} \times \left[1 - \left(\frac{1}{2}\right)^{f+1-j'}\right]\left(\frac{1}{2}\right)^{j'}.$$

From (5.4)–(5.5), we obtain

$$\mu_1(m,n) = \sum_{f=m-1}^{n-2}\sum_{h=0}^{n-f-2}\sum_{j=1}^{f+h+1} C_5(f,h,j) \times \sum_{k=0}^{\infty}(k+1)2^{-k(f+h+2)}\sum_{l=1}^{2^k} l^{j-1},$$

where

$$C_5(f,h,j) := C_3(f,h) \cdot C_4(f,h,j).$$

Here, as described in Section 3.1,

$$\sum_{l=1}^{2^k} l^{j-1} = \sum_{r=0}^{j-1} a_{j,r} 2^{k(j-r)},$$

where $a_{j,r}$ is defined by (3.3). Now define

$$C_6(f,h,j,r) := a_{j,r}\ C_5(f,h,j).$$

Then

$$\mu_1(m,n) = \sum_{a=1}^{n-1} C_7(a)(1-2^{-a})^{-2}, \tag{5.6}$$

where

$$C_7(a) := \sum_{f=m-1}^{n-2}\sum_{h=\alpha}^{n-f-2}\sum_{j=\beta}^{f+h+1} C_6(f,h,j,a+j-(f+h+2)),$$

in which $\alpha := 0 \bigvee (a-f-1)$ and $\beta := 1 \bigvee (f+h+2-a)$.

The procedure described above can be applied to derive analogous exact formulae for $\mu_2(m,n)$ and $\mu_3(m,n)$. In order to derive the analogous exact formula for $\mu_2(m,n)$, one need only start the derivation by changing the indicator function in $C_1(i,j)$ [see (5.2)] to $I\{1 \le i < m < j \le n\}$ and follow each step of the procedure; similarly, for $\mu_3(m,n)$, one need only start the derivation by changing the indicator function to $I\{1 \le i < j \le m \le n\}$.

Using the closed exact formulae of $\mu_1(m,n)$, $\mu_2(m,n)$, and $\mu_3(m,n)$, we computed $\mu(m,n)$ for $n = 2,3,\ldots,20$ and $m = 1,2,\ldots,n$. Figure 1 shows the results, which suggest the following: (i) for fixed n, $\mu(m,n)$ [which of course is symmetric about $(n+1)/2$] increases in m for $m \le (n+1)/2$; and (ii) for fixed m, $\mu(m,n)$ increases in n (asymptotically linearly).

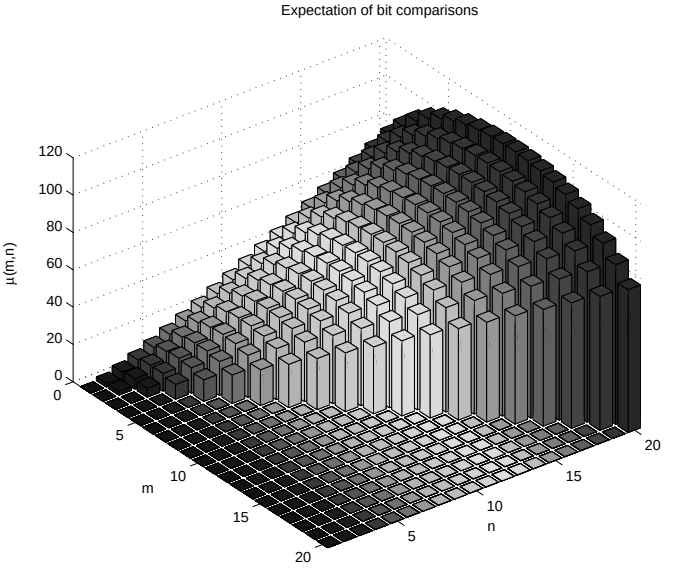


Figure 1: Expected number of bit comparisons for `Quickselect`. The closed formulae for $\mu_1(m,n)$, $\mu_2(m,n)$, and $\mu_3(m,n)$ were used to compute $\mu(m,n)$ for $n = 1,2,\ldots,20$ (n represents the number of keys) and $m = 1,2,\ldots,n$ (m represents the rank of the target key).

6 Discussion

Our investigation of the bit complexity of `Quickselect` revealed that the expected number of bit comparisons required by `Quickselect` to find the smallest or largest key from a set of n keys is asymptotically linear in n with the asymptotic slope approximately equal to

5.27938. Hence asymptotically it differs from the expected number of *key* comparisons to achieve the same task only by a constant factor. (The expectation for key comparisons is asymptotically $2n$; see Knuth [10] and Mahmoud *et al.* [13]). This result is rather contrastive to the `Quicksort` case in which the expected number of bit comparisons is asymptotically $n(\ln n)(\lg n)$ whereas the expected number of key comparisons is asymptotically $2n \ln n$ (see Fill and Janson [3]). Our analysis also showed that the expected number of bit comparisons for the average case remains asymptotically linear in n with the lead-order coefficient approximately equal to 8.20731. Again, the expected number is asymptotically different from that of key comparisons for the average case only by a constant factor. (The expected number of key comparisons for the average case is asymptotically $3n$; see Mahmoud *et al.* [13]).

Although we have yet to establish a formula analogous to (3.4) and (4.6) for the expected number of bit comparisons to find the m-th key for fixed m, we established an exact expression that only requires finite summation and used it to obtain the results shown in Figure 1. However, the formula remains computationally complex. Written as a single expression, $\mu(m, n)$ is a seven-fold sum of rather elementary terms with each sum having order n terms (in the worst case); in this sense, the running time of the algorithm for computing $\mu(m, n)$ is of order n^7. The expression for $\mu(m, n)$ does not allow us to derive an asymptotic formula for it or to prove the two intuitively obvious observations described at the end of Section 5. The situation is substantially better for the expected number of *key* comparisons to find the m-th key from a set of n keys; Knuth [10] showed that the expectation can be written as $2[n+3+(n+1)H_n-(m+2)H_m-(n+3-m)H_{n+1-m}]$.

In this extended abstract, we considered independent and uniformly distributed keys in (0,1). In this case, each bit in a bit-string key is 1 with probability 0.5. In ongoing research, we generalize the model and suppose that each bit results from an independent Bernoulli trial with success probability p. The more general results of that research will further elucidate the bit complexity of `Quickselect` and other algorithms.

Acknowledgment. We thank Philippe Flajolet, Svante Janson, and Helmut Prodinger for helpful discussions.

References

[1] L. Devroye. On the probablistic worst-case time of "Find". *Algorithmica*, 31:291–303, 2001.

[2] J. A. Fill and S. Janson. Quicksort asymptotics. *Journal of Algorithms*, 44:4–28, 2002.

[3] J. A. Fill and S. Janson. The number of bit comparisons used by Quicksort: An average-case analysis. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 293–300, 2004.

[4] J. A. Fill and T. Nakama. Analysis of the expected number of bit comparisons required by Quickselect. *http://front.math.ucdavis.edu/0706.2437*, 2007.

[5] P. J. Grabner and H. Prodinger. On a constant arising in the analysis of bit comparisons in Quickselect. *Preprint*, 2007.

[6] R. Grübel and U. Rösler. Asymptotic distribution theory for Hoare's selection algorithm. *Advances in Applied Probability*, 28:252–269, 1996.

[7] C. R. Hoare. Find (algorithm 65). *Communications of the ACM*, 4:321–322, 1961.

[8] H. Hwang and T. Tsai. Quickselect and the Dickman function. *Combinatorics, Probability and Computing*, 11:353–371, 2002.

[9] C. Knessl and W. Szpankowski. Quicksort algorithm again revisited. *Discrete Mathematics and Theoretical Computer Science*, 3:43–64, 1999.

[10] D. E. Knuth. Mathematical analysis of algorithms. In *Information Processing 71 (Proceedings of IFIP Congress, Ljubljana, 1971)*, pages 19–27. North-Holland, Amsterdam, 1972.

[11] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1998.

[12] J. Lent and H. M. Mahmoud. Average-case analysis of multiple Quickselect: An algorithm for finding order statistics. *Statistics and Probability Letters*, 28:299–310, 1996.

[13] H. M. Mahmoud, R. Modarres, and R. T. Smythe. Analysis of Quickselect: An algorithm for order statistics. *RAIRO Informatique Théorique et Applications*, 29:255–276, 1995.

[14] H. M. Mahmoud and R. T. Smythe. Probabilistic analysis of multiple Quickselect. *Algorithmica*, 22:569–584, 1998.

[15] R. Neininger and L. Rüschendorf. Rates of convergence for Quickselect. *Journal of Algorithm*, 44:51–62, 2002.

[16] M. Régnier. A limiting distribution of Quicksort. *RAIRO Informatique Théorique et Applications*, 23:335–343, 1989.

[17] U. Rösler. A limit theorem for Quicksort. *RAIRO Informatique Théorique et Applications*, 25:85–100, 1991.