Juha Kärkkäinen
Jens Stoye (Eds.)

LNCS 7354

Combinatorial Pattern Matching

23rd Annual Symposium, CPM 2012
Helsinki, Finland, July 2012
Proceedings

Springer

Juha Kärkkäinen   Jens Stoye (Eds.)

Combinatorial Pattern Matching

23rd Annual Symposium, CPM 2012
Helsinki, Finland, July 3-5, 2012
Proceedings

Springer

Volume Editors

Juha Kärkkäinen
University of Helsinki, Department of Computer Science
P.O. Box 68, Gustaf Hällströmin katu 2b, 00014 University of Helsinki, Finland
E-mail: juha.karkkainen@cs.helsinki.fi

Jens Stoye
Bielefeld University, Faculty of Technology
Universitätsstraße 25, 33615 Bielefeld, Germany
E-mail: jens.stoye@uni-bielefeld.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-31264-9 e-ISBN 978-3-642-31265-6
DOI 10.1007/978-3-642-31265-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012939778

CR Subject Classification (1998): I.5, F.2, G.1, G.2.1-2, G.2, E.1, E.4, H.3.3, I.2.7, H.2.8, J.3

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

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

Printed on acid-free paper

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

Preface

The papers contained in this volume were selected to be presented at the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM 2012) held in Helsinki, Finland, during July 3–5, 2012, co-located with the 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2012).

The conference program included 33 contributed papers and two invited talks by Ron Shamir from Tel Aviv University and by Gonzalo Navarro from University of Chile in Santiago. The contributed papers were selected out of 60 submissions from 28 countries; each paper was reviewed by at least three reviewers. The joint program with the co-located SWAT 2012 conference included additional 34 contributed papers and two invited talks, by Joseph S.B. Mitchell from State University of New York at Stony Brook and by Roger Wattenhofer from ETH Zürich, published in a separate volume.

The Program Committee decided to grant the Best Student Paper Award to Paweł Gawrychowski for the paper titled "Simple and Efficient LZW-Compressed Multiple Pattern Matching".

The objective of the annual CPM meetings is to provide an international forum for research in combinatorial pattern matching and related applications. It addresses issues of searching and matching strings and more complicated patterns such as trees, regular expressions, graphs, point sets, and arrays. The goal is to derive non-trivial combinatorial properties of such structures and to exploit these properties in order to either achieve superior performance for the corresponding computational problems or pinpoint conditions under which searches cannot be performed efficiently. The meeting also deals with problems in computational biology, data compression and data mining, coding, information retrieval, natural language processing, and pattern recognition.

The Annual Symposium on Combinatorial Pattern Matching started in 1990, and has since taken place every year. Previous CPM meetings were held in Paris, London (UK), Tucson, Padova, Asilomar, Helsinki, Laguna Beach, Aarhus, Piscataway, Warwick, Montreal, Jerusalem, Fukuoka, Morelia, Istanbul, Jeju Island, Barcelona, London (Canada), Ontario, Pisa, Lille, New York, and Palermo. Helsinki is the first place to host the meeting twice.

Starting from the third meeting, proceedings of all meetings have been published in the LNCS series, as volumes 644, 684, 807, 937, 1075, 1264, 1448, 1645, 1848, 2089, 2373, 2676, 3109, 3537, 4009, 4580, 5029, 5577, 6129, and 6661.

Selected papers from the first meeting appeared in volume 92 of *Theoretical Computer Science,* from the 11th meeting in volume 2 of the *Journal of Discrete Algorithms,* from the 12th meeting in volume 146 of *Discrete Applied Mathematics,* from the 14th meeting in volume 3 of the *Journal of Discrete Algorithms,* from the 15th meeting in volume 368 of *Theoretical Computer Science,* from the 16th meeting in volume 5 of the *Journal of Discrete Algorithms,* from the 19th meeting in volume 410 of *Theoretical Computer Science,* from the 20th meeting in volume 9 of the *Journal of Discrete Algorithms,* from the 21st meeting in volume 213 of *Information and Computation,* and a special issue of *Theoretical Computer Science* is currently in preparation for selected papers from CPM 2011. A special issue of the *Journal of the Discrete Algorithms* is planned for the selected extended abstracts presented at this year's meeting.

We thank the CPM Steering Committee for supporting Helsinki as the site for CPM 2012, and for their advice and help in different issues. Financial and organizational support was provided by the University of Helsinki, Aalto University, and the Federation of the Finnish Learned Societies. The meeting would not have been possible without the laborious work of the local organizing teams of CPM 2012 and SWAT 2012.

The whole submission and review process was carried out with the invaluable help of the EasyChair conference system. Finally, special thanks are due to the members of the Program Committee and their subreviewers who worked very hard to ensure the timely review of all the submitted manuscripts, and participated in stimulating discussions that led to the selection of the papers for the conference.

April 2012

Juha Kärkkäinen
Jens Stoye

Organization

Program Committee

Hiroki Arimura	Hokkaido University, Japan
Diego Arroyuelo	Universidad Técnica Federico Santa María, Chile
Hideo Bannai	Kyushu University, Japan
Nieves R. Brisaboa	University of A Coruña, Spain
Ferdinando Cicalese	University of Salerno, Italy
Martin Farach-Colton	Rutgers University and Tokutek Inc., USA
Simone Faro	University of Catania, Italy
Kimmo Fredriksson	University of Eastern Finland, Finland
Inge Li Gørtz	Technical University of Denmark, Denmark
Shmuel Tomi Klein	Bar Ilan University, Israel
Roman Kolpakov	Moscow University, Russia
Juha Kärkkäinen	University of Helsinki, Finland (Co-chair)
Gad M. Landau	University of Haifa, Israel, and NYU-Poly, USA
Thierry Lecroq	University of Rouen, France
Bin Ma	University of Waterloo, Canada
Jan Manuch	University of British Columbia, Canada, and Simon Fraser University, Canada
Mehryar Mohri	Courant Institute of Mathematical Science, USA, and Google Research, New York, USA
Enno Ohlebusch	University of Ulm, Germany
Heejin Park	Hanyang University, South Korea
Mathieu Raffinot	LIAFA, Université Paris Diderot, France
Sven Rahmann	University of Duisburg-Essen, Germany
Wojciech Rytter	Warsaw University, Poland, and Copernicus University, Poland
Yasmín Ríos-Solís	Autonomous University of Nuevo León, Mexico
Marinella Sciortino	University of Palermo, Italy
Rahul Shah	Louisiana State University, USA
Jens Stoye	Bielefeld University, Germany (Co-chair)
Wing-Kin Sung	National University of Singapore, Singapore
Jorma Tarhio	Aalto University, Finland
German Tischler	The Wellcome Trust Sanger Institute, UK
Alexander Tiskin	University of Warwick, UK
Dekel Tsur	Ben Gurion University of the Negev, Israel

Steering Committee

Alberto Apostolico	University of Padova, Italy, and Georgia Institute of Technology, USA
Maxime Crochemore	Université Paris-Est, France, and King's, College London, UK
Zvi Galil	Georgia Institute of Technology, USA

Organizing Committee

Travis Gagie	Aalto University, Finland
Juha Kärkkäinen	University of Helsinki, Finland
Veli Mäkinen	University of Helsinki, Finland
Simon J. Puglisi	King's College London, UK
Leena Salmela	University of Helsinki, Finland
Jouni Sirén	University of Helsinki, Finland
Jorma Tarhio	Aalto University, Finland
Esko Ukkonen	University of Helsinki, Finland
Niko Välimäki	University of Helsinki, Finland

Additional Reviewers

Allauzen, Cyril
Amit, Mika
Andonov, Rumen
Bader, Martin
Bansal, Mukul S.
Bille, Philip
Blin, Guillaume
Breslauer, Dany
Castiglione, Giuseppa
Cid-García, Néstor
Cording, Patrick
Czeizler, Elena
D'Addario, Marianna
Dondi, Riccardo
Ernst, Corinna
Feng, Guangyu
Fertin, Guillaume
Gagie, Travis
Gawrychowski, Paweł
Gog, Simon
Grabowski, Szymon
Groult, Richard
He, Lin
Hermelin, Danny
Hernández-Landa, Leonardo
I, Tomohiro
Ibarra-Rojas, Omar
Inenaga, Shunsuke
Kaltenbach, Hans-Michael
Kannan, Rajgopal
Karhu, Kalle
Kim, Jin Wook
Kim, Sung-Ryul
Kociumaka, Tomasz
Kopelowitz, Tsvi
Kowaluk, Miroslaw
Lee, Inbok
Lefebvre, Arnaud
Marschall, Tobias
Mnich, Matthias
Morales-Marroquín, Miguel
Na, Joong Chae
Patil, Manish
Prieur-Gaston, Élise
Puglisi, Simon J.
Riley, Michael

Rosone, Giovanna
Rozenberg, Liat
Salmela, Leena
Sammeth, Michael
Sim, Jeong Seop
Sirén, Jouni
Szreder, Bartosz
Thachuk, Chris
Thankachan, Sharma V.
Uno, Takeaki
Venturini, Rossano
Vialette, Stéphane
Vildhøj, Hjalte Wedel
Waleń, Tomasz
Weimann, Oren
Yang, Lian

Table of Contents

Gene Regulation, Protein Networks and Disease: A Computational Perspective

Ron Shamir[⋆]

Blavatnik School of Computer Science, Tel Aviv University
rshamir@tau.ac.il

Abstract. Understanding complex disease is one of today's grand challenges. In spite of the rapid advance of biotechnology, disease understanding is still very limited and further computational tools for disease-related data analysis are in dire need. In this talk I will describe some of the approaches that we are developing for these challenges. I will describe methods for utilizing expression profiles of sick and healthy individuals to identify pathways dysregulated in the disease, methods for integrated analysis for expression and protein interactions, and methods for regulatory motif discovery. If time allows, I'll discuss methods for analysis of genome aberrations in cancer. The utility of the methods will be demonstrated on biological examples.

Joint work with Igor Ulitsky, Ofer Lavi, Yaron Orenstein, Richard M. Karp, Gideon Dror, Akshay Krishnamurthy, Michal Ozery-Flato, Chaim Linhart, Luba Trakhtenbrot, Shai Izraeli, Annelyse Thevenin and Liat Ein-Dor.

[⋆] Supported in part by the Raymond and Beverly Sackler Chair in Bioinformatics, the Israel Science Foundation (Grant 802/08), the European Union 7th Framework (TRIREME and APO-SYS grants), and IBM Research Open Collaborative Research grant.

Wavelet Trees for All[⋆]

Gonzalo Navarro

Dept. of Computer Science, University of Chile
gnavarro@dcc.uchile.cl

Abstract. The wavelet tree is a versatile data structure that serves a number of purposes, from string processing to geometry. It can be regarded as a device that represents a sequence, a reordering, or a grid of points. In addition, its space adapts to various entropy measures of the data it encodes, enabling compressed representations. New competitive solutions to a number of problems, based on wavelet trees, are appearing every year. In this survey we give an overview of wavelet trees and the surprising number of applications in which we have found them useful: basic and weighted point grids, sets of rectangles, strings, permutations, binary relations, graphs, inverted indexes, document retrieval indexes, full-text indexes, XML indexes, and general numeric sequences.

1 Introduction

The *wavelet tree* was invented in 2003 by Grossi, Gupta, and Vitter [54], as a data structure to represent a sequence and answer some queries on it. Curiously, a data structure that has turned out to have a myriad of applications was buried in a paper full of other eye-catching results. The first mention to the name "wavelet tree" appears on page 8 of 10 [54, Sec. 4.2]. The last mention is also on page 8, save for a figure caption on page 9. Yet, the wavelet tree was a key tool to obtain the main result of the paper, a milestone in compressed full-text indexing.

It is interesting that, after some thought, one can see that the wavelet tree is a slight generalization of an old (1988) data structure by Chazelle [25], heavily used in Computational Geometry. This data structure represents a set of points on a two-dimensional grid: it describes a successive reshuffling process where the points start sorted by one coordinate and end up sorted by the other. Kärkkäinen, in 1999 [66], was the first to put this structure in use in the completely different context of text indexing. Still, the concept and usage were totally different from the one Grossi et al. would propose four years later.

We have already mentioned three ways in which wavelet trees can be regarded: (*i*) as a representation of a sequence; (*ii*) as a representation of a reordering of elements; (*iii*) as a representation of a grid of points. Since 2003, these views of wavelet trees, and their interactions, have been fruitful in a surprisingly wide range of problems, extending well beyond the areas of text indexing and computational geometry where the structure was conceived.

⋆ Partially funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Chile.

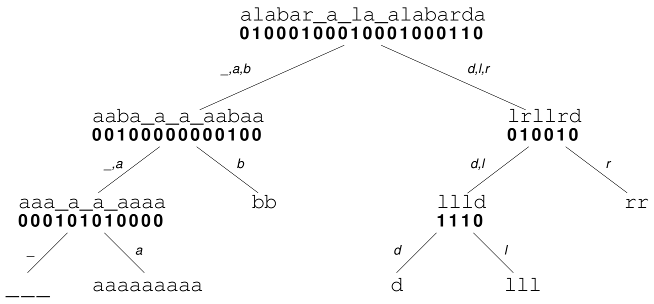


Fig. 1. A wavelet tree on string $S =$ "`alabar a la alabarda`". We draw the spaces as underscores. The subsequences of S and the subsets of Σ labeling the edges are drawn for illustration purposes; the tree stores only the topology and the bitmaps.

Our goal in this article is to give an overview of this marvellous data structure and its many applications. We aim to introduce, to an audience with a general algorithmic background, the basic data organization used by wavelet trees, the information they can model, and the wide range of problems they can solve. We will also mention the most technical results and give the references to be followed by the more knowledgeable readers, advising the rest what to skip.

Being ourselves big fans of wavelet trees, and having squeezed them out for several years, it is inevitable that there will be many references to our own work in this survey. We apologize in advance for this, as well as for oversights of others' results, which are likely to occur despite our efforts.

2 Data Structure

Let $S[1,n] = s_1 s_2 \ldots s_n$ be a sequence of symbols $s_i \in \Sigma$, where $\Sigma = [1..\sigma]$ is called the *alphabet*. Then S can be represented in plain form using $n\lceil \lg \sigma \rceil = n \lg \sigma + O(n)$ bits (we use $\lg x = \log_2 x$).

Structure. A wavelet tree [54] for sequence $S[1,n]$ over alphabet $[1..\sigma]$ can be described recursively, over a sub-alphabet range $[a..b] \subseteq [1..\sigma]$. A wavelet tree over alphabet $[a..b]$ is a binary balanced tree with $b-a+1$ leaves. If $a = b$, the tree is just a leaf labeled a. Else it has an internal root node, v_{root}, that represents $S[1,n]$. This root stores a bitmap $B_{v_{root}}[1,n]$ defined as follows: if $S[i] \leq (a+b)/2$ then $B_{v_{root}}[i] = 0$, else $B_{v_{root}}[i] = 1$. We define $S_0[1,n_0]$ as the subsequence of $S[1,n]$ formed by the symbols $c \leq (a+b)/2$, and $S_1[1,n_1]$ as the subsequence of $S[1,n]$ formed by the symbols $c > (a+b)/2$. Then, the left child of v_{root} is a wavelet tree for $S_0[1,n_0]$ over alphabet $[a..\lfloor (a+b)/2 \rfloor]$ and the right child of v_{root} is a wavelet tree for $S_1[1,n_1]$ over alphabet $[1+\lfloor (a+b)/2 \rfloor..b]$.

Fig. 1 displays a wavelet tree for the sequence $S =$ "`alabar a la alabarda`". Here for legibility we are using $\Sigma = \{$`' '`, `a`, `b`, `d`, `l`, `r`$\}$, so $n = 19$ and $\sigma = 6$.

Note that this wavelet tree has height $\lceil \lg \sigma \rceil$, and it has σ leaves and $\sigma - 1$ internal nodes. If we regard it level by level, it is not hard to see that it stores exactly n bits at each level, and at most n bits in the last one. Thus, $n\lceil\lg\sigma\rceil$ is an upper bound to the total number of bits it stores. Storing the topology of the tree requires $O(\sigma\lg n)$ further bits, if we are careful enough to use $O(\lg n)$ bits for the pointers. This extra space may be a problem on large alphabets. We show in the paragraph "Removing redundancy" how to save it.

Tracking Symbols. This wavelet tree represents S, in the sense that one can recover S from it. More than that, it is a *succinct data structure* for S, in the sense that it takes space asymptotically equal to a plain representation of S, and it permits accessing any $S[i]$ in time $O(\lg\sigma)$, as follows.

To extract $S[i]$, we first examine $B_{v_{root}}[i]$. If it is a 0, we know that $S[i] \le (\sigma+1)/2$, otherwise $S[i] > (\sigma+1)/2$. In the first case, we must continue recursively on the left child; in the second case, on the right child. The problem is to determine where has position i been mapped to on the left (or right) child. In the case of the left child, where $B_{v_{root}}[i] = 0$, i has been mapped to position i_0, which is the number of 0s in $B_{v_{root}}$ up to position i. For the right child, where $B_{v_{root}}[i] = 1$, this corresponds to position i_1, the number of 1s in $B_{v_{root}}$ up to position i. The number of 0s (resp. 1s) up to position i in a bitmap B is called $\mathtt{rank}_0(B,i)$ (resp. $\mathtt{rank}_1(B,i)$). We continue this process recursively until we arrive at a leaf. The label of this leaf is $S[i]$. Note that we do not store the leaf labels; those are deduced as we successively restrict the subrange $[a..b]$ of $[1..\sigma]$ as we descend.

Operation `rank` was already considered by Chazelle [25], who gave a simple data structure using $O(n)$ bits for a bitmap $B[1,n]$, that computed `rank` in constant time (note that we only have to solve $\mathtt{rank}_1(B,i)$, since $\mathtt{rank}_0(B,i) = i - \mathtt{rank}_1(B,i)$). Jacobson [63] improved the space to $n + O(n\lg\lg n/\lg n) = n + o(n)$ bits, and Golynski [48,49] proved this space is optimal as long as we maintain B in plain form and build extra data structures on it. The solution is, essentially, storing `rank` answers every $s = \lg^2 n$ bits of B (using $\lg n$ bits per sample), then storing `rank` answers relative to the last sample every $(\lg n)/2$ bits (using $\lg s = 2\lg\lg n$ bits per sub-sample), and using a universal table to complete the answer to a `rank` query within a sub-sample. We will use in this survey the notation $\mathtt{rank}_b(B,i,j) = \mathtt{rank}_b(B,j) - \mathtt{rank}_b(B,i-1)$.

Above, we have *tracked* a position from the root to a leaf, and as a consequence we have discovered the symbol represented at the root position. It is also useful to carry out the inverse process: given a position at a leaf, we can track it upwards and find out where it is on the root bitmap. This is done as follows.

Assume we start at a given leaf, at position i. If the leaf is the left child of its parent v, then the position i' corresponding to i at v is the i-th occurrence of a 0 in its bitmap B_v. If the leaf is the right child of its parent v, then i' is the position of the i-th occurrence of a 1 in B_v. This procedure is repeated from v until we reach the root, where we find the final position. The operation of finding the i-th 0 (resp. 1) in a bitmap $B[1,n]$ is called $\mathtt{select}_0(B,i)$ (resp. $\mathtt{select}_1(B,i)$), and it can also be solved in constant time using the n bits of B plus $o(n)$ bits [27,79]. Thus the time to track a position upwards is also $O(\lg\sigma)$.

The constant-time solution for `select` [27,79] is analogous to that of `rank`. The bitmap is cut into blocks with s 1s. Those that are long enough to store all their answers within sublinear space are handled in this way. The others are not too long (i.e., $O(\lg^{O(1)} n)$) and thus encoding positions inside them require fewer bits (i.e., $O(\lg\lg n)$). This permits repeating the idea recursively a second time. The third time, the remaining blocks are so short that can be handled in constant time using universal tables. Golynski [48,49] reduced the $o(n)$ extra space to $O(n \lg\lg n/ \lg n)$ and proved this is optimal if B is stored in plain form.

With the support for `rank` and `select`, the space required by the basic binary balanced wavelet tree reaches $n\lceil\lg\sigma\rceil + o(n)\lg\sigma + O(\sigma\lg n)$ bits. This completes a basic description of wavelet trees; the rest of the section is more technical.

Reducing Redundancy. As mentioned, the $O(\sigma \lg n)$ term can be removed if necessary [72,74]. We slightly alter the balanced wavelet tree shape, so that all the leaves are grouped to the left (for this sake we divide the interval $[a..b]$ of $[1..\sigma]$ into $[a..a+2^{\lfloor\lg(b-a+1)\rfloor}-1]$ and $[a+2^{\lfloor\lg(b-a+1)\rfloor}..b]$). Then, all the bitmaps at all the levels belong to consecutive nodes, and they can all be concatenated into a large bitmap $B[1, n\lceil\lg\sigma\rceil]$. We know the bitmap of level ℓ starts at position $1 + n(\ell - 1)$. Moreover, if we have determined that the bitmap of a wavelet tree node corresponds to $B[l, r]$, then the bitmap of its left child is at $B[n+l, n+l+\mathtt{rank}_0(B, l, r)-1]$, and that of the right child is at $B[n+l+\mathtt{rank}_0(B, l, r), n+r]$. Moving to the parent of a node is more complicated, but upward traversals can always be handled by first going down from the root to the desired leaf, so as to discover all the ranges in B of the nodes in the path, and then doing the upward processing as one returns from the recursion.

Using just one bitmap, we do not need pointers for the topology, and the overall space becomes $n\lceil\lg\sigma\rceil + o(n)\lg\sigma$ bits. The time complexities do not change (albeit in practice the operations are slowed down a bit due to the extra `rank` operations needed to navigate [28]).

The redundancy can be further reduced by representing the bitmaps using a structure by Golynski et al. [50], which uses $n + O(n \lg\lg n/ \lg^2 n)$ bits and supports constant-time `rank` and `select` (this representation does not leave the bitmap in plain form, and thus it can break the lower bound [49]). Added over all the wavelet tree bitmaps, the space becomes $n \lg\sigma + O(n\lg\sigma\lg\lg n/\lg^2 n) = n\lg\sigma + o(n)$ bits.[1] This structure has not been implemented as far as we know.

Speeding Up Traversals. Increasing the arity of wavelet trees reduces their height, which dictates the complexity of the downward and upward traversals. If the wavelet tree is d-ary, then its height is $\lceil\lg_d\sigma\rceil$. However, the wavelet tree

[1] We assume $\lg\sigma = O(\lg n)$ here; otherwise there are many symbols that do not appear in S. If this turns out to be the case, one should use a mapping from Σ to the range $[1..\sigma']$, where $\sigma' \le n$ is the number of symbols actually appearing in S. Such a mapping takes constant time and $\sigma'\lg(\sigma/\sigma') + o(\sigma') + O(\lg\lg\sigma)$ bits of space using the "indexable dictionaries" of Raman et al. [93]. Added to the $n\lg\sigma' + o(n)$ bits of the wavelet tree, we are within $n\lg\sigma + o(n) + O(\lg\lg\sigma)$ bits. This is $n\lg\sigma + o(n)$ unless $n = O(\lg\lg\sigma)$, in which case a plain representation of S using $n\lceil\lg\sigma\rceil$ bits solves all the operations in $O(\lg\lg\sigma)$ time. To simplify, a recent analysis [45] claims $n\lg\sigma + O(n)$ bits under similar assumptions. We will ignore the issue from now, and assume for simplicity that all symbols in $[1..\sigma]$ do appear in S.

does not store bitmaps anymore, but rather sequences B_v over alphabet $[1..d]$, so that the symbol at $S_v[i]$ is stored at the child numbered $B_v[i]$ of node v.

In order to obtain time complexities $O(1 + \lg_d \sigma)$ for the operations, we need to handle `rank` and `select` on sequences over alphabet $[1..d]$, in constant time. Ferragina et al. [40] showed that this is indeed possible, while maintaining the overall space within $n \lg \sigma + o(n) \lg \sigma$, for $d = o(\lg n / \lg \lg n)$. Using, for example, $d = \lg^{1-\epsilon} n$ for any constant $0 < \epsilon < 1$, the overall space is $n \lg \sigma + O(n \lg \sigma / \lg^{\epsilon} n)$ bits. Golynski et al. [50] reduced the space to $n \lg \sigma + o(n)$ bits.

To support symbol `rank` and `select` on a sequence $R[1, n]$ over alphabet $[1..d]$, we assume we have d bitmaps $B_c[1, n]$, for $c \in [1..d]$, where $B_c[i] = 1$ iff $R[i] = c$. Then $\mathtt{rank}_c(R, i)$ and $\mathtt{select}_c(R, i)$ are reduced to $\mathtt{rank}_1(B_c, i)$ and $\mathtt{select}_1(B_c, i)$. We cannot afford to store those B_c, but we can store their extra $o(n)$ data for binary `rank` and `select`. Each time we need access to B_c, we access instead R and use a universal table to simulate the bitmap's content. Such table gives constant-time access to chunks of length $\lg_d(n)/2$ instead of $\lg(n)/2$, so the overall space using Golynski et al.'s bitmap index representation [48,49] is $O(dn \lg \lg n / \lg_d n)$, which added over the $\lg_d \sigma$ levels of the wavelet tree gives $O(n \lg \sigma \cdot d \lg d \lg \lg n / \lg n)$. This is $o(n \lg \sigma)$ for any $d = \lg^{1-\epsilon} n$. Further reducing the redundancy to $o(n)$ bits requires more sophisticated techniques [50].

Thus, the $O(\lg \sigma)$ upward/downward traversal times become $O(\lg \sigma / \lg \lg n)$ with multiary wavelet trees. Although theoretically attractive, it is not easy to translate their advantages to practice (see, e.g., a recent work studying interesting practical alternatives [17]). An exception, for a particular application, is described in the paragraph "Positional inverted indexes" of Section 5).

The upward traversal can be speeded up further, using techniques known in computational geometry [25]. Imagine we are at a leaf u representing a sequence $S[1, n_u]$ and want to directly track position i to an ancestor v at distance t, which represents sequence $S[1, n_v]$. We can store at the leaf u a bitmap $B_u[1, n_v]$, so that the n_u positions corresponding to leaf u are marked as 1s in B_u. This bitmap is sparse, so it is stored in compressed form as an "indexable dictionary" [93], which uses $n_u \lg(n_v/n_u) + o(n_u) + O(\lg \lg n_v)$ bits and can answer $\mathtt{select}_1(B_u, i)$ queries in $O(1)$ time. Thus we track position i upwards for t levels in $O(1)$ time.

The space required for all the bitmaps that point to node v is the sum, over at most 2^t leaves u, of those $n_u \lg(n_v/n_u) + o(n_u) + O(\lg \lg n_v)$ bits. This is maximized when $n_u = n_v/2^t$ for all those u, where the space becomes $t \cdot n_v + o(n_v) + O(2^t \lg \lg n_v)$. Added over all the wavelet tree nodes with height multiple of t, we get $n \lg \sigma + o(n \lg \sigma) + O(\sigma \lg \lg n) = n \lg \sigma + o(n \lg \sigma)$. This is in addition to those $n \lg \sigma + o(n)$ bits already used by the wavelet tree.

If we want to track only from the leaves to the root, we may just use $t = \lg \sigma$ and do the tracking in constant time. In many cases, however, one wishes to track from arbitrary to arbitrary nodes. In this case we can use $1/\epsilon$ values of $t = \lg^{i\epsilon} \sigma$, for $i \in [1..1/\epsilon - 1]$, so as to carry out $O(\lg^{\epsilon} \sigma)$ upward steps with one value of t before reaching the next one. This gives a total complexity for upward traversals of $O((1/\epsilon) \lg^{\epsilon} \sigma)$ using $O((1/\epsilon) n \lg \sigma)$ bits of space.

Construction. It is easy to build a wavelet tree in $O(n \lg \sigma)$ time, by a linear-time processing at each node. It is less obvious how to do it in little extra space, which may be important for succinct data structures. Two recent results [31,96] offer various relevant space-time tradeoffs, building the wavelet tree within the time given, or close, and asymptotically negligible extra space.

3 Compression

The wavelet tree adapts elegantly to the compressibility of the data in many ways. Two key techniques to achieve this are using specific encodings on bitmaps, and altering the tree shape. This whole section is technical, yet nonexpert readers may find inspiring the beginning of the paragraph "Entropy coding", and the paragraph "Changing shape".

Entropy Coding. Consider again Fig. 1. The fact that the `'a'` is much more frequent than the other symbols translates into unbalanced 0/1 frequencies in various bitmaps. Dissimilarities in symbol frequencies are an important source of compressibility. The amount of compression that can be reached is measured by the so-called *empirical zero-order entropy* of a sequence $S[1, n]$:

$$H_0(S) \quad = \quad \sum_{c \in \Sigma} (n_c/n) \lg(n/n_c) \quad \leq \quad \lg \sigma$$

where n_c is the number of occurrences of c in S and the sum considers only the symbols that do appear in S. Then $nH_0(S)$ is the least number of bits into which S can be compressed by always encoding the same symbol in the same way.[2]

Grossi et al. [54] already showed that, if the bitmaps of the wavelet tree are compressed to their zero-order entropy, then their overall space is $nH_0(S)$. Let $B_{v_{root}}$ contain n_0 0s and n_1 1s. Then zero-order compressing it yields space $n_0 \lg(n/n_0) + n_1 \lg(n/n_1)$. Now consider its left child v_l. Its bitmap, B_{v_l}, is of length n_0, and say it contains n_{00} 0s and n_{01} 1s. Similarly, the right child is of length n_1 and contains n_{10} 0s and n_{11} 1s. Adding up the zero-order compressed space of both children yields $n_{00} \lg(n_0/n_{00}) + n_{01} \lg(n_0/n_{01}) + n_{10} \lg(n_1/n_{10}) + n_{11} \lg(n_1/n_{11})$. Now adding the space of the root bitmap yields $n_{00} \lg(n/n_{00}) + n_{01} \lg(n/n_{01}) + n_{10} \lg(n/n_{10}) + n_{11} \lg(n/n_{11})$. This would already be $nH_0(S)$ if $\sigma = 4$. It is easy to see that, by splitting the spaces of the internal nodes until reaching the wavelet tree leaves, we arrive at $\sum_{c \in \Sigma} n_c \lg(n/n_c) = nH_0(S)$.

This enables using any zero-order entropy coding for the bitmaps that supports constant-time `rank` and `select`. One is the "fully-indexable dictionary" of Raman et al. [93], which for a bitmap $B[1, n]$ requires $nH_0(B) + O(n \lg \lg n / \lg n)$ bits. A theoretically better one is that of Golynski et al. [50], which we have already mentioned without yet telling that it actually compresses the bitmap, to $nH_0(B) + O(n \lg \lg n / \lg^2 n)$. Pătraşcu [91] showed this can be squeezed up to

[2] In classical information theory [32], H_0 is the least number of bits per symbol achievable by any compressor on an infinite source that emits symbols independently and randomly with probabilities n_c/n.

$nH_0(B)+O(n/\lg^c n)$, answering `rank` and `select` in time $O(c)$, for any constant c, and that this is essentially optimal [92].

Using the second or third encoding, the wavelet tree represents S within $nH_0(S) + o(n)$ bits, still supporting the traversals in time $O(\lg \sigma)$. Ferragina et al. [40] showed that the zero-order compression can be extended to multiary wavelet trees, reaching $nH_0(S) + o(n \lg \sigma)$ bits and time $O(1 + \lg \sigma / \lg \lg n)$ for the operations, and Golynski et al. [50] reduced the space to $nH_0(S) + o(n)$ bits. Recently, Belazzougui and Navarro [12] showed that the times can be reduced to $O(1 + \lg \sigma / \lg w)$, where $w = \Omega(\lg n)$ is the size of the machine word. Basically they replace the universal tables with bit-parallel operations. Their space grows to $nH_0(S) + o(n(H_0(S) + 1))$. (They also prove and match the lower bound time complexity $\Theta(1 + \lg(\lg \sigma / \lg w))$ using techniques that are beyond wavelet trees and this survey, but that do build on wavelet trees [7,4].)

It should not be hard to see at this point that the sums of $n_u \lg(n_v/n_u)$ spaces used for fast upward traversals in Section 2 also add up to $(1/\epsilon)nH_0(S)$.

Changing Shape. The algorithms for traversing the wavelet tree work independently of its balanced shape. Furthermore, our previous analysis of the entropy coding of the bitmap also shows that the resulting space, at least with respect to the entropy part, is independent of the shape of the tree. This was already noted by Grossi et al. [55], who proposed using the shape to optimize average query time: If we know the relative frequencies f_c with which each leaf c is sought, we can create a wavelet tree with the shape of the Huffman tree [62] of those frequencies, thus reducing the average access time to $\sum_{c \in \Sigma} f_c \lg(1/f_c) \le \lg \sigma$.

Mäkinen and Navarro [70, Sec. 3.3], instead, proposed giving the wavelet tree the Huffman shape of the frequencies with which the symbols appear in S. This has interesting consequences. First, it is easy to see that the total number of bits stored in the wavelet tree is exactly the number of bits output by a Huffman compressor that takes the symbol frequencies in S, which is upper bounded by $n(H_0(S) + 1)$. Therefore, even using plain bitmap representations taking $n + o(n)$ bits of space, the total space becomes at most $n(H_0(S) + 1) + o(n(H_0(S) + 1)) + O(\sigma \lg n)$, that is, we compress not only the data, but also the redundancy space. This may seem irrelevant compared to the $nH_0(S) + o(n)$ bits that can be obtained using Golynski et al. [50] over a balanced wavelet tree. However, it is unclear whether that approach is practical; only that of Raman et al. [93] has successful implementations [89,28,84], and this one leads to total space $nH_0(S) + o(n \lg \sigma)$. Furthermore, plain bitmap representations are significantly faster than compressed ones, and thus compressing the wavelet tree by giving it a Huffman shape leads to a much faster implementation in practice.

Another consequence of using Huffman shape, implied by Grossi et al. [55], is that if the accesses to the leaves are done with frequency proportional to their number of occurrences in S (which occurs, for example, if we access at random positions in S), then the average access time is $O(1 + H_0(S))$, better than the $O(\lg \sigma)$ of balanced wavelet trees. A problem is that the worst case could be as bad as $O(\lg n)$ if a very infrequent symbol is sought [70]. However, one can balance wavelet subtrees after some depth, so that the average depth is

$O(1 + H_0(S))$, the maximum depth is $O(\lg \sigma)$, and the total number of bits is at most $n(H_0(S) + 2)$ [70].

Recently, Barbay and Navarro [10] showed that Huffman shapes can be combined with multiary wavelet trees and entropy compression of the bitmaps, to achieve space $nH_0(S)+o(n)$ bits, worst-case time $O(1+\lg \sigma / \lg \lg n)$, and average case time $O(1 + H_0(S)/ \lg \lg n)$.

An interesting extension of Huffman shaped wavelet trees that has not been emphasized much is to use them a mechanism to give direct access on *any* variable-length prefix-free coding. Let $S = s_1, s_2, \ldots, s_n$ be a sequence of symbols, which are encoded in some way into a bit-stream $C = c(s_1)c(s_2)\ldots c(s_n)$. For example, S may be a numeric sequence and c can be a δ-code, to favor small numbers [13], or c can be a Huffman or another prefix-free encoding. Any prefix-free encoding ensures that we can retrieve S from C, but if we want to maintain the compressed form C and access arbitrary positions of S, we need tricks like sampling S at regular intervals and store pointers to C.

Instead, a wavelet tree representation of S, where for each s_i we rather encode $c(s_i)$, uses the same number of bits of C and gives direct access to any $S[i]$ in time $O(|c(s_i)|)$. More precisely, at the bitmap root position $B_{v_{root}}[i]$ we write a 0 if $c(s_i)$ starts with a 0, and 1 otherwise. In the first case we continue by the left child and in the second case we continue by the right child, from the second bit of $c(s_i)$, until the code is exhausted. Gagie et al. [43] combined this idea with multiary wavelet trees to obtain a faster decoding.

Very recently, Grossi and Ottaviano [56] also took advantage of specific shapes, to give the wavelet tree the form of a trie of a set of strings. The goal was to handle a sequence of strings and extend operations like access and `rank` to such strings. The idea extends a previous, more limited, approach [72,74].

High-Order Entropy Coding. High-order compression extends zero-order compression by encoding each symbol according to a context of length k that precedes or follows it. The *k-th order empirical entropy* of S [77] is defined as $H_k(S) = \sum_{A \in \Sigma^k} (|S_A|/n)\ H_0(S_A) \leq H_{k-1}(S)$, where S_A is the string of symbols preceding context A in S. Any statistical compressor assigning fixed codes that depend on a context of length k outputs at least $nH_k(S)$ bits to encode S.

The Burrows-Wheeler transform [22] is a useful tool to achieve high-order entropy. It is a reversible transformation that permutes the symbols of a string $S[1, n]$ as follows. First sort all the suffixes $S[i, n]$ lexicographically, and then list the symbols that precede each suffix (where $S[n]$ precedes $S[1, n]$). The result, $S^{bwt}[1, n]$, is the concatenation of the strings S_A for all the contexts A. By definition, if we compress each substring S_A of S^{bwt} to its zero-order entropy, the total space is the k-th order entropy of S, for $k = |A|$.

The first [54] and second [39] reported use of wavelet trees used a similar partitioning to represent each range of S^{bwt} with a zero-order compressed wavelet tree, so as to reach $nH_k(S) + o(n \lg \sigma)$ bits of space, for any $k \leq \alpha \lg_\sigma n$ and any constant $0 < \alpha < 1$. In the second case [39], the use of S^{bwt} was explicit. The partitioning was not with a fixed context length, but instead an optimal partitioning was used [36]. This way, they obtained the given space simultaneously for any k in the range. In the first case [54], they made no reference to the Burrows-Wheeler transform, but also compressed the sequences S_A of the k-th order entropy formula, for a fixed k. We give more details on the reasons behind the use of S^{bwt} in Section 5.

Already in 2004, Grossi et al. [55] realized that the careful partitioning into many small wavelet trees, one per context, was not really necessary to achieve k-th order compression. By using a proper encoding on its bitmaps, a wavelet tree on the whole S^{bwt} could reach k-th order entropy compression of a string S. They obtained $2nH_k(S)$ bits, plus redundancy, by using γ-codes [13] on the runs of 0s and 1s in the wavelet tree bitmaps. Mäkinen and Navarro [73] observed the same fact when encoding the bitmaps using Raman et al. [93] fully indexable dictionaries. They reached $nH_k(S) + o(n \lg \sigma)$ bits of space, simultaneously for any $k \leq \alpha \lg_\sigma n$ and any constant $0 < \alpha < 1$, using just one wavelet tree for the whole string. This yielded simpler and faster indexes in practice [28].

The key property is that some entropy-compression methods are local, that is, their space is the sum of the zero-order entropies of short substrings of S^{bwt}. This can be shown to be upper-bounded by the entropy of the whole string, but also by the sum of the entropies of the substrings S_A. Even more surprisingly, Kärkkäinen and Puglisi [67] recently showed that the k-th order entropy is still reached if one cuts S^{bwt} into equally-spaced regions of appropriate length, and thus simplified these indexes further by using the faster and more practical Huffman-shaped wavelet trees on each region.

There are also more recent and systematic studies [35,59] of the compressibility properties of wavelet trees, and how they relate to gap and run-length encodings of the bitmaps, as well to the balancing and the arity.

Exploiting Repetitions. Another relevant source of compressibility is repetitiveness, that is, that $S[1,n]$ can be decomposed into a few substrings that have appeared earlier in S, or alternatively, that there is a small context-free grammar that generates S. Many compressors build on these principles [13], but supporting wavelet tree functionality on such compressed representations is harder.

Mäkinen and Navarro [71] studied the effect of repetitions in the Burrows-Wheeler transform of S. They showed that S^{bwt} could be partitioned into at most $nH_k(S)+\sigma^k$ runs of equal letters in S^{bwt}, for any k. It is not hard to see that those runs are inherited by the wavelet tree bitmaps, where run-length compression would take proper advantage of them. Mäkinen and Navarro followed a different path: they built a wavelet tree on the run heads and used a couple of bitmaps to simulate the operations on the original strings. The compressibility of those two bitmaps has been further studied by Mäkinen et al. [95,75] in the context of highly repetitive sequence collections, and also by Simon Gog [47, Sec. 3.6.1].

In some cases, however, we need the wavelet tree of the very same string S that contains the repetition, not its Burrows-Wheeler transform. We describe such an application in the paragraph "Document retrieval indexes" of Section 6.

Recently, Navarro et al. [86] proposed a grammar-compressed wavelet tree for this problem. The key point is that repetitions in $S[1,n]$ induce repetitions in $B_{v_{root}}[1,n]$. They used Re-Pair [69], a grammar-based compressor, on the bitmaps, and enhanced a Re-Pair-based compressed sequence representation [53] to support binary `rank` (they only needed downward traversals). This time, the wavelet tree partitioning into left and right children cuts each repetition into two, so quickly after a few levels such regularities are destroyed and another type of bitmap compression (or none) is preferred. While the theoretical space analysis is too weak to be useful, the result is good in practice and leaves open the challenge of achieving stronger theoretical and practical results.

We will find even more specific wavelet tree compression problems later.

4 Sequences, Reorderings, or Point Grids?

Now that we have established the basic structure, operations, and encodings of wavelet trees, let us take a view with more perspective. Various applications we have mentioned display different ways to regard a wavelet tree representation.

As a Sequence of Values. This is the most basic one. The wavelet tree on a sequence $S = s_1, \ldots, s_n$ represents the values s_i. The most important operations that the wavelet tree must offer to support this view are, apart from accessing any $S[i]$ (that we already explained in Section 2), `rank` and `select` on S. For example, the second main usage of wavelet trees [39,40] used access and `rank` on the wavelet tree built on sequence S^{bwt} in order to support searches on S.

The process to support $\mathtt{rank}_c(S,i)$ is similar to that for access, with a subtle difference. We start at position i in $B_{v_{root}}$, and decide whether to go left or right depending on where is the leaf corresponding to c (and not depending on $B_{v_{root}}[i]$). If we go left, we rewrite $i \leftarrow \mathtt{rank}_0(B_{v_{root}}, i)$, else we rewrite $i \leftarrow \mathtt{rank}_1(B_{v_{root}}, i)$. When we arrive at the leaf c, the value of i is the final answer. The time complexity for this operation is that of a downward traversal towards the leaf labeled c. To support $\mathtt{select}_c(S,i)$ we just apply the upward tracking, as described in Section 2, starting at the i-th position of the leaf labeled c.

As a Reordering. Less obviously, the wavelet tree structure describes a stable ordering of the symbols in S, so that if one traverses the leaves one finds first all the occurrences of the smaller symbols, and within the same symbol (i.e., the same leaf), they are ordered by original position. As it will be clear in Section 5, one can argue that this is the usage of wavelet trees made by their creators [54].

In this case, tracking a position downwards in the wavelet tree tells where it goes after sorting, and tracking a position upwards tells where each symbol is placed in the sequence. An obvious application is to encode a permutation π over $[1..n]$. Our best wavelet tree takes $n \lg n + o(n)$ bits and can compute any $\pi(i)$ and $\pi^{-1}(i)$ in time $O(\lg n / \lg\lg n)$ by carrying out, respectively, downward and upward tracking of position i. We will see improvements on this idea later.

As a Grid of Points. The slightly less general structure of Chazelle [25] can be taken as the representation of a set of points supported by wavelet trees. It is generally assumed that we have an $n \times n$ grid with n points so that no two points share the same row or column (i.e., a permutation). A general set of n points is mapped to such a discrete grid by storing the real coordinates somewhere else and breaking ties somehow (arbitrarily is fine in most cases).

Take the set of points (x_i, y_i), in x-coordinate order (i.e., $x_i < x_{i+1}$). Now define string $S[1, n] = y_1, y_2, \ldots, y_n$. Then we can find the i-th point in x-coordinate order by accessing $S[i]$. Moreover, since the wavelet tree is representing the reordering of the points according to y-coordinate, one can find the i-th point in y-coordinate order by tracking upwards the i-th point in the leaves.

Unlike permutations, here the emphasis is in counting and reporting the points that lie within a rectangle $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. This is solved through a more complicated tracking mechanism, well-known in computational geometry and also described explicitly on wavelet trees [72]. We start at the root bitmap range $B_{v_{root}}[x_l, x_r]$, where $x_l = x_{min}$ and $x_r = x_{max}$. Now we map the interval to the left *and* to the right, using $x_l \leftarrow \texttt{rank}_{0/1}(B_{v_{root}}, x_l - 1) + 1$ and $x_r \leftarrow \texttt{rank}_{0/1}(B_{v_{root}}, x_r)$, and continue recursively. At any node along the recursion, we may stop if (*i*) the interval $[x_l, x_r]$ becomes empty (thus there are no points to report); (*ii*) the interval of leaves (i.e., y-coordinate values) represented by the node has no intersection with $[y_{min}, y_{max}]$; (*iii*) the interval of leaves is contained in $[y_{min}, y_{max}]$. In case (*iii*) we can count the number of points falling in this sub-rectangle as $x_r - x_l + 1$. As it is well known that we visit only $O(\lg n)$ wavelet tree nodes before stopping all the recursive calls (see, e.g., a recent detailed proof, among other more sophisticated wavelet tree properties [45]), the counting time is $O(\lg n)$. Each of the $x_r - x_l + 1$ points found in each node can be tracked up and down to find their x- and y-coordinates, in $O(\lg n)$ time per reported occurrence. There are more efficient variants of this technique that we will cover in Section 7, but they build on this basic idea.

5 Applications as Sequences

Full-Text Indexes. A full-text index built a string $S[1, n]$ is able to count and locate the occurrences of arbitrary patterns $P[1, m]$ in S. A classical index is the *suffix array* [52,76], $A[1, n]$, which lists the starting positions of all the suffixes of S, $S[A[i], n]$, in lexicographic order, using $n\lceil \lg n \rceil$ bits. The starting positions of the occurrences of P in S appear in a contiguous range in A, which can be binary searched in time $O(m \lg n)$, or $O(m + \lg n)$ by doubling the space. A *suffix tree* [98,78,1] is a more space-consuming structure (yet still $O(n \lg n)$ bits) that can find the range in time $O(m)$. After finding the range, each occurrence is reported in constant time, both in suffix trees and arrays.

The suffix array of S is closely related to its Burrows-Wheeler transform: $S^{bwt}[i] = S[A[i] - 1]$ (taking $S[0] = S[n]$). Ferragina and Manzini [37,38] showed how, using at most $2m$ access and $\texttt{rank}$ operations on S^{bwt}, one could count the number of occurrences in S of a pattern $P[1, m]$. Using multiary wavelet trees [40,50] this gives a counting time of $O(m)$ on polylog-sized alphabets, and $O(m \lg \sigma / \lg \lg n)$ in general. Each such occurrence can then be located in time $O(\lg^{1+\epsilon} n \lg \sigma / \lg \lg n)$ for any $\epsilon > 0$, at the price of $O(n / \lg^{\epsilon} n) = o(n)$ further bits of space. This result has been superseded very recently [7,12,11,4], in some cases using wavelet trees as a part of the solution, and in all cases with some extra price in terms of redundancy, such as $o(nH_k(S))$ and $O(n)$ further bits.

Grossi et al. [57,58,54] used wavelet trees to obtain a similar result via a quite different strategy. They represented A by means of a permutation $\Psi(i) = A^{-1}[A[i]+1]$, that is, the cell in A pointing to $A[i]+1$. Ψ turns out to be formed by σ contiguous ascending runs. The suffix array search can be simulated in $O(m \lg n)$ accesses to Ψ. They encode Ψ separately for the range of each context S_A (recall paragraph "High-order entropy coding" in Section 3). As all the Ψ pointers coming from each run are increasing, a wavelet tree is used to describe how the σ ascending sequences of pointers coming from each run are intermingled in the range of S_A. This turns out to be, precisely, the wavelet tree of S_A. This is why both Ferragina et al. and Grossi et al. obtain basically the same space, $nH_k(S) + o(n \lg \sigma)$ bits. Due to the different search strategy, the counting time of Grossi et al. is higher. On the other hand, the representation of Ψ allows them to locate patterns in sublogarithmic time, still using $O(nH_k(S)) + o(n \lg \sigma)$ bits.

This is the best known usage of wavelet trees as sequences, and it is well covered in earlier surveys [82]. New extensions of these basic concepts, supporting more sophisticated search problems, appear every year (e.g., [94,14]). We cover next other completely different applications.

Positional Inverted Indexes. Consider a natural language text collection. A positional inverted index is a data structure that stores, for each word, the list of the positions where it appears in the collection [3]. In compressed form [99] it takes space close to the zero-order entropy of the text *seen as a sequence of words* [82]. This entropy yields very competitive compression in natural language texts. Yet, we need to store both the text (usually zero-order compressed, so that direct access is possible) and the inverted index, adding up to at least $2nH_0(S)$, where S is the text regarded as a sequence of word identifiers. Inverted indexes are by far the most popular data structures to index natural language text collections, so reducing their space requirements is of high relevance.

By representing the sequence of word identifiers using a wavelet tree, we obtain a single representation for both the text and the inverted index, all within $nH_0(S) + o(n)$ bits [28]. In order to access any text word, we just compute $S[i]$. In order to access the i-th element of the inverted list of any word c, we compute $\texttt{select}_c(S, i)$. Furthermore, operation $\texttt{rank}_c(S, i)$ is useful to implement some list intersection algorithms [8], as it finds the position i in the inverted list of word c more efficiently than with a binary or exponential search.

Arroyuelo et al. [2] extended this functionality to *document retrieval*: retrieve the distinct documents where a word appears. They use a special symbol "\$" to mark document boundaries. Then, given the first occurrence of a word c, $p = \texttt{select}_c(S, 1)$, the document where this occurrence lies is $j = \texttt{rank}_\$(S, p)+1$, document j ends at position $p' = \texttt{select}_\$(S, j)$, it contains $o = \texttt{rank}_c(S, p, p')$ occurrences of the word c, and the search for further relevant documents can continue from query $\texttt{select}_c(S, o + 1)$.

An improvement over the basic idea is to use multiary wavelet trees, more precisely of arity up to 256, and using the property that wavelet trees give direct access to any variable-length code. Brisaboa et al. [19] started with a byte-oriented encoding of the text words (using either Huffman with 256 target symbols, or other practical encoding methods [20]) and then organized the sequence of codes into a wavelet tree, as described in the paragraph "Changing shape" of Section 3. A naive byte-based `rank` and `select` implementation on the wavelet tree levels gives good results in this application, with the bytes represented in plain form. The resulting structure is indeed competitive with positional inverted indexes in many cases. A variant specialized on XML text collections, where the codes are also used to distinguish structural elements (tags, content, attributes, etc.) in order to support some XPath queries, is also being developed [18].

Graphs. Another simple application of this idea is the representation of directed graphs [28]. Let G be a graph with n nodes and e edges. An adjacency list, using $n \lg e + e \lg n$ bits (the n pointers to the lists plus the e target nodes) gives direct access to the neighbors of any node v. If we want also to perform reverse nagivation, that is, to know which nodes point to v, we must spend other $n \lg e + e \lg n$ bits to represent the transposed graph.

Once again, representing with a wavelet tree the sequence $S[1, e]$ concatenating all the adjacency lists, plus a compressed bitmap $B[1, e]$ marking the beginnings of the lists, gives access to both types of neighbors within space $n \lg(e/n) + e \lg n + O(n) + o(e)$, which is close to the space of the plain representation (actually, possibly less). To retrieve the i-th neighbor of a node v, we compute the starting point of the list of v, $l \leftarrow \texttt{select}_1(B, v)$, and then access $S[l + i - 1]$. To retrieve the i-th reverse neighbor of a node v, we compute $p \leftarrow \texttt{select}_v(S, i)$ to find the i-th time that v is mentioned in an adjacency list, and then compute with $\texttt{rank}_1(B, p)$ the owner of the list where v is mentioned. Both operations take time $O(\lg n/ \lg \lg n)$. This is also useful to represent undirected graphs, where adjacency lists must usually represent each edge twice. With a wavelet tree we can choose any direction for an edge, and at query time we join direct and reverse neighbors of nodes to build their list.

Note, finally, that the wavelet tree can compress S to its zero-order entropy, which corresponds to the distribution of in-degrees of the nodes. A more sophisticated variant of this idea, combined with Re-Pair compression [69], was shown to be competitive with current Web graph compression methods [29].

6 Applications as Reorderings

Apart from its first usage [54], that can be regarded as encoding a reordering, wavelet trees offer various interesting applications when seen in this way.

Permutations. As explained in Section 4, one can easily encode a permutation with a wavelet tree. It is more interesting that the encoding can take less space when the permutation is, in a sense, compressible. Barbay and Navarro [9,10] considered permutations π of $[1..n]$ that can be decomposed into ρ contiguous ascending runs, of lengths $r_1, r_2, \ldots, r_\rho$. They define the entropy of such a permutation as $H(\pi) = \sum_{i=1}^{\rho} (r_i/n) \lg(n/r_i)$, and show that it is possible to sort an array with such ascending runs in time $O(n(H(\pi) + 1))$. This is obtained by building a Huffman tree on the run lengths (seen as frequencies) and running a mergesort-like algorithm that follows the Huffman tree shape.

They note that, if we encode with 0 or 1 the results of the comparisons of the mergesort algorithm at each node of the merging tree, the resulting structure contains at most $n(H(\pi)+1)$ bits, and it represents the permutation. Starting at position i in the top bitmap $B_{v_{root}}$ one can track down the position exactly as done with wavelet trees, so as to arrive at position j of the t-th leaf (i.e., run). By storing, in $O(\rho \lg n)$ bits, the starting position of each run in π, we can convert the leaf position into a position in π. Therefore the downward traversal solves operation $\pi^{-1}(i)$, because it starts from value i (i.e., position i after sorting π), and gives the position in π from where it started before the merging took place. The corresponding upward traversal, consequently, solves $\pi(i)$. Other types of runs, more and less general, are also studied [9,10].

Some thought reveals that this structure is indeed the wavelet tree of a sequence formed by replacing, in π^{-1}, each symbol belonging to the i-th run, by the run identifier i. Then the fact that a downward traversal yields $\pi^{-1}(i)$ and that the upward traversal yields $\pi(i)$ are natural consequences. This relation is made more explicit in a later article [7,4].

Generic Numeric Sequences. There are several basic problems on sequences of numbers that can be solved in nontrivial ways using wavelet trees. We mention a few that have received attention in the literature.

One such problem is the *range quantile query*: Preprocess a sequence of numbers $S[1,n]$ on the domain $[1..\sigma]$ so that later, given a range $[l,r]$ and a value i, we can compute the i-th smallest element in $S[l,r]$.

Classical solutions to this problem have used nearly quadratic space and constant time. Only a very recent solution [65] reaches $O(n \lg n)$ bits of space (apart from storing S) and $O(\lg n / \lg\lg n)$ time. We show that, by representing S with a wavelet tree, we can solve the problem in $O(\lg \sigma)$ time and just $o(n)$ extra bits [46,45]. This is close to $O(\lg n / \lg\lg n)$ (in this problem, we can always make $\sigma \le n$ hold), and it can be even better if σ is small compared to n.

Starting from the range $S[l,r]$, we compute $\mathtt{rank}_0(B_{v_{root}}, l, r)$. If this is i or more, then the i-th value in this range is stored in the left subtree, so we go to the left child and remap the interval $[l,r]$ as done for counting points in a range (see Section 4). Otherwise we go right, subtracting $\mathtt{rank}_0(B_{v_{root}}, l, r)$ from i and remapping $[l,r]$ in the same way. When we arrive at a leaf, its label is the i-th smallest element in $S[l,r]$.

Another fundamental problem is called *range next value*: Preprocess a sequence of numbers $S[1,n]$ on the domain $[1..\sigma]$ so that later, given a range $[l,r]$ and a value x, we return the smallest value in $S[l,r]$ that is larger than x.

The state of the art also includes superlinear-space and constant-time solutions, as well as one using $O(n \lg n)$ bits of space and $O(\lg n / \lg\lg n)$ time [100]. Once again, we achieve $o(n)$ extra bits and $O(\lg \sigma)$ time using wavelet trees [45] (we improve this time in the paragraph "Binary relations" of Section 7).

Starting at the root from the range $S[l,r]$, we see if value x labels a leaf descending from the left or from the right child. If x descends from the right child, then no value on the left child can be useful, so we recursively descend to the right child and remap the interval $[l, r]$ as done for counting points in a range. Else, there may be values $> x$ on both children, but we prefer those on the left, if any. So we first descend to the left child looking for an answer (there may be no answer if, at some node, the interval $[l, r]$ becomes empty). If the left child returns an answer, this is what we seek and we return it. If, however, there is no value $> x$ on the left child, we seek the smallest value on the right child. We then enter into another mode where we see if there is any 0-bit in $B_v[l, r]$. If there is one, we go to the left child, else we go to the right child. It can be shown that the overall process takes $O(\lg \sigma)$ time.

A variant of the range next value problem is called *prevLess* [68]: return the rightmost value in $S[1, r]$ that is smaller than x. Here we start with $S[1, r]$. If value x labels a leaf descending from the left, we map the interval to the left child and continue recursively from there. If, instead, x descends from the right child, then the answer may be on the left or the right child, and we prefer the rightmost in $[1, r]$. Any 0-bit in $B_v[1, r]$ is a value smaller than x and thus a valid answer. We use `rank` and `select` to find the rightmost 0 in $B_v[1, r]$. We also continue recursively by the right child, and if it returns an answer, we map it to the bitmap $B_v[1, r]$. Then we choose the rightmost between the answer from the right child and the rightmost zero. The overall time is $O(\lg \sigma)$.

Non-positional Inverted Indexes. These indexes store only the list of distinct documents where each word appears, and come in two flavors [99,3]. In the first, the documents for each word are sorted by increasing identifier. This is useful to implement list unions and intersections for boolean, phrase and proximity queries. In the second, a "weight" (measuring importance somehow) is assigned to each document where a word appears. The lists of each word store those weights and are sorted by decreasing weight. This is useful to implement ranked bag-of-word queries, which give the documents with highest weights added over all the query words. It would seem that, unless one stores two inverted indexes, one must choose one order in detriment of the queries of the other type.

By representing a reordering, wavelet trees can store both orderings simultaneously [85,45]. Let us represent the documents where each word appears in decreasing weight order, and concatenate all the lists into a sequence $S[1, n]$. A bitmap $B[1, n]$ marks the starting positions of the lists, and the weights are stored separately. Then, a wavelet tree representation of S simulates, within the space of just one list, both orderings. By accessing $S[l+i-1]$, where $l = \mathtt{select}_1(B, c)$, we obtain the i-th element of the inverted list of word c, in decreasing weight order. To access the i-th element of the inverted list of a word in increasing document order, we also compute the end of its list, $r = \mathtt{select}_1(B, c+1) - 1$, and then run a range quantile query for the i-th smallest value in the range $[l, r]$. Many other operations of interest in information retrieval can be carried out with this representation and little auxiliary data [85,45].

Document Retrieval Indexes. An interesting extension to full-text retrieval is document retrieval, where a collection $S[1, n]$ of general strings (so inverted indexes cannot be used) is to be indexed to answer different document retrieval queries. The most basic one, document listing, is to output the distinct documents where a pattern $P[1, m]$ appears. Muthukrishnan [80] defined a so-called *document array* $D[1, n]$, where $D[i]$ gives the document to which the i-th lexicographically smallest suffix of S belongs (i.e., where the suffix $S[A[i], n]$ belongs, where A is the suffix array of S). He also defined an array $C[1, n]$, where $C[i]$ points to the previous occurrence of $D[i]$ in D. A suffix tree was used to identify the range $A[l, r]$ of the pattern occurrences, so that we seek to report the distinct elements in $D[l, r]$. With further structures to find minima in ranges of C [15], Muthukrishnan gave an $O(m + occ)$ algorithm to find the occ distinct documents where P appears. This is time-optimal, yet the space is impractical.

This is another case where wavelet trees proved extremely useful. Mäkinen and Välimäki [97] showed that, if one implemented D as a wavelet tree, then array C was not necessary, since $C[i] = \mathtt{select}_{D[i]}(D, \mathtt{rank}_{D[i]}(D, i-1))$. They also used a compressed full-text index [39] to identify the range $D[l, r]$, so the total time turned out to be $O(m \lg \sigma + occ \lg d)$, where d is the number of documents in S. Moreover, for each document c output, $\mathtt{rank}_c(D, l, r)$ gave the number of times P appeared in c, which is important for ranked document retrieval.

Gagie et al. [46,45] showed that an application of range quantile queries enabled the wavelet tree to solve this problem elegantly and without any range minima structure: The first distinct document is the smallest value in $D[l, r]$. If it occurs f_1 times, then the second distinct document is the $(1 + f_1)$-th smallest value in $D[l, r]$, and so on. They retained the complexities of Mäkinen and Välimäki, but the solution used less space and time in practice. Later [45] they replaced the range quantile queries by a depth-first traversal of the wavelet tree that reduced the time complexity, after the suffix array search, to $O(occ \lg(d/occ))$. The technique is similar to the two-dimensional range searches: recursively enter into every wavelet tree branch where the mapped interval $[l, r]$ is not empty, and report the leaves found, with frequency $r - l + 1$.

This depth-first search method can easily be extended to support more complex queries, for example t-thresholded ones: given s patterns, we want the documents where at least t of the terms appear. We can first identify the s ranges in D and then traverse the wavelet tree while maintaining the s ranges, stopping when less than t intervals are nonempty, or when we arrive at leaves (where we report the document). Other sophisticated traversals have been proposed for retrieving the documents ranked by number of occurrences of the patterns [33].

An interesting problem is how to compress the wavelet tree of D effectively. The zero-order entropy of D has to do with document lengths, which is generally uninteresting, and unrelated to the compressiblity of S. It has been shown [44,86] that the compressibility of S shows up as repetitions in D, which has stimulated the development of wavelet tree compression methods that take advantage of the repetitiveness of D, as described at the end of Section 3.

7 Applications as Grids

Discrete Grids. Much work has been done in Computational Geometry over structures very similar to wavelet trees. We only highlight some results of interest, generally focusing on structures that use linear space. We assume here that we have an $n \times n$ grid with n points not sharing rows nor columns. Interestingly, these grids with range counting and reporting operations have been intensively used in compressed text indexing data structures [66,81,38,72,26,16,30,68]

Range counting can be done in time $O(\lg n / \lg \lg n)$ and $O(n \lg n)$ bits [64]. This time cannot be improved within space $O(n \lg^{O(1)} n)$ [90], but it can be matched with a multiary wavelet-tree like structure using just $n \lg n + o(n \lg n)$ bits [16]. Reaching this time, instead of the easy $O(\lg n)$ we have explained in Section 4, requires a sophisticated solution to the problem of doing the range counting among several consecutive children of a node, that are completely contained in the x-range of the query. They [16] also obtain a range reporting time (for the occ points in the range) of $O((1+occ) \lg n / \lg \lg n)$. This is not surprising once counting has been solved: it is a matter of upward or downward tracking on a multiary wavelet tree. The technique for faster upward tracking we described in the paragraph "Speeding up traversals" of Section 2 can be used to improve the reporting time to $O((1 + occ) \lg^{\epsilon} n)$, using $O((1/\epsilon) n \lg n)$ bits of space [24].

Wavelet trees offer relevant solutions to other geometric problems, such as finding the dominant points in a grid, or solving visiblity queries. Those problems can be recast as a sequence of queries of the form "find the smallest element larger than x in a range", described in the paragraph "Generic numeric sequences" of Section 6, and therefore solved in time $O(\lg n)$ per point retrieved [83]. That paper [83,87] also studies extensions of geometric queries where the points have weights and statistical queries on them are posed, such as finding range sums, averages, minima, quantiles, majorities, and so on. The way those queries are solved open interesting new avenues in the use of wavelet trees.

Some queries, such as finding the minimum value of a two-dimensional range, are solved by enriching wavelet trees with extra information aligned to the bitmaps. Recall that each wavelet tree node v handles a subsequence S_v of the sequence of points $S[1, n]$. To each node v with bitmap $B_v[1, n_v]$ we associate a data structure using $2n_v + o(n_v)$ bits that answers one-dimensional range minimum queries [41] on $S_v[1, n_v]$. Once built, this structure does not need to access S_v, yet it gives the position of the minimum in constant time. Since, as explained, a two-dimensional range is covered by $O(\lg n)$ wavelet tree nodes, only those $O(\lg n)$ minima must be tracked upwards, where the actual weights are stored, to obtain the final result. Thus the query requires $O(\lg^{1+\epsilon} n)$ time and $O((1/\epsilon) n \lg n)$ bits of space by using the fast upward tracking mechanism.

Other queries, such as finding the i-th smallest value of a two-dimensional range, are handled with a wavelet tree *on the weight values*. Each wavelet tree node stores a grid with the points whose weights are in the range handled by that node. Then, by doing range counting queries on those grids, one can descend left or right, looking for the rightmost leaf (i.e., value) such that the counts of the children to the left of the path followed add up to less than i. The total time is $O(\lg^2 n / \lg \lg n)$, however the space becomes superlinear, $O(n \lg^2 n)$ bits.

Finally, an interesting extension to the typical point grids are grids of rectangles, which are used in geographic information systems as minimum bounding rectangles of complex objects. Then one wishes to find the set of rectangles that intersect a query rectangle. This is well solved with an R-tree data structure [60], but a wavelet tree may offer interesting space reductions. Brisaboa et al. [21] describe a technique to store n rectangles where one does not contain another in the x-coordinate range (so the set is first separated into maximal "x-independent" subsets and each subset is queried separately). Two arrays with the ascending lower and upper x-coordinates of the rectangles are stored (as the sets are x-independent, the same position in both arrays corresponds to the same rectangle). A wavelet tree on those x-coordinate-sorted rectangles is set up, so that each node handles a range of y-coordinate values. This wavelet tree stores two bitmaps per node v: one tells whether the rectangle $S_v[i]$ extends to the y-range of the left child, and the other whether it extends to the right child. Both bitmaps can store a 1 at a position i, and thus the rectangle is stored in both subtrees. To avoid representing a large rectangle more than $O(\lg n)$ times, both bits are set to 0 (which is otherwise impossible) when the rectangle completely contains the y-range of the current node. The total space is $O(n \lg n)$ bits.

Given a query $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$, we look for x_{min} in the array of *upper* x-coordinates, to find position x_l, and look for x_{max} in the array of *lower* x-coordinates, to find position x_r. This is because a query intersects a rectangle on the x-axis if the query does not start after the rectangle ends and the query does not end before the rectangle starts. Now the range $[x_l, x_r]$ is used to traverse the wavelet tree almost like on a typical range search, except that we map to the left child using $\mathtt{rank}_1$ on one bitmap, and to the right child using $\mathtt{rank}_1$ on the other bitmap. Furthermore, we report all the rectangles where both bitmaps contain a 0-bit, and we remove duplicates by merging results at each node, as the same rectangle can be encountered several times. The overall time to report the *occ* rectangles is still $O((1 + occ) \lg n)$.

Binary Relations. A binary relation R between two sets A and B can be thought of as a grid of size $|A| \times |B|$, containing $|R|$ points. Apart from strings, permutations and our grids, that are particular cases, binary relations are good abstractions for a large number of more applied structures. For example, a non-positional inverted index is a binary relation between a set of words and a set of documents, so that a word is related to the documents where it appears. As another example, a graph is a binary relation between the set of nodes and itself.

The most typical operations on binary relations are determining the elements $b \in B$ that are related to some $a \in A$ and vice versa, and determining whether a pair $(a, b) \in A \times B$ is related in R. However, more complex queries are also of interest. For example, counting or retrieving the documents related to any term in a range enables on-the-fly stemming and query expansion. Retrieving the terms associated to a document permits vocabulary analyses. Accessing the documents in a range related to a term enables searches local to subcollections. Range counting and reporting allows regarding graphs at a larger granularity (e.g., a Web graph can be regarded as a graph of hosts, or of pages, on the fly).

Barbay et al. [5,6] studied a large number of complex queries for binary relations, including accessing the points in a range in various orders, as well as reporting rows or columns containing points in a range. They proposed two wavelet-tree-like data structures for handling the operations. One is basically a wavelet tree of the set of points (plus a bitmap that indicates when we move from one column to the next). It turns out that almost all the solutions described so far on wavelet trees find application to solve some of the operations.

In the extended version [6] they use multiary wavelet trees to reduce the times of most of the operations. Several nontrivial structures and algorithms are designed in order to divide the times of various operations by $\lg \lg n$ (the only precedent we know of is that of counting the number of points in a range [16]). For example, it is shown how to solve the *range next value* problem (recall paragraph "Generic numeric sequences" of Section 6) in time $O(\lg n / \lg \lg n)$. Others, like the *range quantile query*, stay no better than $O(\lg n)$.

Barbay et al. also propose a second data structure that is analogous to the one described for rectangles in the paragraph "Discrete grids". Two bitmaps are stored per node, indicating whether a given column has points in the first and in the second range of rows. This extension of a wavelet tree is less powerful than the previous structure, but it is shown that its space is close to the *entropy* of the binary relation: $(1+\sqrt{2})H+O(|A|+|B|+|R|)$ bits, where $H = \lg \binom{|A|\cdot|B|}{|R|}$. This is not achieved with the classical wavelet tree. A separate work [34] builds on this to obtain a fully-compressed grid representation, within $H + o(H)$ bits.

Colored Range Queries. A problem of interest in areas like query log and web mining is to count the different *colors* in a sequence $S[1,n]$ over a universe of σ colors. Inspired in the idea of Muthukrishnan [80] for document retrieval (recall paragraph "Document retrieval indexes" in Section 6), Gagie et al. [44] showed that this is a matter of counting how many values smaller than l are there in $C[l,r]$, where $C[i] = \max\{j<i, S[j]=S[i]\}$. This is a range counting query for $[l,r] \times [1,l-1]$ on C seen as a grid, that can be solved in time $O(\lg n)$ using the wavelet tree of C. Note that this wavelet tree, unlike that of S, uses $n \lg n + o(n)$ bits. Gagie et al. compressed it to $n \lg \sigma + O(n \lg \lg n)$ bits, by taking advantage of the particular structure of C, which shows up in the bit-vectors. Gagie and Kärkkäinen [42] then reduced the space to $nH_0(S) + o(nH_0(S)) + O(n)$ with more advanced techniques, and also reduced the query time to $O(\lg(r-l+1))$.

8 Conclusions and Further Challenges

We have described the wavelet tree, a surprisingly versatile data structure that offers nontrivial solutions to a wide range of problems in areas like string processing, computational geometry, and many more. An important additional asset of the wavelet tree is its simplicity to understand, teach, and program. This makes it a good data structure to be introduced at an undegraduate level, at least in its more basic variants. In many cases, solutions with better time complexity than the ones offered by wavelet trees are not so practical nor easy to implement.

Wavelet trees seem to be unable to reach access and `rank`/`select` times of the form $O(\lg \lg \sigma)$, as other structures for representing sequences do [51], close to the lower bounds [12]. However, both have been combined to offer those time complexities and good zero-order compression of data and redundancy [7,4,12]. Yet, the lower bounds on some geometric problems [24], matched with current wavelet trees [16,6], suggest that this combination cannot be carried out much further than those three operations. Still, there are some complex operations where it is not clear that wavelet trees have matched lower bounds [45].

We have described the wavelet tree as a static data structure. However, if the bitmaps or sequences stored at the nodes support insertions and deletions in time $indel(n)$, then the wavelet tree easily supports insertions and deletions in the sequence $S[1, n]$ it represents, in time $O(h \cdot indel(n))$, where h is its height. This has been used to support indels in time $O((1 + \lg \sigma / \lg \lg n) \lg n / \lg \lg n)$ [61,88]. The alphabet, however, is still fixed in those solutions. While such a limitation may seem natural for sequences, it looks definitely artificial when representing grids: one can insert and delete new x-coordinates and points, but the y-coordinate universe cannot change. Creating or removing alphabet symbols requires changing the shape of the wavelet tree, and the bitmaps or sequences stored at the nodes undergo extensive modifications upon small tree shape changes (e.g., AVL rotations). Extending dynamism to support this type of updates, with good time complexities at least in the amortized sense, is an important challenge for this data structure. It is also unclear what is the dynamic lower bound on a general alphabet; on a constant-size alphabet it is $\Theta(\lg n / \lg \lg n)$ [23]. Very recently [56] a dynamic scheme for a particular case (sequences of strings) has been proposed.

A path that, in our opinion, has only started to be exploited, is to enhance the wavelet tree with "one-dimensional" data structures at its nodes v, so that, by efficiently solving some kind of query over the corresponding subsequences S_v, we solve a more complex query on the original sequence S. In most cases along this survey, these one-dimensional queries have been `rank` and `select` on the bitmaps, but we have already shown some examples involving more complicated queries [44,87,83]. This approach may prove to be very fruitful.

In terms of practice, although there are many successful and publicly available implementations of wavelet tree variants (see, e.g., `libcds.recoded.cl` and `http://www.uni-ulm.de/in/theo/research/sdsl.html`), there are some challenges ahead, such as carrying to practice the theoretical results that promise fast and small multiary wavelet trees [40,50,17] and lower redundancies [49,91,50].

Acknowledgements. Thanks to Jérémy Barbay, Ana Cerdeira, Travis Gagie, Juha Kärkkäinen, Susana Ladra, Simon Puglisi, and all those who took the time to read early versions of this manuscript.

References

1. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words. NATO ISI Series, pp. 85–96. Springer (1985)
2. Arroyuelo, D., González, S., Oyarzún, M.: Compressed Self-indices Supporting Conjunctive Queries on Document Collections. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 43–54. Springer, Heidelberg (2010)

3. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval, 2nd edn. Addison-Wesley (2011)
4. Barbay, J., Claude, F., Gagie, T., Navarro, G., Nekrich, Y.: Efficient fully-compressed sequence representations. CoRR, abs/0911.4981v4 (2012)
5. Barbay, J., Claude, F., Navarro, G.: Compact Rich-Functional Binary Relation Representations. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 170–183. Springer, Heidelberg (2010)
6. Barbay, J., Claude, F., Navarro, G.: Compact binary relation representations with rich functionality. CoRR, abs/1201.3602 (2012)
7. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet Partitioning for Compressed Rank/Select and Applications. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 315–326. Springer, Heidelberg (2010)
8. Barbay, J., López-Ortiz, A., Lu, T., Salinger, A.: An experimental investigation of set intersection algorithms for text searching. ACM J. Exp. Alg. 14 (2009)
9. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: Proc. 26th STACS, pp. 111–122 (2009)
10. Barbay, J., Navarro, G.: On compressing permutations and adaptive sorting. CoRR, abs/1108.4408 (2011)
11. Belazzougui, D., Navarro, G.: Alphabet-Independent Compressed Text Indexing. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 748–759. Springer, Heidelberg (2011)
12. Belazzougui, D., Navarro, G.: New lower and upper bounds for representing sequences. CoRR, abs/1111.2621 (2011)
13. Bell, T., Cleary, J., Witten, I.: Text Compression. Prentice Hall (1990)
14. Beller, T., Gog, S., Ohlebusch, E., Schnattinger, T.: Computing the Longest Common Prefix Array Based on the Burrows-Wheeler Transform. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 197–208. Springer, Heidelberg (2011)
15. Bender, M., Farach-Colton, M.: The LCA Problem Revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
16. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct Orthogonal Range Search Structures on a Grid with Applications to Text Indexing. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 98–109. Springer, Heidelberg (2009)
17. Bowe, A.: Multiary Wavelet Trees in Practice. Honours thesis, RMIT Univ., Australia (2010)
18. Brisaboa, N.R., Cerdeira-Pena, A., Navarro, G.: A Compressed Self-indexed Representation of XML Documents. In: Agosti, M., Borbinha, J., Kapidakis, S., Papatheodorou, C., Tsakonas, G. (eds.) ECDL 2009. LNCS, vol. 5714, pp. 273–284. Springer, Heidelberg (2009)
19. Brisaboa, N., Fariña, A., Ladra, S., Navarro, G.: Reorganizing compressed text. In: Proc. 31st SIGIR, pp. 139–146 (2008)
20. Brisaboa, N., Fariña, A., Navarro, G., Paramá, J.: Lightweight natural language text compression. Inf. Retr. 10, 1–33 (2007)
21. Brisaboa, N.R., Luaces, M.R., Navarro, G., Seco, D.: A Fun Application of Compact Data Structures to Indexing Geographic Data. In: Boldi, P. (ed.) FUN 2010. LNCS, vol. 6099, pp. 77–88. Springer, Heidelberg (2010)
22. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)

23. Chan, H.-L., Hon, W.-K., Lam, T.-W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Trans. Alg. 3(2), article 21 (2007)
24. Chan, T., Larsen, K.G., Pătraşcu, M.: Orthogonal range searching on the RAM, revisited. In: Proc. 27th SoCG, pp. 1–10 (2011)
25. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM J. Comp. 17(3), 427–462 (1988)
26. Chien, Y.-F., Hon, W.-K., Shah, R., Vitter, J.: Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In: Proc. 18th DCC, pp. 252–261 (2008)
27. Clark, D.: Compact Pat Trees. PhD thesis, Univ. of Waterloo, Canada (1996)
28. Claude, F., Navarro, G.: Practical Rank/Select Queries over Arbitrary Sequences. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 176–187. Springer, Heidelberg (2008)
29. Claude, F., Navarro, G.: Extended Compact Web Graph Representations. In: Elomaa, T., Mannila, H., Orponen, P. (eds.) Ukkonen Festschrift 2010. LNCS, vol. 6060, pp. 77–91. Springer, Heidelberg (2010)
30. Claude, F., Navarro, G.: Self-indexed grammar-based compression. Fund. Inf. 111(3), 313–337 (2010)
31. Claude, F., Nicholson, P.K., Seco, D.: Space Efficient Wavelet Tree Construction. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 185–196. Springer, Heidelberg (2011)
32. Cover, T., Thomas, J.: Elements of Information Theory. Wiley (1991)
33. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k Ranked Document Search in General Text Databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
34. Farzan, A., Gagie, T., Navarro, G.: Entropy-Bounded Representation of Point Grids. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 327–338. Springer, Heidelberg (2010)
35. Ferragina, P., Giancarlo, R., Manzini, G.: The myriad virtues of wavelet trees. Inf. Comp. 207(8), 849–866 (2009)
36. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. J. ACM 52(4), 688–713 (2005)
37. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. 41st FOCS, pp. 390–398 (2000)
38. Ferragina, P., Manzini, G.: Indexing compressed texts. J. ACM 52(4), 552–581 (2005)
39. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An Alphabet-Friendly FM-Index. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 150–160. Springer, Heidelberg (2004)
40. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. 3(2), article 20 (2007)
41. Fischer, J.: Optimal Succinctness for Range Minimum Queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010)
42. Gagie, T., Kärkkäinen, J.: Counting Colours in Compressed Strings. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 197–207. Springer, Heidelberg (2011)
43. Gagie, T., Navarro, G., Nekrich, Y.: Fast and Compact Prefix Codes. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., Rumpe, B. (eds.) SOFSEM 2010. LNCS, vol. 5901, pp. 419–427. Springer, Heidelberg (2010)

44. Gagie, T., Navarro, G., Puglisi, S.J.: Colored Range Queries and Document Retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 67–81. Springer, Heidelberg (2010)
45. Gagie, T., Navarro, G., Puglisi, S.J.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comp. Sci. 426-427, 25–41 (2012)
46. Gagie, T., Puglisi, S.J., Turpin, A.: Range Quantile Queries: Another Virtue of Wavelet Trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
47. Gog, S.: Compressed Suffix Trees: Design, Construction, and Applications. PhD thesis, Univ. of Ulm, Germany (2011)
48. Golynski, A.: Optimal Lower Bounds for Rank and Select Indexes. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 370–381. Springer, Heidelberg (2006)
49. Golynski, A.: Optimal lower bounds for rank and select indexes. Theor. Comp. Sci. 387(3), 348–359 (2007)
50. Golynski, A., Grossi, R., Gupta, A., Raman, R., Rao, S.S.: On the Size of Succinct Indices. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 371–382. Springer, Heidelberg (2007)
51. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. 17th SODA, pp. 368–373 (2006)
52. Gonnet, G., Baeza-Yates, R., Snider, T.: New indices for text: Pat trees and Pat arrays. In: Information Retrieval: Data Structures and Algorithms, ch. 3, pp. 66–82. Prentice-Hall (1992)
53. González, R., Navarro, G.: Compressed Text Indexes with Fast Locate. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 216–227. Springer, Heidelberg (2007)
54. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA, pp. 841–850 (2003)
55. Grossi, R., Gupta, A., Vitter, J.: When indexing equals compression: Experiments with compressing suffix arrays and applications. In: Proc. 15th SODA, pp. 636–645 (2004)
56. Grossi, R., Ottaviano, G.: The wavelet trie: Maintaining an indexed sequence of strings in compressed space. In: Proc. 31st PODS (to appear, 2012)
57. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. 32nd STOC, pp. 397–406 (2000)
58. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comp. 35(2), 378–407 (2006)
59. Grossi, R., Vitter, J., Xu, B.: Wavelet trees: From theory to practice. In: Proc. 1st CCP, pp. 210–221 (2011)
60. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Proc. 10th SIGMOD, pp. 47–57 (1984)
61. He, M., Munro, J.I.: Succinct Representations of Dynamic Strings. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 334–346. Springer, Heidelberg (2010)
62. Huffman, D.: A method for the construction of minimum-redundancy codes. Proceedings of the I.R.E. 40(9), 1090–1101 (1952)
63. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th FOCS, pp. 549–554 (1989)
64. JáJá, J., Mortensen, C.W., Shi, Q.: Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 558–568. Springer, Heidelberg (2004)

65. Jørgensen, A.G., Larsen, K.D.: Range selection and median: Tight cell probe lower bounds and adaptive data structures. In: Proc. 22nd SODA, pp. 805–813 (2011)
66. Kärkkäinen, J.: Repetition-Based Text Indexing. PhD thesis, Univ. of Helsinki, Finland (1999)
67. Kärkkäinen, J., Puglisi, S.J.: Fixed Block Compression Boosting in FM-Indexes. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 174–184. Springer, Heidelberg (2011)
68. Kreft, S., Navarro, G.: Self-indexing Based on LZ77. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 41–54. Springer, Heidelberg (2011)
69. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proceedings of the IEEE 88(11), 1722–1732 (2000)
70. Mäkinen, V., Navarro, G.: New search algorithms and time/space tradeoffs for succinct suffix arrays. Tech. Rep. C-2004-20, Univ. of Helsinki, Finland (April 2004)
71. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic J. Comp. 12(1), 40–66 (2005)
72. Mäkinen, V., Navarro, G.: Position-Restricted Substring Searching. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 703–714. Springer, Heidelberg (2006)
73. Mäkinen, V., Navarro, G.: Implicit Compression Boosting with Applications to Self-indexing. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 229–241. Springer, Heidelberg (2007)
74. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theor. Comp. Sci. 387(3), 332–347 (2007)
75. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Comp. Biol. 17(3), 281–308 (2010)
76. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comp. 22(5), 935–948 (1993)
77. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM 48(3), 407–430 (2001)
78. McCreight, E.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (1976)
79. Munro, I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
80. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proc. 13th SODA, pp. 657–666 (2002)
81. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. Discr. Alg. 2(1), 87–114 (2004)
82. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
83. Navarro, G., Nekrich, Y., Russo, L.: Space-efficient data-analysis queries on grids. CoRR, abs/1106.4649v2 (2012)
84. Navarro, G., Providel, E.: Fast, small, simple rank/select on bitmaps. In: Proc. 11th SEA (to appear, 2012)
85. Navarro, G., Puglisi, S.J.: Dual-Sorted Inverted Lists. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 309–321. Springer, Heidelberg (2010)
86. Navarro, G., Puglisi, S.J., Valenzuela, D.: Practical Compressed Document Retrieval. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 193–205. Springer, Heidelberg (2011)

87. Navarro, G., Russo, L.M.S.: Space-Efficient Data-Analysis Queries on Grids. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 323–332. Springer, Heidelberg (2011)
88. Navarro, G., Sadakane, K.: Fully-functional static and dynamic succinct trees. CoRR, abs/0905.0768v5 (2010)
89. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. 9th ALENEX (2007)
90. Pătraşcu, M.: Lower bounds for 2-dimensional range counting. In: Proc. 39th STOC, pp. 40–46 (2007)
91. Pătraşcu, M.: Succincter. In: Proc. 49th FOCS, pp. 305–313 (2008)
92. Pătraşcu, M., Viola, E.: Cell-probe lower bounds for succinct partial sums. In: Proc. 21st SODA, pp. 117–122 (2010)
93. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. 13th SODA, pp. 233–242 (2002)
94. Schnattinger, T., Ohlebusch, E., Gog, S.: Bidirectional Search in a String with Wavelet Trees. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 40–50. Springer, Heidelberg (2010)
95. Sirén, J., Välimäki, N., Mäkinen, V., Navarro, G.: Run-Length Compressed Indexes Are Superior for Highly Repetitive Sequence Collections. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 164–175. Springer, Heidelberg (2008)
96. Tischler, G.: On Wavelet Tree Construction. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 208–218. Springer, Heidelberg (2011)
97. Välimäki, N., Mäkinen, V.: Space-Efficient Algorithms for Document Retrieval. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 205–215. Springer, Heidelberg (2007)
98. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)
99. Witten, I., Moffat, A., Bell, T.: Managing Gigabytes, 2nd edn. Morgan Kaufmann (1999)
100. Yu, C.-C., Hon, W.-K., Wang, B.-F.: Efficient Data Structures for the Orthogonal Range Successor Problem. In: Ngo, H.Q. (ed.) COCOON 2009. LNCS, vol. 5609, pp. 96–105. Springer, Heidelberg (2009)

The Maximum Number of Squares in a Tree

Maxime Crochemore[1,3], Costas S. Iliopoulos[1,4], Tomasz Kociumaka[2], Marcin Kubica[2], Jakub Radoszewski[2,⋆], Wojciech Rytter[2,5,⋆⋆], Wojciech Tyczyński[2], and Tomasz Waleń[2,6]

[1] Dept. of Informatics, King's College London, London WC2R 2LS, UK
{maxime.crochemore,csi}@dcs.kcl.ac.uk
[2] Faculty of Mathematics, Informatics and Mechanics,
University of Warsaw, Warsaw, Poland
{kociumaka,kubica,jrad,rytter,w.tyczynski,walen}@mimuw.edu.pl
[3] Université Paris-Est, France
[4] Faculty of Engineering, Computing and Mathematics,
University of Western Australia, Perth WA 6009, Australia
[5] Faculty of Mathematics and Computer Science,
Copernicus University, Toruń, Poland
[6] Laboratory of Bioinformatics and Protein Engineering,
International Institute of Molecular and Cell Biology in Warsaw, Poland

Abstract. We show that the maximum number of different square substrings in unrooted labelled trees behaves much differently than in words. A substring in a tree corresponds (as its value) to a simple path. Let $\mathsf{sq}(n)$ be the maximum number of different square substrings in a tree of size n. We show that asymptotically $\mathsf{sq}(n)$ is strictly between linear and quadratic orders, for some constants $c_1, c_2 > 0$ we obtain:

$$c_1 n^{4/3} \leq \mathsf{sq}(n) \leq c_2 n^{4/3}.$$

1 Introduction

Repetitions are a fundamental notion in combinatorics and algorithmics on words. The basic type of a repetition are squares: words of the type zz, where $z \neq \varepsilon$. (By ε we denote the empty word.) In this paper we consider square substrings corresponding to simple paths in labelled trees. If a tree is a single path then it is a problem of classical repetitions in strings. Combinatorics of squares in classical strings has been investigated in [7,9,10] and for partial words in [3]. Squares were also studied in the context of games, e.g. in [8].

Repetitions in trees and graphs have already been considered, for example in [4,1,2]. The number of square substrings in general graphs dramatically increases — it can be exponential, even in case of binary alphabet.

Assume we have a tree T whose edges are labelled with symbols from an alphabet Σ. By $|T|$ we denote the size of the tree, that is the number of nodes.

⋆ The author is supported by grant no. N206 568540 of the National Science Centre.
⋆⋆ The author is supported by grant no. N206 566740 of the National Science Centre.

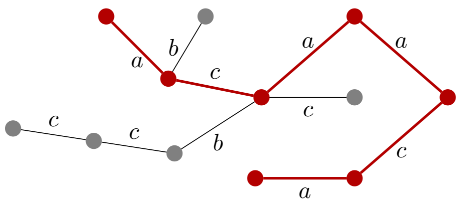


Fig. 1. There are 4 square substrings in this tree: aa, $acaaca$, $bcbc$, cc. Note that cc occurs twice. The longest is $acaaca$ and it corresponds to a path marked with a solid line in the figure.

If u and v are two nodes of T, then by $val(u,v)$ we denote the sequence of labels of edges on the path from u to v. We call $val(u,v)$ a substring of T. (Note that a substring is a string, not a path.) Figure 1 illustrates a square substring in a sample tree. We consider only simple paths: this means that vertices of a path do not repeat (though edges on the path do not need to have distinct labels).

For a tree T, by $\mathsf{sq}(T)$ we denote the number of different square substrings in T. For the tree T from Fig. 1, we have $\mathsf{sq}(T) = 4$. Let $\mathsf{sq}(n)$ be the maximum of $\mathsf{sq}(T)$ over all trees of size n. We show that $\mathsf{sq}(n) = \Theta(n^{4/3})$. Thus $\mathsf{sq}(n)$ has different asymptotics than the maximum number of different square substrings in a standard word (a single path tree) of length n, which is known to be $\Theta(n)$ [7].

We introduce a family of trees which we call combs. The lower bound for $\mathsf{sq}(n)$ turns out to be realized by trees from this family, and such trees also play an important role in the proof of the upper bound. Before we show the general upper bound, we provide some intuition behind this proof by showing the same upper bound for combs and for *special squares* of the form $(a^iba^j)^2$ in general trees.

2 Bounds for Combs

Before we show a general $O(n^{4/3})$ bound on the number of squares in a tree, we analyze the number of squares for a family of trees which we call *standard combs*. The notion of combs is generalized later in the paper.

A *standard comb* is a labelled tree that consists of a path called the *spine*, with at most one *branch* attached to each node of the spine. All spine-edges are labelled with the letter a. Each branch is a path starting with the letter b, followed by a number of a-labelled edges, see Fig. 2.

As we show in the theorem below, there exists a family T_m of standard combs for which $\mathsf{sq}(T_m) = \Omega(|T_m|^{4/3})$. From this one easily obtains $\mathsf{sq}(n) = \Omega(n^{4/3})$ for any n. In this section we also prove an upper bound of $O(n^{4/3})$ for the number of squares in a standard comb of size n. This proof is extensively used throughout the proof of the same upper bound for general trees, given in the following sections. Hence, our family of standard combs T_m meets the asymptotic upper bound for $\mathsf{sq}(n)$ for general trees.

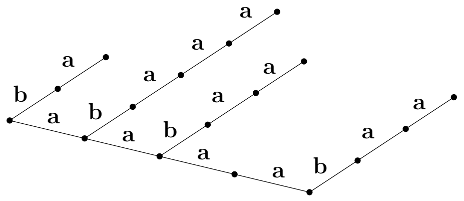


Fig. 2. A standard comb containing 11 square substrings

For $m = k^2$ we define a set $Z_m = \{1, \ldots, k\} \cup \{i \cdot k : 1 \leq i \leq k\}$. For example, if $m = 9$, then $Z_m = \{1, 2, 3, 6, 9\}$.

Lemma 1. *Assume m is a square of a positive integer. Then for each $0 < j < m$ there exist $u, v \in Z_m$ such that $u - v = j$.*

Proof. Each number $0 < j < m$ can be written as $p\sqrt{m} - q$, where $0 < p, q \leq \sqrt{m}$. This formula corresponds to distance between points q and $p\sqrt{m}$. □

For $m = k^2$ we define a standard comb T_m as follows: T_m consists of a spine of length m with vertices numbered from 1 to m, and branches of the form ba^m attached to each vertex $j \in Z_m$ of the spine, see Fig. 3.

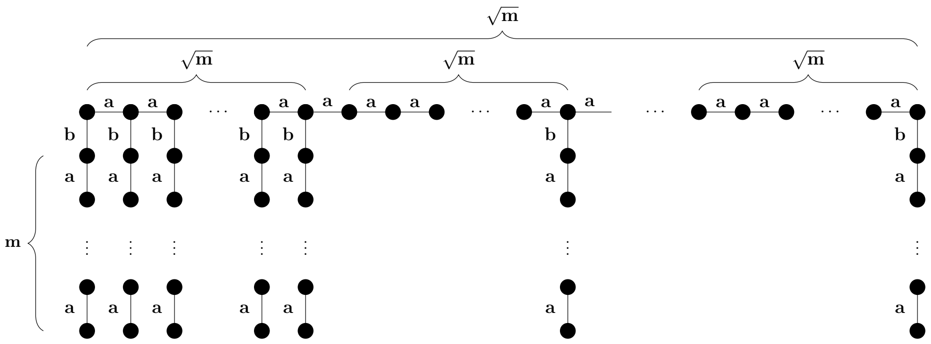


Fig. 3. The structure of a standard comb T_m

Theorem 1. [Lower Bound Theorem]
For each tree T_m we have $\mathsf{sq}(T_m) = \Omega(|T_m|^{4/3})$.

Proof. From Lemma 1, for every $0 < j < m$ there are two nodes u, v of degree 3 on the spine with $distance(u, v) = j$. Hence, T_m contains all squares of the form $(a^i b a^{j-i})^2$ for $0 \leq i \leq j$ and $0 < j < m$. Altogether this gives $\Omega(m^2)$ different squares. Note that $|T_m| = O(m\sqrt{m})$. Hence, the number of square substrings in T_m is $\Omega(|T_m|^{4/3})$. □

Lemma 2. *The number of squares in a standard comb of size n is $O(n^{4/3})$.*

Proof. Let T be a standard comb of size n. Note that T contains only square substrings of the form $(a^i)^2$ or $(a^iba^j)^2$. The number of squares of the former type is $O(n)$. We need to bound the number of squares of the latter type (special squares). Any occurrence of a special square starts and ends within two different branches of T.

There are at most $n^{4/3}$ different special squares for which $i < n^{2/3}$ and $j < n^{2/3}$. Hence, it suffices to prove that there are $O(n^{4/3})$ special square substrings of T for $i \geq n^{2/3}$ or $j \geq n^{2/3}$, we call such special squares *long*.

A branch of a standard comb is called *long* if it contains at least $n^{2/3}$ nodes. Note that there are $O(n^{1/3})$ long branches in T. Any occurrence of a long special square has at least one endpoint in a long branch.

Consider a node u located in a branch B of T and a long branch B'. There is at most one occurrence of a long special square that starts in u and ends within the branch B'. Indeed, if there are i a-labelled edges on the path from u to the spine and k edges on the path connecting the branches B and B' then the considered square $(a^iba^{k-i})^2$ uniquely determines its other endpoint. Hence, the total number of long special squares is bounded by the number of nodes u multiplied by the number of long branches B', that is, by $O(n^{4/3})$. This completes the proof. □

3 Prelude to Upper Bound Proof

In this section we show a tight upper bound for *special squares*, defined at the end of Section 1. Along the way we introduce some part of the machinery for the general proof. Define a *double tree* $\mathcal{D} = (T_1, T_2, R)$ as a labelled tree consisting of two disjoint (except one vertex) trees T_1, T_2 with a common root R. The size of $\mathcal{D}$ is defined as $|\mathcal{D}| = |T_1| + |T_2| - 1$. The substrings of $\mathcal{D}$ are defined as values of paths which start within T_1 and end in T_2. An example of a double tree is shown in Fig. 4, T_1 lies below R (lower tree) while T_2 above R (upper tree).

A directed rooted labelled tree is *deterministic* if the edges going down from the same vertex have different labels. Note that a tree is deterministic if and only if it is a trie (also called a prefix tree) of the values of the paths from R to the leaves. A double tree is *deterministic* if each of the trees T_1, T_2 treated as a directed tree with root R is deterministic. A double deterministic tree is also called here a *D-tree*. The double tree in Fig. 4 is a sample D-tree.

Lemma 3. *For each double (possibly nondeterministic) tree there exists a D-tree with at most the same number of vertices and the same set of substrings (going from T_1 to T_2).*

Proof. For a moment let us direct each tree T_i down from R (treated as a root). Assume we have a vertex v with edges (v, u), (v, w) going to its children and labelled with the same letter a. Then we can *glue* the vertices u, w. We can perform such operation going top-down from the root in a BFS traversal. Note that the resulting trees T_i are deterministic, their sizes could only decrease, and the set of the substrings of the D-tree remains unchanged. $\square$

A path in a tree T is said to be anchored in a node $R \in T$ if R lies on this path. A square is *anchored* in R if it is a value of a path anchored in R.

A path p from v to u in a D-tree is called a *D-square* if $v \in T_1$, $u \in T_2$, $val(v,u)$ is a square and its midpoint lies within T_1, and amongst all such paths of the same value p has its starting node closest to R. Since the D-tree is deterministic, no two D-squares have the same value. Below we bind the number of D-squares in D-trees with the number of squares in ordinary trees. Recall that a *centroid* of a tree T is a node R such that each component of $T \setminus \{R\}$ contains at most $n/2$ nodes. It is a well-known fact that each tree has a centroid.

Lemma 4. *Assume that the number of D-squares in any D-tree of size n is $O(n^{4/3})$. Then the number of squares in any tree is also $O(n^{4/3})$.*

Proof. Let T be a tree of size n and let R be its centroid. Consider a D-tree $\mathcal{D} = (T_1, T_2, R)$ composed of two copies T_1 and T_2 of T, determinised as in Lemma 3.

Let xx be a square in T anchored in R. Either this square or its reverse corresponds to a D-square in $\mathcal{D}$. Obviously $|\mathcal{D}| = O(n)$, therefore, by the hypothesis of the lemma, there are $O(n^{4/3})$ D-squares in this D-tree. Hence, the number of squares in T anchored in R is also $O(n^{4/3})$.

Now we need to count the squares in T that are not anchored in R. After removing the node R, the tree is partitioned into components $T_1, \ldots, T_k$, such that $\sum_i |T_i| = n - 1$ and $|T_i| \le n/2$. Hence, the number of squares in T can be written as:

$$\mathsf{sq}(T) \le O(|T|^{4/3}) + \textstyle\sum_i \mathsf{sq}(T_i).$$

A solution to this recurrence yields the upper bound $\mathsf{sq}(n) = O(n^{4/3})$. $\square$

The proof of the assumption of the previous lemma is the core of this paper. In full generality it is provided in the last section. Here we limit ourselves to a very special type of squares. There is a useful connection between D-trees and combs, as expressed by the following observation, see Fig. 4.

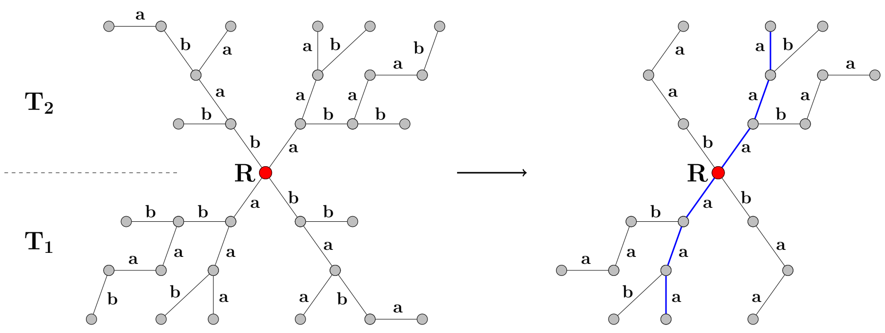


Fig. 4. Illustration of Observation 1

Observation 1. *Assume we have a D-tree labelled with letters a, b. Let us take only paths from a vertex in T_1 to R or from R to a vertex in T_2 which contain at most one b, with other edges labelled with a. Then the resulting labelled tree is a standard comb (with at most one additional branch attached to R).*

Corollary 1. *Assume binary alphabet $\{a, b\}$. The maximum number of special squares in any tree is $O(n^{4/3})$.*

Proof. By Lemma 4, it suffices to consider a D-tree $\mathcal{D} = (T_1, T_2, R)$ and only special D-squares in $\mathcal{D}$. The special D-squares with both occurrences of b in T_2 are uniquely determined by their upper end and those with both occurrences in T_1 by their lower end. Hence the number of such D-squares in linear. By Lemma 2 and Observation 1, there are $O(n^{4/3})$ special D-squares which have one b in T_1 and one b in T_2. □

4 (p, q)-Representations of Substrings

In this section w denotes a word of length n. We start by recalling a few basic notions of word periodicity, see e.g. [5,6,11]. A *border* of w is defined as a prefix of w which is also a suffix of w. We say that a positive integer p is a *period* of $w = w_1 w_2 \dots w_n$ if $w_i = w_{i+p}$ holds for all $1 \le i \le n - p$. A non-empty word w is called *periodic* if it has a period p such that $2p \le |w|$. The *primitive root* of a word w, denoted $\mathsf{root}(w)$, is the shortest word u such that $u^k = w$ for some positive integer k. We call a word w *primitive* if $\mathsf{root}(w) = w$, otherwise it is called *non-primitive.* We recall several periodic properties of words [5,6,11].

Fact 1. *A word w has a border of length b if and only if w has a period $|w| - b$.*

Fact 2 (Periodicity Lemma). *If a word of length n has two periods p and q, such that $p + q \le n + \gcd(p, q)$, then $\gcd(p, q)$ is also a period of the word.*

Fact 3 (Synchronizing Properties)

(a) If $uv = vu$ then both words u, v are powers of the same primitive word.
(b) Let $q \neq \varepsilon$ be a primitive word. Then q has exactly two occurrences in qq.
(c) Let $p \neq \varepsilon, q \neq \varepsilon$ be such that pq is primitive. Then qp has exactly one occurrence in $pqpq$.

As a consequence of the synchronizing properties of primitive words, we obtain the following auxiliary fact that will be useful in the proof of the main result (Lemma 9).

Fact 4. *Let p, p', q, q' be words such that: $q \neq \varepsilon$ and $q' \neq \varepsilon$, pq is primitive, $pq = p'q'$, and $qp = q'p'$. Then $p = p'$ and $q = q'$.*

Proof. First assume $p = \varepsilon$. Then $q'p' = qp = pq = p'q'$. From Fact 3a, since $q' \neq \varepsilon$, we get $p' = \varepsilon$. This naturally implies that $q = q'$. Now assume that $p \neq \varepsilon$. We have $pqpq = p'q'p'q' = p'qpq'$ and from Fact 3c we know that there is only one occurrence of qp in $pqpq$. Thus $p = p'$ and $q = q'$. □

Assume that w is periodic. There exists a unique representation of w: $w = (pq)^k p$ such that $k \geq 2$, $q \neq \varepsilon$ and pq is primitive. This representation is called a canonical representation of w. Here $|pq|$ is the shortest period of w. We say that w is *of periodic type* (p, q).

Example 1. The word abbabbab has a canonical representation $(\mathtt{abb})^2\mathtt{ab}$, with $p = \mathtt{ab}$ and $q = \mathtt{b}$. On the other hand, bababa has a representation $(\mathtt{ba})^3$ with $p = \varepsilon$ and $q = \mathtt{ba}$.

Fact 5. *Borders of w that are periodic belong to $O(\log n)$ periodic types. Additionally, w may have $O(\log n)$ borders which are not periodic.*

Proof. As for the first part of the lemma, let u, v be periodic borders of w such that $|u| < |v| \leq 1.5|u|$. We show that u and v are of the same periodic type.

Indeed, let $v = (pq)^k p$, where $d = |pq|$ is the shortest period of v, and $u = (p'q')^{k'} p'$ be the canonical representations of v and u. The border u is also a border of v. Due to Fact 1, both d and $|v| - |u|$ are periods of v. Moreover $d < \frac{1}{2}|v|$ (since $k \geq 2$) and $|v| - |u| \leq \frac{1}{3}|v|$ (since $|v| \leq 1.5|u|$). Hence, by the Periodicity Lemma, $|v| - |u|$ is a multiple of d. The word u is a prefix of v, hence $u = (pq)^\ell p$ for some $\ell < k$. Now, let us show that $\ell \geq 2$. Assume to the contrary that $\ell \leq 1$. Then:

$$3|p| + 2|q| \leq (k+1)|p| + k|q| = |v| \leq 1.5|u| \leq 3|p| + 1.5|q|.$$

This is clearly a contradiction, since $|q| > 0$. Hence $\ell \geq 2$. Now by the uniqueness of canonical representations we obtain $p = p'$, $q = q'$ and $k' = \ell$. This concludes that the borders u and v are of the same periodic type.

As for the second part, let u, v be non-periodic borders of w such that $|u| < |v|$. We show that $|v| > 2|u|$.

Assume to the contrary that $|u| < |v| \leq 2|u|$. As in the previous part of the proof, we see that $|v| - |u|$ is a period of v. However,

$$2(|v| - |u|) = |v| + |v| - 2|u| \leq |v| + 2|u| - 2|u| = |v|,$$

therefore v is periodic, a contradiction. □

A periodic border v of w is called *global* if its period is the period of the whole word w. Equivalently, v is global if v, w are of the same periodic type. If w is of periodic type (p, q) and its canonical representation is $w = (pq)^k p$, then all its global borders are $(pq)^{k'} p$ for $2 \leq k' \leq k$.

Definition 1. *Let p, q be such words that $q \neq \varepsilon$ and pq is primitive. The representation $w = p(qp)^\ell y(pq)^r p$ is called the (p, q)-representation of w if:* **(a)** *$\ell, r \geq 1$;* **(b)** *y has a prefix qp but not $(qp)^2$;* **(c)** *y has a suffix pq but not $(pq)^2$, see Fig. 5 and 6.*

Lemma 5. *Assume w has a non-global periodic border of periodic type (p, q). Then w has a (p, q)-representation $w = p(qp)^\ell y(pq)^r p$. Moreover:* **(a)** *this (p, q)-representation is unique (i.e., ℓ, r and y are unique);* **(b)** *y is not a prefix of $(qp)^2$;* **(c)** *y is not a suffix of $(pq)^2$;* **(d)** *all borders of w of periodic type (p, q) are: $(pq)^{k'} p$ for $2 \leq k' \leq \min(\ell, r) + 1$.*

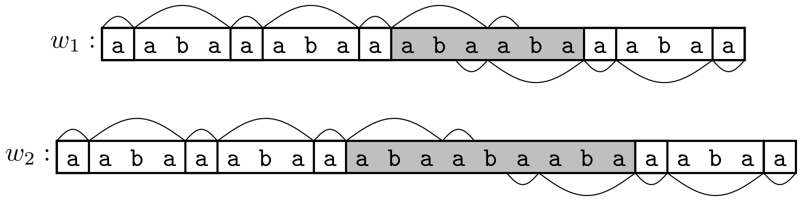

w_1 :
a a b a a a b a a a b a a b a a a b a a
w_2 :
a a b a a a b a a a b a a b a a b a a a b a a

Fig. 5. The (p, q)-representations: $w_1 = \mathtt{a}(\mathtt{abaa})^2\mathtt{abaaba}(\mathtt{aaba})^1\mathtt{a}$ and $w_2 = \mathtt{a}(\mathtt{abaa})^2\mathtt{abaabaaba}(\mathtt{aaba})^1\mathtt{a}$. In both cases $p = \mathtt{a}$, $q = \mathtt{aba}$ and y is marked grey.

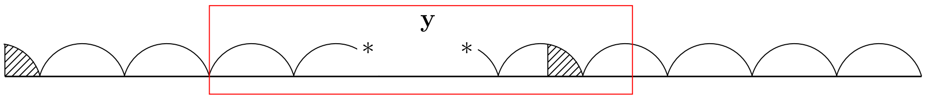


Fig. 6. A schematic view of a (p, q)-representation. The $*$-symbols correspond to the first mismatch for the continuation of the period qp from the left side and the period pq from the right side.

Proof. Let $u = (pq)^k p$ be the longest border of w of type (p, q). Clearly p is a prefix of w and $|w| > |u| > 2|p|$, so let us write $w = pzp$. Now, let $(qp)^{\ell+1}$ be the maximal power of qp that is a prefix of z. We have for some z': $w = p(qp)^\ell z' p$. Let $(pq)^{r+1}$ be the maximal power of pq that is a suffix of z'. Now we can write $w = p(qp)^\ell y(pq)^r p$. Let us prove that this representation satisfies the required conditions. We get the following easily:

– z has a prefix qp and a suffix pq.
– z' has a prefix qp but not $(qp)^2$, and a suffix pq
– y has a prefix qp but not $(qp)^2$, and a suffix pq but not $(pq)^2$.

Let us now show that y is not a prefix of $(qp)^2$. Assume to the contrary. Recall that pq is a suffix of y. Thus we get an occurrence of pq in $qpqp$. If $p \neq \varepsilon$, from Fact 3 we get that $y = qpq$. But then, w would be of type (p, q). Therefore $p = \varepsilon$. From Fact 3 we conclude that $y = q$ and w is a power of q. This is, however, impossible since w has a non-global periodic border $(pq)^k p = q^k$. This contradiction implies that y is not a prefix of $(qp)^2$.

A symmetric argument proves that y is not a suffix of $(pq)^2$. Since none of $(qp)^2$ and y is a prefix of the other, $p(qp)^{\ell+2}$ is not a prefix of w. Similarly $(pq)^{r+2}p$ is not a suffix of w. Thus $u = (pq)^{\min(\ell,r)+1}$. In particular $\ell, r \geq 1$. Clearly $(pq)^{k'}p$ for $2 \leq k' \leq \min(\ell, r) + 1$ are the only periodic borders of w of the type (p, q).

Now it remains to show the uniqueness of the representation. Assume there was another representation $w = p(qp)^{\ell'} y'(pq)^{r'} p$. Since y' has a prefix qp but not $(qp)^2$, $\ell' + 1$ is the largest m such that $p(qp)^m$ is a prefix of m, that is $\ell' = \ell$. Similarly $r' = r$ and finally $y' = y$. □

A periodic border is called *maximal* if it is the longest border of its periodic type. By Fact 5, w has $O(\log n)$ maximal borders.

We call a border *regular* if it is periodic and is neither global nor maximal.

5 General Combs and General Upper Bound

Due to Lemma 4, in this section we are only dealing with D-squares in a deterministic double tree $\mathcal{D} = (T_1, T_2, R)$ of size n.

For a node $v \in T_1$ we define the set $SQ(v)$ of all D-squares which start in v. Each D-square in $SQ(v)$ of value xx induces a period $|x|$ of $val(v, R)$, and thus corresponds to a border u of $val(v, R)$. This D-square is called regular if u is a regular border of $val(v, R)$. The periodic type of a D-square is defined as the periodic type of the underlying border u.

The following lemma lets us concentrate only on the regular D-squares.

Lemma 6. *At most $O(n \log n)$ D-squares in $\mathcal{D}$ are not regular.*

Proof. We show that in $SQ(v)$ at most $O(\log n)$ D-squares are not regular. Each D-square in $SQ(v)$ corresponds to a different border of $val(v, R)$. The borders corresponding to non-regular D-squares are non-periodic, global or maximal; we extend these terms to D-squares as in the case of regular D-squares and borders. We have the following claim.

Claim. In $SQ(v)$ at most one D-square is global.

Proof. Let xx and $x'x'$ be values of two global D-squares starting in v. Assume $|x| < |x'|$. Let $w = val(v, R) = (pq)^k p$. The global D-squares are of the form $(pq)^{k'}$. Since a global border is periodic of periodic type (p, q), we have $1 \le k' \le k - 2$. Let $x = (pq)^\ell$ and $x' = (pq)^{\ell'}$, $\ell < \ell'$.

Let u be an ancestor of v the defined by $val(u, R) = (pq)^{k-1}p$. A path starting in u and going to the upper end of $x'x'$ has the value $(pq)^{2\ell'-1}$, which has a prefix $(pq)^{2\ell} = xx$. We have $\ell < k - 1$, so this occurrence has a centre in the lower part of the D-tree. Hence, it is a candidate for a D-square of the value xx. This concludes that the original path of value xx starting in v could not be a D-square, which is a contradiction. □

As we noticed in Section 4, only $O(\log n)$ borders of $val(v, R)$ can be non-periodic or maximal. Hence only $O(\log n)$ D-squares starting in v can correspond to a non-regular border. Thus there can be only $O(n \log n)$ D-squares which are not regular. □

We introduce an important notion of a general comb. Before we give a formal definition, we provide a few sentences of intuition behind it. Assume $val(v, u)$ is a regular D-square of type (p, q). By Lemma 5 we have the representation $val(v, R) = p(qp)^\ell y(pq)^r p$. All D-squares of type (p, q) starting in v correspond to the same representation. Those squares induce a particular structure of paths labelled with p, q and y in the upper part of the D-tree $\mathcal{D}$. A similar structure is also present in the lower part.

Definition 2. *Let p, q, y satisfy the conditions of Lemma 5. A D-tree (T_1, T_2, R) is called a (p, q, y)-comb if*

- *for each leaf $v \in T_1$, $val(v, R) = p(qp)^m y(pq)^k p$ for some integers k, m,*

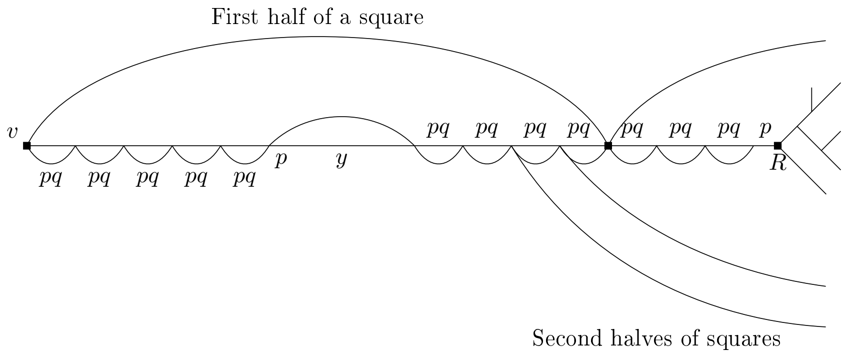


Fig. 7. Correspondence between squares in trees and borders

– *for each leaf* $u \in T_2$*,* $val(R,u) = (qp)^m y(pq)^k$ *for some integers* k, m*.*

Let $\mathcal{D}$ *be a D-tree containing a regular D-square of periodic type* (p,q)*. Then by* $Comb(\mathcal{D},p,q,y)$ *we denote the maximal subtree of* $\mathcal{D}$ *that is a* (p,q,y)*-comb.*

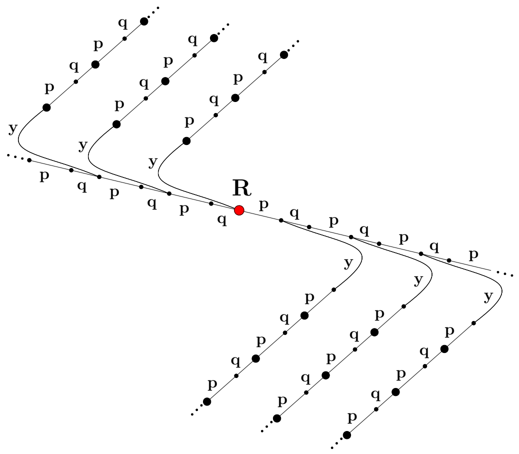


Fig. 8. A sample (p,q,y)-comb with the root R; the main nodes are shown as larger circles; the bended edges are partially glued to the spine due to determinisation. All (p,q,y)-combs are subtrees of this infinite D-tree. For $p = \varepsilon$, $q =$ `a` and $y =$ `aba` we obtain a standard comb.

Note that the conditions of Definition 1 and Lemma 5 in particular imply that neither y is a prefix of $(qp)^2$ nor $(qp)^2$ a prefix of y. Similarly neither y is a suffix of $(pq)^2$ nor $(pq)^2$ a suffix of y. Hence, all combs have a regular structure, see Fig. 8. Each (p,q,y)-comb consists of a path labelled with $p(qp)^m$ (for an integer m) and containing the root, which we call the *spine*, and the *branches* which are paths attached directly to the spine.

Some nodes of the combs are particularly important for the D-squares. These are nodes v of values $val(v, R) = p(qp)^k y(pq)^m p$ in the lower part, and nodes u of values $val(R, u) = (qp)^k y(pq)^m$ in the upper part (k, m are arbitrary nonnegative integers in both cases). Such nodes are called *main*. For a comb $\mathcal{C}$, by $Main(\mathcal{C})$ we denote the set of main nodes in $\mathcal{C}$, and by $\|\mathcal{C}\|$ we denote $|Main(\mathcal{C})|$. D-squares in $\mathcal{C}$ with both endpoints in main nodes are said to be *induced* by the comb.

The following lemma confirms a strong relation between combs and regular D-squares.

Lemma 7. *Each regular D-square of type (p, q) in $\mathcal{D}$ is induced by the corresponding comb $Comb(\mathcal{D}, p, q, y)$.*

Proof. Let $val(v, u)$ be a regular D-square in $\mathcal{D}$ of type (p, q). By Lemma 5, $val(v, R)$ has a following representation $val(v, R) = p(qp)^\ell y(pq)^r p$. The underlying border is regular, that is $(pq)^k p$ for some $2 \le k \le \min(\ell, r)$, hence the value of the D-square is $(p(qp)^\ell y(pq)^{r-k})^2$. Thus $val(R, u) = (qp)^{\ell-k} y(pq)^{r-k}$. Now it is clear that both v and u are main nodes of $Comb(\mathcal{D}, p, q, y)$. □

The following result is a simple extension of the upper bound for standard combs.

Lemma 8. *A comb $\mathcal{C}$ induces $O(\|\mathcal{C}\|^{4/3})$ D-squares.*

Proof. Let $\mathcal{C}$ be a (p, q, y)-comb. We can construct a (ε, a, aba)-comb $\mathcal{C}'$ of the same structure of branches and main nodes as $\mathcal{C}$. Clearly, $\|\mathcal{C}\| = \|\mathcal{C}'\|$ and the number of squares induced by both combs is the same. But now $\mathcal{C}'$ is a standard comb.

For the comb $\mathcal{C}'$ we have an upper bound $\mathsf{sq}(\mathcal{C}') = O(|\mathcal{C}'|^{4/3})$ from Lemma 2. In order to obtain an $O(\|\mathcal{C}'\|^{4/3})$ bound for the number of squares *induced* by $\mathcal{C}'$, it suffices to restrict the proof of that lemma to special squares $(a^i ba^j)$ for $i \ge 2$ and $j \ge 1$. This way we obtain an upper bound of $O(\|\mathcal{C}'\|^{4/3})$ for the number of D-squares induced by $\mathcal{C}'$, consequently an $O(\|\mathcal{C}\|^{4/3})$ upper bound for an arbitrary comb $\mathcal{C}$. □

Finally, we can prove the main lemma.

Lemma 9 (Key lemma). *A D-tree of size n contains $O(n^{4/3})$ regular D-squares.*

Proof. We show that combs in a D-tree are almost disjoint with regard to their main nodes. More precisely, due to combinatorial properties of words, any two different such combs can have at most two common main nodes in upper branches, and same for lower branches (Claim 2). Before that, we show that certain pairs of combs (with $|pq| = |p'q'|$) have none common main nodes at all (Claim 1).

Claim 1. *If $\mathcal{C} = Comb(\mathcal{D}, p, q, y)$ and $\mathcal{C}' = Comb(\mathcal{D}, p', q', y')$ are different combs satisfying $|pq| = |p'q'|$, then $Main(\mathcal{C}) \cap Main(\mathcal{C}') = \emptyset$.*

Proof. Assume u is a common main node of the two combs. It can lie either in the upper part or in the lower part of $\mathcal{D}$. First, let us consider the first case. Let $w = val(R, u)$. Since qp and $q'p'$ is a prefix of w and pq and $p'q'$ is a suffix of w, we get that $qp = q'p'$ and $pq = p'q'$. By Fact 4, $p = p'$ and $q = q'$. We now know that $w = (qp)^{\ell} y(pq)^r = (qp)^{\ell'} y'(pq)^{r'}$. Assume $\ell \neq \ell'$, without the loss of generality $\ell < \ell'$. Since y' has a prefix qp, $y(pq)^r$ has a prefix $(qp)^2$.i This is impossible by the definition of a comb. By a similar argument, $r = r'$. Hence $y = y'$, so $\mathcal{C}$ and $\mathcal{C}'$ cannot be different combs.

Now, consider a common main node u in the lower part. Let $w = val(u, R)$. As previously we easily obtain that $p = p'$ and $q = q'$. This time we have $w = p(qp)^{\ell} y(pq)^r p = p(qp)^{\ell'} y'(pq)^{r'} p$. Exactly in the same way as before, we get $\ell = \ell'$, $r = r'$ and conclude that $y = y'$. □

Claim 2. *Let $\mathcal{C} = Comb(\mathcal{D}, p, q, y)$ and $\mathcal{C}' = Comb(\mathcal{D}, p', q', y')$. Then either $\mathcal{C} = \mathcal{C}'$ or $|Main(\mathcal{C}) \cap Main(\mathcal{C}')| \leq 4$.*

Proof. By Claim 1, it suffices to show that if $\mathcal{C}$ and $\mathcal{C}'$ have at least 5 common main nodes, then $|pq| = |p'q'|$. First we show that no two common main nodes may lie on a single branch (in the upper or in the lower tree). Assume we have such two nodes u and u' and u is the lower among them. Then $val(u, u')$ is a power of both pq and $p'q'$. But pq and $p'q'$ are primitive, so $|pq| = |\mathsf{root}(val(u, u'))| = |p'q'|$, which by claim concludes the proof of this case.

Now we show that no three common main nodes may lie in the upper tree. Assume that u, u', u'' are such nodes. Since no two of them can lie on the same branch, they are aligned as in Fig. 9 (up to a permutation of u, u', u'').

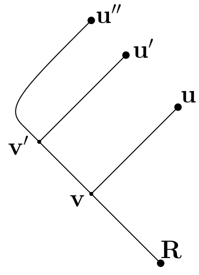


Fig. 9. Main nodes on three different branches of a D-tree

Note that nodes v and v' need to be branching nodes of both combs. Obviously, $|pq| = |\mathsf{root}(val(v, v'))| = |p'q'|$. This again concludes the proof of the current case.

Finally, it remains to show that no three common main nodes may lie in the lower tree. The proof is exactly the same as in the previous case. □

Now let us divide combs into small combs, for which $\|\mathcal{C}\| \leq n^{0.6}$, and the remaining big combs. Due to the following claim, we can restrict the further analysis to big combs.

Claim 3. *The number of regular D-squares in $\mathcal{D}$ induced by small combs is $o(n^{4/3})$.*

Proof. Consider a node v in the lower part of the D-tree $\mathcal{D}$. Assume $SQ(v)$ contains $s > 0$ regular D-squares of type (p, q), and $val(v, R) = w = p(qp)^\ell y(pq)^r p$ is a corresponding representation. Let $\mathcal{C} = Comb(\mathcal{D}, p, q, y)$. We will show that $|Main(\mathcal{C})| = \Omega(s^2)$.

Indeed, let $x_1x_1, \ldots, x_sx_s$ be those s regular D-squares ordered by increasing lengths. As in the proof of Lemma 7 the values of these D-squares are of regular form. Namely, we have $x_i = p(qp)^\ell y(pq)^{k_i}$ for some $\max(0, r-\ell) \le k_1 < \ldots < k_s < r$. Let $u_1, \ldots, u_s$ be the other endpoints of these D-squares. We have $val(R, u_i) = (qp)^{\ell-r+k_i} y(pq)^{k_i}$. The nodes in the upper tree of $\mathcal{C}$ corresponding to paths of the form $(qp)^{\ell-r+k_i} y(pq)^k$ for $0 \le k \le k_i$ are all distinct main nodes, hence $|Main(\mathcal{C})| \ge ((k_1+1)+(k_2+1)+\ldots+(k_s+1)) \ge (1+2+\ldots+s) = \Omega(s^2)$.

As a consequence, we get that $O(n^{0.3})$ D-squares from $SQ(v)$ can be induced by a single small comb. Moreover, by Fact 5, regular squares starting in v are induced by $O(\log n)$ combs. Consequently, the number of elements of $SQ(v)$ that are induced by small combs is $O(n^{0.3}) \cdot O(\log n) = o(n^{1/3})$. In total, small combs induce $o(n^{4/3})$ squares. □

Let $\mathcal{C}_1, \ldots, \mathcal{C}_k$ denote all big combs of $\mathcal{D}$. As a consequence of Claim 2, the total size of all these combs, measured in the number of main nodes, turns out to be linear in terms of n.

Claim 4. *For any D-tree of n nodes, $\sum_{i=1}^{k} \|\mathcal{C}_i\| = O(n)$.*

Proof. We will show the following inequality:

$$\sum_{i=1}^{k} \|\mathcal{C}_i\| \le n + 2(k-1)(k-2). \tag{1}$$

From this inequality, by $\|\mathcal{C}_i\| \ge n^{0.6}$, we get

$$k \cdot n^{0.6} \le n + 2(k-1)(k-2).$$

Comparing asymptotics of both sides of the inequality, we conclude that for *almost all* values of n (that is, all values excluding only a finite number) $k < n^{0.5}$. For such values of k the right side of the inequality (1) is $O(n)$, which will conclude the proof of the claim provided that we show that inequality.

As for the proof of (1), using Claim 2 we obtain that:

$$\begin{aligned}\left|\bigcup_{i=1}^{k} Main(\mathcal{C}_i)\right| &= \left|\bigcup_{i=1}^{k}\left(Main(\mathcal{C}_i) \setminus \bigcup_{j=1}^{i-1} Main(\mathcal{C}_j)\right)\right| \\ &= \sum_{i=1}^{k}\left|Main(\mathcal{C}_i) \setminus \bigcup_{j=1}^{i-1} Main(\mathcal{C}_j)\right| \\ &\ge \sum_{i=1}^{k}(\|\mathcal{C}_i\| - 4\cdot(i-1)) \;=\; \sum_{i=1}^{k}\|\mathcal{C}_i\| - 2(k-1)(k-2).\end{aligned}$$

Consequently:

$$\sum_{i=1}^{k} \|\mathcal{C}_i\| - 2(k-1)(k-2) \;\le\; \left|\bigcup_{i=1}^{k} Main(\mathcal{C}_i)\right| \;\le\; n$$

which is equivalent to the inequality (1). □

Let $\mathcal{D}$ be a D-tree of size n. Due to Lemma 7, each regular D-square in $\mathcal{D}$ is induced by a comb in $\mathcal{D}$. By Claim 3, there are $o(n^{4/3})$ such D-squares induced by small combs. Finally, by Lemma 8 and Claim 4, the number of regular D-squares induced by big combs $\mathcal{C}_1, \ldots, \mathcal{C}_k$ of $\mathcal{D}$ is bounded by:

$$\sum_{i=1}^{k} O\left(\|\mathcal{C}_i\|^{4/3}\right) = O\left(\sum_{i=1}^{k} \|\mathcal{C}_i\|\right)^{4/3} = O(n^{4/3}).$$

This completes the proof of the key lemma. □

As a corollary of the key lemma, by Lemma 4 and 6 we obtain the desired upper bound.

Theorem 2. *The number of squares in a tree with n nodes is $O(n^{4/3})$.*

References

1. Alon, N., Grytczuk, J.: Breaking the rhythm on graphs. Discrete Mathematics 308(8), 1375–1380 (2008)
2. Alon, N., Grytczuk, J., Haluszczak, M., Riordan, O.: Nonrepetitive colorings of graphs. Random Struct. Algorithms 21(3-4), 336–346 (2002)
3. Blanchet-Sadri, F., Mercas, R., Scott, G.: Counting distinct squares in partial words. Acta Cybern. 19(2), 465–477 (2009)
4. Bresar, B., Grytczuk, J., Klavzar, S., Niwczyk, S., Peterin, I.: Nonrepetitive colorings of trees. Discrete Mathematics 307(2), 163–172 (2007)
5. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007)
6. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2003)
7. Fraenkel, A.S., Simpson, J.: How many squares can a string contain? J. of Combinatorial Theory Series A 82, 112–120 (1998)
8. Grytczuk, J., Przybylo, J., Zhu, X.: Nonrepetitive list colourings of paths. Random Struct. Algorithms 38(1-2), 162–173 (2011)
9. Ilie, L.: A simple proof that a word of length n has at most $2n$ distinct squares. J. Comb. Theory, Ser. A 112(1), 163–164 (2005)
10. Ilie, L.: A note on the number of squares in a word. Theor. Comput. Sci. 380(3), 373–376 (2007)
11. Lothaire, M.: Combinatorics on Words. Addison-Wesley, Reading (1983)

Faster and Simpler Minimal Conflicting Set Identification
(Extended Abstract)

Aïda Ouangraoua[1] and Mathieu Raffinot[2]

[1] INRIA Lille, LIFL - Université Lille 1, Villeneuve d'Ascq, France
aida.ouangraoua@inria.fr
[2] CNRS/LIAFA, Université Paris Diderot - Paris 7, France
raffinot@liafa.jussieu.fr

Abstract. Let $\mathcal{C}$ be a finite set of n elements and $\mathcal{R} = \{r_1, r_2, \ldots, r_m\}$ a family of m subsets of $\mathcal{C}$. A subset $\mathcal{X}$ of $\mathcal{R}$ satisfies the *Consecutive Ones Property (C1P)* if there exists a permutation P of $\mathcal{C}$ such that each r_i in $\mathcal{X}$ is an interval of P. A *Minimal Conflicting Set (MCS)* $\mathcal{S} \subseteq \mathcal{R}$ is a subset of $\mathcal{R}$ that does not satisfy the C1P, but such that any of its proper subsets does. In this paper, we present a new simpler and faster algorithm to decide if a given element $r \in \mathcal{R}$ belongs to at least one MCS. Our algorithm runs in $O(n^2m^2 + nm^7)$, largely improving the current $O(m^6n^5(m+n)^2 \log(m+n))$ fastest algorithm of [Blin *et al*, CSR 2011]. The new algorithm is based on an alternative approach considering minimal forbidden induced subgraphs of interval graphs instead of Tucker matrices.

1 Introduction

Let $\mathcal{C} = \{c_1, \ldots, c_n\}$ be a finite set of n elements and $\mathcal{R} = \{r_1, r_2, \ldots, r_m\}$ a family of m subsets of $\mathcal{C}$. Those sets can be seen as a $m \times n$ 0-1 matrix $M = (\mathcal{R}, \mathcal{C})$, such that the set $\mathcal{C}$ represents the columns of the matrix, and the set $\mathcal{R}$ the rows of the matrix: each $r_i \in \mathcal{R}$ represents the set of columns where row i has an entry 1.

A subset $\mathcal{X}$ of $\mathcal{R}$ satisfies the consecutive ones property (C1P) if there exists a permutation P of $\mathcal{C}$ such that each r_i in $\mathcal{X}$ is an interval of P. Testing the consecutive ones property is the core of many algorithms that have applications in a wide range of domains, from VLSI circuit conception through planar embeddings [8] to computational biology for the reconstruction of ancestral genomes [1,2,4,5,10]. We focus on this last field in this paper.

On real biological matrices, the C1P is rarely satisfied, and only some subsets of rows might satisfy the desired property. However, the combinatorics of such sets is difficult to handle, and a strategy to deal with them has been proposed in [1,5,10]. It consists in identifying the rows belonging to minimal conflicting subsets of rows that do not satisfy the C1P, but such that any of their proper subset does.

Definition 1. *A set $\mathcal{S} \subseteq \mathcal{R}, \mathcal{S} \neq \emptyset$ is a* Minimal Conflicting Set *(MCS) if $\mathcal{S}$ does not satisfy the C1P, but such that $\forall \mathcal{X}, \mathcal{X} \subset S$, the set $\mathcal{X}$ satisfies the C1P.*

However, it is not difficult to build examples of matrices such that the number of MCS is polynomial or even exponential in the number of rows. Figure 1 shows an example of a matrix for which the number of MCS is exponential in the number of rows.

(a)
$$\begin{array}{c|cccccc}
 & \mathbf{c_1} & c_2 & \mathbf{c_3} & c_4 & \mathbf{c_5} & c_6 \\
r_1 & 1 & 0 & 0 & 0 & 0 & 1 \\
r_2 & 1 & 1 & 0 & 0 & 0 & 0 \\
r_3 & 0 & 1 & 1 & 0 & 0 & 0 \\
r_4 & 0 & 0 & 1 & 1 & 0 & 0 \\
r_5 & 0 & 0 & 0 & 1 & 1 & 0 \\
r_6 & 0 & 0 & 0 & 0 & 1 & 1 \\
r_7 & 1 & 1 & 0 & 0 & 0 & 0 \\
r_8 & 0 & 0 & 1 & 1 & 0 & 0 \\
r_9 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}$$

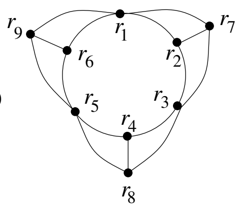
(b)


Fig. 1. (a) A matrix not satisfying the C1P and such that the number of MCS is exponential in the number of rows. (b) A row intersection graph of the matrix whose vertices correspond to the rows of the matrix, and such that there exists an edge between two rows r_i and r_j if $r_i \cap r_j \neq \emptyset$.

From a computational point of view, the first question that arises is the following: is a given row $r \in \mathcal{R}$ included in at least one MCS ? This question has been raised in [1], recalled in [4,5] and recently solved in polynomial time $O(m^6 n^5 (m+n)^2 \log(m+n))$ in [3]. This currently fastest algorithm is based on the identification of minimal Tucker forbidden submatrices [6,11].

In this paper we present a new simpler $O(m^2 n^2 + nm^7)$ time algorithm for deciding if a given row belongs to at least one MCS and if true exhibit one. Our algorithm is based on an alternative approach considering minimal forbidden induced subgraphs of interval graphs [7] instead of Tucker matrices. Moreover, our central paradigm consists in reducing the recognition of complex forbidden induced subgraphs to the detection of induced cycles in ad-hoc graphs. Our approach is faster and simpler, but a limit shared by both approaches resides in avoiding to report the number of MCS to which a given row belongs.

2 MCS and Forbidden Induced Subgraphs

The *row-column intersection graph* of a 0-1 matrix $M = (\mathcal{R}, \mathcal{C})$ is a vertex-colored bipartite graph $G_{RC}(M)$ whose set of vertices is $\mathcal{R} \cup \mathcal{C}$; the vertices corresponding to rows (resp. columns) are black (resp. white) ; there exists an edge between two rows $r_i \in \mathcal{R}$ and $r_j \in \mathcal{R}$ if $r_i \cap r_j \neq \emptyset$, and there exists an edge between a row $r \in \mathcal{R}$ and a column $c \in \mathcal{C}$ if $c \in r$.

It should be noted that a column vertex (white) is only connected to row vertices (black). The *neighborhood* $N(r)$ of a row r is the set of rows intersecting r, $N(r) = \{x \in \mathcal{R} \ : \ r \cap x \neq \emptyset\}$ and $N(r_i, r_j) = N(r_i) \cap N(r_j)$. The *span* $L(c)$ of a column c is the set of rows containing c, $L(c) = \{r \in \mathcal{R} \ : \ c \in r\}$.

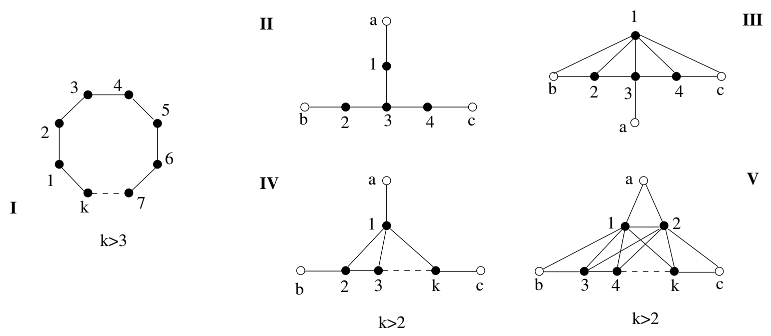


Fig. 2. Forbidden induced subgraphs for the row-column intersection graph of $M = (\mathcal{R}, \mathcal{C})$ to satisfy the C1P. Black vertices correspond to rows, and white vertices to colums.

Theorem 1 ([7], Theorem 4). *A 0-1 matrix $M = (\mathcal{R}, \mathcal{C})$ satisfies the C1P if and only if its row-column intersection graph does not contain a forbidden induced subgraph of the form I, II, III, IV, or V (Figure 2).*

Property 1. From Theorem 1, a set $\mathcal{S} \subseteq \mathcal{R}$ is a MCS if the row-column intersection graph $G_{RC}(\mathcal{S}, \mathcal{C})$ contains a subgraph of the form I, II, III, IV, or V; and for any $\mathcal{T} \subset \mathcal{S}$, $G_{RC}(\mathcal{T}, \mathcal{C})$ does not contain a subgraph of the form I, II, III, IV, or V.

Given a MCS $\mathcal{S} \subseteq \mathcal{R}$, a forbidden induced subgraph contained in $G_{RC}(\mathcal{S}, \mathcal{C})$ is said to be *responsible* for the MCS $\mathcal{S}$. If this forbidden induced subgraph is of the form I (resp. II; III; IV; V), we simply say that $\mathcal{S}$ is a MCS of the form I (resp. II; III; IV; V).

Definition 2. *A row of a MCS $\mathcal{S}$ that intersects all other rows of $\mathcal{S}$ is called a* kernel *of $\mathcal{S}$. In a forbidden induced subgraph responsible for $\mathcal{S}$, any kernel of $\mathcal{S}$ constitutes a black vertex that is connected to all other black vertices.*

Property 2. An induced subgraph of the form II, III, IV, or V always contains at least one kernel, while an induced subgraph of the form I contains no kernel. We denote by $G_R(M)$, the subgraph of $G_{RC}(M)$ induced by the set of rows $\mathcal{R}$, thus containing only black vertices.

Graph Sizes. $G_R(M)$ has m vertices and at most $m(m-1)/2$ edges, while $G_{RC}(M)$ has $m + n$ vertices and at most $nm + m(m-1)/2$ edges.

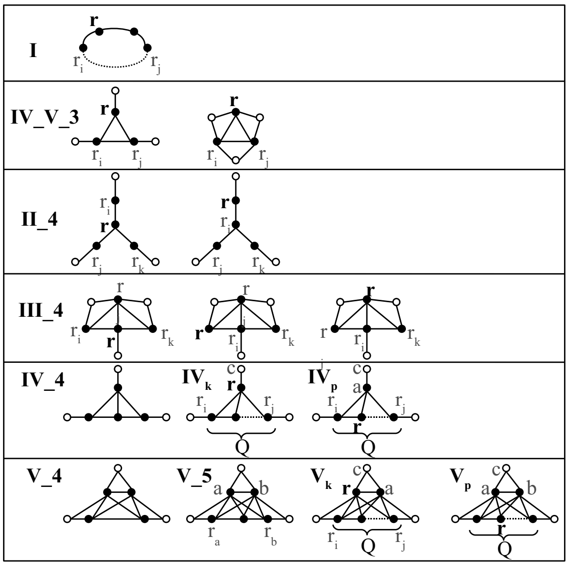


Fig. 3. The different steps of the algorithm: in each case, when row r has a specific location in the forbidden induced subgraph that is looked for, this location is indicated in bold character. Other rows and columns of the forbidden induced subgraph are indicated in grey color characters.

3 A Global Algorithm

Our algorithm to decide if a row $r \in \mathcal{R}$ of a 0-1 matrix $M = (\mathcal{R}, \mathcal{C})$ belongs to at least one MCS, is based on a sequence of algorithms for finding a forbidden subgraph of $G_{RC}(M)$ responsible for a MCS containing r. It looks for some MCS of the form I, III, II, IV, or V, containing r, in the following order:

1. MCS of type I, 2. MCS of size 3 (types IV or V), 3. MCS of type II, 4. MCS of type III, 5. MCS of type IV and size larger than 3, and MCS of type V and size larger than 3. See Figure 3 for an overview. The steps 2 to 4 are based on straightforward brute-force algorithms to detect forbidden induced subgraphs of $G_{RC}(M)$ containing at most 4 black vertices (rows) including r. For this reason, their detailed descriptions are not presented in this extended abstract. They can however be found in [9]. The steps 1, 5, and 6 rely on a reduction to the detection of induced chordless cycles in ad-hoc graphs. In the following, we simply write $G_{RC}(M)$ as G and $G_R(M)$ as G_R.

STEP 1: Forbidden induced subgraph I. We first test if r belongs to a MCS of the form I. The rows (black vertices) of such a MCS necessarily consitute an induced chordless cycle in G_R. Thus it suffices to search for an induced chordless cycle in G_R containing r (see Figure 3.I and Algorithm 1 ; a P_4 of a graph is an induced chordless path of the graph containing 4 vertices).

Algorithm 1.1. Check_I (r, G_R) – $O(m^5)$

Input: a row r, the subgraph G_R.
Output: returns a MCS $\mathcal{S}$ given by a forbidden induced subgraph of the form I containing r if such a MCS exists, otherwise returns "NO".

```
1: for any $P_4$ of $G_R$ containing $r$ do
2:   Consider the graph $G'$ obtained from $G_R$ after removing the two internal
     vertices of the $P_4$ and their neighborhood from the graph, and consider
     the extremities $r_i$ and $r_j$ of the $P_4$
3:   if there exists a $r_i r_j$-path in $G'$ then
4:     find a chordless path $P$ in this graph linking $r_i$ and $r_j$.
5:     return the set of vertices of the $P_4$ plus the set of vertices of $P$
6:   end if
7: end for
8: return "NO"
```

Proposition 1. *Algorithm Check_I is correct and worst case $O(m^5)$ time.*

Proof. The correctness of Algorithm Check_I comes from the fact that, r is contained in a MCS of the form I if and only if r belongs to an induced chordless cycle of G_R of length at least 4 whose set of vertices $\mathcal{S}$ constitutes the MCS (Figure 3.I). A P_4 of G_R is an induced chordless path of G_R containing 4 vertices. In this case, Algorithm Check_I returns such a set of vertices since an induced chordless cycle of G_R of length at least 4 containing r is a P_4 containing r whose extremities are linked by a chordless path in the subgraph of G that does not contain the neighborhood of the internal vertices of the P_4. This set $\mathcal{S}$ cannot contain a smaller subset of rows that is a MCS, as no subset of S can be a MCS of the form I, or a MCS of any other form because of Property 2.

Algorithm Check_I might be implemented in $O(m^5)$. The test performed on a given P_4 containing r (lines 2-5 of the algorithm) can be achieved in $O(m^2 + m \log m)$ as follows: removing the neighborhood of its internal vertices might be done in m^2 time, and finding a chordless path between the two extremities might be performed using Dijkstra's algorithm in $O(m^2 + m \log m)$ time. Enumerating all P_4 containing r might be done in time $O(m^3)$ using a BFS from r stopping at depth 4. Eventually, the whole algorithm is in $O(m^3(m^2 + m \log m)) = O(m^5)$ time. □

Precomputation. In the following steps, we assume that the following precomputations have been achieved: (a) for any triplet of rows (r, r_i, r_j) in G that are pairwise connected, $r - (r_i \cup r_j)$ and $(r_i \cap r_j) - r$ are precomputed ; (b) two rows r_i and r_j are *overlapping* if $r_i \cap r_j \neq \emptyset$ and $r_i - r_j \neq \emptyset$ and $r_j - r_i \neq \emptyset$. The overlapping relation between any couple of rows in G is precomputed ; (c) for any quadruplet of rows (r, r_i, r_j, r_k) in G such that r_i, r_j, and r_k overlap r, $r - (r_i \cap r_j \cap r_k)$ is precomputed. All those precomputations can simply be performed in $O(nm^4)$ time using straightforward algorithms, that are, scanning the n columns of the input matrix for each triplet or quadruplet of rows.

STEP 2: Forbidden induced subgraph responsible for a MCS of size 3. We test here if r belongs to a MCS of size 3. A MCS of size 3 is necessarily caused by a forbidden induced subgraph of the form IV or V. As a consequence, the following property is immediate.

Property 3. A MCS of size 3 is always composed of 3 rows that are pairwise overlapping.

Thus, in this step, it suffices to use a brute-force algorithm, Check_IV_V_3, running in $O(m^2)$ time to search for a triplet of rows in G including r, satisfying one the two configurations shown in Figure 3_IV_V_3. The pseudocode of Algorithm Check_IV_V_3 and its proof of correctness and complexity might be found in [9].

STEP 3: Forbidden induced subgraph II. We test here if r belongs to a MCS of the form II, with the assumption that r is not contained in a MCS of size 3. Note that such a MCS is of size 4. In this step, it suffices to use a brute-force algorithm, Check_II_4, running in $O(m^3)$ time to search for a quatruplet of rows in G including r, satisfying one the two configurations shown in Figure 3_II_4. The pseudocode of Algorithm Check_II_4 and its proof of correctness and complexity might be found in [9].

STEP 4: Forbidden induced subgraph III. We test here if r belongs to a MCS of the form III, with the assumption that r is not contained in a MCS of size 3. Note that such a MCS is of size 4. In this step, we use a brute-force algorithm, Check_III_4, running in $O(m^3)$ time to search for a quatruplet of rows in G including r, satisfying one the three configurations shown in Figure 3_III_4. The pseudocode of Algorithm Check_III_4 and its proof of correctness and complexity might be found in [9].

STEP 5: Forbidden induced subgraph IV

We test here if r belongs to a MCS of the form IV, with the assumption that r is contained, neither in a MCS of size 3, nor in a MCS of type I. Depending on whether the size of the MCS is 4 or larger than 4, we describe two algorithms.

MCS of Size 4. We first test if r belongs to a MCS of the form IV of size 4. In this step, it suffices to use a brute-force algorithm, Check_IV_4, running in $O(m^3)$ time to search for a quatruplet of rows in G including r, satisfying the configurations shown in Figure 3_IV_4. The pseudocode of Algorithm Check_IV_4 and its proof of correctness and complexity might be found in [9].

MCS of Size Larger than 4. We test here if r belongs to a MCS of the form IV of size larger than 4. A MCS of the form IV of size larger than 4 contains one and only one kernel. Depending on whether r is the kernel or not, we distinguish two cases here.

Case 1: If row r is the kernel of the MCS. Algorithm Check_IV$_k$ recovers a MCS $\mathcal{S}$ of the form IV of size larger than 4 containing r as a kernel,

Algorithm 1.2. Check_IV$_k$ (r, G) – $O(nm^2)$

Input: a row r, the row-column intersection graph G.
Assumption: r is not contained in a MCS of size 3.
Output: returns a MCS $\mathcal{S}$ of size larger that 4 given by a forbidden induced subgraph of the form IV whose kernel is r if such a MCS exists, otherwise returns "NO".

```
1: for any column $c \in r$ do
2:    $H = G[N(r) - L(c)]$
3:    for any connected component $C$ of $H$ do
4:       pick a a couple $(r_i, r_j)$ of black vertices in $C$ that satisfies 1) $r_i$ and $r_j$
         are not connected, and 2) $r_i, r_j$ overlap $r$.
5:       find a chordless path $P$ in $C$ linking $r_i$ and $r_j$
6:       pick the smallest subpath $Q$ of $P$ linking two vertices $r'_i$ and $r'_j$, such
         that the couple $(r'_i, r'_j)$ also satisfies 1) and 2)
7:       return $\{r\} \cup Q$
8:    end for
9: end for
10: return "NO"
```

with the assumption that r is not contained in a MCS of size 3 (Figure 3.IV$_k$). The principle of the algorithm relies in first choosing the column $c \in \mathcal{C}$, of the forbidden induced subgraph of type IV responsible for the MCS $\mathcal{S}$, such that c is contained in r, and in no other row of the MCS (see Figure 3.IV$_k$). Next, it considers the subgraph H of G induced by the set of black vertices (rows) that are neighbors of r, but do not contain the column c. We denote this subgraph by $H = G[N(r) - L(c)]$. Then, it looks for a set of rows Q, constituting a chordless path in H, such that $\{r\} \cup Q$ is a MCS of the form IV.

Proposition 2. *Algorithm Check_IV$_k$ is correct and runs in $O(nm^2)$ time.*

Proof. Note that, if the MCS exists, then all the rows belonging to the MCS, except r, belong to a same connected component of H. Thus, in each connected component of H, the algorithm looks for a chordless path Q linking two vertices r_i, r_j satisfying 1) r_i and r_j are not connected, and 2) r_i, r_j overlap r, and 3) Q does not contain any smaller subpath satisfying conditions 1) and 2). These conditions are necessary and sufficient for the set $\{r\} \cup Q$ to form the rows of a induced subgraph of the form IV. The set $\{r\} \cup Q$ cannot contain a subset that is a MCS as such a smaller MCS should be:

– either a MCS of size 3 including r, which impossible by assumption,
– or a MCS of type II or III necessarily including r as kernel,
– or a MCS of type IV and size larger than 3 having r as kernel.

The two last cases are also impossible, since Q would not have satisfy condition 3) in these cases.

Next, there might be n columns $c \in r$ and up to m^2 couples (r_i, r_j) of black vertices to test before finding a valid couple (r_i, r_j) satisfying the conditions in line 4 of the algorithm. Up to this point, the complexity is in $O(nm^2)$. Assume now that such a couple exist. Then finding a chordless path between r_i and r_j might be done by searching for a shortest path between r_i and r_j in the connected component C using Dijkstra's algorithm, which thus requires at worst $O(m^2 + m \log m)$ time. The path is of length at most m, and thus identifying r'_i and r'_j is bounded by testing each pair on this path in C, which requires at worst $O(m^2)$ time. Thus, in total, the algorithm is $O(nm^2)$ worst case time. □

Case 2: If row r is not the kernel of the MCS. Algorithm Check_IV$_p$ recovers a MCS $\mathcal{S}$ of the form IV of size larger than 4 containing r, such that r is not a kernel of the MCS, with the assumptions that r is not contained in a MCS of size 3, and r does not belong to an induced chordless cycle of G_R (Figure 4.IV$_p$). The principle of the algorithm consists in first choosing the kernel a of the MCS $\mathcal{S}$ among the black vertices (rows) neighbors of r, and the column $c \in \mathcal{C}$, of the induced subgraph of type IV responsible for the MCS $\mathcal{S}$, that is contained in a, but in no other row of the MCS (see Figure 3.IV$_p$). Next, the algorithm calls Algorithm Check_IV to look for the MCS $\mathcal{S}$ with r, a, c, and G given as parameters.

Algorithm 1.3. Check_IV$_p$ (r, G) – $O(nm^6)$

Input: a row r, the row-column intersection graph G.
Assumption: r is not contained in a MCS of size 3.
r does not belong to an induced chordless cycle of G_R.
Output: returns a MCS $\mathcal{S}$ of size larger that 4 given by a forbidden induced subgraph of the form IV containing r whose kernel is not r if such a MCS exists, otherwise returns "NO".

```
1: for any black vertex $a \in N(r)$ do
2:   for any column $c \in a - r$ do
3:     return Check_IV($r$, $a$, $c$, $G$)
4:   end for
5: end for
6: return "NO"
```

Algorithm Check_IV is called in Algorithm Check_IV$_p$. It recovers a MCS $\mathcal{S}$ of the form IV of size larger than 4 containing r, given the row r, the kernel a of the MCS $\mathcal{S}$, and the column $c \in \mathcal{C}$, of the induced subgraph of type IV responsible for $\mathcal{S}$, that is contained in a, but in no other row of the MCS (Figure 3.IV$_p$).

Proposition 3. *Algorithm Check_IV$_p$ is correct, and runs in $O(nm^6)$ time.*

Proof. The correctness and the complexity of Check_IV$_p$ follows directly from the correctness and the complexity of Algorithm Check_IV that is called in Algorithm Check_IV$_p$.

The correctness of Check_IV comes from the fact that, r does not belong to any chordless cycle in the graph C computed at line 2 of the algorithm by assumption. Then at line 6 of the algorithm, any chordless cycle in the graph D containing vertex r necessarily contains at least one edge (r_i, r_j) belonging to the set E_a.

Algorithm 1.4. Check_IV (r, a, c, G)– $O(m^5)$

Input: two rows r and a, and a column $c \in a$ such that $r \in (N(a) - L(c))$.
Assumption: r is not contained in a MCS of size 3.
r does not belong to an induced chordless cycle of G_R.
Output: returns a MCS $\mathcal{S}$ of size larger that 4 given by a forbidden induced subgraph of the form IV containing r and a, whose kernel is a if such a MCS exists, otherwise returns "NO".

```
 1: $H = G[N(a) - L(c)]$
 2: let $C = (V_C, E_C)$ be the connected component of $H$ to which $r$ belongs.
 3: let $V_a$ be the set of vertices $V_a = \{u \in V_C \ : \ u - a \neq \emptyset\}$.
 4: let $E_a$ be the set of edges $E_a = \{(u, v) \in V_a^2 \ : \ u \cap v = \emptyset\}$.
 5: let $D = (V_D, E_D)$ be the graph such that $V_D = V_C$ and $E_D = E_C \cup E_a$.
 6: $Q$ = Check_I ($r$, $D$)
 7: if $Q \neq$ "NO" then
 8:     return $\{a\} \cup Q$
 9: end if
10: return "NO"
```

The number of edges belonging to the set E_a in such a chordless cycle Q cannot be greater than 1 as any couple of such edges in the chordless cycle would induce a chord. Indeed, if Q contains more than one edge belonging to E_a, any two such edges would have two extremities in V_a, one from each of the two edges, that are not connected in the graph C. These extremities would thus be linked by an edge in E_a, creating a chord for the cycle Q in the graph D.

Therefore, the set of vertices of the chordless cycle Q induces a chordless path in G such that each vertex of Q is connected to vertex a by definition of the graph H, and the extremities r_i and r_j of Q satisfy 1) r_i and r_j are not connected in G, and 2) r_i, r_j overlap r, and 3) Q does not contain any smaller subpath satisfying conditions 1) and 2). These conditions are necessary and sufficient for the set $\{a\} \cup Q$ to form the rows of an induced subgraph of the form IV, and this set cannot contain a smaller MCS since such a MCS would be: (a) either a MCS of size 3 including a; (b) or a MCS of type II or III necessarily including a as kernel; (c) or a MCS of type IV and size larger than 3 having a as kernel.

The 3 cases are impossible, since they would induce a chord from the set E_a in the chordless cycle induced by Q in the graph D.

Algorithm Check_IV calls Algorithm Check_I. Both algorithms have the same time complexity in $O(m^5)$ time. It follows immediately that Algorithm Check_IV$_p$ runs in $O(nm^6)$ time. □

STEP 6: Forbidden induced subgraph V

We test here if r belongs to a MCS of the form V, with the assumption that r is contained neither in a MCS of size 3, nor in a MCS of type I. Depending on whether the size of the MCS is 4, 5 or larger than 5, we describe three algorithms.
MCS of size 4 or 5.

We first test if r belongs to a MCS of the form V of size 4 or 5.

In this step, it suffices to use two brute-force algorithms, Check_V_4 (resp. Check_V_5), running in $O(m^3)$ time (resp. $O(m^4)$ time) to search for a quatruplet (resp. a quintuplet) of rows in G including r, satisfying the configurations shown in Figure 3._V_4 (resp. Figure 3._V_5).

The pseudocode of Algorithm Check_V_4 and its proof of correctness and complexity might be found in [9].

Next, for a MCS of size 5, we look for a quadruplet of rows (r_i, r_j, r_k, r_l) such that the set $\{r, r_i, r_j, r_k, r_l\}$ is a MCS of the form V (Figure 3.V_5). Algorithm Check_V_5 looks for an induced subgraph of the form V, consisting of 5 rows (black vertices) r, r_i, r_j, r_k, r_l that are pairwise connected, except for a on missing edge, say (r_a, r_b) in $\{r, r_i, r_j, r_k, r_l\} \times \{r, r_i, r_j, r_k, r_l\}$, and three columns (white vertices) satisfying the configuration of Figure 3.V_5.

Algorithm 1.5. Check_V_5 $(r\ ,\ G)$ – $O(m^4)$

Input: a row r, the row-column intersection graph G.
Assumption: r is not contained in a MCS of size 3 or 4.
Output: returns a MCS $\mathcal{S}$ of size 5 given by a forbidden induced subgraph of the form V containing r if such a MCS exists, otherwise returns "NO".

```
1: for any quadruplet $(r_i, r_j, r_k, r_l)$ of black vertices such that $r, r_i, r_j, r_k, r_l$
   are pairwise connected, except for one edge $(r_a, r_b)$ in $\{r, r_i, r_j, r_k, r_l\} \times$
   $\{r, r_i, r_j, r_k, r_l\}$ missing do
2:   if $\{r_i, r_j, r_k, r_l\}$ is C1P then
3:     for any pair $(a, b)$ in $(\{r, r_i, r_j, r_k, r_l\} - \{r_a, r_b\}) \times (\{r, r_i, r_j, r_k, r_l\} -$
       $\{r_a, r_b\})$ do
4:       if $(a \cap b) - \cup(\{r, r_i, r_j, r_k, r_l\} - \{a, b\}) \neq \emptyset$, and $(r_k \cap a) -$
         $\cup(\{r, r_i, r_j, r_k, r_l\} - \{r_k, a\}) \neq \emptyset$, and $(r_l \cap b) - \cup(\{r, r_i, r_j, r_k, r_l\} -$
         $\{r_l, b\}) \neq \emptyset$ then
5:         return $\{r, r_i, r_j, r_k, r_l\}$
6:       end if
7:     end for
8:   end if
9: end for
10: return "NO"
```

Proposition 4. *Algorithm Check_V_5 is correct and runs in* $O(m^4)$ *time.*

Proof. Algorithm Check_V_5 looks for an induced subgraph with 5 black vertices $\{r, r_i, r_j, r_k, r_l\}$, that are pairwise connected, except for one missing edge (r_a, r_b) in $\{r, r_i, r_j, r_k, r_l\} \times \{r, r_i, r_j, r_k, r_l\}$. The 4 black vertices that belong to the set with r, should correspond to a set of rows that is C1P. Moreover, there should exist two particular rows (black vertices) of the set, with three columns (white vertices) that satisfy the conditions on line 4 of the algorithm in order to fit the configuration depicted in Figure 3.V_5.

Next, all the tests performed by Algorithm Check_V_5 (lines 2-8 of the algoritm) on a given quatruplet (r_i, r_j, r_k, r_l) are achieved in $O(1)$ thanks to the precomputations, and given r there might be $O(m^4)$ such triplets. Thus, Algorithm Check_V_5 runs in $O(m^4)$ time. □

MCS of size larger than 5**.**

A MCS of the form V of size larger than 5 contains exactly two kernels. Depending on whether r is a kernel or not, we distinguish two cases.

Case 1: If row r **is a kernel of the MCS.** Algorithm Check_V$_k$ recovers a MCS $\mathcal{S}$ of the form V of size larger than 5 containing r as a kernel, with the assumption that r is not contained in a MCS of size 3, or 4 (Figure 3.V$_k$). The principle of the algorithm is similar to Algorithm Check_IV$_k$. It relies in first choosing the second kernel a of the MCS, and the column c, of the induced subgraph of type V responsible for the MCS $\mathcal{S}$, such that c is contained in both r and a, but in no other row of the MCS (see Figure 3.V$_k$). Next, it considers the subgraph H of G induced by the set of black vertices (rows) that are neighbors of both r and a, but do not contain c. We denote this subgraph by $H = G[N(r,a) - L(c)]$. Then, it looks for a set of rows Q, constituting a chordless path in H, such that $\{r\} \cup Q$ is a MCS of the form V.

Algorithm 1.6. Check_V$_k$ (r, G) – $O(n^2m^2)$

Input: a row r, the row-column intersection graph G.
Assumption: r is not contained in a MCS of size 3, or 4.
Output: returns a MCS $\mathcal{S}$ of size larger that 5 given by a forbidden induced subgraph of the form V such that r is one of its kernel, if such a MCS exists, otherwise returns "NO".

```
1: for any black vertex $a \in N(r)$ do
2:    for any column $c \in (r \cap a)$ do
3:       $H = G[N(r,a) - L(c)]$
4:       for any connected component $C$ of $H$ do
5:          pick a a couple $(r_i, r_j)$ of black vertices in $C$ that satisfies 1) $r_i$ and
            $r_j$ are not connected, and 2) $(r_i \cap r) - a \neq \emptyset$, and 3) $(r_j \cap a) - r \neq \emptyset$.
6:          find a chordless path $P$ in $C$ linking $r_i$ and $r_j$
7:          pick the smallest subpath $Q$ of $P$ linking two vertices $r'_i$ and $r'_j$, such
            that the couple $(r'_i, r'_j)$ also satisfies 1) and 2) and 3)
8:          return $\{r\} \cup Q$
9:       end for
10:   end for
11: end for
12: return "NO"
```

Proposition 5. *Algorithm Check_V$_k$ is correct and runs in $O(n^2m^2)$ time.*

Proof. The proofs are similar to the proofs for the correctness and the complexity of Algorithm Check_IV$_k$ as the two algorithms are based on the same principle. However, here the complexity is multiplied by a factor n due to considering all black vertices $a \in N(r)$. □

Case 2: If row r **is not a kernel of the MCS.** Algorithm Check_V$_p$ recovers a MCS S of the form V of size larger than 5 containing r, such that r is not a kernel of the MCS, with the assumptions that r is not contained in a

MCS of size 3 or 4, and r does not belong to an induced chordless cycle of G_R (Figure 3.V).

The principle of the algorithm is similar to the principle of Algorithm Check_IV$_p$. It consists in first choosing the two kernels (a, b) of the MCS S among the black vertices (rows) neighbors of r, and the column c, of the induced subgraph responsible for S, that is contained in both a and b, but in no other row of the MCS. Next, the algorithm calls Algorithm Check_V to look for the MCS S with r, (a, b), c, and G given as parameters.

Proposition 6. *Algorithm Check_V$_p$ is correct and runs in $O(nm^7)$ time.*

Proof. In order to prove the correctness and the complexity of Algorithm Check_V$_p$, we need to prove the correctness and give the complexity of Algorithm Check_V that is called in Check_V$_p$.

The correctness of Check_V comes from the fact that r does not belong to any chordless cycle in the graph C computed at line 2 of the algorithm by assumption. Let Q be a chordless cycle in the graph D containing vertex r, computed at line 9 of the algorithm. Since r does not belong to an induced chordless cycle of the C by assumption, then Q necessarily contains at least one edge belonging to the set $E_{AB} \cup E_a \cup E_b$. We first give two trivial but useful properties for the remaining of the proof: (i) for any two edges of Q, there always exist two extremities u and v of these edges, one from each edge, that are not connected by an edge in the graph C, i.e $u \cap v = \emptyset$; (ii) $V_A \subseteq V_b$, and $V_B \subseteq V_a$. We also prove the following useful property: (iii) $V_a \subseteq (V_B \cup V_b)$ and $V_b \subseteq (V_A \cup V_a)$. Let $x \in V_a$, there exists c such that $c \in x$ and $c \notin a$. Then, either $c \notin b$ in which case $x \in V_b$, or $c \in b$, which implies that $x \in V_B$. The proof is similar for $V_b \subseteq (V_A \cup V_a)$.

We now prove that the cycle Q necessarily contains at most one edge of the set $E_{AB} \cup E_a \cup E_b$. Indeed, let us suppose that Q contains two edges of $E_{AB} \cup E_a \cup E_b$. Then, let u, v be two disconnected extremities of these edges in the graph C (Property (i)). We show that (u, v) necessarily constitutes an edge that is a chord for the chordless cycle Q in the graph D : contradiction. (1) If $(u, v) \in V_A^2$ (resp. $(u, v) \in V_B^2$), then from Property (ii), $(u, v) \in V_b^2$ (resp. $(u, v) \in V_a^2$), and thus $(u, v) \in E_b$ (resp. $(u, v) \in E_a$); (2) If $(u, v) \in V_a^2$ (resp.

Algorithm 1.7. Check_V$_p$ (r, G) – $O(nm^7)$

Input: a row r, the row-column intersection graph G.
Assumption: r is not contained in a MCS of size 3, 4.
r does not belong to an induced chordless cycle of G_R.
Output: returns a MCS $\mathcal{S}$ of size larger that 5 given by a forbidden induced subgraph of the form V containing r, but not as a kernel, if such a MCS exists, otherwise returns "NO".

```
1: for any couple of connected black vertices $(a, b) \in N(r)^2$ do
2:    for any column $c \in (a \cap b) - r$ do
3:       return Check_V $(r, (a, b), c, G)$
4:    end for
5: end for
6: return "NO"
```

Algorithm 1.8. Check_V (r, (a,b), c, G)– $O(m^5)$

Input: three rows r, a and b, and a column $c \in a \cap b$ such that $r \in (N(a,b) - L(c))$.
Assumption: r is not contained in a MCS of size 3, 4 or 5.
r is not contained in a MCS of type IV.
r does not belong to an induced chordless cycle of G_R.
Output: returns a MCS $\mathcal{S}$ of size larger that 5 given by a forbidden induced subgraph of the form V containing a, b, and r, and whose kernels are a and b, if such a MCS exists, otherwise returns "NO".

```
1: $H = G[N(a,b) - L(c)]$
2: let $C = (V_C, E_C)$ be the connected component of $H$ to which $r$ belongs.
3: let $V_A$ be the set of vertices $V_A = \{u \in V_C \ : \ (u \cap a) - b \neq \emptyset\}$.
4: let $V_B$ be the set of vertices $V_B = \{v \in V_C \ : \ (v \cap b) - a \neq \emptyset\}$.
5: let $E_{AB}$ be the set of edges $E_{AB} = \{(u,v), u \in V_A, v \in V_B \ : \ u \cap v = \emptyset\}$.
6: let $V_a$ be the set of vertices $V_a = \{u \in V_C \ : \ u - a \neq \emptyset\}$, and $E_a$ be the set
   of edges $E_a = \{(u,v) \in V_a^2 \ : \ u \cap v = \emptyset\}$.
7: let $V_b$ be the set of vertices $V_b = \{u \in V_C \ : \ u - b \neq \emptyset\}$, and $E_b$ be the set
   of edges $E_b = \{(u,v) \in V_b^2 \ : \ u \cap v = \emptyset\}$.
8: let $D = (V_D, E_D)$ be the graph such that $V_D = V_C$ and $E_D = E_C \cup E_{AB} \cup$
   $E_a \cup E_b$
9: $Q$ = Check_I ($r$, $D$)
10: **if** $Q \neq$ "NO" **then**
11:    return $\{a,b\} \cup Q$
12: **end if**
13: return "NO"
```

$(u,v) \in V_b^2$), then $(u,v) \in E_a$ (resp. $(u,v) \in E_b$); (3) If $(u,v) \in V_A \times V_B$ (or the symmetric), then $(u,v) \in E_{AB}$. (4) If $(u,v) \in V_A \times V_a$ (or the symmetric), then from Property (iii); $(u,v) \in V_A \times V_B$ or $(u,v) \in V_A \times V_b$, and thus $(u,v) \in E_{AB}$ or $(u,v) \in E_b$ from cases 3. and 6; (5) If $(u,v) \in V_B \times V_b$ (or the symmetric), then from Property (iii), $(u,v) \in V_B \times V_A$ or $(u,v) \in V_B \times V_a$, and thus $(u,v) \in E_{AB}$ or $(u,v) \in E_a$ from cases 3. and 6; (6) If $(u,v) \in V_A \times V_b$ (resp. $(u,v) \in V_B \times V_a$) (or the symmetric), then from Property (ii), $(u,v) \in V_b^2$ (resp. $(u,v) \in V_a^2$), and thus $(u,v) \in E_b$ (resp. $(u,v) \in E_a$); (7) If $(u,v) \in V_a \times V_b$ (or the symmetric), then from Property (iii), $(u,v) \in V_b \times V_b$ or $(u,v) \in V_B \times V_b$, and thus $(u,v) \in E_b$ or $(u,v) \in E_{AB} \cup E_a$ from cases 1 and 5.

In consequence, there exits at most one edge, and then exactly one edge of the set $E_{AB} \cup E_a \cup E_b$ in the cycle Q in the graph D. Next, let (r_i, r_j) be the only edge of Q belonging to $E_{AB} \cup E_a \cup E_b$. We show that $(r_i, r_j) \notin E_a \cup E_b$. Indeed, if $(r_i, r_j) \in E_a$ (resp. $(r_i, r_j) \in E_b$), then the set $\{a\} \cup Q$ (resp. $\{b\} \cup Q$) satisfies the conditions to be a MCS of type IV with a (resp. b) as kernel, which is impossible by assumption.

So, we have $(r_i, r_j) \in E_{AB} - (E_a \cup E_b)$. Finally, removing the edge (r_i, r_j) from the cycle yields a chordless path Q in G containing r such that each vertex of Q is connected to vertices a and b, and the extremities r_i and r_j of Q satisfy 1) r_i and r_j are not connected, and 2) $(r_i \cap a) - b \neq \emptyset$, and 3) $(r_j \cap b) - a \neq \emptyset$. and 4) Q does not contain any smaller subpath satisfying conditions 1) and 2) and 3). These conditions are necessary and sufficient for the set $\{a,b\} \cup Q$ to form the rows of an induced subgraph of the form V, and this set cannot contain a smaller MCS since such a MCS would be: either a MCS of size 3 including a or b, or a MCS of type II or III necessarily including a or b as kernel, or a MCS of type IV and size larger than 3 having a or b as kernel, or a MCS of type V and size larger than 3 having a and b as kernels. These 3 last cases are impossible, since they would induce a chord from the set $E_{AB} \cup E_a \cup E_b$ in the chordless cycle induced by Q in the graph D.

The correctness of Algorithm Check_V$_p$ follows immediately from the correctness of Algorithm Check_V. Algorithm Check_V calls Algorithm Check_I. Both algorithms have the same time complexity in $O(m^5)$ time. It follows immediately that Algorithm Check_IV$_p$ runs in $O(nm^7)$ time. □

4 Conclusion

We describe a $O(n^2m^2 + nm^7)$ time algorithm for deciding if a given row r of a $m \times n$ 0-1 matrix M belongs to at least one MCS.

The algorithm consists in a precomputation phase that runs in $O(nm^4)$, followed by 6 consecutive steps in which we look for some MCS containing r, and constituting the set of rows of a forbidden induced subgraph in the row-column intersection graph of M. The core of the algorithm relies in reducing the recognition of complex forbidden induced subgraphs to the detection of induced cycles in ad-hoc graphs.

Steps	Algorithms	Comp.
STEP 1	Check_I	$O(m^5)$
STEP 2	Check_IV_V_3	$O(m^2)$
STEP 3	Check_II_4	$O(m^3)$
STEP 4	Check_III_4	$O(m^3)$
STEP 5	Check_IV_4	$O(m^3)$
	Check_IV$_k$	$O(nm^2)$
	Check_IV$_p$	$O(nm^6)$
STEP 6	Check_V_4	$O(m^3)$
	Check_V_5	$O(m^4)$
	Check_V$_k$	$O(n^2m^2)$
	Check_V$_p$	$O(nm^7)$

Acknowledgment. This work is partly supported by the french MAPPI project (ANR-2010-COSI-004). We would like to thanks Nicolas Trotignon for his valuable comments on induced subgraphs and also Juraj Stacho for his participation to some meeting on the subject.

References

1. Bergeron, A., Blanchette, M., Chateau, A., Chauve, C.: Reconstructing Ancestral Gene Orders Using Conserved Intervals. In: Jonassen, I., Kim, J. (eds.) WABI 2004. LNCS (LNBI), vol. 3240, pp. 14–25. Springer, Heidelberg (2004)
2. Blin, G., Rizzi, R., Vialette, S.: A Faster Algorithm for Finding Minimum Tucker Submatrices. In: Ferreira, F., Löwe, B., Mayordomo, E., Mendes Gomes, L. (eds.) CiE 2010. LNCS, vol. 6158, pp. 69–77. Springer, Heidelberg (2010)

3. Blin, G., Rizzi, R., Vialette, S.: A Polynomial-Time Algorithm for Finding a Minimal Conflicting Set Containing a Given Row. In: Kulikov, A., Vereshchagin, N. (eds.) CSR 2011. LNCS, vol. 6651, pp. 373–384. Springer, Heidelberg (2011)
4. Chauve, C., Haus, U.-U., Stephen, T., You, V.P.: Minimal Conflicting Sets for the Consecutive Ones Property in Ancestral Genome Reconstruction. In: Ciccarelli, F.D., Miklós, I. (eds.) RECOMB-CG 2009. LNCS, vol. 5817, pp. 48–58. Springer, Heidelberg (2009)
5. Chauve, C., Tannier, E.: A methodological framework for the reconstruction of contiguous regions of ancestral genomes and its application to mammalian genomes. PLoS Comput. Biol. 4(11), 11 (2008)
6. Dom, M.: Algorithmic aspects of the consecutive-ones property. Bulletin of the Eur. Assoc. for Theor. Comp. Science (EATCS) 98, 27–59 (2009)
7. Lekkerkerker, C.G., Boland, J.C.: Representation of a finite graph by a set of intervals on the real line. Fund. Math. 51, 45–64 (1962)
8. Nishizeki, T., Rahman, M.S.: Planar Graph Drawing. World Scientific (2004)
9. Ouangraoua, A., Raffinot, M.: Faster and Simpler Minimal Conflicting Set Identification. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp.41–55. Springer, Heidelberg (2012)
10. Stoye, J., Wittler, R.: A unified approach for reconstructing ancient gene clusters. IEEE/ACM Trans. Comput. Biol. Bioinf. 6(3), 387–400 (2009)
11. Tucker, A.C.: A structure theorem for the consecutive 1s property. Journal of Combinatorial Theory. Series B 12, 153–162 (1972)

Partitioning into Colorful Components by Minimum Edge Deletions

Sharon Bruckner[1,⋆], Falk Hüffner[2,⋆⋆], Christian Komusiewicz[2], Rolf Niedermeier[2], Sven Thiel[3], and Johannes Uhlmann[2,⋆⋆⋆]

[1] Institut für Mathematik, Freie Universität Berlin
`sharonb@mi.fu-berlin.de`
[2] Institut für Softwaretechnik und Theoretische Informatik, TU Berlin
{`falk.hueffner`,`christian.komusiewicz`,`rolf.niedermeier`}`@tu-berlin.de`,
`johannes.uhlmann@campus.tu-berlin.de`
[3] Institut für Informatik, Friedrich-Schiller-Universität Jena
`sven.thiel@uni-jena.de`

Abstract. The NP-hard Colorful Components problem is, given a vertex-colored graph, to delete a minimum number of edges such that no connected component contains two vertices of the same color. It has applications in multiple sequence alignment and in multiple network alignment where the colors correspond to species. We initiate a systematic complexity-theoretic study of Colorful Components by presenting NP-hardness as well as fixed-parameter tractability results for different variants of Colorful Components. We also perform experiments with our algorithms and additionally develop an efficient and very accurate heuristic algorithm clearly outperforming a previous min-cut-based heuristic on multiple sequence alignment data.

1 Introduction

We study a maximum parsimony approach to the discovery of heterogeneous components in vertex-colored graphs:

Colorful Components
Instance: An undirected graph $G = (V, E)$ and a coloring of the vertices $\chi : V \to \{1, \ldots, c\}$.
Task: Find a minimum subset of edges $E' \subseteq E$ such that in $G' = (V, E \setminus E')$, all connected components are *colorful*, that is, they do not contain two vertices of the same color.

Such an edge set E' is called a *solution*, and we denote its size by k. Colorful Components is an edge modification problem originating from biological applications in sequence and network alignment as described next.

⋆ Supported by project NANOPOLY (PITN-GA-2009–238700).
⋆⋆ Supported by DFG project PABI (NI 369/7-2).
⋆⋆⋆ Supported by DFG project PABI (NI 369/7-2).

The first application of COLORFUL COMPONENTS stems from *Multiple Sequence Alignment*. This is the process of aligning at least three protein, DNA, or RNA sequences such that positions believed to be homologous, that is, resulting from inheritance from a common ancestor, are written in a common column. This serves to illustrate the similarity or dissimilarity between the sequences and makes it possible to investigate their evolutionary relationship. [6] present an algorithm for this problem where a central step is to find connected subgraphs in graphs whose vertices are positions of the sequences, edges indicate that a pair of positions should be aligned, and the colors one-to-one correspond to sequences. These subgraphs correspond to partial alignment columns and thus may contain at most one vertex from each input sequence. This yields the COLORFUL COMPONENTS problem. The solution of COLORFUL COMPONENTS is then used by the DIALIGN software to compute a multiple alignment. [6] solve COLORFUL COMPONENTS using a greedy algorithm, subsequently called "min-cut heuristic": Find two vertices of the same color in some connected component, find a minimum edge cut between them, and remove it; repeat this until all connected components are colorful.

A second biological motivation for COLORFUL COMPONENTS arises in *Network Alignment* for multiple protein–protein interaction (PPI) networks. We propose a method for network alignment that is based on solving COLORFUL COMPONENTS. Given networks $G_i = (V_i, E_i)$ and a similarity relation S between the proteins of different species, first create a network whose vertex set is $\bigcup V_i$ and in which vertex $v \in V_i$ receives color i. Then, add an edge $\{u, v\}$ if uSv. The detected colorful components are then sets of matched proteins. Every protein appears in exactly one component, and every component has at most one protein from each species, which is a very strict model. The results can then be viewed as functional orthologs [14], or they can form the basis for further analysis. [7] suggest a three-step framework for network alignment where the first step is to aggregate the proteins from the different species into subsets, and COLORFUL COMPONENTS offers a way of performing this task that results in consistent, disjoint aggregated groups.

Related combinatorial problems. COLORFUL COMPONENTS can be seen as the problem of destroying by edge deletions all *bad paths*, that is, simple paths between two vertices of the same color. Thus, it is a special case of the well-known NP-hard MULTICUT problem, which has as input an undirected graph and a set of vertex pairs and asks for a minimum number of edges to delete to disconnect each given vertex pair. MULTICUT is fixed-parameter tractable with respect to the number k of edge deletions, with a running time of $2^{O(k^3)} \cdot |V|^{O(1)}$ [4, 13].

COLORFUL COMPONENTS is also a special case of MULTI-MULTIWAY CUT [1]. This problem asks to disconnect by edge deletions all paths between vertices from the same vertex set of some given vertex sets. Thus, COLORFUL COMPONENTS is the special case where the vertex sets form a partition. Finally, there is related work on "clustering with diversity" [11] which extends a traditional clustering problem by asking that in each resulting cluster all points of the underlying colored metric space must have different colors.

Contributions. On the theoretical side, we present a first systematic study on the computational complexity of Colorful Components, exhibiting both tractable and intractable cases. First, we observe that Colorful Components is NP-hard even in trees. Then, we present a complexity dichotomy concerning the number c of colors showing that Colorful Components is polynomial-time solvable for two or less colors and NP-hard otherwise. For three or more colors, we also obtain super-polynomial running time lower bounds (based on the Exponential Time Hypothesis) even in the case that the input graph has bounded degree. On the positive side, we present fixed-parameter algorithms with running time $2^c \cdot |V|^{O(1)}$ for Colorful Components in trees and with running time $O((c-1)^k \cdot |E|)$ in general graphs. In experimental work we demonstrate that, somewhat surprisingly, we can get better results by solving the more general Weighted Multi-Multiway Cut problem, since this allows us to merge vertices. We take advantage of this in data reduction rules, a simplified branching, and a new heuristic. With the branching algorithms, we can solve to optimality more than half of the instances generated from the BAliBASE 3.0 benchmark [15] each time within five minutes on a standard PC, with up to 5 000 vertices and 13 000 edges. Our heuristic has an average error of 0.6 %, a large improvement over the 29.2 % of the previously suggested min-cut heuristic [6]. We also show the strength of the developed data reduction rules.

Preliminaries. We consider only undirected and simple graphs $G = (V, E)$ where $n := |V|$ and $m := |E|$. We assume that $n = O(m)$ since isolated vertices can be removed from the input in linear time. A *bad path* is a simple (that is, cycle-free) path between two vertices of the same color. The length of a path is the number of its edges. An *edge cut* is a set of edges whose deletion increases the number of connected components. For a nonnegative number t, a graph is *t-edge connected* if it does not have an edge cut of size less than t.

The Exponential Time Hypothesis (ETH) states that, for all $x \geq 3$, x-SAT, which asks whether a boolean input formula in conjunctive normal form with n variables and m clauses and at most x variables per clause is satisfiable, cannot be solved within a running time of $2^{o(n)}$ or $2^{o(m)}$; see [12] for a recent survey. A problem is *fixed-parameter tractable* with respect to a parameter k if it can be solved in $f(k) \cdot n^{O(1)}$ time for an arbitrary (typically exponential) function f in k.

2 Computational Hardness

In this section, we present hardness results for two restricted variants of Colorful Components.

First, we consider the special case where the input graph is a tree. For obtaining our hardness result, we exploit the connection between Colorful Components and Multicut. Note that Multicut is NP-hard and MaxSNP-hard even if the input is a star, that is, a tree consisting of a central vertex with attached degree-1 vertices [8]. Multicut in stars can be reduced to Colorful

Components as follows: for every pair $\{s, t\}$ to be disconnected, create degree-1 vertices s' and t' attached to s and t, respectively, and color s' and t' with the same unique color. Each original vertex gets a further unique color. Since this reduction produces trees whose diameter is four, we arrive at the following.

Proposition 1. Colorful Components *is NP-hard even in trees with diameter four.*

In stars, however, Colorful Components turns out to be polynomial-time solvable: If the central vertex v has two neighbors with the same color, one can delete the edge between v and one of the two identically colored degree-one vertices. If v has no two neighbors of the same color, then every connected component is colorful.

Second, we study the computational complexity of Colorful Components if the number of colors is fixed. This is of interest since the number of colors may be small in practical cases.

Theorem 1. Colorful Components *with three colors in graphs with maximum degree six is NP-hard; it cannot be solved in $2^{o(k)} \cdot n^{O(1)}$, $2^{o(n)} \cdot n^{O(1)}$, or $2^{o(m)} \cdot n^{O(1)}$ time unless the ETH is false.*

Proof. We present a polynomial-time many-to-one reduction from the NP-hard 3-SAT problem which has as input a Boolean formula ϕ in 3-CNF.[1] For simplicity, we assume that every clause contains *exactly* three literals.

The basic idea of the reduction is as follows. For each variable x_i of a given 3-CNF formula ϕ, we construct a *variable cycle* of length $4m_i$, where m_i denotes the number of clauses that contain x_i. These cycles are colored alternatingly with two colors c_e and c_o such that deleting every second edge yields a minimum-cardinality edge deletion set for obtaining colorful components for this cycle. The corresponding two possibilities are used to represent the two choices for the value of x_i. Then, for each clause C_j of ϕ containing the variables x_p, x_q, and x_r, we connect the three corresponding variable cycles by a clause gadget. This gadget has the property that if the solutions for the variable gadgets correspond to an assignment that satisfies C_j, then one needs only four edge deletions for the clause gadget. Conversely, if four edge deletions are sufficient, then the assignment corresponding to the deletions in the variable cycle satisfies C_j. Let m be the number of clauses in ϕ and observe that, since ϕ is a 3-CNF formula, the overall number of vertices in the variable cycles is $12m$. Our construction guarantees that there is a satisfying assignment for ϕ if and only if the constructed graph can be transformed into one with colorful components by exactly $6m + 4m = 10m$ edge deletions, where $6m$ edge deletions are used for the variable cycles and $4m$ modifications are used for the clause gadgets. The details follow.

Given a 3-CNF formula ϕ consisting of the clauses $C_0, \ldots, C_{m-1}$ over the variables $\{x_0, \ldots, x_{n-1}\}$, construct a Colorful Components-instance $(G =$

[1] A similar reduction type was previously used to show analogous results for Transitivity Editing [16] and Cluster Editing [10], which, in contrast, are defined on uncolored graphs.

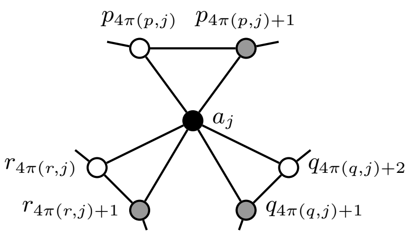


Fig. 1. The clause gadget for clause $C_j = (x_p \vee \bar{x}_q \vee x_r)$. White vertices have color c_e, gray vertices have color c_o, and black vertices have color c_g. The vertex a_j is the reserved vertex for C_j, the other vertices lie on the variable cycles for x_p, x_q, and x_r, respectively.

$(V, E), k)$ as follows. For each variable x_i, $0 \leq i < n$, G contains a *variable cycle* consisting of the vertices $V_i^v := \{i_0, \ldots, i_{4m_i-1}\}$ and the edges $E_i^v := \{\{i_k, i_{k+1}\} \mid 0 \leq k < 4m_i\}$ (for ease of presentation let $i_{4m_i} = i_0$). An edge $\{i_x, i_{x+1}\}$ is *even* if x is even, and *odd* otherwise. A vertex i_x receives color c_e if x is even; otherwise, it receives color c_o. So far, the constructed graph consists of a disjoint union of cycles and has $12m$ vertices and edges.

Next, add a *clause gadget* to G for each clause of ϕ. In the construction of the clause gadgets, we need for each clause C_j in the variable cycles of C_j's variables a fixed set of vertices that are "reserved" for C_j. To this end, suppose that for each variable x_i an arbitrary but fixed ordering of the clauses containing x_i is given, and let $\pi(i, j) \in \{0, \ldots, 4m_i - 1\}$ denote the position of a clause C_j that contains x_i in this ordering. We now give the details of the construction of the clause gadgets. Let C_j be a clause containing the variables x_p, x_q, and x_r (either negated or nonnegated). We construct a clause gadget connecting the variable cycles of x_p, x_q, and x_r. First, let a_j be a new vertex that appears only in the clause gadget for clause C_j and color a_j with a third color c_g. Let E_j^g denote the edge set of the clause gadget and let E_j^g contain for each $i \in \{p, q, r\}$ the edges $\{a_j, i_{4\pi(i,j)}\}$ and $\{a_j, i_{4\pi(i,j)+1}\}$ if x_i occurs nonnegated in C_j or the edges $\{a_j, i_{4\pi(i,j)+1}\}$ and $\{a_j, i_{4\pi(i,j)+2}\}$, otherwise. See Fig. 1 for an illustration. The construction of $G = (V, E)$ is completed by setting $V := \bigcup_{i=0}^{n-1} V_i^v \cup \bigcup_{j=0}^{m-1} \{a_j\}$ and $E := \bigcup_{i=0}^{n-1} E_i^v \cup \bigcup_{j=0}^{m-1} E_j^g$.

We show the correctness of the reduction by showing the following claim.

> ϕ is satisfiable $\Leftrightarrow$ G can be transformed into a graph with colorful components by deleting at most $k := 10m$ edges.

"$\Rightarrow$": Given a satisfying assignment β for ϕ, we can transform G into a graph with colorful connected components as follows. For each variable x_i delete the odd edges of the variable cycle of x_i if $\beta(x_i) = \text{true}$ and the even edges otherwise. After these deletions, there are no bad paths that contain only vertices from the variable cycles. Then, proceed as follows for each clause C_j. Assume without loss of generality that C_j contains the variables x_p, x_q, and x_r, and that the literal corresponding to x_p is true. Then, delete the four edges that are incident with a_j and with one vertex of the variable cycles of x_q and x_r. After the deletion of these four edges, there are no bad paths that contain a_j, which can be seen as follows. Clearly, a_j is only adjacent to two vertices on the variable cycle of x_p. Since the literal corresponding to x_p in C_j is true, the edge between these two vertices corresponds to the truth assignment that makes this literal true. Consequently, this edge is *not* deleted and the edges of the variable cycle that are before and after this edge are deleted. Hence, a_j and its two neighbors in the variable cycle of x_p form an isolated triangle with three different colors.

Summarizing, this means that after deleting the four edges as described above for each clause gadget, all bad paths containing some a_j have been destroyed. The overall number of edge deletions is $10m$: For the variable cycles, we perform altogether $\sum_{0 \leq i < n} 4m_i/2 = 6m$ edge deletions, and for each clause gadget four edges are deleted.

"$\Leftarrow$": Let S denote an optimal solution for G with $|S| \leq k := 10m$. To show that ϕ is satisfiable, we make some observations about the structure of G and S.

First, we show that $10m$ is a lower bound on any solution for G, that is, $|S| \geq 10m$ and thus $|S| = 10m$. First, note that for each variable x_i the variable cycle contains $4m_i/2$ edge-disjoint bad paths. Hence, G contains overall $6m$ edge-disjoint induced bad paths such that all vertices of the bad paths are in the variable cycles. Clearly, at least $6m$ edge deletions are needed for these bad paths. For each clause C_j, $0 \leq j < m$, at least four edges incident with a_j have to be deleted since a_j has degree six and can have degree at most two in a colorful component. Hence, every solution has size at least $6m + \sum_{0 \leq j < m} 4 = 10m$ and thus $|S| = 10m$.

Now, since at least $6m$ edges are deleted in the variable cycles, this means that for each clause C_j *exactly* four edges incident with a_j are deleted by S. Consequently, for each variable cycle either all even or all odd edges are deleted.

Consider the assignment β for ϕ that, for each x_i, $0 \leq i < n$, sets $\beta(x_i) :=$ true if all odd edges of V_i^v are deleted and sets $\beta(x_i) :=$ false if all even edges of V_i^v are deleted. We show that β is a satisfying assignment. Consider an arbitrary clause C_j containing the variables x_p, x_q, and x_r. Since a_j is in a colorful component after the edge deletions, it can have at most two neighbors. Furthermore, these neighbors must be on the same variable cycle: otherwise, a_j would be in a connected component of size five, because after the edge deletions, every vertex is adjacent to exactly one further vertex on its variable cycle. Hence, after the edge deletions, a_j is adjacent to at most two vertices of the variable cycle of one of x_p, x_q, and x_r of C_j. Let x_p be this variable. Furthermore, since exactly four edge deletions are incident with a_j, *both* edges that are incident with the vertices of the variable cycle of x_p are not deleted by S. Without loss of generality, assume that x_p appears nonnegated in C_j. Then the two vertices of V_p^v that are adjacent to a_j are $p_{4\pi(p,j)}$ and $p_{4\pi(p,j)+1}$. Since S is a solution, the edge $\{p_{4\pi(p,j)}, p_{4\pi(p,j)-1}\}$ is not deleted by S: otherwise, a_j is in a connected component of size five. Hence, all odd edges of V_p^v are deleted, and therefore the assignment β fulfills clause C_j.

Altogether, this shows the correctness of the reduction. Since the reduction can be performed in polynomial time and produces a graph with maximum degree six, it implies NP-hardness in graphs with maximum degree six. Furthermore, for formulas with m clauses, the reduction produces graphs with $|V| < 13m$, $k = 10m$ and $|E| = 18m$. Hence, any algorithm with running time $2^{o(k)} \cdot n^{O(1)}$, $2^{o(|V|)}$, or $2^{o(|E|)}$ implies an algorithm with running time $2^{o(m)}$ for 3-SAT, contradicting the ETH. □

3 Algorithms

Two colors. While Theorem 1 shows that Colorful Components is NP-hard for three colors, for two colors it can be solved in polynomial time via computing a maximum matching in bipartite graphs.

Proposition 2. Colorful Components *in two-colored graphs can be solved in $O(\sqrt{n}m)$ time.*

Proof. We begin by removing all edges $\{u, v\}$ where u and v have the same color. The remaining graph is bipartite, since it has a proper 2-coloring. This instance can be solved by computing a maximum matching: First, observe that the edges that are *not* deleted by a solution to Colorful Components must be a matching in the bipartite graph. This is because the degree of every vertex in the solution is either one (it is a part of a component of size two, which is the largest possible component size) or zero. Since maximizing the number of undeleted edges is equivalent to minimizing the number of deleted edges, we can obtain a minimum-cardinality solution by computing a maximum matching M, and then deleting all edges not contained in M.

Removing all edges between vertices with identical colors can be done in $O(m)$ time. A maximum matching in a bipartite graph can be found in $O(\sqrt{n}m)$ time using the Hopcroft–Karp algorithm. □

An efficient algorithm in trees with few colors. Let $T = (V, E)$ denote the input tree and assume that T is rooted at an arbitrary vertex. The idea is to do dynamic programming bottom-up from the leaves, storing for each $v \in V$ and $C \subseteq \{1, \ldots, c\}$ the minimal cost $T(v, C)$ of a solution for the subtree rooted at v where the connected component containing v contains exactly the colors of C.

We describe the algorithm for binary trees. Define T_v to be the subtree of the tree T that is rooted in node v. We then find, for every vertex v and every subset of colors $C \subseteq \{1, \ldots, c\}$, the minimal cost $T(v, C)$. We compute this by dynamic programming using a table $T[\cdot, \cdot]$. Performing a bottom-up traversal starting at the leaves, for each v we compute $T(v, C)$ for every C. When computing the cost $T(v, C)$, we choose the minimal cost between four options: In the first case, we do not delete the two edges from v to its children. The cost of the solution then is the minimum value of the sum of the costs of the two subtrees for every combination of colors that will give C. In the second and third case, we delete the edge to the left or the right subtree, respectively. The cost is obtained by

summing the cost of the solution for the subtree taken and the minimal possible cost for the rest of the tree. In the last case both edges are deleted. More formally, this reads as follows.
Initialization: For each leaf v, set $T[v, \{\chi(v)\}] := 0$ and $T[v, C] := \infty$ for $C \subseteq \{1, \ldots, c\}$ and $C \neq \{\chi(v)\}$.
Recursion: Let l and r denote the two children of an inner node v.

$$T[v, C] := \min \begin{cases} \min_{C_1 \uplus C_2 \uplus \{\chi(v)\} = C} T[l, C_1] + T[r, C_2], \\ T[r, C \setminus \{\chi(v)\}] + 1 + \min_{C' \subseteq X} T[l, C'], \\ T[l, C \setminus \{\chi(v)\}] + 1 + \min_{C' \subseteq X} T[r, C'], \\ 2 + \min_{C_1 \subseteq X} T[l, C_1] + \min_{C_2 \subseteq X} T[r, C_2] \end{cases}$$

The running time of this algorithm is $O(3^c \cdot n)$. The size of our dynamic programming table is $O(2^c \cdot n)$ to include all possible color subsets. Overall, the computation can be executed in $O(3^c \cdot n)$ time: for each vertex we need to consider at most $O(3^c)$ combinations of color subsets (every subset and the possibilities to split it into two). For each such combination the computation of the recursion can be performed in constant time if we maintain for each v the minimum cost of T_v. To extract the actual colorful components found, one can use a traceback procedure within the same running time bound. The exponential factor can be further improved (increasing the polynomial factor) to 2^c by using the convolution-based techniques of [2].

To extend this algorithm to work in general trees, we use a standard trick for dynamic programming in trees: Order the children of every node and add an additional dimension $i = 1, \ldots, d$ to the dynamic programming table, where d is the maximum degree in the tree. We then compute $T[v, i, C]$ by an adaption of the above approach. We omit the straightforward details.

Theorem 2. Colorful Components *on trees can be solved in* $2^c \cdot n^{O(1)}$ *time.*

An efficient fixed-parameter algorithm for graphs with few colors. Whereas on general graphs due to Theorem 1 there is no hope for fixed-parameter tractability with respect to the parameter "number c of colors", additionally using the parameter "number k of edge deletions" leads to fixed-parameter tractability.

First, we describe a simple $O(c^k \cdot m)$-time search tree algorithm. Using breadth-first search, it finds a bad path between two vertices of the same color. This strategy will be referred to as *bad-path branching* in the experimental part. This path has length at most c, since after visiting $c+1$ vertices in the breadth-first search there must be a bad path. Now, branch into the c cases to destroy this bad path by edge deletion, and for each case recursively solve the resulting instance. Since at most k edge deletions are needed, the search tree has depth at most k and therefore size $O(c^k)$; a bad path can be found in $O(m)$ time.

We can get a speed-up by using the observation that we can either find a bad path of length at most $c - 1$ or solve the problem in polynomial time. As a motivation for this improvement, observe that in practical applications the parameter c denoting the number of colors can be quite small with values in the one-digit range. Moreover, according to Theorem 1 we cannot expect a $2^{o(k)} \cdot n^{O(1)}$-time algorithm for Colorful Components on three-colored graphs.

Theorem 3. *For $c \geq 3$,* Colorful Components *can be solved in $O((c-1)^k \cdot m)$ time.*

Proof. We first describe the algorithm and then bound its running time. In the following analysis, assume that k is fixed in advance.

As long as the input graph contains a vertex v with degree at least three, perform a breadth-first search starting at v until either two vertices with the same color have been found or all vertices of v's connected component have been visited. In the second case, v's connected component is colorful and can therefore be removed from the graph. In the first case, we have visited at most $c+1$ vertices until a vertex pair with the same color has been found. The bad path between these two vertices has length at most $c-1$: since v has degree at least three, at least one neighbor of v is not on this path.

After a bad path has been found, branch into the at most $c-1$ edges to destroy it. Clearly, one of the edges has to be deleted. Hence, a solution can be found by recursively solving Colorful Components for each of these cases; now with $k-1$ edge deletions.

In case the graph has only vertices of degree at most two, proceed as follows. If the connected component is a cycle, then one of the following two cases can occur. If there is a bad path of length at most $c-1$ between any pair of two vertices, then branch as described above. Otherwise, the coloring on the cycle is ordered, that is, we can assume without loss of generality that each vertex with color 1 is adjacent to one vertex with color 2 and one vertex with color c; each vertex with color 2 is adjacent to one vertex with color 1 and one vertex with color 3, and so on. In this case, a solution for the connected component is simply to delete all edges between vertices of color c and 1. In the last remaining case, the connected component is a simple path. A solution for a path can be found by visiting the edges along the path starting from one of the two degree-one vertices until there is a color that has already been visited. Then, the last visited edge can be deleted; this is repeated until the path is colorful.

The running time of the algorithm can be shown as follows. The search tree has size $O((c-1)^k)$ since at each search tree node, we branch into at most $c-1$ cases, and the depth of the tree is at most k. A path to branch on can be found in $O(m)$ time since the procedure only uses breadth-first search. Finally, all presented algorithms for the polynomial-time special cases can be performed in $O(m)$ time as well; the overall running time follows.

If k is not given in advance, we can start the algorithm described above for increasing values of k until a solution is found; the running time bound remains the same since $\sum_{1 \leq i \leq k}(c-1)^i = O(c-1)^k$. □

The observation that we can either find a bad path of length $c-1$ or solve the problem in polynomial time also implies the following factor-$(c-1)$ approximation algorithm: As long as the graph contains a bad path of length at most $c-1$, delete all $c-1$ of these edges. If the graph has none of these bad paths, solve the problem in linear time. The approximation factor follows from the observation that at least one of the $c-1$ edges has to be deleted, and that deleting "unnecessary" edges does not create new bad paths.

Corollary 1. Colorful Components *can be approximated within a factor of* $c-1$ *in* $O(m^2)$ *time.*

Note that for $c \geq 11$ the factor-$(4\ln(c+1))$ approximation which is implied by the relation to Multi-Multiway Cut [1] gives better approximation ratios, for $c < 11$ our bound is better.

Data reduction. The following two polynomial-time executable data reduction rules for Colorful Components are relevant for the experimental work.

Rule 1. *If a connected component is colorful, then remove it from* G*.*

Rule 1 can be executed in linear time. We note that Rule 1 provides a trivial *kernelization* [9][2] for Colorful Components with respect to the combined parameter (k,c): obviously, after exhaustive data reduction, the instance has at most $2kc$ vertices, since an edge deletion can produce at most two colorful components, each of size at most c. This can be improved to a kernelization yielding only $(1+\epsilon)kc$ vertices for any $\epsilon > 0$: The idea of the corresponding data reduction is to choose any constant ℓ and to check (by say brute-force) for every connected component C and for all $1 \leq i \leq \ell$ whether (C,i) forms a yes-instance of Colorful Components and, if so, decrease the parameter k accordingly by i. The larger we choose ℓ, the smaller ϵ gets. We omit the details.

Rule 2 is less obvious.

Rule 2. *Let* $B = \{b_1, \ldots, b_t\}$ *be a minimal edge cut, let* G_B *be one side of the cut (that is, a connected component of* $G-B$ *such that each edge in* B *has exactly one endpoint in* G_B*), and let* N *denote the vertices that are incident with* B *but not in* G_B*. If* G_B *is colorful and* t*-edge connected and each color of* N *also occurs in* G_B*, then delete* B *and decrease* k *by* $|B|$*.*

Proof (of correctness). The correctness of Rule 2 can be seen as follows. Let S be a solution that does not contain some $\{u,v\} \in B$ with $u \in N$. Then, the bad path from u to the vertex in G_B with color $\chi(u)$ is destroyed by a set X of at least t edge deletions within G_B. Hence, the set $S' := (S \setminus X) \cup B$ is also a solution: First, $|S'| \leq |S|$. Second, the deletion of X only destroys bad paths that visit at least one vertex of $V(G_B)$ and all of these bad paths are also destroyed by deleting B since G_B is colorful. □

Note that so far it is not clear whether Rule 2 is applicable in polynomial time if t is not a constant.

[2] Informally, a kernelization transforms in polynomial time the original instance into a smaller equivalent instance whose size is upper-bounded by a function solely depending on the parameter; ideally, this function is a small polynomial.

4 Formulation as Weighted Multi-Multiway Cut

In the Colorful Components formulation, it is not possible to simplify a graph based on the knowledge that two vertices belong to the same connected component; we would like to be able to merge two such vertices. For this, we first need to allow not just a single color per vertex, but a set; moreover, we need to allow edge weights. Thus, we arrive at the edge-weighted version of Multi-Multiway Cut [1]: given an undirected graph $G = (V, E)$ with edge weights $w : E \to \{x \in \mathbb{Q} \mid x \geq 1\}$ and vertex sets $S_1, \ldots, S_c \subseteq V$, find a minimum-weight subset of edges $E' \subseteq E$ such that in $G' = (V, E \setminus E')$ no connected component contains two vertices from the same S_i.

To emphasize the connection to Colorful Components, for Weighted Multi-Multiway Cut we define the colors $\chi(u)$ of a vertex u as $\{i \mid u \in S_i\}$. Note that we require weights to be at least 1.

Now, we can *merge* two vertices u and v with disjoint colors. This means to replace them by a new vertex u' with colors $\chi(u) \cup \chi(v)$ and $N(u') := N(u) \cup N(v) \setminus \{u, v\}$, where $w(\{u', x\}) := w(\{u, x\}) + w(\{v, x\})$ (assuming $w(\{x, y\}) = 0$ for $\{x, y\} \notin E$).

Edge branching. Using the merge operation, we can do a simple branching on an edge [3]: either delete the edge, or merge its endpoints; in the experimental part this will be referred to as *edge branching.* Note that merging does not necessarily decrease the parameter; but it is easy to see that if we branch on each edge of a forbidden path successively, then the last edge of the path cannot be merged since it connects vertices with an intersecting color set. This allows us to immediately delete the edge; thus, the $O(c^k \cdot m)$-time branching is still possible.

Data reduction. We can also adapt Rule 2 to Weighted Multi-Multiway Cut; the proof is similar to that of Rule 2.

Rule 3. *Let $V' \subseteq V$ be a colorful subgraph. If the cut between V' and $V \setminus V'$ is at least as large as the connectivity of V', then merge V' into a single vertex. Herein, connectivity is defined as the minimum total* weight *of edges to be deleted to obtain at least one more connected component.*

Merge heuristic. The idea of the heuristic is to repeatedly merge the two vertices "most likely" to be in the same component. During the process, we immediately delete edges connecting vertices with intersecting color sets. The *merge cost* of two vertices u and v is the weight of the edges that would need to be deleted when merging u and v, while the *cut cost* is defined as

$$3w(\{u, v\}) + \sum_{w \in V \mid \{\{u,w\},\{v,w\}\} \subseteq E} \min\{w(\{u, w\}), w(\{v, w\})\}$$

as a rough approximation of the minimum cut between u and v. The factor 3 has been tuned heuristically. We then always merge the endpoints of the edge that maximizes cut cost minus merge cost.

Table 1. Instances before and after data reduction. Herein, n, m, and c are the number of vertices, edges, and colors, respectively, for the whole graph while n',m',c' denote those values for the largest connected component of the instances.

	original						after Rule 2						after Rule 3					
	n	m	c	n'	m'	c'	n	m	c	n'	m'	c'	n	m	c	n'	m'	c'
min.	178	156	3	8	7	3	0	0	0	0	0	0	0	0	0	0	0	0
max.	9311	26344	10	3048	6063	10	2769	5583	10	2769	5583	10	2602	5336	10	2602	5336	10
avg.	1702	3159	6.2	504	921	6.2	486	839	5.2	407	697	4.7	429	712	5.9	354	607	5.3
med.	1187	1401	6	149	232	6	172	262	6	46	90	5	119	128	6	42	58	5

5 Experiments

We performed experiments with instances from the multiple sequence alignment application. The search tree algorithm, data reduction, and merge heuristic were implemented in OCaml and compiled with the OCaml native-code compiler version 3.11.2. The test machine is a 2.66 GHz Intel Xeon X5550 with 8 MB cache and 16 GB main memory, running under openSUSE 11.3 Linux.

The source code and the test instances are available under the GNU GPL license at `http://fpt.akt.tu-berlin.de/colcom/`.

Data. We generated one Colorful Components instance for each multiple alignment instance from the BAliBASE 3.0 benchmark [15], using the diafragm 1.0 software [6]. We restricted the experiments to the 135 of the 386 instances that have at most 10 colors (that is, 10 sequences to be aligned). Instances with more colors can mostly not be solved with our exact methods.

Implementation details. To speed up the branching algorithms from Section 3 and Section 4, we use a transposition table in order to avoid recomputing the solutions of identical search tree nodes. We also track upper and lower bounds. These bounds are seeded with the result of the heuristic and a simple greedy packing, respectively. The branching is always on a shortest bad path. If during the branching process the instance decomposes into connected components, we solve them separately.

To efficiently find data reduction opportunities with Rule 2 and Rule 3, we try starting with each vertex and successively add more vertices with disjoint colors that minimize the cut to other edges, until we have either found a reduction opportunity or no more vertices can be added.

Results. We first examine the effect of data reduction (see Table 1), that is, we compare the size of the instances before and after exhaustively applying the data reduction rules. With Rule 2, we can solve 47 instances by data reduction alone, the largest among those having 3115 vertices and 4383 edges. The number of edges is reduced on average by 76.1 % (median 92.8 %). When considering the largest connected component, we get an average reduction of 64.5 % (median 65.8 %). Rule 2 reduces the largest component only by 55.4 % on average (median 54.9 %). Thus, clearly for many instances only data reduction makes the exact approaches feasible.

Next, we consider the running times of the branching algorithms. For the bad-path branching, we have 61 instances that can be solved in less than 1 second, 6 instances that can be solved in 1 second to 10 minutes, and 68 cannot be solved in 10 minutes. With edge branching, we can solve 70 instances in less than 1 second, 9 in 1 second to 10 minutes, and 56 remain unsolved. We note that in ongoing research, we are able to solve several more instances to optimality with integer linear programming (ILP) based approaches.

For the heuristics, we compare the solution quality for the 112 instances for which we know the optimal solution. The min-cut heuristic [6] has an error between 0 % (once) and 70.0 %, with an average error of 29.2 % (median 27.8 %). In contrast, the merge heuristic has an error between 0 % (76 times) and 12.7 %, with the average error 0.6 % (median 0 %). Without data reduction, the results are slightly worse with 66 times an optimal result and an average error of 1.0 % (median 0 %). Thus, clearly the merge heuristic is much superior for these instances, and in fact solves the majority of the instances optimally. Both heuristics take at most two seconds to solve an instance.

Finally, for the instances for which an exact solution was found, we compared the solution quality of the alignments obtained by using DIALIGN with and without the partial alignment columns indicated by an exact solution for Colorful Components, by the merge heuristic, and by the min-cut heuristic. The found alignments were compared with the BAliBASE reference alignments concerning the reconstruction of total columns (TC score) and position pairs (SP score). The exact algorithm had a TC score of 56.6 %, the merge heuristic achieved 55.1 %, the min-cut heuristic 53.6 %, and the alignment without anchors achieved 54 %; for SP-score the results are similar. This indicates that minimizing edge deletions for obtaining Colorful Components is indeed helpful for obtaining better alignments. Note that, concerning TC score, DIALIGN with the min-cut heuristic is about 10 percentage points worse than current state-of-the-art multiple alignment methods [6]. Hence, an improvement of roughly 3 percentage points is a sizable step towards closing the gap between DIALIGN and these methods.

6 Outlook

It is open to obtain a smaller problem kernel, a problem kernel with size independent of c, and branching algorithms with branching number less than $c - 1$. So far, it is also undetermined whether Rule 2 and Rule 3 can be exhaustively applied in polynomial time. From the modeling perspective, it is interesting to consider a relaxation of the colorfulness constraint: In preliminary experiments with network alignment data, we found that allowing only one protein of each species to be matched was, while a natural model, too strict. Generalizing Color Components to allow a constant number of occurrences of each color for the connected components could result in improved network alignments.

References

[1] Avidor, A., Langberg, M.: The multi-multiway cut problem. Theoretical Computer Science 377(1-3), 35–42 (2007)

[2] Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Fourier meets Möbius: fast subset convolution. In: Proc. 39th STOC, pp. 67–74. ACM (2007)

[3] Böcker, S., Briesemeister, S., Bui, Q.B.A., Truß, A.: Going weighted: Parameterized algorthims for cluster editing. Theoretical Computer Science 410(52), 5467–5480 (2009)

[4] Bousquet, N., Daligault, J., Thomassé, S.: Multicut is FPT. In: Proc. 43rd STOC, pp. 459–468. ACM (2011)

[5] Bruckner, S., Hüffner, F., Komusiewicz, C., Niedermeier, R.: Entity disambiguation by partitioning under heterogeneity constraints (February 2012) (manuscript, submitted), http://fpt.akt.tu-berlin.de/publications/disambiguation.pdf

[6] Corel, E., Pitschi, F., Morgenstern, B.: A min-cut algorithm for the consistency problem in multiple sequence alignment. Bioinformatics 26(8), 1015–1021 (2010)

[7] Deniélou, Y.-P., Boyer, F., Viari, A., Sagot, M.-F.: Multiple Alignment of Biological Networks: A Flexible Approach. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 263–273. Springer, Heidelberg (2009)

[8] Garg, N., Vazirani, V.V., Yannakakis, M.: Primal–dual approximation algorithms for integral flow and multicut in trees. Algorithmica 18(1), 3–20 (1997)

[9] Guo, J., Niedermeier, R.: Invitation to data reduction and problem kernelization. ACM SIGACT News 38(1), 31–45 (2007)

[10] Komusiewicz, C.: Parameterized Algorithmics for Network Analysis: Clustering & Querying. PhD thesis, Technische Universität Berlin, Berlin, Germany (2011)

[11] Li, J., Yi, K., Zhang, Q.: Clustering with Diversity. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 188–200. Springer, Heidelberg (2010)

[12] Lokshtanov, D., Marx, D., Saurabh, S.: Lower bounds based on the Exponential Time Hypothesis. Bulletin of the EATCS 105, 41–71 (2011)

[13] Marx, D., Razgon, I.: Fixed-parameter tractability of multicut parameterized by the size of the cutset. In: Proc. 43rd STOC, pp. 469–478. ACM (2011)

[14] Park, D., Singh, R., Baym, M., Liao, C.-S., Berger, B.: IsoBase: a database of functionally related proteins across PPI networks. Nucleic Acids Research 39(Database), 295–300 (2011)

[15] Thompson, J.D., Koehl, P., Ripp, R., Poch, O.: BAliBASE 3.0: latest developments of the multiple sequence alignment benchmark. Proteins: Structure, Function, and Bioinformatics 61(1), 127–136 (2005)

[16] Weller, M., Komusiewicz, C., Niedermeier, R., Uhlmann, J.: On making directed graphs transitive. Journal of Computer and System Sciences 78(2), 559–574 (2012)

Approximation Algorithms and Hardness Results for Shortest Path Based Graph Orientations*

Dima Blokh[1,**], Danny Segev[2,**], and Roded Sharan[1]

[1] Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
{blokhdmi,roded}@post.tau.ac.il

[2] Department of Statistics, University of Haifa, Haifa 31905, Israel
segevd@stat.haifa.ac.il

Abstract. The graph orientation problem calls for orienting the edges of an undirected graph so as to maximize the number of pre-specified source-target vertex pairs that admit a directed path from the source to the target. Most algorithmic approaches to this problem share a common preprocessing step, in which the input graph is reduced to a tree by repeatedly contracting its cycles. While this reduction is valid from an algorithmic perspective, the assignment of directions to the edges of the contracted cycles becomes arbitrary, and the connecting source-target paths may be arbitrarily long. In the context of biological networks, the connection of vertex pairs via shortest paths is highly motivated, leading to the following variant: Given an undirected graph and a collection of source-target vertex pairs, assign directions to the edges so as to maximize the number of pairs that are connected by a shortest (in the original graph) directed path. Here we study this variant, provide strong inapproximability results for it and propose an approximation algorithm for the problem, as well as for relaxations of it where the connecting paths need only be approximately shortest.

1 Introduction

Protein-protein interactions form the skeleton of signal transduction in the cell. While many of these interactions carry directed signaling information, current interaction measurement technologies, such as yeast two hybrid [5] and co-immunoprecipitation [7], reveal the presence of an interaction, but not its directionality. Identifying this directionality is fundamental to our understanding of how these protein networks function.

To tackle the arising orientation problem, previous work has relied on information from perturbation experiments [13], in which a gene is perturbed (cause) and as a result other genes change their expression levels (effects). The fundamental assumption is that, for an effect to take place, there must be a directed

* Due to space limitations, some proofs are omitted from this extended abstract. These will appear in the full version of this paper.

** These authors contributed equally to this work.

path in the network from the causal gene to the affected gene. This setting calls for an orientation, that is, an assignment of directions to the edges of the network, such that a maximum number of pairs admit a directed path from the cause (source of response) to the affected genes (targets of the response).

Recently, large scale networks for many organisms have become available, leading to increasing interest in orientation problems of this nature. Medvedovsky et al. [10], Gamzu et al. [6], and later on Elberfeld et al. [3], were the first to study the MAXIMUM GRAPH ORIENTATION problem (MGO), where the objective is to direct the edges of a given (undirected) network so as to maximize the number of vertex pairs that are connected by directed source-target paths, which are allowed to be of arbitrary length. They proved that MGO is NP-hard to approximate to within a factor better than 12/13 and provided an $\Omega(\log\log n/\log n)$ approximation algorithm for it. It was further shown that MGO, as well as several natural extensions, admit efficient integer programming formulations [10, 11].

The main caveat of these approaches is that they all employ a preprocessing step in which cycles in the input graph are contracted one after the other, ending up with a tree network. Such structural modifications do not affect the optimization criterion, since directed connectivity can be preserved when cycles are consistently oriented in advance, either in clockwise or counter-clockwise direction. However, in practice, this preprocessing step results in a large fraction of the edges being arbitrarily oriented and in arbitrarily long directed source-target paths.

Other approaches to the problem concentrated on short connecting paths, which are more plausible biologically [13]. Gitter et al. [8] focused on paths whose length is bounded by a parameter k, showing that while the resulting problem is NP-hard, it can still be approximated within factor $O(k/2^k)$. Vinayagam et al. [12] developed a Bayesian learning strategy to predict the directionality of each edge based on the shortest paths that contain it.

Problem definition and our contribution. In this paper, we study the latter biologically-motivated setting [8], in which the directed paths connecting each pair of source-target vertices are required to be shortest. Let $G = (V, E)$ be an undirected graph with a vertex set V of size n and an edge set E of size m. Denote by $\delta_G(s,t)$ the length (number of edges) of a shortest path between s and t. An *orientation* $\vec{G}$ of G is a directed graph on the same vertex set whose edge set contains a single directed instance of every undirected edge, but nothing more. We say that a pair of vertices (s,t) is *satisfied* by an orientation $\vec{G}$ when the latter contains a directed s-t path of length $\delta_G(s,t)$. The MAXIMUM SHORTEST-PATH ORIENTATION (MSPO) problem is defined as follows:

Input: An undirected graph G and a collection $P = \{(s_1,t_1),\ldots,(s_k,t_k)\}$ of source-target vertex pairs.

Objective: Compute an orientation of G that satisfies a maximum number of pairs.

Our contribution is three-fold: (i) We relate the hardness of approximating MSPO to that of the Independent Set problem through a combinatorial construction called the "single-pair gadget", which may be interesting in its own right. Consequently, we show that this problem is NP-hard to approximate within factors $O(k^{1-\epsilon})$ and $O(m^{1/3-\epsilon})$, for any fixed $\epsilon > 0$ (Section 2). (ii) On the positive side, we adapt the approximation algorithm of [3], which was initially suggested for MGO in mixed graphs, and attain a performance guarantee of $\Omega(1/\max\{n,k\}^{1/\sqrt{2}})$ (Section 3.1). (iii) Last, we show that significantly better upper bounds can be obtained when one is willing to settle for bi-criteria approximations, where the strict requirement of connecting pairs only via shortest paths is relaxed and, instead, approximately-shortest paths are allowed. Here, we make use of random embeddings to compute $\tilde{O}(\log n)$-approximate shortest paths connecting an $\Omega(1/\log n)$ fraction of all pairs, with constant probability. Additionally, we show that by using $(1+\epsilon)$-approximate shortest paths one can satisfy an $\tilde{\Omega}(1/\sqrt{k})$ fraction of the pairs (Section 3.2).

2 Hardness of Approximation

In this section we provide a reduction from Independent Set showing that it is NP-hard to approximate MSPO to within factors $\Omega(1/k^{1-\epsilon})$ and $\Omega(1/m^{1/3-\epsilon})$ of optimum for any fixed $\epsilon > 0$. To this end, we first construct a *single-pair gadget*, which shows that there are MSPO instances in which even optimal orientations satisfy only one out of k source-target pairs. This construction will serve as the main building block of our hardness reduction. The single-pair gadget is also interesting in its own right, as it creates a strong separation between our definition of satisfying a given pair via a shortest path and the one studied by Medvedovsky et al. [10], in which pairs could be satisfied via any directed path, a setting where a logarithmic fraction of all pairs can always be satisfied.

2.1 The Single-Pair Gadget

For convenience, we describe the single-pair gadget using an edge-weighted mixed graph, in which some of the edges are pre-directed. Later on, we show how to remove these extra constraints. In what follows, given any integer k, we show how to create an MSPO instance (G, P) with k pairs, $O(k^2)$ vertices and $O(k^2)$ edges, such that the following properties are satisfied: (1) For every pair in P there is some orientation that satisfies it, and (2) Any orientation of G satisfies at most one pair in P. To this end, we will argue that, in the instance described below, there is a unique shortest path connecting any given source-target pair. Moreover, these will be contradicting paths, in the sense that when one sets the direction of any such path from source to target all other paths can no longer be similarly directed (due to overlapping edges that need to be oriented in opposite directions).

Our construction is schematically drawn in Figure 1. In detail, the graph vertices are partitioned into k layers, $\mathcal{V}_1, \ldots, \mathcal{V}_k$, where $\mathcal{V}_i$ contains $2k-i$ vertices, $\{v_{i,1}, \ldots, v_{i,2k-i}\}$. There are three types of edges:

– Cross edges, E_{cross}: For every $1 \le i \le k-1$ and $i < j \le k$, we have a pair of directed edges $(v_{j,i}, v_{i,2j-i-1})$ and $(v_{i,2j-i-2}, v_{j,i+1})$. The weight of these edges is 1.
– Contradiction edges, E_{cont}: For every $1 \le i \le k-1$ and $i < j \le k$, we have an undirected edge $(v_{i,2j-i-2}, v_{i,2j-i-1})$. The weight of these edges is 0.
– Direction edges, E_{dir}: For every $1 \le i \le k-1$ and $i < j \le k+1$, we have a directed edge $(v_{i,2j-i-1}, v_{i,2j-i})$. The weight of these edges is 2.

Finally, the collection of pairs is $P = \{(s_i, t_i) : 1 \le i \le k\}$, where $s_i = v_{i,1}$ and $t_i = v_{i,2k-i}$.

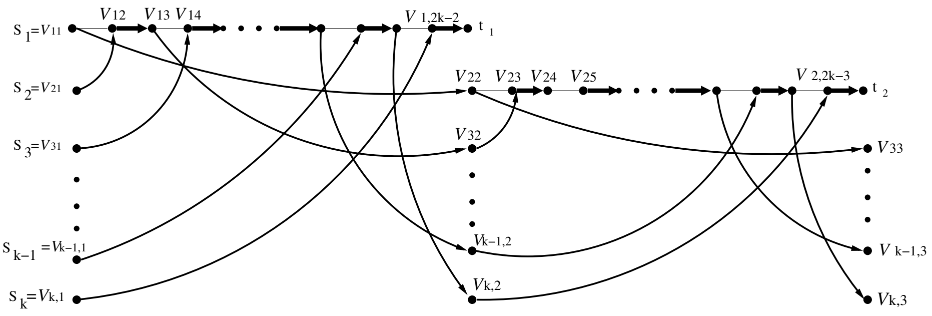


Fig. 1. The single-pair gadget (only the first two layers are shown). Here, direction edges are drawn as thick lines, cross edges as regular lines, and contradiction edges as thin lines.

We begin to analyze the single-pair gadget by highlighting a couple of structural properties that will be required to establish the uniqueness of shortest paths and the way in which they intersect. Observations 1 and 2 characterize the unique paths that connect vertices in one vertical column of the gadget (i.e, $v_{i,i}, \ldots, v_{k,i}$) to its successive column ($v_{i+1,i+1}, \ldots, v_{k,i+1}$). Somewhat informally, these observations will allow us to argue that for any s_i-t_i path, the sequence of column entry points $s_i = v_{i,1} \rightsquigarrow v_{i_2,2} \rightsquigarrow \cdots \rightsquigarrow v_{i_i,i}$ is non-decreasing in its vertical distance from s_i, that is, $i \le i_2 \le \cdots \le i_i$.

Observation 1. For every $1 \le i \le k-1$ and $i < j_1 \le j_2 \le k$, there is only one path from $v_{j_1,i}$ to $v_{j_2,i+1}$. More specifically,

– If $j_1 = j_2$, this path takes the cross edge from $v_{j_1,i}$ to $v_{i,2j_1-i-1}$, then a single contradiction edge (in right-to-left direction), and finally the cross edge from $v_{i,2j_1-i-2}$ to $v_{j_1,i+1}$. Hence, the total weight of this path is 2.
– If $j_1 < j_2$, this path takes the cross edge from $v_{j_1,i}$ to $v_{i,2j_1-i-1}$, then travels in left-to-right direction in $\mathcal{V}_i$, alternating between direction and contradiction edges, and finally takes the cross edge from $v_{i,2j_2-i-2}$ to $v_{j_2,i+1}$. Hence, the total weight of this path is $2 + 2(j_2 - j_1)$.

Observation 2. For every $1 \leq i \leq k-1$ and $i < j_1 < j_2 \leq k$, there are no paths from $v_{j_2,i}$ to $v_{j_1,i+1}$.

With these observations in place, let us focus on one particular s_i-t_i path, p_i, which is schematically drawn in Figure 2 (for $i = 3$). This path repeatedly takes two cross edges and one contradiction edge $i-1$ times until it arrives to $v_{i,i}$, and then traverses $\mathcal{V}_i$ in left-to-right direction to reach $v_{i,2k-i} = t_i$. The next lemma shows that p_i must be shortest and unique.

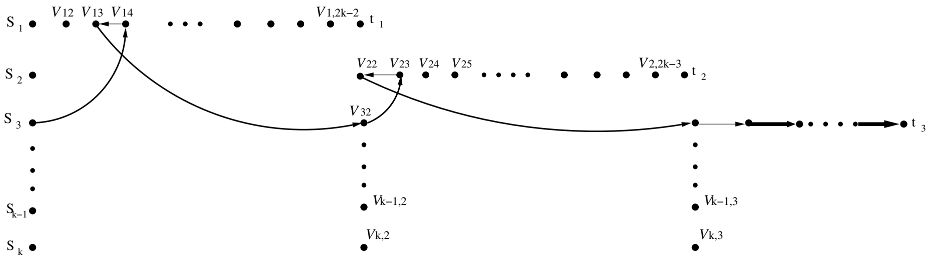


Fig. 2. The path p_3 connecting s_3 to t_3

Lemma 1. *For every $1 \leq i \leq k$, the path p_i is the unique shortest s_i-t_i path.*

Proof. By definition of p_i, this path traverses $2(i-1)$ cross edges and $i-1$ contradiction edges prior to arriving at $v_{i,i}$. Then it traverses $k-i$ additional pairs of direction and cross edges before reaching t_i. Therefore, the total weight of p_i is exactly $2(i-1) + 2(k-i) = 2k-2$.

Now consider some other s_i-t_i path, $p \neq p_i$, and let $v_{j,i}$ be the entry point of p into the ith column (whose vertices are $v_{i,i}, \ldots, v_{k,i}$). Suppose $j = i$ and consider all the entry points of p into columns $2, \ldots, i-1$. By Observation 2 all these points must be at layer i and, hence, p identifies with p_i, contradicting our initial assumption. Thus, we may assume that $j > i$. By Observations 1 and 2, it follows that p traverses $2(i-1)$ cross edges and $j-i$ direction edges prior to arriving at $v_{j,i}$. The combined weight of those edges is $2(i-1)+2(j-i) = 2j-2$. From $v_{j,i}$, the path p must traverse the cross edge to $v_{i,2j-i-1}$ and then $k-j+1$ additional direction edges before reaching t_i. Consequently, the total weight of p is $(2j-2)+1+2(k-j+1) = 2k+1$, which is strictly greater than the weight of p_i, a contradiction. □

We conclude that for every pair $(s_i, t_i) \in P$ there exists an orientation satisfying this pair, in which all contradiction edges along p_i are oriented from s_i to t_i. It remains to show that any orientation satisfies at most one pair. Suppose to the contrary that there exists an orientation $\vec{G}$ that satisfies both (s_{i_1}, t_{i_1}) and (s_{i_2}, t_{i_2}), for some $i_1 < i_2$, meaning in particular that both p_{i_1} and p_{i_2} must agree with $\vec{G}$. However, these paths intersect in exactly one contradiction edge, $(v_{i_1,2i_2-i_1-2}, v_{i_1,2i_2-i_1-1})$, where in p_{i_1} it is orientated from left to right, while in p_{i_2} its direction is from right to left, a contradiction.

2.2 Reduction from Independent Set

We are now ready to make use of the single-pair gadget in order to prove the hardness of approximating MSPO. To simplify the presentation, we first establish this result for the more general setting in which the underlying graph is mixed (i.e., contains both directed and undirected edges) and weighted, similar to the construction described in Section 2.1.

Theorem 3. *For any fixed $\epsilon > 0$, it is NP-hard to approximate MSPO to within factors $\Omega(1/k^{1-\epsilon})$ and $\Omega(1/m^{1/2-\epsilon})$ of optimum in mixed weighted graphs.*

Proof. The basis for our reduction is the Independent Set problem, which is known to be hard to approximate to within a factor of $\Omega(1/n^{1-\epsilon})$ on an n-vertex graph for any fixed $\epsilon > 0$ [9]. Given an Independent Set instance $G = (V, E)$, we begin by constructing a single-pair gadget for $k = |V|$. In this construction, every layer $\mathcal{V}_i$ represents a vertex $v_i \in V$. Next, for every pair of vertices v_i and v_j such that $(v_i, v_j) \notin E$, we replace the cross edges $(v_{j,i}, v_{i,2j-i-1})$ and $(v_{i,2j-i-2}, v_{j,i+1})$ by a single directed edge $(v_{j,i}, v_{j,i+1})$ of weight 2. This modification is illustrated in Figure 3.

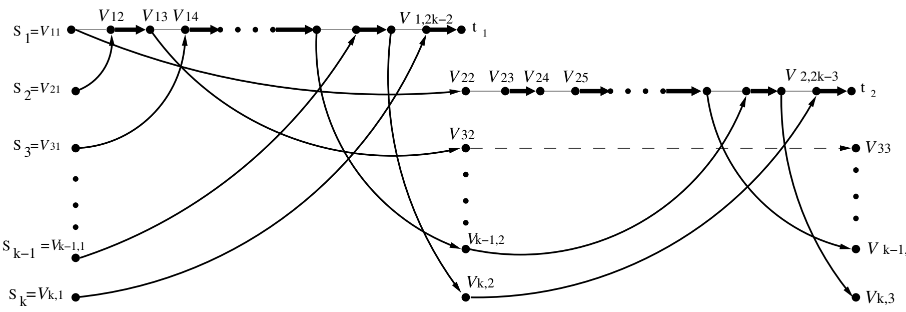


Fig. 3. An example modification for v_2 and v_3, where their newly added edge is drawn as a dashed line

Now, for an original vertex v_i, let us focus once again on one particular s_i-t_i path, $\tilde{p}_i$. This path is created from the unique shortest path p_i in the original single-pair gadget by replacing every $\langle$cross, contradiction, cross$\rangle$ sequence of edges along p_i with its corresponding newly-added edge, whenever this modification has been made. By adapting the analysis given in Section 2.1, it is easy to verify that $\tilde{p}_i$ becomes the unique shortest s_i-t_i path. We proceed by observing that for every pair of original vertices v_i and v_j, $i < j$, the unique shortest paths $\tilde{p}_i$ and $\tilde{p}_j$, respectively connecting s_i to t_i and s_j to t_j, are edge-disjoint if and only if $(v_i, v_j) \notin E$. This follows from the way in which $\tilde{p}_i$ and $\tilde{p}_j$ were derived from p_i and p_j, along with our previous observation that p_i and p_j intersect in exactly one contradiction edge. This edge, $(v_{i_1,2i_2-i_1-2}, v_{i_1,2i_2-i_1-1})$, will be skipped in the modified instance by $\tilde{p}_j$ if and only if $(v_i, v_j) \notin E$.

It follows that there is a one-to-one correspondence between solutions $\{v_i : i \in I\}$ to the Independent Set instance and sets of pairs $\{(s_i, t_i) : i \in I\}$ that can be satisfied by some orientation. As the resulting MSPO instance consists of n pairs and $O(n^2)$ edges, the hardness of approximation for Independent Set implies bounds of $\Omega(1/k^{1-\epsilon})$ and $\Omega(1/m^{1/2-\epsilon})$ on the approximability of MSPO. □

It remains to show that the above reduction can be extended to the setting of undirected and unweighted graphs. For the former, we will show that when every directed edge is replaced in the single-pair gadget by an undirected edge, shortest paths remain unchanged. The following lemmas establish the correctness of this alteration. Due to space limitations and the rather involved nature of the corresponding proofs, these are deferred to the full version of our paper.

Lemma 2. *For every $1 \leq i \leq k$, a shortest s_i-t_i path in the undirected single-pair gadget cannot traverse cross edges in a direction different than the one defined in the mixed gadget.*

Lemma 3. *For every $1 \leq i \leq k$, a shortest s_i-t_i path in the undirected single-pair gadget cannot traverse direction edges from right to left.*

It remains to show how to remove edge weights from our construction. To this end, we first transform the original weights in the single-pair gadget so that these become positive integers. While cross and direction edges are associated with weights 1 and 2, respectively, contradiction edges are associated with zero weights. Our objective is to "scale" these values without changing the shortest path structure on the one hand, and while avoiding the use of large values on the other hand so as not to affect the inapproximability bound by much.

We begin by setting the weight of contradiction edges to $1/k$. This implies that for every $1 \leq i \leq k$, the total weight of the unique shortest s_i-t_i path p_i (see Section 2.1), which has been preserved during the reduction from mixed to undirected graphs, is at most $2k - 2 + (k-1)/k$. This is lighter than any other s_i-t_i path, which has weight at least $2k + 1$ according to the proof of Lemma 1. We proceed by scaling all edge weights by a factor of k to make them integral. Last, we replace each edge e of weight $w(e)$ by a path consisting of $w(e)$ unit-weight edges. As a result, the number of vertices and edges blows up to $O(k^3)$ instead of $O(k^2)$ as in the original gadget. Combined with our reduction from the Independent Set problem, the next inapproximability result follows.

Theorem 4. *For any fixed $\epsilon > 0$, it is NP-hard to approximate MSPO to within factors of $\Omega(1/k^{1-\epsilon})$ and $\Omega(1/m^{1/3-\epsilon})$ of optimum.*

Interestingly, we can use our construction to provide similar hardness of approximation results for the problem variant studied by Gitter at al. [8], for which non-trivial bounds were not known before. Further details will be provided in the full version of this paper.

3 Approximation Algorithms

In this section we provide an approximation algorithm for MSPO whose performance guarantee is sub-linear in either the number of vertices of the underlying graph or in the number of input pairs. In light of the hardness results established in Section 2, we cannot expect to come significantly closer to the optimal number of satisfied pairs, and the only possible avenue for improvement is decreasing the exponent we attain. However, a detailed inspection of Theorem 4 and its proof reveals that these do not exclude the possibility of obtaining better performance guarantees when one is willing to relax the strict requirement of satisfying pairs only via shortest paths and, instead, make use of approximately shortest paths. We explore this option as well, and show how to improve our previously-mentioned algorithm by utilizing such paths.

3.1 Exact Shortest Paths

To tackle MSPO, we adapt the approximation algorithm of Elberfeld et al. [3], which was initially suggested for MGO in mixed graphs. In that setting, pairs could be satisfied via any connecting path, regardless of its length, whereas in the current setting, connecting paths are required to be shortest.

Let (G, P) be an MSPO instance. For every $(s_i, t_i) \in P$, choose arbitrarily a shortest path p_i between them. Let $\mathcal{P} = \{p_i : (s_i, t_i) \in P\}$. The algorithm is iterative. At any point in time, we will be holding a partial orientation G_ℓ of G and a subset $\mathcal{P}_\ell \subseteq \mathcal{P}$ of shortest paths, where these sets are indexed according to the step number that has just been completed. Initially $G_0 = G$ and $\mathcal{P}_0 = \mathcal{P}$. Now, as long as none of the termination conditions described below is met, we proceed as follows:

1. Let $\hat{p} = (s, \dots, t)$ be a minimum-length path in $\mathcal{P}_\ell$.
2. Orient $\hat{p}$ in the direction from s to t to obtain $G_{\ell+1}$.
3. To prevent the edges in $\hat{p}$ from being re-oriented in subsequent iterations, discard from $\mathcal{P}_\ell$ the path $\hat{p}$ as well as any path that overlaps (in edges) with it, obtaining $\mathcal{P}_{\ell+1}$.

There are two conditions that will cause the greedy iterations to terminate. For now, we state both conditions in terms of two parameters, $\alpha \geq 0$ and $\beta \geq 0$, whose values will be optimized later on.

1. $|\mathcal{P}_\ell| \leq n^\alpha$. In this case, we orient an arbitrary path from $\mathcal{P}_\ell$.
2. There exists a vertex v such that at least $|\mathcal{P}_\ell|^\beta$ paths in $\mathcal{P}_\ell$ go through v. Let $\mathcal{P}'_\ell$ be this sub-collection of paths and let P' be the collection of corresponding pairs. We show in the full version of this paper that one can satisfy at least $1/4$ of these pairs.

Under both termination conditions, we complete the orientation by directing the remaining edges in an arbitrary manner. With some modifications through their analysis, the arguments of Elberfeld et al. [3] essentially give rise to the next claim.

Lemma 4. *When the algorithm terminates due to condition 1, the number of satisfied pairs is $\Omega(k/n^{\max\{1-\alpha(1-2\beta),\alpha\}})$. Termination due to condition 2 leads to $\Omega(k/\max\{n^{1-\alpha(1-2\beta)}, k^{1-\beta}\})$ satisfied pairs.*

To obtain the best-possible performance guarantee, we pick values for α and β so as to minimize the maximum of all exponents mentioned above. To this end, the optimal values are $\alpha^* = \sqrt{1/2}$ and $\beta^* = 1 - \sqrt{1/2}$, in which case the maximal exponent becomes $\sqrt{1/2} \approx 0.707$.

Theorem 5. *MSPO can be approximated to within factor $\Omega(1/\max\{n,k\}^{1/\sqrt{2}})$.*

3.2 Approximate Shortest Paths

In order to improve on the performance guarantee attained in Theorem 5, we proceed by providing bi-criteria approximation algorithms for MSPO. Here, we relax the strict requirement of satisfying pairs only via shortest paths and, instead, allow approximately-shortest paths.

The precise setting we consider is as follows: For $\sigma \geq 1$, we say that a given orientation $\vec{G}$ σ-satisfies the pair (s_i, t_i) when it contains a directed s_i-t_i path of length at most σ times that of a shortest path, i.e., $\delta_{\vec{G}}(s_i, t_i) \leq \sigma \cdot \delta_G(s_i, t_i)$. For $\alpha \leq 1$ and $\sigma \geq 1$, we say that a given algorithm guarantees an (α, σ)-approximation when, for any instance of the problem, it computes an orientation that σ-satisfies at least $\alpha \cdot \mathrm{OPT}$ pairs. Here, OPT stands for the maximal number of pairs that can be 1-satisfied by any orientation.

An $(\Omega(1/\log n), \tilde{O}(\log n))$-approximation via embedding. With a slight adaptation of the metric embeddings terminology to our particular setting, the basic idea in this approach is to compute a random spanning tree $T \subseteq G$, sampled from a distribution $\mathcal{T}$ over a set of spanning trees in a way that pairwise distances do not get "stretched" by much in expectation. This line of work [2, 4] has evolved into a near-optimal bound due to Abraham, Bartal, and Neiman [1], who showed how to sample a random spanning tree such that the expected stretch is $\tilde{O}(\log n)$ uniformly over all vertex pairs, that is,

$$\max_{(u,v)\in V\times V} \mathrm{E}_{T\sim\mathcal{T}}\left[\frac{\delta_T(u,v)}{\delta_G(u,v)}\right] \leq \psi(n) = O(\log n \log\log n (\log\log\log n)^3) \ .$$

Here, $\mathrm{E}_{T\sim\mathcal{T}}[\cdot]$ denotes expectation with respect to the random choice of T, and $\psi(n)$ is our notation for the precise upper bound on the maximal expected stretch. In what follows, we argue that this result can be exploited to obtain logarithmic error bounds in both the number of satisfied pairs and in the extent to which distances are stretched.

Theorem 6. *There is a randomized algorithm that $\tilde{O}(\log n)$-satisfies $\Omega(k/\log n)$ pairs, with constant probability.*

Proof. We begin by computing a random spanning tree T using the embedding method of Abraham et al. [1]. With respect to this tree, let $P_{\text{small}} \subseteq P$ be the collection of pairs whose shortest path distances have not been significantly stretched beyond a factor of $\psi(n)$, which will be formally defined as $P_{\text{small}} = \{(s_i, t_i) \in P : \delta_T(s_i, t_i) \leq 2\psi(n) \cdot \delta_G(s_i, t_i)\}$. Since $\mathrm{E}_{T \sim \mathcal{T}}[\delta_T(s_i, t_i)] \leq \psi(n) \cdot \delta_G(s_i, t_i)$ for every pair $(s_i, t_i) \in P$, by Markov's inequality, each of these pairs is indeed a member of P_{small} with probability at least $1/2$. For this reason, $\mathrm{E}[|P_{\text{small}}|] \geq k/2$, which implies that $|P_{\text{small}}| \geq k/4$ with probability at least $1/3$, since

$$\begin{aligned}
\frac{k}{2} &\leq \mathrm{E}\left[|P_{\text{small}}|\right] \\
&= \Pr\left[|P_{\text{small}}| \geq \frac{k}{4}\right] \cdot \mathrm{E}\left[|P_{\text{small}}| \,\middle|\, |P_{\text{small}}| \geq \frac{k}{4}\right] \\
&\quad + \Pr\left[|P_{\text{small}}| < \frac{k}{4}\right] \cdot \mathrm{E}\left[|P_{\text{small}}| \,\middle|\, |P_{\text{small}}| < \frac{k}{4}\right] \\
&\leq \Pr\left[|P_{\text{small}}| \geq \frac{k}{4}\right] \cdot k + \left(1 - \Pr\left[|P_{\text{small}}| \geq \frac{k}{4}\right]\right) \cdot \frac{k}{4} \ .
\end{aligned}$$

Thus, with constant probably we obtain a spanning tree for which $|P_{\text{small}}|$, i.e., the number of pairs in P with stretch smaller than $2\psi(n) = \tilde{O}(\log n)$, contains a constant fraction of the pairs in P. Since we formed a tree instance, the maximum tree orientation algorithm of Medvedovsky et al. [10] can be used to compute an orientation that satisfies $\Omega(1/\log n) \cdot |P_{\text{small}}| = \Omega(k/\log n)$ pairs. □

An $(\tilde{\Omega}(1/\sqrt{k}), 1 + \epsilon)$*-approximation.* Even though our embedding-based algorithm improves on the one described in Section 3.1 by orders of magnitude, at least as far as the number of satisfied pairs is concerned, it uses paths that may be $\tilde{\Omega}(\log n)$-fold longer than needed. In the remainder of this section, we propose another direction for improvement, in which pairs are guaranteed to be $(1 + \epsilon)$-satisfied, for any required degree of accuracy $\epsilon > 0$. As it turns out, by resorting to ϵ-approximate paths, it is possible to satisfy $\tilde{\Omega}(1/k^{1/2})$ pairs, rather than $\Omega(1/\max\{n, k\}^{1/\sqrt{2}})$ as in the exact case.

Prior to formally describing our algorithm, it is worth pointing out that when a constant fraction of the pairs $(s_i, t_i) \in P$ are connected via very short paths, or more precisely, when $\delta_G(s_i, t_i) \leq 1/\epsilon$, the setting in question becomes very simple. In this case, a random orientation where the direction of each edge is picked at random, with equal probabilities for both options (independently of other edges), 1-satisfies each pair with probability at least $2^{-1/\epsilon}$. Therefore, the expected fraction of pairs that are satisfied is $\Omega(2^{-1/\epsilon})$. For this reason, we focus attention only on pairs for which $\delta_G(s_i, t_i) > 1/\epsilon$, and assume from this point on that all other pairs have already been discarded from P.

Let $\beta = \beta(n, k, \epsilon)$ be a parameter whose value will be optimized later on. As in the greedy algorithm, we use p_i to denote some shortest s_i-t_i path, arbitrarily picked in advance, and define $\mathcal{P} = \{p_i : (s_i, t_i) \in P\}$. Moreover, for a path $p \in \mathcal{P}$, let $I_p(\mathcal{P})$ be the set of paths in $\mathcal{P}$ that intersect p, i.e, share at least one common edge. With these definitions in place, our algorithm works in two phases:

1. As long as there exists a path $p \in \mathcal{P}$, say from s to t, such that $|I_p(\mathcal{P})| < \beta$:
   (a) Orient p in the direction from s to t.
   (b) Discard from $\mathcal{P}$ the path p as well as all paths in $I_p(\mathcal{P})$.
2. Once the condition in phase 1 is no longer satisfied, let p be the shortest among all paths in $\mathcal{P}$, connecting s to t.
   (a) Partition the path p into at most $1/\epsilon$ edge-disjoint subpaths, each of length at most $\lceil \epsilon \cdot \delta_G(s,t) \rceil \leq 2\epsilon \cdot \delta_G(s,t)$, where this inequality holds since $\delta_G(s,t) \geq 1/\epsilon$.
   (b) Identify a subpath $\tilde{p}$ for which $|I_{\tilde{p}}(\mathcal{P})| \geq (\epsilon/2) \cdot |I_p(\mathcal{P})| \geq \epsilon\beta/2$, and let r be some arbitrary vertex in $\tilde{p}$.
   (c) Construct an r-rooted shortest-path tree T in the subgraph that results from unifying $\tilde{p}$ and all paths in $I_{\tilde{p}}(\mathcal{P})$. At this point in time, we have just created an instance of the maximum tree orientation problem, where the underlying tree is T and the collection of pairs are those corresponding to the paths in $I_{\tilde{p}}(\mathcal{P})$. Hence, we can use the algorithm of [10] to compute an orientation that satisfies $\Omega(1/\log n) \cdot |I_{\tilde{p}}(\mathcal{P})| = \Omega(\epsilon\beta/\log n)$ pairs.

Obviously, all pairs that were connected in phase 1 are 1-satisfied, since these connections are due to exact shortest paths. For this reason, it remains to show that every connection in phase 2 uses a $(1+\epsilon)$-approximate shortest path. This follows from the next claim, where we derive an upper bound on the factor by which pairwise distances can grow in T (for the relevant subset of pairs).

Lemma 5. *For every path $p_i \in I_{\tilde{p}}(\mathcal{P})$ connecting s_i to t_i,*

$$\delta_T(s_i,t_i) \leq (1+4\epsilon) \cdot \delta_G(s_i,t_i) \ .$$

Proof. Consider some path $p_i \in I_{\tilde{p}}(\mathcal{P})$, and let y_{s_i} be its first vertex (in the direction from s_i to t_i) that also belongs to the subpath $\tilde{p}$. Similarly, let y_{t_i} be the last vertex in p_i that still resides in $\tilde{p}$. Since T is an r-rooted shortest path tree in the union of $\tilde{p}$ and all paths in $I_{\tilde{p}}(\mathcal{P})$, and since the entire length of $\tilde{p}$ is at most $2\epsilon \cdot \delta_G(s,t)$ and $\delta_G(s,t) \leq \delta_G(s_i,t_i)$, we must have

$$\begin{cases} \delta_T(r,s_i) \leq \delta_G(r,y_{s_i}) + \delta_G(y_{s_i},s_i) \leq 2\epsilon \cdot \delta_G(s_i,t_i) + \delta_G(y_{s_i},s_i) \\ \delta_T(r,t_i) \leq \delta_G(r,y_{t_i}) + \delta_G(y_{t_i},t_i) \leq 2\epsilon \cdot \delta_G(s_i,t_i) + \delta_G(y_{t_i},t_i) \end{cases}$$

These inequalities can now be used to prove the desired claim, since:

$$\begin{aligned} \delta_T(s_i,t_i) &\leq \delta_T(s_i,r) + \delta_T(r,t_i) \\ &\leq (2\epsilon \cdot \delta_G(s_i,t_i) + \delta_G(y_{s_i},s_i)) + (2\epsilon \cdot \delta_G(s_i,t_i) + \delta_G(y_{t_i},t_i)) \\ &\leq (\delta_G(s_i,y_{s_i}) + \delta_G(y_{s_i},y_{t_i}) + \delta_G(y_{t_i},t_i)) + 4\epsilon \cdot \delta_G(s_i,t_i) \\ &= \delta_G(s_i,t_i) + 4\epsilon \cdot \delta_G(s_i,t_i) \\ &\leq (1+4\epsilon) \cdot \delta_G(s_i,t_i) \ . \end{aligned}$$

□

We conclude the description of the algorithm by showing how to optimize the value of $\beta = \beta(n, k, \epsilon)$ such that it balances between the worst-case performances of phases 1 and 2.

Theorem 7. *For any fixed $\epsilon > 0$, there is a deterministic algorithm that $(1+\epsilon)$-satisfies a fraction of $\Omega(1/\sqrt{(k \log n)/\epsilon})$ of the pairs.*

Proof. Let D be the number of paths that were eliminated from $\mathcal{P}$ in phase 1. By the condition to terminate this phase, at least D/β of these paths must have been oriented so that the corresponding pairs are satisfied. In addition, as shown above, the number of $(1+\epsilon)$-satisfied pairs in phase 2 is $\Omega(\epsilon\beta/\log n)$. Therefore, the overall number of $(1+\epsilon)$-satisfied pairs is at least

$$\begin{aligned}\frac{D}{\beta} + \Omega\left(\frac{\epsilon\beta}{\log n}\right) &= \frac{1}{\beta} \cdot D + \Omega\left(\frac{\epsilon\beta}{(|P| - D)\log n}\right) \cdot (|P| - D) \\ &= \Omega\left(\min\left\{\frac{1}{\beta}, \frac{\epsilon\beta}{(|P| - D)\log n}\right\}\right) \cdot |P| \\ &= \Omega\left(\min\left\{\frac{1}{\beta}, \frac{\epsilon\beta}{k\log n}\right\}\right) \cdot k \ .\end{aligned}$$

To obtain the best-possible performance guarantee, we pick a value for β so as to maximize $\min\{\frac{1}{\beta}, \frac{\epsilon\beta}{k \log n}\}$. The latter term attains its maximal value at $\beta^* = \sqrt{(k \log n)/\epsilon}$. □

Acknowledgments. We would like to thank Ofer Neiman for valuable discussions and pointers regarding metric embeddings. RS was supported by a research grant from the Israel Science Foundation (grant no. 241/11).

References

[1] Abraham, I., Bartal, Y., Neiman, O.: Nearly tight low stretch spanning trees. In: Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science, pp. 781–790 (2008)

[2] Alon, N., Karp, R.M., Peleg, D., West, D.B.: A graph-theoretic game and its application to the k-server problem. SIAM Journal on Computing 24(1), 78–100 (1995)

[3] Elberfeld, M., Segev, D., Davidson, C.R., Silverbush, D., Sharan, R.: Approximation Algorithms for Orienting Mixed Graphs. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 416–428. Springer, Heidelberg (2011)

[4] Elkin, M., Emek, Y., Spielman, D.A., Teng, S.-H.: Lower-stretch spanning trees. SIAM Journal on Computing 38(2), 608–628 (2008)

[5] Fields, S.: High-throughput two-hybrid analysis. The promise and the peril. The FEBS Journal 272(21), 5391–5399 (2005)

[6] Gamzu, I., Segev, D., Sharan, R.: Improved Orientations of Physical Networks. In: Moulton, V., Singh, M. (eds.) WABI 2010. LNCS, vol. 6293, pp. 215–225. Springer, Heidelberg (2010)

[7] Gavin, A.-C., Bosche, M., Krause, R., Grandi, P., Marzioch, M., Bauer, A., Schultz, J., Rick, J.M., Michon, A.-M., Cruciat, C.-M., Remor, M., Hofert, C., Schelder, M., Brajenovic, M., Ruffner, H., Merino, A., Klein, K., Hudak, M., Dickson, D., Rudi, T., Gnau, V., Bauch, A., Bastuck, S., Huhse, B., Leutwein, C., Heurtier, M.-A., Copley, R.R., Edelmann, A., Querfurth, E., Rybin, V., Drewes, G., Raida, M., Bouwmeester, T., Bork, P., Seraphin, B., Kuster, B., Neubauer, G., Superti-Furga, G.: Functional organization of the yeast proteome by systematic analysis of protein complexes. Nature 415(6868), 141–147 (2002)

[8] Gitter, A., Klein-Seetharaman, J., Gupta, A., Bar-Joseph, Z.: Discovering pathways by orienting edges in protein interaction networks. Nucleic Acids Research 39(4), e22 (2011)

[9] Håstad, J.: Clique is hard to approximate within $n^{1-\mathrm{epsilon}}$. In: Proceedings of the 37th Annual Symposium on Foundations of Computer Science, pp. 627–636 (1996)

[10] Medvedovsky, A., Bafna, V., Zwick, U., Sharan, R.: An Algorithm for Orienting Graphs Based on Cause-Effect Pairs and Its Applications to Orienting Protein Networks. In: Crandall, K.A., Lagergren, J. (eds.) WABI 2008. LNCS (LNBI), vol. 5251, pp. 222–232. Springer, Heidelberg (2008)

[11] Silverbush, D., Elberfeld, M., Sharan, R.: Optimally Orienting Physical Networks. In: Bafna, V., Sahinalp, S.C. (eds.) RECOMB 2011. LNCS, vol. 6577, pp. 424–436. Springer, Heidelberg (2011)

[12] Vinayagam, A., Stelzl, U., Foulle, R., Plassmann, S., Zenkner, M., Timm, J., Assmus, H.E., Andrade-Navarro, M.A., Wanker, E.E.: A directed protein interaction network for investigating intracellular signal transduction. Science Signaling 4(189), rs8 (2011)

[13] Yeang, C.-H., Ideker, T., Jaakkola, T.: Physical network models. Journal of Computational Biology 11(2/3), 243–262 (2004)

Constant-Time Word-Size String Matching*

Dany Breslauer[1], Leszek Gąsieniec[2], and Roberto Grossi[3]

[1] Caesarea Rothschild Institute, University of Haifa, Haifa, Israel
[2] University of Liverpool, Liverpool, United Kingdom
[3] Dipartimento di Informatica, Università di Pisa, Pisa, Italy

Abstract. We present a novel string-matching algorithm that requires constant time for text scanning in an unusual model where (a) the input pattern and text are each packed into a single word, (b) the output is a one word bit-mask identifying the pattern occurrences in the text, and (c) there are constant-time arithmetic, bitwise, and shift instructions that operate on words whose size is proportional to the arbitrarily long input length. Our bit-parallelism techniques build upon and also greatly simplify existing parallel random access machine algorithms by using two "simple structure" rather than "small size" deterministic samples, i.e., one deterministic sample is very small (size two), while the other is a potentially very long prefix of the pattern. Pattern preprocessing takes time proportional to the word size. Our results also establish, by recent reductions, new bounds for the packed string matching problem.

1 Introduction

In modern computers, a machine ω-bit word is typically larger than the size in bits of an alphabet character and the machine level instructions are optimized to operate on whole words, i.e., 64-bit or longer words versus 8-bit ASCII, 16-bit UCS, 2-bit 4-character biological DNA, 5-bit 20-character amino acid alphabets, etc. Thus, it is quite natural to store strings where consecutive characters are packed into one larger word and compare the characters in bulk rather than compare them individually. Specifically, if the characters of a string are drawn from an alphabet Σ, then a word with at least $\omega \geq \log_2 n$ bits fits up to α packed characters as a base $|\Sigma|$ number, where the packing factor is $\alpha = \omega / \log_2 |\Sigma| \geq \log_{|\Sigma|} n$. We assume that $|\Sigma|$ is a power of two, ω is divisible by $\log_2 |\Sigma|$, and the packing factor $\alpha \leq \omega$ is a whole integer.

In this settings, it takes just $O(n/\alpha)$ time to read a packed string of length n, instead of the $O(n)$ time required to examine all of its characters individually. In the *packed string-matching problem*, both the pattern of length m and the text of length n are given packed: the idea of using the packed string representation in the string matching problem was considered already in early string matching

* Work partially supported by the European Research Council (ERC) project SFEROT, and by the Israeli Science Foundation Grants 686/07, 347/09 and 864/11, and by Italian project PRIN AlgoDEEP (2008TFBWL4) of MIUR.

papers by Knuth, Morris and Pratt [22, §4] and Boyer and Moore [6, §8-9] but they are far from reaching the optimal time bound of $O(n/\alpha + m/\alpha)$.

In this paper, we focus on the case where the whole pattern and text fit into a single word (i.e. $m, n \leq \alpha$) and so the output is a bit-mask representing all occurrences of the pattern in the text. Our main result is a constant-time word-size packed string-matching algorithm in the word RAM model, that has the standard constant-time arithmetic, bitwise, and shift instructions that operate on words whose size is proportional to the input length. Our new algorithm's text processing takes $O(1)$ time, but requires an $O(\alpha)$-time pattern preprocessing.

Our result has implications for the general version of the packed string-matching problem. In the known literature, the problem has also been considered by various authors in [1,2,4,5,13,15,16,24,25] to give efficient algorithm in theory and practice. Ben-Kiki et al. [3] show that the problem can be optimally solved in $O(n/\alpha + m/\alpha)$ time using constant auxiliary space and even in real time, using two *specialized* AC^0 constant-time word-size packed string instructions called WSLM and WSSM (e.g. in Intel's SSE4.2): in the absence of these specialized instructions, Ben-Kiki et al. [3] give emulation algorithms in the ω-word RAM model, as shown in Table 1, where the state of the art on the problem for the ω-word RAM is summarized, and *occ* denotes the number of occurrences.

Table 1. Comparison of packed string matching algorithms in the ω-word RAM

Time	Space	Reference
$O(\frac{n}{\log_{\lvert\Sigma\rvert} n} + n^{\varepsilon} m + occ)$	$O(n^{\varepsilon} m)$	Fredriksson [15,16]
$O(\frac{n}{\log_{\lvert\Sigma\rvert} n} + m + occ)$	$O(n^{\varepsilon} + m)$	Bille [5]
$O(\frac{n}{\alpha} + \frac{n}{m} + m + occ)$	$O(m)$	Belazzougui [2]
$O(\frac{n}{\alpha} + \frac{m}{\alpha} + occ)$	$O(1)$	Ben-Kiki et al. [3] WSSM & WSLM
$O(\frac{n \log\log \omega}{\alpha} + \frac{m}{\log_{\lvert\Sigma\rvert} n} + \omega + occ)$	$O(m^{\varepsilon} + \omega)$	Ben-Kiki et al. [3] emulation
$O(\frac{n}{\alpha} + \frac{m}{\log_{\lvert\Sigma\rvert} n} + \omega + occ)$	$O(m^{\varepsilon} + \omega)$	This paper

The motivation behind our study is now clear. Our new word-size string-matching algorithm in this paper can be employed in Ben-Kiki et al.'s [3] reduction, showing that the packed string matching problem can be solved in $O(n/\alpha)$ text scanning time in the ω-word RAM model. Here the word-size pattern preprocessing takes $O(m/\log_{\lvert\Sigma\rvert} n + \alpha)$ time, but this overhead is only incurred once, resulting in an $O(n/\alpha + m/\log_{\lvert\Sigma\rvert} n + \alpha)$ time packed string-matching algorithm.

Our result borrows techniques from algorithms in the parallel random access machine model, and surprisingly, also greatly simplifies on the existing string-matching algorithms in that model [7,8,9,10,11,12,18,19,20,21,26,27]. Perhaps, it is worthwhile to note that our algorithm uses integer multiplication, which is not an AC^0 operation, since it can be used to compute the parity [17], while the string matching problem is in AC^0. We also contribute in this paper with a constant-time algorithm for finding the most significant bit in a word: this cannot be easily derived from the least significant bit as we do not know how to bitwise reverse a word in $O(1)$ time.

2 Basic Concepts

Period. A string u is *a period* of a string x, if x is a prefix of u^k for some integer k, or equivalently if x is a prefix of ux. Thus, x may be written as $u^l v$, where v is prefix of x's period u. The shortest period of x is called *the period* of x and its length is denoted by $\pi(x)$. Galil [18] and all subsequent parallel string-matching algorithms reduce the search for a highly periodic pattern $x = u^l v$, such that $l > 2$, to a search for occurrences of the pattern prefix $u^2 v$, that is followed by counting runs of consecutive occurrences of $u^2 v$ that must fall in an arithmetic progression, by the *periodicity lemma*. Breslauer and Galil [9] showed that $\Omega(\log \log n)$ time is required by any optimal parallel algorithm that finds the pattern's period length.

Witness. If the pattern x is not a prefix of yx, namely y is not a period of x, then there must be at least one character mismatch between x and the prefix of yx of length $\pi(x)$; such a mismatch is called *a witness* for the non-periodicity of length $|y|$ and it exists for all length $|y|$, such that $1 \leq |y| < \pi(x)$. A witness may be used to eliminate at least one of two close-by occurrences candidates in a process called a *duel*, where a text symbol that is aligned with the witness is used to eliminate at least one of the two occurrence candidates, or even both, as that text symbol cannot be equal simultaneously to the two different pattern symbols. Vishkin [26] introduced witnesses in an optimal $O(\log n)$ time parallel string-matching algorithm, improving on an earlier alphabet dependent result by Galil [18], and Breslauer and Galil [8] subsequently used witnesses to improve the time further to $O(\log \log n)$.

Deterministic Sample. A *k-deterministic sample* for the pattern x is a small set of locations DS in x such that if we verify that the pattern occurrence candidate matches all the text symbols aligned at the locations in DS, then no other occurrence candidates that are closer than k locations to each other are plausible; such occurrence candidates are not entirely eliminated by verifying the symbols in DS, but rather must be still sparsified by eliminating candidates that are too close-by to each other. Vishkin [27] introduced deterministic samples and proved that a $\pi(x)$-deterministic sample of size $|DS| \leq \log \pi(x)$ always exist, in an optimal parallel algorithm that has faster optimal $O(\log^* n)$ time text processing, but slower optimal $O(\log^2 n)$ pattern preprocessing. Galil [19] improved the text processing to $O(1)$ time and Crochemore et al. [11] used constant-size $\log \log \pi(x)$-deterministic samples to improve the pattern processing time. Ben-Kiki et al. [3] used deterministic samples in the bit-parallel settings, to get a constant-time word-size packed string-matching algorithm that uses $O(\omega \log \log \alpha)$ bit words.

Word-size RAM Operations. Consider the set of binary words $\mathcal{B} = \{0, 1\}^\omega$ of length ω. In our fast solution to text search we use a number of constant time operations defined on whole words from $\mathcal{B}$ or their partitions into consecutive blocks of uniform length. We assume also that indices of bits in words drawn from $\mathcal{B}$ are enumerated from $w - 1$ down to 0, counting from the left. This notation is adopted to reflect a natural representation of polynomials. To be precise, any word $A \in \mathcal{B}$ can be interpreted as the polynomial $P_A(x) = A_{\omega-1} \cdot x^{\omega-1} + A_{\omega-2} \cdot x^{\omega-2} + \ldots + A_1 \cdot x^1 + A_0 \cdot x^0$, where $V(A)$ is defined as $P_A(2)$. For this reason we will also refer to elements of $\mathcal{B}$ as *vectors*. We will need the following operations defined on words from $\mathcal{B}$:

(1) $X = \texttt{and}(A, B)$ - *bit AND*, where $X_i = (A_i \wedge B_i)$, for all $i = 0, .., \omega - 1$.
(2) $X = \texttt{or}(A, B)$ - *bit OR*, where $X_i = (A_i \vee B_i)$, for all $i = 0, .., \omega - 1$.
(3) $X = \texttt{neg}(A)$ - *bit negation*, where $X_i = \neg A_i$, for all $i = 0, .., \omega - 1$.
(4) $X = \texttt{xor}(A, B)$ - *bit exclusive OR*, where $X_i = (\neg A_i \wedge B_i) \vee (A_i \wedge \neg B_i)$, for all $i = 0, .., \omega - 1$.
(5) $X = \texttt{add}(A, B)$ - *addition*, where X satisfies $V(X) = V(A) + V(B)$.
(6) $X = \texttt{sub}(A, B)$ - *subtraction*, where X satisfies $V(X) = V(A) - V(B)$.
(7) $X = \texttt{mul}(A, B)$ - *multiplication*, where X satisfies $V(X) = V(A) \cdot V(B)$.
(8) $X = \texttt{shl}(A, k)$ - *shift left* by k positions, where X satisfies $V(X) = V(A) \cdot 2^k$.
(9) $X = \texttt{shr}(A, k)$ - *shift right* by k positions, where X satisfies $V(X) = \lfloor V(A)/2^k \rfloor$.
(10) $X = \texttt{rmo}(A, k)$ - rightmost 1 in each consecutive block of size k, i.e., $X_i = 1$ iff $A_i = 1 \wedge A_{i-l} = 0$, for all $0 \leq l < (i \bmod k)$.
(11) $X = \texttt{lmo}(A, k)$ - leftmost 1s in each consecutive block of size k, i.e., $X_i = 1$ iff $A_i = 1 \wedge A_{i+l} = 0$, for all $(i \bmod k) < l < k$.

It has been shown in [14,23] that operations (1-10) can be implemented in $O(1)$ time in the model adopted in this paper. One can also derive from [14] the following lemma.

Lemma 1. *Any word in* $\mathcal{B}$ *of the form* $(0^a(0^b1^c0^d)^e0^f)$, *for non-negative integer values* a, b, c, d, e, f, *s.t.,* $a + f + e(b + c + d) = \omega$ *can be generated in* $O(1)$ *time.*

In contrast, a fast implementation of operation (11), i.e., finding the leftmost (most significant) 1s in consecutive non-overlapping blocks of uniform size is not known. As one of the contributions of this paper we discuss the implementation of this operation in $O(1)$ time in Section 4.

3 Word-Size Text Search

In the bit-parallel setting, we need an algorithm that may easily map to the standard primitives in the ω-word RAM model. Inspiration comes from parallel algorithms: we describe here a new text scanning algorithm with a simple and regular memory access pattern, which in turn, also greatly simplifies on previous work. The resulting algorithm bares some resemblance to Gąsieniec et al.'s [20] sequential algorithm and to Crochemore et al.'s [11] parallel algorithm.

Slub. Our approach is based on two stage deterministic samples whose size is not necessarily small, as given by the following definition.

Definition 1. *Given a string* x *of period length* $\pi(x)$, *we say that the substring* z *is a* slub *in* x *if there exist two distinct symbols* a *and* b *such that*

1. both za and zb occur in x, but,
2. za occurs only once among the first $\pi(x)$ locations in x.

Note that za may occur also elsewhere in x among locations $\pi(x)+1,\ldots,|x|$: this is not in contrast with the fact that it occurs only once among $1,\ldots,\pi(x)$. Also, if x contains just one distinct symbol, i.e. $x = a^m$ for a symbol a, the string-matching problem is trivially solved. Hence, we assume that $x \neq a^m$ for all symbol a and show next that a slub always exists in x: we actually choose z to be a pattern *prefix*, but other choices are possible. The length of the slub may be anything between 0 and $|x|-1$. Observe that any string that occurs in x at location $i \geq \pi(x)+1$ must also occur starting at location $i-\pi(x)$.

Lemma 2. *If $x \neq a^m$ for all symbol a, there is a prefix z of x that is a slub.*

Proof. Let za be the shortest prefix of x that occurs only once among the first $\pi(x)$ locations of x. Such shortest prefix clearly exists since $za = x$ only occurs once in x. Since $x \neq a^m$, the shorter prefix z occurs more than once in the first $\pi(x)$ location and at least one such occurrence is some zb, $a \neq b$. Alternatively, if we consider the suffix tree built on x without any terminating symbol, then z is the branching parent node of the leaf x: clearly, za is a prefix of x and some zb, $a \neq b$, also appears in x. □

For example, the pattern $x = a^n b$ has a slub $z = a^{n-1}$ that appears as its prefix followed by the symbol a and starting at the second location followed by the symbol b. The pattern ab^n has a slub that is the empty string $z = \epsilon$ that appears at its beginning followed by the symbol a and starting at all other locations followed by the symbol b.

Lemma 3. *Given a slub z and symbols a and b, the deterministic sample consisting of the locations of the symbols a and b in x is a $|z|+1$-deterministic sample of size 2.*

Proof. Observe that any occurrence candidate starting fewer than $|z|+1$ locations *after* an occurrence candidate with a verified deterministic sample $\{a, b\}$ may be eliminated, since a character in z that is aligned with a in one z instance must match b in the other z instance, leading to non-constructive evidence that such an occurrence candidate may be eliminated. See Figure 1-a. □

Goldberg and Zwick [21] also used larger deterministic samples, but our new algorithm is unusual in that it uses only two "simple structure" rather than "small size" deterministic samples, i.e., the first deterministic sample is very small and eliminates occurrence candidates via non-constructive evidence, while the second deterministic sample is a potentially very long prefix of the pattern.

The algorithm. Let $x \neq a^m$ be the input pattern with period length $\pi(x)$—and so $x = u^l v$ where u is made up of the first $\pi(x)$ symbols of x—and z be its slub as in Lemma 2. Slubs may be used to obtain a simple parallel string matching algorithm that works by starting out from all the n text positions as candidates:

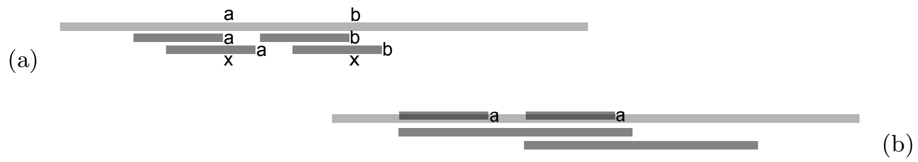


Fig. 1. The new string-matching algorithm using slubs: (a) sparsify to one candidate in each $|z|+1$ block: after verifying the size-2 deterministic sample $\{a,b\}$, any surviving occurrence candidate that has another close-by candidate on its left, may be eliminated. (b) sparsify to one candidate in each period length block: after verifying the deterministic sample za, any surviving occurrence candidate that has another close-by candidate on its right, may be eliminated.

1. For each occurrence candidate, verify the size-2 deterministic sample for a and b. In each block of length $|z|+1$, eliminate all remaining occurrence candidates except for the *leftmost* viable candidate by Lemma 3. Note that $O(n/(|z|+1))$ candidates remain.
2. For each remaining occurrence candidate, verify the pattern prefix za. In each block of length $\pi(x)$ eliminate all the remaining occurrence candidates except for the *rightmost* viable candidate since za is unique within the first $\pi(x)$ pattern locations. Note that $O(n/\pi(x))$ candidates survive. See Figure 1-b.
3. For each surviving occurrence candidate, verify the pattern prefix u of length $\pi(x)$. All the occurrences of the period u of x are known at this point.
4. Count periodic runs of u; verify pattern tail v (a prefix of u) too if it is last in the current run.

Theorem 1. *There exist a word-size string-matching algorithm that takes $O(1)$ time in the ω-word RAM, following pattern preprocessing.*

The proof of Theorem 1 is in the implementation of the steps 1–4 of the basic algorithm via bit-parallel techniques as outlined next. We assume without loss of generality that the pattern x and the text y are binary strings of length at most ω with only 0 and 1 symbols. The pattern preprocessing is described in Section 5 and finds the *period length* $\pi(x)$ of the pattern and the *slub* z with deterministic sample $\{a,b\}$.

Step 1. We first need to verify the small deterministic sample for the slub z. Let $i_a < i_b$ be the indices of the deterministic sample symbols a and b in the pattern x. Without loss of generality suppose that $a=1$ and $b=0$. We copy x and complement it into x'. We then perform a left shift `shl` of x' by $d = i_b - i_a$ locations, so as to align the symbols a and b that need to be verified. We then perform a bitwise `and` followed by a left shift `shl` by i_a locations to obtain r, so that the resulting 1s in r mark the candidate occurrences.

At this point, we need to $(|z|+1)$-sparsify these candidate occurrences in r. We conceptually divide r into blocks of length $|z|+1$ and keep only the leftmost candidate in each block using the bitwise `lmo` operation described in Section 4.

Note that since candidates in consecutive blocks might not be $|z|+1$ apart, we consider odd and even blocks separately and remove candidates that have a near neighbor. To do this, we employ a binary mask $1^{|z|+1}0^{|z|+1}1^{|z|+1}0^{|z|+1}\ldots$ in bitwise `and` with r for odd blocks, and repeat the same for even blocks. We make a bitwise `or` of the outcomes, storing it into r. As a result, the remaining candidates in r are now represented by 1s separated by at least $|z|$ 0s.

Step 2. We can now verify the sparse candidate occurrences marked in r against za, namely, check if each of these candidate occurrence starts with the prefix za, when the candidates are at least $|z|+1$ locations apart. We again proceed by odd and even blocks, so let us assume without loss of generality that r contains only the candidate occurrences in its odd blocks to avoid interference with those in the even blocks. Consider the word p which contains za followed by a run of 0s. If we multiply p and r and store the result in c, we obtain that c has a copy of za in each location corresponding to a marked location of r, while the rest are 0s. If we perform a right shift `shr` of r by $|z|+1$ locations and make it in `or` with y, storing the result in y', we have that each potential occurrence of za in y is also an occurrence in y' but terminated by 1. Since the bits in r at least $|z|+1$ apart and we are considering the candidate occurrences in the odd blocks, we get za at the candidate occurrences without interference in this way. Then, we perform a bitwise `xor` between c and y', storing the result in d. Consider now an odd block and its next (even) block: for the current candidate in location i, there is an occurrence of za if and only if the nearest 1 is in position $i+|z|+1$. We conceptually divide d into larger blocks of length $2(|z|+1)$ and keep only the leftmost candidate in each larger block using the bitwise `lmo` operation described in Section 4. We then perform a left shift of d by $|z|+1$ positions and store the result in s, so now each 1 in s marks an occurrence of za.

At this point, we need to $\pi(x)$-sparsify these candidate occurrences in s. This is logically done the same way as the $|z|+1$ sparsification above, only keeping the rightmost surviving candidate in each block of size $\pi(x)$ through the bitwise `rmo` operation described in Section 4.

Step 3. We can now verify the sparse candidate occurrences marked in s again the period u of x, namely, check if each of these candidate occurrence starts with the prefix u, when the candidates are at least $\pi(x)$ locations apart. This step is done the same way as the verification again the prefix za in Step 2, using the pattern period u instead of za.

Step 4. Recall that the pattern is $x = u^l v$. If $l = 1$, we can also check v in a similar way we did for u, and complete our task by suitably `and`ing the results. Hence we focus here on the interesting case $l \geq 2$. While general counting is difficult in our setting, our task is simpler since occurrences mean that the periods are lined up in an arithmetic progress.

We first select the candidates in which u^2v occurs: note that this is a minor variation of what discussed so far for u and uv. Hence, let t be the word in which each 1 marks an occurrence of u^2v. We filter the runs of these occurrences by storing in t the bitwise `or` of two words: the former is obtained by putting t in `and` with its left shift `shl` by $\pi(x)$ positions; the latter is obtained by putting t in `and` with its right shift `shr` by $\pi(x)$ positions. At this point, the word t thus obtained has 1 in correspondence of aligned occurrences of u^2v, and we have runs of 1s at distance $\pi(x)$ from each other inside each run. All we have to do is to remove the last $l-1$ 1s of each run in t, since they are shorter than the pattern: the remaining ones are the pattern occurrences. Summing up: counting is not possible, but removing those shorter than the pattern is doable.

To avoid interferences we consider blocks of $l \times \pi(x)$ bits and work separately on the odd and even blocks, as already discussed before. First mark the last occurrence of u^2v inside each run of t. This is done by making the `xor` of t with itself shifted right `shr` by $\pi(x)$ locations, and with the result shifted left `shl` by $\pi(x)$ locations and put in `and` with t, storing the outcome in q. Now the 1s in q corresponds to the last occurrence of u^2v inside each run of t. If we multiply the word $(10^{\pi(x)-1})^{l-1}$ (followed by 0s) by q, and we store in q the result shifted left `shl` by $(l-2) \times \pi(x)$ locations, we capture the last $l-1$ entries of each run, exactly those to be removed. At this point, if we complement q and make an `and` with t, storing the result in t, we obtain that the 1s in t finally corresponds to the occurrences of x in y.

4 Implementing `lmo` Operation

We show how to implement the $\texttt{lmo}(A, k)$ operation, that was defined in Section 3 and constitutes an integral part of our fast text search procedure, in constant time in the ω-word RAM. Knuth [23] observes that "big-endian and little-endian approaches aren't readily interchangeable in general, because the laws of arithmetic send signals leftward from the bits that are least significant" and therefore assumes that a less traditional bit reversal AC^0 instruction, that can be easily implemented in hardware, is available in his model: such instruction would trivially map between `lmo` and `rmo` operations. Related endian conversion byte reversal instructions, often used in network applications to convert between big and little endian integers, are available in contemporary processors.

Given a word $A \in \mathcal{B}$ we are asked to determine the leftmost (the most significant) 1 in each consecutive k-block K_j of A, for $j = 0, \ldots, \omega/k - 1$. Each block K_j is formed of contiguous k positions $(j+1) \cdot k + 1, \ldots, j \cdot k$. We propose the solution when the size of the consecutive blocks $k = q^2$ without loss of generality, for some integer $q > 0$. One of the main ideas behind the solution lies in partitioning of each block K_j into consecutive sub-blocks $K_j^{q-1}, \ldots, K_j^0$, each of length $q = \sqrt{k}$. If a block (sub-block) contains 1s we say that this is a non-zero block (sub-block). The `lmo` algorithm operates in three stages.

During *Stage 1* we identify all non-zero sub-blocks K_j^i. Later in *Stage 2* we identify in each K_j the leftmost non-zero sub-block $K_j^{i^*}$. Finally, in *Stage 3* we identify the leftmost 1 located at position $j \cdot k + i^* \cdot q + l^*$ in each sub-block $K_j^{i^*}$ which also refers to the leftmost 1 in K_j.

STAGE 1 (DETERMINE ALL NON-ZERO SUB-BLOCKS)

During this stage we identify all non-zero sub-blocks K^i_j in each K_j. More precisely, for the input word A we compute a word $B \in \mathcal{B}$, in which each non-zero sub-block K^i_j in A, for $i = 0, \ldots, q-1$ and $j = 0, \ldots, \omega/k-1$, is identified in B by the most significant (leftmost) bit $j \cdot k + i \cdot q + q - 1$ in K^j_i set to 1 in B. The remaining bits in B are set to 0. A more detail description of this stage follows.

Part A: Extract the most significant bits in sub-blocks. For a rather technical reason, we will extract first the most significant bit in each sub-block K^j_i of A. We store extracted bits in a separate word X replacing the corresponding positions in A by 0s to form a new word A_1. This process is performed in four steps:

A.1 Create a bit mask $M_1 = (10^{q-1})^{\omega/q}$, see Lemma 1.
A.2 $X = \texttt{and}(A, M_1)$, where X contains the most significant bits extracted from sub-blocks.
A.3 $M_2 = \texttt{neg}(M_1)$, where $M2$ is built to extract all but the most significant bits in sub-blocks.
A.4 $A_1 = \texttt{and}(A, M_2)$, where A_1 is the requested new word.

Part B: Determine empty (zero) sub-blocks. During this part we create a new word A_2 in which the most significant bit in each sub-block is set to 1 if the remaining $k-1$ bits in this sub-block in A contain 0s. This process is performed in four steps:

B.1 $A^- = \texttt{neg}(A)$, where A^- contains negated bits of A.
B.2 $A' = \texttt{and}(A^-, M_2)$, where A' is obtained from A^- by removal of the most significant bit in each sub-block.
B.3 Create a bit mask $M_3 = (0^{q-1}1)^{\omega/q}$, see Lemma 1.
B.4 $A_2 = \texttt{and}(\texttt{add}(A', M_3), M_1)$, where A_2 is the requested new word.

Part C: Determine non-empty (non-zero) sub-blocks. Non-zero sub-blocks in A are represented by 1s present at the leftmost bits of these sub-blocks in B and all other bits set to 0s. The process is performed in two steps:

C.1 $B = \texttt{and}(\texttt{neg}(A_2), M_1)$, where A_3 identifies sub-blocks with 1s located outside of the most significant bits.

STAGE 2 (DETERMINE IN EACH BLOCK THE LEFTMOST NON-ZERO SUB-BLOCK)

The second stage is devoted to computation of the leftmost non-zero sub-block $K^{i^*}_j$ in each block K_j, for $j = 0, \ldots, \omega/k-1$ on the basis of word B.

More precisely, for the input word B (obtained on the conclusion of stage 1) we compute a word $C \in \mathcal{B}$, in which each the leftmost non-zero sub-block $K^{i^*}_j$, for $j = 0, \ldots, \omega/k-1$, is identified in C by the most significant (leftmost) bit $j \cdot k + i^* \cdot q + q - 1$ in $K^{i^*}_j$ set to 1 in C. The remaining bits in C are set to 0. A more detail description of this stage follows.

In order to make computation of the leftmost non-zero sub-block viable, more particularly to avoid unwanted clashes of 1s, we consider separately three different words based on B. In each of the three words the bits of selected blocks are *culled*, i.e., reset to 0s as follows, where $B[K_j]$ denotes the segment of B corresponding to block K_j.

(1) $B_L = B[K_{\frac{\omega}{k}-1}]0^{2k}B[K_{\frac{\omega}{k}-4}]0^{2k}...B[K_2]0^{2k}$,
(2) $B_M = 0^kB[K_{\frac{\omega}{k}-2}]0^{2k}B[K_{\frac{\omega}{k}-5}]0^{2k}...B[K_1]0^k$, and
(3) $B_R = 0^{2k}B[K_{\frac{\omega}{k}-3}]0^{2k}B[K_{\omega k-6}]0^{2k}...B[K_0]$.

In each of these 3 words, the blocks that are spared from resetting are called *alive blocks*. Without loss of generality (computation on other two words and their alive blocks are analogous) lets show how to compute the most significant non-zero sub-blocks in alive blocks in B_R.

Part D: Fishing out B_R

D.1 Create a bit mask $M_4 = (0^{2k}1^k)^{n/3k}$, see Lemma 1.
D.2 $B_R = \mathtt{and}(B, M_4)$, where all bits irrelevant to B_R are culled.

The following lemma holds:

Lemma 4. *The only positions at which 1s may occur in B_R refer to the leftmost bits in sub-blocks of alive blocks, i.e., $3j \cdot k + iq - 1$, for all $j = 0, \ldots, \frac{\omega}{3k} - 1$ and $i = 1, \ldots, q$.*

Proof. The proof follows directly from the definition of B and M_4. □

Part E: Reversing the leftmost bits of sub-blocks in each block. During this step we multiply and shuffle 1s of B_R. To be precise, we compute the product of B_R and the shuffling palindrome of the form $P = (10^q)^{q-1}1$, where $Y = \mathtt{mul}(B_R, P)$. The main purpose of this process is to generate a sequence of bits from B_R that appear in the reverse order in Y. These bits can be later searched for the rightmost bit using the procedure `rmo` that already has efficient implementation. We need a couple of more detail observations.

Lemma 5 (Global independence of bits). *In the product of B_R and P there is no interference (overlap) between 1s coming from different alive blocks.*

Proof. Since (alive) blocks with 1s in B_R are separated by the sequence of 0s of length $\geq 2k$ and $|P| \leq k$ any 1s belonging to different alive blocks are simply too far to interfere during the multiplication process. □

Lemma 6 (Local independence of bits). *Only one pair of 1s can meet at any position of the convolution vector of the product B_R and P.*

Proof. Observe that bits from the two words meet at the same position in the convolution vector if the sum of the powers associated with the bits are the same.

Recall that 1s in B_R in jth alive block may occur only at positions $3j \cdot k + iq - 1$, for $i = 1, \dots, q$, and all 1s in P are located at positions 0 and $l(q+1)$, for $l = 0, \dots, q-1$. Assume by contradiction that there exist $1 \le i_1 < i_2 \le q$ and $0 \le l_1 < l_2 \le q-1$, s.t., $3j \cdot k + i_2 q - 1 + l_1(q+1) = 3j \cdot k + i_1 q - 1 + l_2(q+1)$. The latter equation is equivalent with $(l_2 - l_1)(q+1) = (i_2 - i_1)q$. Since we know that q and $q+1$ are relatively prime and $l_2 - l_1$ and $i_2 - i_1$ are both smaller than q we can conclude that the equivalence cannot hold. □

We focus now our attention on the specific contiguous chunks of bits in Y that contain the leftmost bits of sub-blocks in the same alive block arranged in the reverse order. Consider jth alive block in B_R (note that now i ranges form 1 to q).

Lemma 7 (Reversed bits). *The bit $3j \cdot k + iq - 1$, for $i = 1, \dots, q$, from the jth alive block in B_R appears in $Y = \mathtt{mul}(B_R, P)$ at position $(3j+1) \cdot k + q - (i+1)$.*

Proof. The proof comes directly from the definition of the product of two vectors and the content of B_R and P. In particular, for $i = 1, \dots q$ the bit $3j \cdot k + iq - 1$ in B_R meets 1 at position $(q-i)(q+1)$ in P and it gets replicated at position $3j \cdot k + iq - 1 + (q-i)(q+1) = 3j \cdot k + iq - 1 + q^2 + q - iq - i = (3j+1)k + q - (i+1)$. □

The sequence of bits $(3j+1)k+q-(i+1)$ in Y, for $i = 1, \dots q$, and $j = 0, \dots, \frac{\omega}{3k} - 1$, is called the *$j$-significant sequence*. The main purpose of the next step is to extract the significant sequences from Y and later to determine the rightmost 1 in each j-significant sequence. Note that the rightmost 1 in j-significant sequence corresponds to the leftmost 1 in jth sub-block.

Part F: Find the rightmost bits in significant sequences

F.1 Create bit mask $M_5 = 0(0^{3k-q}1^q)^{\frac{\omega}{3k}}0^{k-1}$ by Lemma 1, where 1s in this bit-mask correspond to the positions located in significant sequences in Y.

F.2 Y_1= $\mathtt{and}(Y, M_5)$, where all positions outside of the significant sequences are culled.

F.3 Y_2= $\mathtt{rmo}(Y_1, 3k)$ where `rmo` operation computes the rightmost bit in each significant sequence of Y_1.

When the rightmost 1 in each significant sequence is determined and stored in Y_2 we apply the shuffle (reverse) mechanism once again. In particular, we use multiplication by palindrome P and shift mechanism to recover the original position of the leftmost 1 (representing the leftmost non-zero sub-block) in each active block of B_R.

Part G: Find the leftmost non-zero sub-block in each active block Consider the jth active block and assume that its i^* sub-block is the leftmost non-zero sub-block in this block, where $1 \le i^* \le q$. This sub-block is represented by 1 located at position $3jk + i^*q - 1$ in B_R. According to Lemma 7 after Part E this 1 is present in Y, and in turn in Y_2, at position $(3j+1)k + q - (i^* + 1)$.

G.1 Compute Y_3= $\texttt{mul}(Y_2, P)$, where the bits are shuffled again.
G.2 Compute bit mask M_6= $(0^{2k}(10^{q-1})^q)^{\frac{\omega}{3k}}0^{k+q}$, see Lemma 1.
G.3 Y_4= $\texttt{and}(Y_3, M_6)$, where we fish out bits of our interest.
G.4 C_R= $\texttt{shr}(Y_4, k+q)$, where we reinstate the original location of bits.

During multiplication of Y_2 by P (step G.1) we focus on the instance of this 1 in the product vector $Y_3 = \texttt{mul}(Y_2, P)$ generated by 1 located at position $i^*(q+1)$ in P. This specific instance of 1 from Y_2 is located at position $(3j+1)k+q-(i^*+1)+i^*(q+1) = (3j+1)k+q(i^*+1)-1$. This last expression can be also written as $3jk+qi^*-1+(k+q)$.

We still need to cull unwanted 1s resulting from multiplication of Y_2 and P. In order to do this, we define a bit mask $M_6 = (0^{2k}(10^{q-1})^q)^{\frac{\omega}{3k}}0^{k+q}$ (Steps G.2) in which 1s are located at positions $3jk+ql-1+(k+q)$, for $j = 0, \ldots \frac{\omega}{3k}-1$ and $l = 1, \ldots, q$. And we apply this mask to vector Y_3 (Step G.3).

Lemma 8. *The 1s in M_6 are only those located at position $3jk+qi^*-1+(k+q)$ for any choice of i^*.*

Proof. We need to show that other 1s in Y_3 obtained on the basis of 1s located at position $3jk+i^*q-1$ in B_R and 1s in P at positions $i(q+1)$, for $i \neq i^*$, will not survive the culling process. Assume to contrary that there exists 1 in Y_3 formed on the basis of 1 located at the position $(3j+1)k+q-(i^*+1)$ in Y_2 and 1 located at a position $i(q+1)$ in P, for $i \neq i^*$. The location of this 1 in Y_3 is $(3j+1)k+q-(i^*+1)+i(q+1) = 3jk+qi+(i-i^*)-1+(k+q)$. But since $0 < |i-i^*| < q$ the location of this 1 cannot overlap with 1s in M_6. □

We conclude the process (Step G.4) by shifting Y_4 by $k+q$ positions to the right. This is required to relocate the leftmost 1s in the leftmost non-zero sub-blocks to the correct positions. The computations on sequences B_L and B_M can be performed analogously to form C_L and C_M. We conclude by forming C= $\texttt{or}(\texttt{or}(C_L, C_M), C_R)$ which is the combination of C_L, C_M and C_R.

STAGE 3 (FINDING THE LEFTMOST 1 IN EACH BLOCK)

The third stage is devoted to computation of the leftmost 1 in sub-block $K_j^{i^*}$ in each K_j, for $j = 0, \ldots, \frac{\omega}{k}-1$ on the basis of C.

More precisely, for the input word C (obtained on the conclusion of Stage 2) containing information on location of $K_j^{i^*}$ in each block K_j we compute a word $D \in \mathcal{B}$, in which the leftmost 1 in $K_j^{i^*}$ located at position $j \cdot k + i^* \cdot q + l^*$ in $K_j^{i^*}$ is present in D while the remaining bits in D are set to 0. Note that the selected 1s stand also for the leftmost 1s in blocks, i.e., they form the solution to our problem.

The operations used in Stage 3 are analogous to those used in Stage 2. Due to limited space the detail description of this stage is omitted here.

5 Word-Size Pattern Preprocessing

The pattern preprocessing is the more difficult part in parallel string matching algorithms and also in our case using bit-parallelism: as demonstrated by

Breslauer and Galil's [9] $\Omega(\log \log n)$ period computation lower bound and the faster text processing by Vishkin [27] and Galil [19]. We assume that the preprocessing is done in $O(m/\alpha + \omega)$ time by a purely sequential algorithm, with additional $O(\frac{m}{\log_{|\Sigma|} n})$ for the WSLM emulation required in [3]. Observe that the period may be computed in the ω-word RAM model in constant time using larger $O(\omega \log \omega)$-bit words, or in $O(\log \omega)$ time using ω-bit words.

6 Parallel Random Access Machine

The new algorithm presented in Section 3 offers a much simpler constant time CRCW parallel random access machine algorithm than Galil [19], Crochemore et al. [11] and Goldberg and Zwick [21]. The pattern preprocessing is composed of two parts: the first is the witness computation as in Breslauer and Galil's [8] work, and the second computes the information needed by the new algorithm in this paper in constant time, but only for the pattern prefix of length $\sqrt{m}$. Applying first the algorithm to the pattern prefix of length $\sqrt{m}$ and using periodicity properties, we can sparsify the remaining candidate occurrences to one in every $\sqrt{m}$ block, and then use witnesses to eliminate the rest.

7 Conclusions

We presented a theoretical way of simulating a string-matching instruction already available in modern machines on the ω-bit word RAM. There is a number of questions to be answered. We are curious whether there exist constant-time word-size packed string-matching algorithms in the ω-bit word RAM model using only AC^0 instructions, i.e., no integer multiplication. As a first step, it would be interesting to obtain an algorithm where integer multiplication is only used in the pattern processing, but not in the text preprocessing. We are also curious whether the parallel random access machine pattern preprocessing can be done faster than $O(\log \log n)$ if the alphabet is small integers [7].

References

1. Baeza-Yates, R.A.: Improved string searching. Softw. Pract. Exper. 19(3), 257–271 (1989)
2. Belazzougui, D.: Worst Case Efficient Single and Multiple String Matching in the RAM Model. In: Iliopoulos, C.S., Smyth, W.F. (eds.) IWOCA 2010. LNCS, vol. 6460, pp. 90–102. Springer, Heidelberg (2011)
3. Ben-Kiki, O., Bille, P., Breslauer, D., Gąsieniec, L., Grossi, R., Weimann, O.: Optimal Packed String Matching. In: Proc. FSTTCS. LIPIcs, vol. 13, pp. 423–432. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
4. Klein, S.T., Kopel Ben-Nissan, M.: Accelerating Boyer Moore Searches on Binary Texts. In: Holub, J., Žďárek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 130–143. Springer, Heidelberg (2007)
5. Bille, P.: Fast searching in packed strings. J. Discrete Algorithms 9(1), 49–56 (2011)

6. Boyer, R., Moore, J.: A fast string searching algorithm. Comm. of the ACM 20, 762–772 (1977)
7. Breslauer, D., Czumaj, A., Dubhashi, D.P., Meyer auf der Heide, F.: Comparison Model Lower Bounds to the Parallel-Random-Access-Machine. Inf. Process. Lett. 62(2), 103–110 (1997)
8. Breslauer, D., Galil, Z.: An optimal $O(\log\log n)$ time parallel string matching algorithm. SIAM J. Comput. 19(6), 1051–1058 (1990)
9. Breslauer, D., Galil, Z.: A Lower Bound for Parallel String Matching. SIAM J. Comput. 21(5), 856–862 (1992)
10. Cole, R., Crochemore, M., Galil, Z., Gąsieniec, L., Hariharan, R., Muthukrishnan, S., Park, K., Rytter, W.: Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions. In: Proc. FOCS, pp. 248–258 (1993)
11. Crochemore, M., Galil, Z., Gąsieniec, L., Park, K., Rytter, W.: Constant-Time Randomized Parallel String Matching. SIAM J. Comput. 26(4), 950–960 (1997)
12. Czumaj, A., Galil, Z., Gąsieniec, L., Park, K., Plandowski, W.: Work-time-optimal parallel algorithms for string problems. In: Proc. STOC, pp. 713–722. ACM (1995)
13. Faro, S., Lecroq, T.: Efficient pattern matching on binary strings. In: Proc. SOFSEM (2009)
14. Fich, F.E.: Constant Time Operations for Words of Length w. Technical report, University of Toronto (1999), http://www.cs.toronto.edu/~faith/algs.ps
15. Fredriksson, K.: Faster String Matching with Super-Alphabets. In: Laender, A.H.F., Oliveira, A.L. (eds.) SPIRE 2002. LNCS, vol. 2476, pp. 44–57. Springer, Heidelberg (2002)
16. Fredriksson, K.: Shift-or string matching with super-alphabets. IPL 87(4), 201–204 (2003)
17. Furst, M.L., Saxe, J.B., Sipser, M.: Parity, circuits, and the polynomial-time hierarchy. Mathematical Systems Theory 17(1), 13–27 (1984)
18. Galil, Z.: Optimal parallel algorithms for string matching. Inform. and Control 67, 144–157 (1985)
19. Galil, Z.: A Constant-Time Optimal Parallel String-Matching Algorithm. J. ACM 42(4), 908–918 (1995)
20. Gąsieniec, L., Plandowski, W., Rytter, W.: Constant-space String Matching with Smaller Number of Comparisons: Sequential Sampling. In: Galil, Z., Ukkonen, E. (eds.) CPM 1995. LNCS, vol. 937, pp. 78–89. Springer, Heidelberg (1995)
21. Goldberg, T., Zwick, U.: Faster parallel string matching via larger deterministic samples. J. Algorithms 16(2), 295–308 (1994)
22. Knuth, D., Morris, J., Pratt, V.: Fast pattern matching in strings. SIAM J. Comput. 6, 322–350 (1977)
23. Knuth, D.E.: Combinatorial Algorithms. The Art of Computer Programming, vol. 4A. Addison-Wesley Professional (January 2011)
24. Navarro, G., Raffinot, M.: A Bit-Parallel Approach to Suffix Automata: Fast Extended String Matching. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 14–33. Springer, Heidelberg (1998)
25. Tarhio, J., Peltola, H.: String matching in the DNA alphabet. Software Practice Experience 27, 851–861 (1997)
26. Vishkin, U.: Optimal parallel pattern matching in strings. Inform. and Control 67, 91–113 (1985)
27. Vishkin, U.: Deterministic sampling - A new technique for fast pattern matching. SIAM J. Comput. 20(1), 22–40 (1990)

Pattern Matching in Multiple Streams[⋆]

Raphaël Clifford[1], Markus Jalsenius[1], Ely Porat[2], and Benjamin Sach[3]

[1] University of Bristol, Department of Computer Science, Bristol, UK
[2] Bar-Ilan University, Department of Computer Science, Ramat-Gan, Israel
[3] University of Warwick, Department of Computer Science, Coventry, UK

Abstract. We investigate the problem of deterministic pattern matching in multiple streams. In this model, one symbol arrives at a time and is associated with one of s streaming texts. The task at each time step is to report if there is a new match between a fixed pattern of length m and a newly updated stream. As is usual in the streaming context, the goal is to use as little space as possible while still reporting matches quickly. We give almost matching upper and lower space bounds for three distinct pattern matching problems. For exact matching we show that the problem can be solved in constant time per arriving symbol and $O(m + s)$ words of space. For the k-mismatch and k-difference problems we give $O(k)$ time solutions that require $O(m + ks)$ words of space. In all three cases we also give space lower bounds which show our methods are optimal up to a single logarithmic factor. Finally we set out a number of open problems related to this new model for pattern matching.

1 Introduction

We introduce a new set of problems centered on pattern matching in multiple streaming texts. In this model, one symbol arrives at a time and is added to the tail of exactly one of s streaming texts. The task at each time step is to report if there is a new match between a fixed pattern P of length m and a newly updated stream. Our interest is in deterministic algorithms with guaranteed worst case time complexity. The goal is to use as little space as possible while still reporting matches quickly.

The problem of pattern matching in a single stream using limited space had a major breakthrough in 2009 when it was shown that exact matching can be performed using only $O(\log m)$ words of space and $O(\log m)$ time [20]. This result was subsequently simplified [10] and then improved [4] to run in constant time per new symbol. To achieve this small space, these methods are however all necessarily randomised and allow some small probability of error. Where neither randomisation nor error is permitted, a straightforward argument shows us that there is no hope of using space sublinear in the pattern size. This follows directly from the observation that we could use such a matching algorithm to reproduce the pattern in its entirety and that it therefore must use at least linear space.

⋆ This work was partially supported by EPSRC.

Where there are multiple streams however the situation is not as clear cut even where no randomisation is allowed. A naive approach would simply be to store s copies of the working space, one for each stream. This will typically then require $\Theta(ms)$ space overall. Where the number of streams s is large, the space usage of such an approach is therefore likely to be prohibitive.

Our contribution is first to show that for three particularly common pattern matching problems, pattern matching in multiple streams can be solved in considerably less than $\Theta(ms)$ space without the use of randomisation. In Section 3 we give a constant time algorithm for exact matching that requires only $O(m+s)$ words of space. In Section 4 we review a recently introduced data structure which allows us to perform longest common extension (LCE) queries between a streaming text and a fixed pattern in constant time and $O(m)$ space. This data structure is then used in Sections 5 and 6 where we give $O(k)$ time and $O(m + ks)$ space solutions to the k-mismatch and k-difference problems. In Section 7 we give almost matching space lower bounds for our problems as well as lower bounds for some other common pattern matching problems. Finally in Section 8 we set out a number of open problems that immediately arise for this new model of pattern matching.

2 Related Work

Randomised space lower bounds for a wide range of pattern matching problems in a single stream were given in [7]. In [6,9] it was also shown that a large set of pattern matching algorithms could be converted from offline to online with only at worst a multiplicative logarithmic factor overhead in their time complexity. This therefore provided an effective deamortisation of almost the entire field of combinatorial pattern matching and a ready tool for the construction of fast streaming pattern matching algorithms.

In the more usual offline setting, a great deal of progress has been made in finding fast algorithms for a variety of approximate matching problems. One of the most studied is the Hamming distance which measures the number of single character mismatches between two strings. Given a text of length n and a pattern of length m, the task is to report the Hamming distance at every possible alignment. Solutions running in $O(n\sqrt{m \log m})$ time which are based on repeated applications of the FFT were given independently by both Abrahamson and Kosaraju in 1987 [1,15]. Particular interest has been paid to the bounded variant we also consider called the k-mismatch problem. Here a bound k is given and we need only report the Hamming distance if it is less than or equal to k. If the number of mismatches is greater than the bound, the algorithm need only report that fact and not give the actual Hamming distance. In 1986 Landau and Vishkin gave a beautiful $O(nk)$ time algorithm that is not FFT based and uses constant time lowest common ancestor (LCA) operations on the suffix tree of the pattern and text [18]. This was subsequently improved to $O(n\sqrt{k \log k})$ time by a method based on filtering and FFTs again [3]. A separate line of research considered the question of how to find approximations within a

$(1+\varepsilon)$ multiplicative factor of the Hamming distance [12,14]. The edit distance measures the minimum number of single character insert, delete and mismatch operations required to transform one string into another. We consider a bounded problem called k-difference which reports the edit distance at every alignment only if it is at most k. An $O(nk)$ solution to this problem was given in [19] using LCE queries to perform jumps within the dynamic programming table.

2.1 Preliminaries and a New Model for Multiple Streams

In order to design algorithms for pattern matching in multiple streams we first define a new computational model which we will use throughout. In this new model there is only one stream, however we will carefully distinguish between space that any pattern matching algorithm uses that is associated with the pattern and space associated with the text. We call this model, which may be of independent interest, the *single stream read-only preprocessing model* or simply the *read-only preprocessing model* for short.

Any algorithm that operates in this model operates in two phases: a preprocessing phase followed by an online phase. During the preprocessing phase, the algorithm processes the pattern without any knowledge of the text. The output of the preprocessing phase is termed the *pattern space*. The online phase is provided read-only access to a copy of the pattern space. It is important to note that the online phase is not provided direct access to the pattern unless the pattern is explicitly stored in the pattern space. The online phase continues as in the original single stream model, processing each text character as it arrives. Any space used by the online phase in addition to the read-only pattern space is termed *text space*. The text space can therefore only store information that is associated with the text and its relation to information already stored in the pattern space.

Algorithms developed in the read-only preprocessing model translate directly to the multiple stream model that we are interested in. Consider an algorithm in the read-only preprocessing model. Let f be the time taken per new arriving symbol, g_{p} the pattern space and g_{t} the text space of this algorithm. Since the pattern space is independent of the text, it can be shared across multiple texts. Therefore we can directly derive a new pattern matching algorithm in the multiple stream model which runs in $O(f)$ time per character using $O(g_{\mathrm{p}}+sg_{\mathrm{t}})$ space. The space and time requirements of the preprocessing stage are arguably of less significance given that they are a function only of the pattern size. Nevertheless, for all three pattern matching problems we consider, the preprocessing stage can be implemented in $O(m)$ space and $O(m\log m)$ time. We assume throughout that all computation is performed in the unit cost RAM model.

3 Exact Matching

We begin by giving a real-time variant of the KMP algorithm. We will apply it to the read-only preprocessing model ensuring it takes $O(1)$ time, using $O(m)$

pattern space and $O(1)$ text space, so that in the multiple stream model it will take $O(1)$ time and use $O(m+s)$ space. The more usual real-time variant provided by Galil [11] will not suffice for our purposes as it buffers the text and therefore would use $O(ms)$ space in the multiple stream model.

First recall that at each stage of the standard KMP algorithm we keep track of the longest prefix of the pattern that matches a suffix of the text seen so far. When a new character $T[i]$ arrives, we extend the matching prefix by $T[i]$ if possible, otherwise we shift the pattern according to a pre-computed prefix table, also known as the failure function. More precisely, suppose that $P[0 \dots j-1]$ matches $T[i-j \dots i-1]$ when $T[i]$ arrives. If $P[j] = T[i]$, we extend the match, otherwise we look at position j of the prefix table, which gives us the largest value $0 < j' < j$ such that $P[0 \dots j'-1]$ matches $T[i-j' \dots i-1]$ and $P[j'] \neq P[j]$. Then we compare $P[j']$ with $T[i]$ to see if we can extend the new match by $T[i]$. If not, we shift the pattern again using the prefix table, and so on.

While it is well-known that the time complexity per character is $O(1)$ amortised, our motivations call for an unamortised solution. Instead of using a prefix table, for each position j of the pattern we store a dictionary D_j that contains every pair $(\sigma, j') \in \Sigma \times \{1, \dots, m-1\}$ where $0 < j' < j$ is the largest index such that $P[0 \dots j'-1]$ matches $P[j-j' \dots j-1]$ and $P[j'] = \sigma \neq P[j]$. The pairs (σ, j') in D_j are indexed by the symbol σ and Σ denotes the alphabet. In other words, whenever a match $P[0 \dots j-1]$ cannot be extended to $P[0 \dots j]$ (i.e., $P[j] \neq T[i]$), instead of repeatedly shifting the pattern according to the prefix table until $T[i]$ is matched, we look up symbol $T[i]$ in D_j and immediately get the length j' of the prefix of P that the KMP algorithm would eventually align with $T[i-j' \dots i-1]$ in order to be able to extend the match to $P[j'] = T[i]$. The dictionaries D_j can be pre-computed as all shifts are based on self-matches with the pattern itself. By using a static perfect dictionary we can do lookups in constant time. Thus, a KMP algorithm equipped with these dictionaries instead of a prefix table will run in unamortised $O(1)$ time per character. An interesting fact is that the total space usage to store these dictionaries is only $O(m)$. The following lemma was proved in [22], where it was stated in the language of finite automata. For completeness we include a proof.

Lemma 1 (Theorem 1 of [22]). *The sum of the sizes of all dictionaries,*

$$\sum_{j=0}^{m-1} |D_j| \leqslant m\,.$$

Proof. A lookup in the dictionary D_j results in a shift of the pattern. We show that for every length $\ell \in \{1, \dots, m-1\}$ there is at most one element over all dictionaries that moves the pattern along by ℓ positions. For contradiction, suppose that some $(\sigma_1, j_1') \in D_{j_1}$ and $(\sigma_2, j_2') \in D_{j_2}$, where $j_1 < j_2$, both shift the pattern by ℓ. By definition, we have $P[j_1] \neq P[j_1']$. If the same shift ℓ is applied when the j_2-length prefix of P is moved along, we must have $P[j_1] = P[j_1 - \ell] = P[j_1']$, which leads to a contradiction. □

By using static perfect hashing, we can store all dictionaries $D_0, \ldots, D_{m-1}$ in $O(m)$ pattern space. Although preprocessing time is not our focus, we briefly discuss how to construct the dictionaries in $O(m \log\log m)$ time.

We begin by constructing the standard KMP prefix table in $O(m)$ time. We then construct each dictionary D_j by considering j in increasing order, starting with $j = 0$. For any $j \geqslant 0$, the dictionary D_j is constructed as follows. Let $j' < j$ be the index given by the original KMP prefix function for $P[0 \ldots j]$. The elements in $\{\, (\sigma, j'') \in D_{j'} \mid \sigma \neq P[j] \,\} \cup \{(P[j'], j')\}$ are added to D_j. These are precisely the elements that belong to D_j according to its definition. Over all j, gathering the elements to be added to the dictionaries takes $O(m)$ time. The running time is therefore dominated by the time it takes to insert the elements into the dictionaries. By using the the static dictionary of Ružić [21], construction takes

$$\sum_{j=0}^{m-1} O\big(|D_j| \log\log |D_j|\big) \leqslant O(m \log\log m)\,.$$

time, where Lemma 1 has been applied.

The next lemma summarises the result, which together with the properties of the read-only preprocessing model gives us Theorem 3.

Lemma 2. *Exact matching can be solved in the read-only preprocessing model in $O(1)$ time per character and using $O(m)$ pattern space and $O(1)$ text space.*

Theorem 3. *Exact matching in the multiple stream model with s texts can be solved in $O(1)$ time per character and $O(m+s)$ space.*

4 LCE Queries in a Stream

In preparation for the algorithms we give for the k-mismatch and k-difference problems, we will be required to maintain a data structure that allows us to compute LCE queries in a streaming text in $O(1)$ time and $O(m)$ space. This method was first introduced in [8], although the idea of representing the text in terms of substrings of the pattern was first used in a different setting in [2]. We provide a brief recap here.

We will split the streaming text into contiguous substrings which are encoded as triplets, (i', j, ℓ), each representing an ℓ-length text substring $T[i' \ldots (i'+\ell-1)]$ that equals a pattern substring $P[j \ldots (j+\ell-1)]$. We refer to such a triple as *p-region* and a disjoint ordered sequence of triplets that encode the entire text as a *p-representation* . For example, with $P =$ `babbac` and $T =$ `abcaababba`, a p-representation of T is

$$(0,1,2), (2,5,1), (3,4,1), (4,1,2), (6,1,4).$$

The p-representation is not necessarily unique. We say that it is of minimal length if it contains a minimal number of triplets.

In [8] it was shown that we can extend a minimal length p-representation of $T[0 \ldots i-1]$ to a minimal length p-representation of $T[0 \ldots i]$ in $O(1)$ time when symbol $T[i]$ arrives. More precisely, the extended p-representation is obtained greedily by updating the last p-region if possible, otherwise adding a new p-region $(i, j, 1)$, where j is some position such that $P[j] = T[i]$. To accomplish this task in $O(1)$ time, a suffix tree of the pattern can be used [8]. For our purposes in terms of pattern space and text space, we may construct the suffix tree during the pattern preprocessing phase and store it together with the pattern itself in the pattern space. Deciding whether to update the last p-region or add a new one when a new symbol $T[i]$ arrives can then be done in constant text space. For simplicity of explanation we will assume that all symbols in T occur at least once in P. For further details of the method and how to handle symbols which occur in the text but not the pattern, we refer the interested reader to [8].

As we will see shortly, a benefit of using a minimal length p-representation of T is that we can answer longest common extension (LCE) queries between the pattern and $T[0 \ldots i]$ in $O(1)$ time. We write $\mathrm{LCE}(i', j)$ to denote the length of the longest prefix of $P[j \ldots m-1]$ that is also a prefix of $T[i' \ldots i]$. The following lemma was stated as Lemma 1 in [8].

Lemma 4. *For a minimal length p-representation of T, at most three p-regions overlap $T[i' \ldots (i'+\ell-1)]$, where ℓ is the length returned by any $\mathrm{LCE}(i', j)$ query.*

It is well known that we can precompute a static data structure using $O(m)$ space, denoted $\mathrm{LCE_p}$, to support LCE queries between the pattern and itself in $O(1)$ time. This is traditionally achieved with a suffix tree on which lowest common ancestor queries are answered in constant time. It now follows from Lemma 4 that any $\mathrm{LCE}(i', j)$ query on the streaming text can be answered in $O(1)$ time by performing at most three pattern-pattern LCE queries in the $\mathrm{LCE_p}$ structure.

5 *k*-Mismatch in Multiple Streams

We described in Section 4 how a p-representation of the text can be maintained in $O(1)$ time and $O(m)$ pattern space. The actual p-representation of $T[0 \ldots i]$ will be stored in the text space of size $O(i)$. Instead of storing every p-region, we could store only the most recent p-regions of the text. This representation will of course only give us access to some suffix of the text seen so far. Our algorithm for the k-mismatch problem, which we present in the read-only preprocessing model in the first instance, will store the most recent $4(k+1)$ p-regions, requiring $O(k)$ text space.

In order to determine whether $T[(i-m+1) \ldots i]$ has at most k mismatches with P when $T[i]$ arrives, we apply the kangaroo technique [17] consisting of up to $k+1$ LCE queries between the text and the pattern. We will perform these LCE queries in the *reverse* of T and P, starting from the rightmost character $T[i]$. It should not be too hard to see that the data structures described in Section 4 can be modified to support reverse LCE queries between the pattern and the text with no effect on the asymptotic time and space complexities. Also, Lemma 4 holds for reverse LCE queries. In the next lemma we prove that all

LCE queries performed by the algorithm fall within the $4(k+1)$ most recent p-regions.

Lemma 5. *The k-mismatch problem can be solved in the read-only preprocessing model in $O(k)$ time per character and using $O(m)$ pattern space and $O(k)$ text space.*

Proof. The time complexity follows immediately from the algorithm description, similarly with the pattern and text space complexities. For the correctness of the algorithm we need to show that none of the at most $k+1$ reverse LCE queries performed to determine the number of mismatches between $T[(i-m+1)\ldots i]$ and P fall outside the text substring represented by the $4(k+1)$ most recent p-regions.

From Lemma 4 it follows that one LCE query could span three p-regions. As part of the kangaroo technique, we skip over a mismatch position between each LCE query. The mismatch could fall inside a new p-region, hence no more than a total of $4(k+1)$ p-regions are involved in a series of up to $k+1$ LCE queries. □

From Lemma 5 and the properties of the read-only preprocessing model, we immediately have the following theorem.

Theorem 6. *The k-mismatch problem can be solved in the multiple stream model with s texts in $O(k)$ time per character and $O(m+ks)$ space.*

6 *k*-Difference in Multiple Streams

Let $D[j,i]$ denote the minimum of all k-bounded edit distances between the pattern prefix $P[0\ldots j]$ and all suffixes of $T[0\ldots i]$. For the k-difference problem we want to output $D[m-1,i]$ as soon as $T[i]$ arrives. We have the standard dynamic programming recurrence,

$$D[j,i] = \min \begin{cases} D[j,\, i-1]+1 & \text{(insert)} \\ D[j-1,\, i]+1 & \text{(delete)} \\ D[j-1,\, i-1]+1-\mathrm{eq}(i,j) & \text{(mismatch)} \\ k+1 & \text{(k-bounded)} \end{cases}$$

where $\mathrm{eq}(i,j)=1$ if $T[i]=P[j]$ and 0 otherwise. For the base cases we have $D[j,-1]=\min(k+1,j+1)$ and $D[-1,i]=0$ for all i,j.

We now present a solution for the k-difference problem, first in the read-only preprocessing model and then give the final result in the multiple stream model as required. Whenever a text character $T[i]$ arrives such that i is a multiple of k, we start a *child process* which will be responsible for outputting $D[m-1,i']$ for all $i' \in \{(i+k),\ldots,(i+2k-1)\}$ as each such $T[i']$ arrives (Interval 2 in Fig. 1). Therefore, there is a child process responsible for each and every output. The k text arrivals between $T[i]$ and the first output for a child process will, as explained below, give us enough time to prepare for the outputs. Observe that at most two child processes are running at any one time, hence we only need

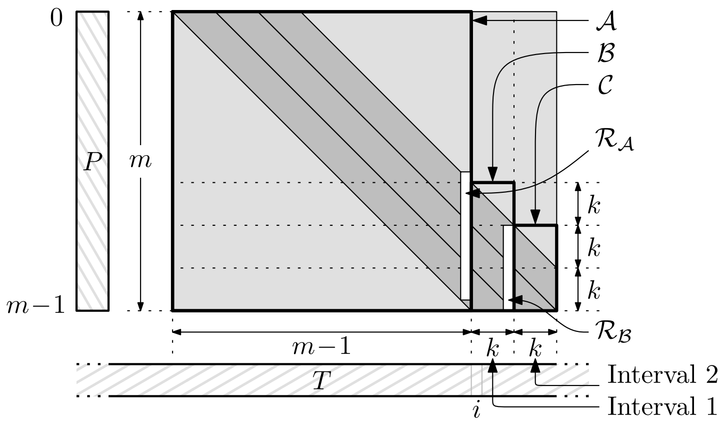


Fig. 1. The dynamic programming table for k-difference

to show that one child process can be implemented in $O(k)$ time per character, $O(m)$ pattern space and $O(k)$ text space.

The operation of a child process is divided into three stages, each responsible for computing cells of the blocks denoted $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, respectively, of the dynamic programming table in Fig. 1. In the first stage, which runs during the $k/2$-length text interval starting with the arrival of $T[i]$ (first half of Interval 1 in Fig. 1), the child process will compute the cells marked $\mathcal{R}_\mathcal{A}$ in Fig. 1, that is the cells $D[(m-3k-1)\dots(m-2),\, i-1]$. To do this we use the technique of Landau-Vishkin [19] which is an offline algorithm that we run on block $\mathcal{A}$. Their algorithm takes $O(k^2)$ time and uses $O(m)$ space by operating along the diagonals of the dynamic programming table (shaded in dark gray in the figure). Their algorithm is based around LCE queries of the text, which we can take advantage of in a similar fashion to the k-mismatch algorithm from the previous section. We will see below that we only need to store $O(k)$ p-regions of the text to perform the LCE queries, hence we can run their algorithm in $O(k)$ text space. The running time of $O(k^2)$ is spread evenly over $k/2$ text arrivals, and therefore takes $O(k)$ time per character.

Once the first stage is completed we have obtained the values of the cells marked $\mathcal{R}_\mathcal{A}$ in the figure. Now the second stage takes over. Here we compute the cells of block $\mathcal{B}$ of the dynamic programming table by direct use of the recurrence above. The work is spread evenly over the next $k/2$ text arrivals (second half of Interval 1) by computing two columns of the block $\mathcal{B}$ per arrival. The final column, marked $\mathcal{R}_\mathcal{B}$ in the figure, is therefore done by the arrival of $T[i+k]$. Thus, the second stage takes $O(k)$ time per character and uses only $O(k)$ space. We should mention that the values of cells on the boundary of $\mathcal{B}$ (excluding $\mathcal{R}_\mathcal{A}$) that are used in the recurrence when computing block $\mathcal{B}$ are all set to ∞. This however does not affect the correctness of the computed values of $\mathcal{R}_\mathcal{B}$.

In the final and third stage we compute all cell values of block $\mathcal{C}$ by direct use of the recurrence, like we did for block $\mathcal{B}$ during the second stage. We use the values of $\mathcal{R}_{\mathcal{B}}$ and set the other boundary cells to ∞ (which does not affect the correctness of the final output). For each arriving character during Interval 2 in Fig. 1, we compute the corresponding column of block $\mathcal{C}$. Therefore the running time is $O(k)$ per character, and space usage is $O(k)$.

All steps of the child process described above, except for the real-time LCE processing, require only $O(k)$ space. During the pattern preprocessing phase, as for the k-mismatch algorithm above, we will construct the required LCE_p structure and the suffix tree of the pattern, and store both these structures as well as a copy of the pattern, using a total of $O(m)$ pattern space. We modify the LCE processing to store only the most recent $5(k+1)$ p-regions. We will show that whenever the edit distance is k or less, storing the $5(k+1)$ most recent p-regions is sufficient to support every LCE query. Thus, if any LCE query stretches beyond these p-regions, the edit distance must be more than k. The following lemma summarises the result. The result for the multiple stream model follows immediately from the above observation about the relationship between the models.

Lemma 7. *The k-difference problem can be solved in the read-only preprocessing model in $O(k)$ time per character using $O(m)$ pattern space and $O(k)$ text space.*

Proof (sketch). The time and space complexities follow from inspection of the algorithm description. For correctness, suppose there exists an $i' \in \{(i+k), \ldots, (i+2k-1)\}$ such that $D[m-1, i'] \leqslant k$. Therefore, by the problem definition, there exists an ℓ such that P can be transformed into $T[(i'-\ell+1)\ldots i']$ in at most k insert, delete and mismatch operations. We will first show that this immediately implies the existence of a p-representation of $T[(i'-\ell+1)\ldots i']$ containing at most $2k+1$ p-regions.

Consider any transformation of P into $P' := T[(i'-\ell+1)\ldots i']$ which contains at most k operations. We denote by C the ℓ-length array which states the 'origin' of each character in P': $C[j'] = j$ if the transformation aligns $P'[j']$ with $P[j]$ and $P'[j'] = P[j]$, otherwise $C[j'] = -\infty$, which means that $C[j']$ is the result of an insert operation or is aligned with a symbol different from $P'[j']$.

We can construct a p-representation R of P' by a single pass of C as follows. If $C[0] \neq -\infty$, we begin by creating the p-region $(0, C[0], 1)$. Otherwise, we create the region $(0, j'', 1)$ where j'' is any index such that $P[j''] = P'[0]$. For each $j' > 0$, we consider three disjoint cases:

1. $C[j'] = C[j'-1]+1$. Increase the length of the most recent region by one.
2. $C[j'] > C[j'-1]+1$. Create a new p-region $(j', C[j'], 1)$.
3. $C[j'] = -\infty$. Create a new p-region $(j', j'', 1)$, where j'' is any index such that $P'[j'] = P[j'']$.

We now consider the number of p-regions in the p-representation R. An additional p-region is only created when either case 2 or case 3 occurs in the construction. Case 3 occurs only when $C[j'] = -\infty$. However, by the definition of C, each $-\infty$ corresponds $P[j']$ to being a result of a mismatch or insert operation of which there are at most k in total. Therefore case 3 occurs at most k times.

Case 2 occurs when $C[j'] > C[j'-1]+1$. By the definition of C, this implies that either some character $P[j'']$ with $C[j'-1] < j'' < C[j']$ was deleted or $C[j'-1] = -\infty$ which in turn implies that either a mismatch or insert operation occurred at $C[j'-1]$. Hence case 2 can occur at most k times. The total number of p-regions is therefore upper bounded by $2k+1$ as a mismatch or insert can cause two new p-regions to be created, one at some $C[j'-1]$ and another at $C[j']$.

To see that $5(k+1)$ p-regions are enough, first observe that every LCE query performed ends in the substring $T[(i-m+1)\dots(i+2k-1)]$. As P can be transformed into $T[(i'-\ell+1)\dots i']$ in at most k moves, we have that $\ell \geqslant m-k$. We also have that $i' \in \{(i+k),\dots,(i+2k-1)\}$ and hence $T[(i'-\ell+1)\dots i']$ is a substring of $T[(i-m+1)\dots(i+2k-1)]$. We can then then convert the p-representation of $T[(i'-\ell+1)\dots i']$ into a p-representation of $T[(i-m+1)\dots(i+2k-1)]$ by adding at most $3k$ p-regions of length one each. Thus, $(2k+1)+3k \leqslant 5(k+1)$ p-regions suffice to support any LCE query. □

Theorem 8. *The k-difference problem can be solved in the multiple stream model with s texts in $O(k)$ time per character and $O(m+ks)$ space.*

7 Space Lower Bounds

In this section we show that our space upper bounds for k-mismatch and k-difference are optimal (up to a log factor). We also show that for several other common distance measures, pattern matching in s streams requires $\Omega(ms)$ bits of space, implying that we may not do better than treating each stream independently.

The log sized gap between our lower bounds and upper bounds comes from the fact that we state the lower bounds in bits whereas the upper bounds are given in words. A smaller gap could be obtained by considering large alphabets (see e.g. [7]), however for simplicity we give our lower bounds assuming binary alphabets.

Our results are based on reductions from two one-way communication complexity problems with known lower bounds. In a one-way communication model, only Alice can send messages to Bob and then Bob must output the correct answer. In the EQUALITY problem, Alice has a string $X \in \{0,1\}^m$ and Bob has a string $Y \in \{0,1\}^m$. Bob must determine whether $X = Y$. The communication complexity is $\Omega(m)$ bits [16]. In the INDEXING problem, Alice has a string $X \in \{0,1\}^m$ and Bob has an index $n \in \{0,\dots,m-1\}$. Bob must find $X[n]$. The problem is known to have an $\Omega(m)$ bit lower bound [16].

Theorem 9. *The k-mismatch and k-difference problems in s streams both require $\Omega(m+ks)$ bits of space.*

Proof. First consider the case where $m \geqslant ks$. We reduce from the EQUALITY problem, where Alice has a string $X \in \{0,1\}^m$ and Bob has a bit string $Y \in \{0,1\}^m$. Let the pattern P be the string X. Let A be any algorithm that solves either k-mismatch or k-difference on the pattern P. Alice sends the internal state of A to Bob, who feeds the algorithm with the string Y in one of the streams. The output is 0 if and only if $X = Y$, hence $\Omega(m)$ bits of space is required.

Now consider the case where $m < ks$. We reduce from the Indexing problem, where Alice has a string $X \in \{0,1\}^{ks}$ and Bob has an index $n \in \{0, \ldots, ks-1\}$. Let A be any algorithm that solves either k-mismatch or k-difference on the pattern $P = \{0\}^m$. Alice feeds each of the s streams with k bits from her string X such that the first stream is fed the first k bits of X, the second stream is fed the next k bits of X, and so on. Alice then sends the internal state of A to Bob who now wants to determine $X[n]$ which was fed into stream $r = \lfloor n/k \rfloor$. Bob feeds the stream r with $m - k + (n \bmod k)$ 0s, which ensures that $X[n]$ is aligned with the first position of P. Let d be the output by A at this alignment and observe that for both k-mismatch and k-difference, d equals the number of 1s in the last m symbols of the stream r. Bob now feeds another 0 into the stream r. Let d' be the new output by A. It follows that $X[n] = 0$ if and only if $d = d'$, hence $\Omega(ks)$ bits of space is required. The space lower bound of $\Omega(m + ks)$ bits is obtained by combining the two cases. □

Theorem 10. *Exact matching in s streams requires $\Omega(m + s)$ bits of space.*

Proof. Following the proof of Theorem 9, we reduce from the Equality problem if $m > s$, otherwise we reduce from Indexing where Alice feeds each stream with either one 0 or one 1. By feeding $m - 1$ 0s into any stream, Bob can determine any of the s bits. □

We now turn our attention to the L_1, L_2 and Hamming distance problems. For any constant p, the L_p distance between two equal length strings X and Y is given by $d_p(X, Y) = \left(\sum_j |X[j] - Y[j]|^p\right)^{1/p}$.

Theorem 11. *Computing the L_1, L_2 and Hamming distances, as well as the cross-correlation/convolution in s streams, requires $\Omega(ms)$ bits of space.*

Proof. We reduce from the Indexing problem, where Alice has a string $X \in \{0,1\}^{ms}$ and Bob has an index $n \in \{0, \ldots, ms-1\}$. Let A be any algorithm that solves either of the problems in the statement of the lemma on the pattern $P = \{1\}^m$, where instead of computing the L_2 distance, the square of the distance is computed. Observe that a lower bound for computing the square of the L_2 distance is also a lower bound for the L_2 distance. Each of the problems is now *local* in the sense that the output is the sum of m position-wise values. Following the idea in the proof of Theorem 9, Alice feeds each stream with m bits of her string X before sending the internal state to Bob. In order to determine $X[n]$, Bob feeds the appropriate stream r with enough 1s to align $X[n]$ with the first position of P. By feeding another 1 into the stream r and comparing the two outputs, Bob can determine the value of $X[n]$. □

8 Open Problems

The space complexity of the results we give are tight to within a log factor when no randomisation is permitted. Further, the time complexity of both the exact matching and k-difference algorithms we give match that of the fastest known offline algorithms per arriving symbol. However, for k-mismatch there remains a gap of approximately $O(\sqrt{k})$ between the fastest single stream algorithm [8] and the time complexity we give for multiple streams. There is an even more pronounced gap for the special case of constant sized alphabets when the bound k is relatively large. Here the k-mismatch problem can be solved in a single stream in $O(\log^2 m)$ time per symbol and $O(m)$ space [5], independent of the value of k. It would be interesting to consider whether there is a $O(\text{poly}(\log m))$ time, multiple stream algorithm using only $O(m + ks)$ space in this case.

We have seen that the read-only preprocessing model is a useful conceptual tool for developing algorithms in the multiple stream model. In particular we have used the fact that any efficient algorithm in the former model immediately gives an efficient algorithm in the latter. It is natural to wonder whether these models are in fact equivalent. We conjecture that for any $O(g_\text{p} + sg_\text{t})$ space algorithm for a pattern matching problem in the multiple stream model (where g_p, g_t do not depend on s), there is an $O(g_\text{p})$ pattern space, $O(g_\text{t})$ text space algorithm in the read-only preprocessing model with the same time complexity per character.

If we are concerned with randomised computation where each output has to be correct with some (arbitrarily large) constant probability, we can derive new space lower bounds for all three problems with some modification. In particular, k-mismatch and k-difference will require at least $\Omega(\log m + ks)$ bits of space and exact matching $\Omega(\log m + s)$ bits of space. These lower bounds follow from the randomised counterpart of Equality and Indexing. The randomised one-way communication complexity with private randomness for Equality is $\Theta(\log m)$ bits [23], and for Indexing it remains $\Omega(m)$ bits (see [13] for an elementary proof). It is not yet clear whether these randomised lower bounds for the multiple streams problem can be met by matching algorithmic upper bounds.

References

1. Abrahamson, K.: Generalized string matching. SIAM Journal on Computing 16(6), 1039–1051 (1987)
2. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic text and static pattern matching. ACM Transactions on Algorithms (TALG) 3(2) (2007)
3. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. Journal of Algorithms 50(2), 257–275 (2004)
4. Breslauer, D., Galil, Z.: Real-Time Streaming String-Matching. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 162–172. Springer, Heidelberg (2011)
5. Clifford, R., Efremenko, K., Porat, B., Porat, E.: A Black Box for Online Approximate Pattern Matching. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 143–151. Springer, Heidelberg (2008)
6. Clifford, R., Efremenko, K., Porat, B., Porat, E.: A black box for online approximate pattern matching. Information and Computation 209(4), 731–736 (2011)
7. Clifford, R., Jalsenius, M., Porat, E., Sach, B.: Space Lower Bounds for Online Pattern Matching. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 184–196. Springer, Heidelberg (2011)

8. Clifford, R., Sach, B.: Pseudo-realtime Pattern Matching: Closing the Gap. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 101–111. Springer, Heidelberg (2010)
9. Clifford, R., Sach, B.: Pattern matching in pseudo real-time. Journal of Discrete Algorithms 9(1), 67–81 (2011)
10. Ergun, F., Jowhari, H., Sağlam, M.: Periodicity in Streams. In: Serna, M., Shaltiel, R., Jansen, K., Rolim, J. (eds.) APPROX and RANDOM 2010, LNCS, vol. 6302, pp. 545–559. Springer, Heidelberg (2010)
11. Galil, Z.: String matching in real time. Journal of the ACM 28(1), 134–149 (1981)
12. Indyk, P.: Faster algorithms for string matching problems: Matching the convolution bound. In: FOCS 1998: Proc. 39th Annual Symp. Foundations of Computer Science, pp. 166–173 (1998)
13. Jayram, T.S., Kumar, R., Sivakumar, D.: The one-way communication complexity of hamming distance. Theory of Computing 4(1), 129–135 (2008)
14. Karloff, H.: Fast algorithms for approximately counting mismatches. Information Processing Letters 48(2), 53–60 (1993)
15. Kosaraju, S.R.: Efficient string matching (1987) (manuscript)
16. Kushilevitz, E., Nisan, N.: Communication complexity. Cambridge University Press (1997)
17. Landau, G.M., Vishkin, U.: Efficient string matching in the presence of errors. In: FOCS 1985: Proc. 26th Annual Symp. Foundations of Computer Science, pp. 126–136 (1985)
18. Landau, G.M., Vishkin, U.: Efficient string matching with k mismatches. Theoretical Computer Science 43, 239–249 (1986)
19. Landau, G.M., Vishkin, U.: Fast string matching with k differences. Journal of Computer System Sciences 37(1), 63–78 (1988)
20. Porat, B., Porat, E.: Exact and approximate pattern matching in the streaming model. In: FOCS 2009: Proc. 50th Annual Symp. Foundations of Computer Science, pp. 315–323 (2009)
21. Ružić, M.: Constructing Efficient Dictionaries in Close to Sorting Time. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 84–95. Springer, Heidelberg (2008)
22. Simon, I.: String matching algorithms and automata. In: First American Workshop on String Processing, pp. 151–157 (1993)
23. Yao, A.C.-C.: Some complexity questions related to distributive computing. In: STOC 1979: Proc. 11th Annual ACM Symp. Theory of Computing, pp. 209–213 (1979)

An Efficient Linear Pseudo-minimization Algorithm for Aho-Corasick Automata[⋆]

Omar AitMous[1], Frédérique Bassino[1], and Cyril Nicaud[2]

[1] LIPN, UMR 7030, Université Paris 13 - CNRS, 93430 Villetaneuse, France
[2] LIGM, UMR 8049, Université Paris-Est - CNRS, 77454 Marne-la-Vallée, France
{aitmous,bassino}@lipn.univ-paris13.fr, nicaud@univ-mlv.fr

Abstract. A classical construction of Aho and Corasick solves the pattern matching problem for a finite set of words X in linear time, where the size of the input X is the sum of the lengths of its elements. It produces an automaton that recognizes A^*X, where A is a finite alphabet, but which is generally not minimal. As an alternative to classical minimization algorithms, which yields a $\mathcal{O}(n \log n)$ solution to the problem, we propose a linear pseudo-minimization algorithm specific to Aho-Corasick automata, which produces an automaton whose size is between the size of the input automaton and the one of its associated minimal automaton. Moreover this algorithm generically computes the minimal automaton: for a large variety of natural distributions the probability that the output is the minimal automaton of A^*X tends to one as the size of X tends to infinity.

1 Introduction

Pattern matching issues arise naturally in various fields of computer science, motivating an ever renewed search for efficient algorithms that answer the following question: given a set of words X, the set of patterns, what is the best way to compute the number of occurrences of words of X in a large text?

The literature mostly focuses on finite sets of patterns, and a first efficient solution to the pattern matching problem was given by Aho and Corasick [1]: they described an elegant construction for an automaton that recognizes A^*X, where A is a finite alphabet, which can thereafter be used to sequentially parse the text. The algorithm proceeds in two steps. The prefix tree of X is built first (we call its transitions *tree transitions*), and is then cleverly completed with *failure transitions*, using properties on borders of words. Defining the size of X as the sum of the lengths of its words, their solution runs in linear time. This algorithm is still widely used as a primitive brick in many situations (see for instance Baker [3] and Bird [5] algorithms for 2D pattern recognition).

Aho-Corasick automaton is always deterministic and complete, but unfortunately not necessarily minimal, as depicted on Figure 1. However, unless for

[⋆] This work was completed with the support of the ANR project MAGNUM number 2010-BLAN-0204.

specific patterns, Aho-Corasick automaton and the corresponding minimal one seem to differ only in a few number of states. This asks for some simpler process than the usual minimization algorithms [8], whose worst-case complexity is at least $\mathcal{O}(n \log n)$. It is however still not known whether Aho-Corasick automata can be minimized in linear time. The purpose of this article is to give a natural probabilistic framework on sets of patterns, under which we explain the observations above. Based on this study, we also propose a linear algorithm which generically[1] computes the minimal automaton. When this algorithm fails to output the minimal automaton, it still produces an automaton that recognizes A^*X and whose size is intermediate between the size of Aho-Corasick automaton and the one of the corresponding minimal automaton. For this reason, we call it a *pseudo-minimization*[2] *algorithm*.

Contrary to what we stated, the algorithm proposed in [2] does not always build the minimal automaton; we can nonetheless prove that it generically does. The present article can be seen as a natural continuation of this framework of analysis, which results in a much simpler algorithm, which works for more general probabilistic models. Especially, the number of words do not need to be fixed anymore in our new settings.

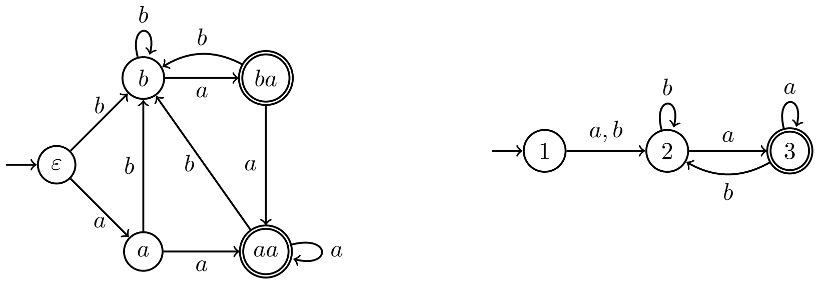


Fig. 1. On the left, an Aho-Corasick automaton that is not minimal, for $X = \{aa, ba\}$, and on the right, the corresponding minimal automaton with three states

Our probabilistic framework can be described informally as follows. A set of patterns is generically of size $\mathcal{O}(n)$, n being the asymptotic parameter[3], with patterns of length at least $\Omega(\log n)$ and with low correlation. Low correlation means that a factor of length ℓ in a pattern of X has probability exponentially small in ℓ to appear elsewhere in X. These conditions are natural, since they are satisfied by classical probabilistic models on sets of words, as we shall see in Section 3.2. In particular they hold for the uniform distribution on sets of m words whose sum of lengths is n, with $m = \mathcal{O}(n^\gamma)$, for $0 \leq \gamma < \frac{1}{2}$.

[1] A property holds "generically" when the probability it holds tends to one as the size tends to infinity.

[2] The notion of pseudo-minimal automaton we define in this article is not to be confused with the one introduced by Dominique Revuz in [11] for acyclic automata.

[3] It is sometime convenient to allow a size $\mathcal{O}(n)$ instead of strictly equal to n.

Recall that most minimization algorithms proceed by merging Nerode-equivalent states, which leads to the minimal automaton. We first prove that under our probabilistic models, the Aho-Corasick automata generically have the two following properties:

- The failure transitions all end at states "near the root" (at distance at most $\mathcal{O}(\log n)$ from the initial state).
- Any state that is near the root is not Nerode-equivalent to another state.

This leads to consider another equivalence relation on states defined by: p and q are $\equiv$-equivalent when p and q are either both final or both not final, and for any letter a, either both $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ are failure transitions and $p' = q'$, or both transitions are tree transitions and $p' \equiv q'$. Notice that it differs from Nerode-equivalence in the fact that failure transitions must end at the same state instead of at equivalent states. Therefore merging $\equiv$-equivalent states results in a partially minimized automaton. However, generically, the two properties above ensure that $\equiv$-equivalence and Nerode-equivalence are equal, and that the partial minimization is in fact a full minimization. Moreover, $\equiv$-equivalence is easier to calculate than Nerode-equivalence, and we provide an algorithm that computes it in linear time, by adapting Revuz famous algorithm for minimizing acyclic automata [12].

The article is organized as follows. After recalling basic facts about words and automata in Section 2, we formally define and illustrate the considered probabilistic models in Section 3. In this section are also stated results on typical properties on associated Aho-Corasick automata. Our algorithms and the analysis of their complexities are then given in Section 4, along with experimental results.

2 Definitions and Notations

In the sequel, we always work on a fixed finite alphabet A with $k \geq 2$ letters.

Words. A word y is a *factor* of a word x if there exist two words u and v such that $x = u{\cdot}y{\cdot}v$. The word y is a *prefix* (resp. a *suffix*) of x if $u = \varepsilon$ (resp. $v = \varepsilon$), ε being the empty word. We say that y is a *proper* prefix (resp. suffix) of x if y is a prefix (resp. suffix) such that $y \neq x$. Given a set of words X, we denote by $\texttt{Pref}(X)$ the set of all prefixes of words in X and by $\|X\|$ its *size*, defined as the sum of the lengths of its words. If u is a word of length n, its letters have indices in $\{0, \dots, n-1\}$ and we denote by $u[i,j]$, for $0 \leq i \leq j \leq n-1$, the factor of u starting at index i and ending at index j.

Deterministic Automata. A *deterministic and complete finite automaton* (or just *automaton* for short since we do not consider other kinds of automata in this article) over a finite alphabet A is a quintuple $\mathcal{A} = (A, Q, q_0, F, \delta)$, where Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the *set of final states* and δ, the transition function, is a complete mapping from $Q \times A$ to Q. The transition function δ is extended inductively to $Q \times A^*$ by setting, for any $p \in Q$,

$\delta(p, \varepsilon) = p$ and for any word $u \in A^*$ and any letter $a \in A$, $\delta(p, ua) = \delta(\delta(p, u), a)$. A word u is *recognized* by the automaton when $\delta(q_0, u) \in F$. The *size* of an automaton is its number of states. A classical result states that to each regular language L (a set of the words recognized by an automaton) one can associate its smallest automaton which recognizes it. This automaton is unique and is called *the minimal automaton* of L. See [9] for more results about automata.

Aho-Corasick Algorithm. Let $X = \{u_1, u_2, \ldots, u_m\}$ be a set of m non-empty words, whose sum of lengths $\|X\| = \sum_{i=1}^{m} |u_i|$ is n. Aho-Corasick algorithm [1] builds an automaton that recognizes A^*X but which is not always minimal (see Figure 1). It has linear time and space complexities and proceeds in two main steps: The first step consists in constructing a tree-automaton $\mathcal{A}_X = (A, \texttt{Pref}(X), \varepsilon, X, \delta)$, whose states are labelled by the prefixes of the words of X. The initial state is the empty word ε and the set of final states is made of the words of X. The transition function δ is defined by $\delta(u, a) = ua$ for any word u and any letter a such that $ua \in \texttt{Pref}(X)$. Transitions of this kind are referred to as *tree transitions*. The second step consists in completing the transition function of the automaton $\mathcal{A}_X$, and possibly in adding some final states, to obtain the automaton $\mathcal{AC}_X$: For any state u and any letter a, $\delta(u, a)$ is the longest suffix of ua that is also in $\texttt{Pref}(X)$, and the set of final states is $\texttt{Pref}(X) \cap A^*X$; this can be done in linear time since the tree-automaton has already been calculated (more details on the complexity of the construction can be found in [6], proposition 2.9). The new transitions, which are not tree transitions, are called *failure transitions*. An example of Aho-Corasick's construction is depicted on Figure 2. For more information on Aho-Corasick automata and related algorithms, the reader is referred to [7].

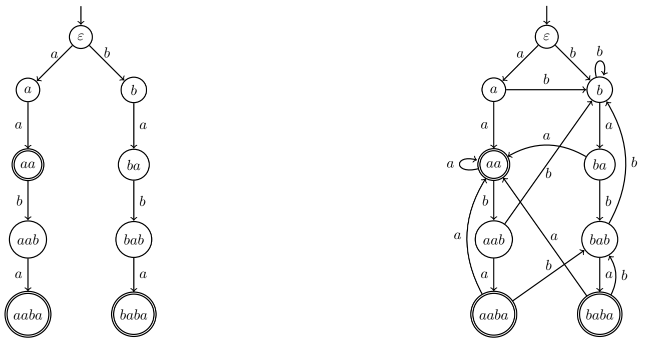


Fig. 2. The two steps construction of the Aho-Corasick automaton $\mathcal{AC}_X$ recognizing A^*X, where $X = \{aa, aaba, baba\}$. This automaton is not minimal because states aa, $aaba$ and $baba$ are equivalent, as are states a, aab and bab. Its minimal automaton has five states.

Note that since states of $\mathcal{AC}_X$ are words, we can define the size $|q|$ of a state as its length as a word of A^*.

3 Probabilistic Models and Generic Properties

In this section we define the probabilistic models that are studied throughout this article. Then we illustrate them on natural examples and, finally, we exhibit the two main properties that hold generically for these models, and that will be used in the algorithmic section (Section 4).

3.1 Probabilistic Models with Low Correlation

A *probabilistic model* is a sequence $(\mathbb{P}_n)_{n\geq 1}$ of probability measures on the same space. A property P is said to be *generic* for the probabilistic model $(\mathbb{P}_n)_{n\geq 1}$ when the probability that P is satisfied, tends to 1 when n tends to infinity.

Let $\mathcal{T}_m$ denote the set of all m-tuples of non-empty words and $\mathcal{T} = \cup_{m\geq 1}\mathcal{T}_m$. Under a probabilistic model $(\mathbb{P}_n)_{n\geq 1}$ over $\mathcal{T}$, a probability such as $\mathbb{P}_n\left(u_i \text{ satisfies } P\right)$ is to be understood as $\mathbb{P}_n\left(X \text{ has at least } i \text{ words and } u_i \text{ satisfies } P\right)$.

Definition 1. *A* probabilistic model with low correlation *is a probabilistic model $(\mathbb{P}_n)_{n\geq 1}$ on $\mathcal{T}$ such that there exist $C > 0$, $\alpha > 0$ and $\beta > 1$, and some $\lambda > \frac{8}{\log\beta}$ such that, the three following conditions hold generically:*
(1) *the tuple is of size at most Cn;*
(2) $\forall n \geq 1$, $\forall i, i' \geq 1$, $\forall j, j' \geq 0$, $\forall \ell \geq 0$, *if* $(i, j) \neq (i', j')$ *then*

$$\mathbb{P}_n\left(u_i[j, j+\ell-1] = u_{i'}[j', j'+\ell-1]\right) \leq \alpha\beta^{-\ell};$$

(3) *every word of the tuple is of length at least $\lambda \log n$.*

The definition above can be roughly described as follows:
• Because of Condition (1), the parameter n is, with high probability, an upper bound of the magnitude of the size of X: it is the parameter that will be used to measure the complexity. Note that it is convenient in some situations to allow this upper bound of $\mathcal{O}(n)$ instead of requiring the length to be exactly equal to n.
• Condition (2) means that generically, for two coordinates i and i' in the tuples and two starting positions j and j' in the words u_i and $u_{i'}$, the probability that the ℓ next letters coincide in u_i and $u_{i'}$ is exponentially small in ℓ.
• Condition (3) means that generically, the words in a tuple are not too small with respect to n. The value $\frac{8}{\log\beta}$ can be improved, but this is the smallest one that keeps our proof simple.

Hence, Condition (1) controls the size of the tuple, whereas Conditions (2) and (3) force the words of the tuple to have low self and mutual correlations. For instance it generically prevents that a word is a factor of another one.

▶ **Example:** Section 3.2 is devoted to examples of such distributions, but let us mention a first simple one to illustrate the definition. Let $A = \{a, b\}$ be a binary alphabet and for any $n \geq 1$, let $\mathbb{P}_n$ be the uniform distribution on words of length n, which are seen as 1-tuples of words. Condition (1) and Condition (3) hold, since the size of the tuple is the length of its only word, which is equal to n. Condition (2) also holds, since with probability 1 there is only one word: we only have to consider the case $i = i'$ with two different starting positions $j < j'$ in a random word u of length n. Using classical arguments from combinatorics on words [10], the number of words such that $u[j, j + \ell - 1]$ and $u[j', j' + \ell - 1]$ exist and are equal is at most $2^{n-\ell}$. Hence, Condition (2) holds with $\alpha = 1$ and $\beta = 2$. This simple probabilistic model is therefore of low correlation.

Lemma 1 below will be useful in the sequel and illustrates our terminology of "low correlation". If $X = (u_1, \ldots, u_k)$ is a tuple of words, a word w appears more than once as a factor in X if there exist $i, i' \geq 1$ and $j, j' \geq 0$, with $(i, j) \neq (i', j')$ such that $u_i[j, j + |w| - 1]$ and $u_{i'}[j', j' + |w| - 1]$ both exist and are equal to w.

Lemma 1. *Under a probabilistic model with low correlation, generically, no word of length* $\ell \geq \frac{\lambda}{2} \log n - 1$ *appears more than once as a factor in a tuple.*

Proof. The statement is similar to the one of Condition (2), except that it is not for fixed i, i', j, j' and ℓ. We bound the probability from above by summing the probabilities for each choice of the five parameters that satisfies our hypothesis.

Note that since Condition (1), Condition (2) and Condition (3) hold generically, it is sufficient to assume that they hold to prove the lemma, as we are looking for a generic result. We refer to such a tuple as a *generic tuple* in the rest of the proof.

By Condition (1), the size of a generic tuple is at most Cn, hence each of its words is also of length at most Cn. And since each word is non-empty, its length is at least 1, so that the number of components in a generic tuple is at most Cn. The length of a factor that can appear in a generic tuple is also at most Cn.

Hence, for $n \geq 1$, the probability that a word of length $\ell \geq \frac{\lambda}{2} \log n - 1$ appears more than once as a factor in a generic tuple is bounded from above by (we use Iverson bracket notation: $[\![P]\!]$ equals to 1 is P is true and 0 if P is false)

$$\begin{aligned}
\sum_{\ell=\frac{\lambda}{2}\log n-1}^{Cn} \sum_{i,i'=1}^{Cn} \sum_{j,j'=0}^{Cn-\ell-1} &\mathbb{P}_n \left(u_i[j, j+\ell-1] = u_{i'}[j', j'+\ell-1]\right) [\![(i,j) \neq (i',j')]\!] \\
&\leq C^4 n^4 \sum_{\ell=\frac{\lambda}{2}\log n-1}^{Cn} \alpha\beta^{-\ell} \leq C^4 n^4 \alpha \beta^{1-\frac{\lambda}{2}\log n} \sum_{m=0}^{\infty} \beta^{-m} \\
&\leq \frac{C^4 \alpha \beta^2}{\beta - 1} n^{4-\frac{\lambda}{2}\log\beta},
\end{aligned}$$

and this quantity tends to zero when n tends to infinity, since $\frac{\lambda}{2} \log \beta > 4$. □

3.2 Examples of Probabilistic Models with Low Correlation

▶ **Tuples of at most $\mathcal{O}(n^\gamma)$ words of total length n, $0 < \gamma < \frac{1}{2}$:** In this model, let $(m_n)_{n\geq 1}$ be a sequence of positive integers that is $\mathcal{O}(n^\gamma)$, for some $\gamma \in (0, \frac{1}{2})$. For any $n \geq 1$, we consider the uniform distribution on tuples $X = (u_1, \ldots, u_{m_n})$ of m_n words of total length n. A typical instance of this model is when m_n is a fixed integer (which is a distribution of tuple studied in [4]).

Obviously, Condition (1) always holds, with $C = 1$. Moreover, as mentioned in [4], such a tuple can be seen as a word u drawn uniformly at random in A^n and a composition of n into m_n parts also chosen uniformly at random and independently. For any integers i, i', j, j' such that $(i, j) \neq (i', j')$, if $u_i[j, j + \ell - 1] = u_{i'}[j', j' + \ell - 1]$, this factor appears at two different positions in u (the concatenation of the u_i's).The probability for a word of length n to contain a repetition of length ℓ is classically equal to $k^{-\ell}$. Therefore Condition (2) is satisfied for $\alpha = 1$ and $\beta = k$.

In order to prove that Condition (3) also holds generically, we use the following technical lemma:

Lemma 2. *For any γ and δ in $(0, 1)$ such that $2\gamma + \delta < 1$, generically, a composition of n into $m_n = \mathcal{O}(n^\gamma)$ parts contains no part of length smaller than n^δ.*

Since $\gamma < \frac{1}{2}$ in the model, there exists some $\delta > 0$ such that $2\gamma + \delta < 1$, and thus Lemma 2 applies. This proves Condition (3).

▶ **Tuples of n^γ words of length at most $n^{1-\gamma}$, $0 < \gamma < 1$:** Conditions (1) and (2) are direct consequences of the definition. Condition (3) is still easily checked since generically a word of length at most $n^{1-\gamma}$ is of length greater than $\frac{1}{2}n^{1-\gamma}$.

For an example of such a distribution, choose $\gamma = \frac{1}{2}$ and consider the set of tuples containing $m = \lfloor \sqrt{n} \rfloor$ words, each word being chosen uniformly at random (and independently) in the set $A^{\leq m}$ of all words of length at most m.

▶ **More advanced models for words:** For the same choices of distributions on sizes as above, if we enrich the model by generating the words according to a Bernoulli model (which associates a positive probability to each letter) or by an ergodic finite Markov chain, it is not difficult to check that the conditions for having low correlation still hold. It is mainly a consequence of the exponentially fast forgetfulness property of finite Markov chains (Bernoulli models are simpler instances of finite Markov chains).

3.3 Generic Properties of the Associated Aho-Corasick Automaton

Lemma 3. *Under a probabilistic model with low correlation over the set of tuples X from which the automaton is built, any state q of the Aho-Corasick automaton such that $|q| \geq \frac{\lambda}{2} \log n$ has generically only one incoming transition.*

Proof. For any $n \geq 1$, let X be a tuple that satisfies all conditions of low correlation and therefore the generic properties of Lemma 1.

If the property of Lemma 3 does not hold, there exists a state q, with $|q| \geq \frac{\lambda}{2} \log n$ and a failure transition $p \xrightarrow{a} q$ ending on q. By construction of the transition function δ in an Aho-Corasick automaton, q is the longest proper suffix of the word $p \cdot a$ that is also a prefix of a word of X. Therefore, $q = w \cdot a$ for some word w of length at least $\frac{\lambda}{2} \log n - 1$, since $|w| = |q| - 1$. The word w is a proper suffix of p, and a prefix of q; it is therefore a factor that appears at least twice in the tuple. By Lemma 1, this event does not happen generically, concluding the proof. □

Lemma 4. *Under a probabilistic model with low correlation over the set of tuples X from which the automaton is built, generically, every state q of the Aho-Corasick automaton such that $|q| < \frac{\lambda}{2} \log n$ is not Nerode-equivalent to another state of the automaton.*

Proof. For any $n \geq 1$, let X be a tuple that satisfies all conditions of low correlation and therefore the generic properties of Lemma 1 and Lemma 3. Let $\mathcal{AC}_X$ be the Aho-Corasick automaton associated with X and q be a state such that $|q| < \frac{\lambda}{2} \log n$.

We claim that q is not Nerode-equivalent to another state of $\mathcal{AC}_X$. By contradiction, assume that there exists a state $p \neq q$ that is Nerode-equivalent to q, with $|p| \geq |q|$ (if $|p| < |q|$ then $|p| < \frac{\lambda}{2} \log n$ and one can switch the roles of p and q). Let u be a word such that there exists a path from p to a final state labelled by u that uses tree transitions only.

Consider the path labelled by u starting from q; since p and q are equivalent, this path ends at a final state f. Hence, by Condition (3) and since $|q| < \frac{\lambda}{2} \log n$, we have $|u| \geq \frac{\lambda}{2} \log n$. This path is made of tree and failure transitions, and there is at least one failure transition by Lemma 1, otherwise X would contain two occurrences of u, but $|u| \geq \frac{\lambda}{2} \log n$. Let r be the last state reached by a failure transition along this path, and let w be the suffix of $u = vw$ that labels this path from r to f. By construction, the path starting from r and labelled by w uses tree transitions only. Hence w labels paths using tree transitions only, from both states r and $p \cdot v$. These two states are different: since the path starting at state q and labelled by v uses at least one failure transition, $|r| = |\delta(q, v)| < |q| + |v|$ and since $|q| \leq |p|$, this implies that $|r| < |p \cdot v|$.

At this point, we have proved that w appears at least twice as a factor in X. It only remains to prove that w is long enough to make this situation impossible in our settings: since X satisfies the property of Lemma 3 and r is the target of a failure transition, it has at least two incoming transitions and then $|r| < \frac{\lambda}{2} \log n$. Moreover, since $r \cdot w \in X$ and X satisfies Condition (3), then $|r| + |w| \geq \lambda \log n$. Hence $|w| \geq \frac{\lambda}{2} \log n$, yielding the contradiction for a generic X, by Lemma 1. □

4 A Pseudo-minimization Algorithm for Aho-Corasick Automata

In this section we present a simple algorithm that, under a low correlation probabilistic model for the inputs, generically minimizes the Aho-Corasick automaton in linear time and space.

4.1 Algorithm

The pseudo-minimization algorithm[4] we propose is an adaptation of Revuz algorithm [12] that minimizes acyclic deterministic automata.

Let us first introduce the notion of $\equiv$-*equivalence* on states of an automaton. Two states p and q are $\equiv$-*equivalent* when both p and q are terminal or both non-terminal, and for every letter a, either both $p \xrightarrow{a}$ and $q \xrightarrow{a}$ are tree transitions and $\delta(p, a)$ is $\equiv$-equivalent to $\delta(q, a)$, or both $p \xrightarrow{a}$ and $q \xrightarrow{a}$ are failure transitions and $\delta(p, a) = \delta(q, a)$. Therefore if the two transitions are not of the same nature, the two states are not $\equiv$-equivalent. Note that if two states are $\equiv$-equivalent they have to be at the same distance from terminal states. We use this observation to define the *distance function* d on the states of the Aho-Corasick automaton. For any state p, $d(p)$ is the maximal distance from p to a leaf in the prefix tree: $d(p) = \max\{|w| \mid p \cdot w \in X\}$. Denote also by $\mathcal{D}_i$ the set of states at distance i; the distance function defines a partition $\mathcal{D} = (\mathcal{D}_i)_{0 \le i \le d(\varepsilon)}$ of the set of states Q. The distance function relies on tree transitions only and can be computed bottom-up by a depth-first traversal of the tree.

If we have already computed $\equiv$-equivalent states for all the classes $\mathcal{D}_j$ with $j < i$, we can easily compute the $\equiv$-equivalent states in the class $\mathcal{D}_i$, using the definition. This leads to a simple pseudo-minimization method (see Algorithm 1 below).

Algorithm 1. Pseudo-Minimization Method

```
  Inputs  : Aho-Corasick Automaton $\mathcal{AC}_X$
  Outputs: Pseudo-Minimal Automaton
1 compute $\mathcal{D}$;
2 for $i = 0$ to $d(\varepsilon)$ do
3     detect $\equiv$-equivalent states at distance $i$;
4     merge them;
```

More precisely, to detect $\equiv$-equivalent states at distance i, we proceed as follows. Let p and q be two states with $d(p) = d(q) = i$. Assuming that for $j < i$, $\equiv$-equivalent states in classes $\mathcal{D}_j$ have already been merged, p and q are $\equiv$-equivalent if and only if both p and q are terminal or non-terminal, and for every $a \in A$, $\delta(p, a) = \delta(q, a)$. The last item differs from the definition because of the dynamic merging of $\equiv$-equivalent states in the process of the method.

Algorithm 2 explains how $\equiv$-equivalent states at distance i are detected. Given the list of the states of $\mathcal{D}_i$, it first splits the list in two: terminal and non-terminal states. Then these groups are, in turn, split according to each letter of the alphabet, by using a kind of bucket sort algorithm on outgoing transitions, but in which we are only interested in identifying states having identical outgoing transitions (not actually sorting them). In order to test only non-empty lists in

[4] An implementation of the algorithm can be found in http://lipn.fr/~aitmous/

Algorithm 2. State-Distinguishing Algorithm

```
Inputs  : Set of states $S$
Outputs: A set of lists of pseudo-equivalent states
1  $a \leftarrow$ first letter of $A$;
2  split $S$ into two lists $L_1$ (of terminal states) and $L_2$ (of non-terminal states);
3  $R_1 \leftarrow \{L_1, L_2\}$;
4  $R_2 \leftarrow \emptyset$;
5  while $R_1 \neq \emptyset$ and $a \neq$ null do
6      $R_2 \leftarrow R_1$;
7      $R_1 \leftarrow \emptyset$;
8      while $R_2 \neq \emptyset$ do
9          $L \leftarrow$ first list of $R_2$;
10         create an array $T$ of $|Q|$ empty lists;
11         for all states $p \in L$ do
12             $q \leftarrow \delta(p, a)$;
13             add $p$ to the list $T[q]$;
14         for all non-empty lists $\ell \in T$ do
15             if $|\ell| \geq 2$ then
16                 add $\ell$ to $R_1$;
17     $a \leftarrow$ next letter of $A$; // returns null if there is no more letter in $A$
18 return $R_1$;
```

line 14, we can use a list NE to keep track of non-empty lists. Whenever a state p is inserted in an empty list $T[q]$ in line 13, q is inserted in the list NE.

Figure 3 illustrates the pseudo-minimization of the automaton given Figure 2, showing an example where the pseudo-minimization algorithm fails to output the minimal automaton.

Lemma 5. *With the states of $\mathcal{D}_i$, and all equivalent states at distance less than i already merged, Algorithm 2 computes the classes of $\equiv$-equivalent states of $\mathcal{D}_i$.*

Proof. The proof of the proposition uses the same arguments as in [12]. We prove the validity of the algorithm by induction on the number of executions of the outer loop (lines 5-16). After c executions, R_1 contains only lists of states that are either all terminal or all non-terminal, and have the same outgoing transitions labelled by the c first letters of the alphabet.

In lines 8-16, the list of states L is split by the next outgoing transitions. Thus the lists added to R_1 only contain states with the same $c + 1$ first transitions. Since the inner loop is executed on every list in R_2, the property holds.

At the end of the outer-loop, if there are $\equiv$-equivalent states, they are in the same lists of R_1. □

Theorem 1. *Under a probabilistic model with low correlation on the inputs of the algorithm, the pseudo-minimization algorithm generically returns the minimal automaton of A^*X.*

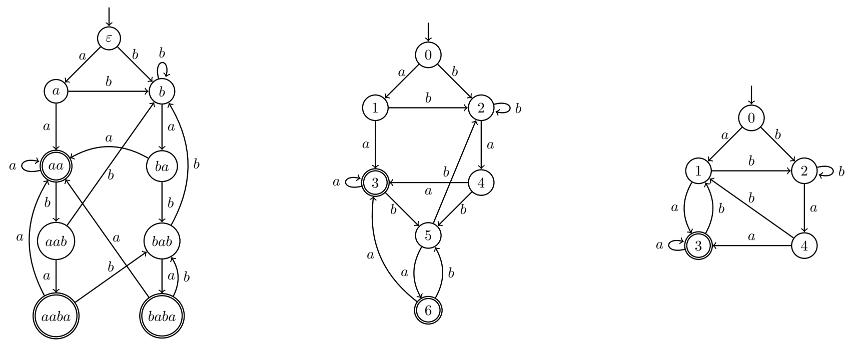


Fig. 3. The pseudo-minimization of the Aho-Corasick automaton recognizing A^*X, where $X = \{aa, aaba, baba\}$ (left), produces the automaton (center). The automaton is not minimal since states 3 and 6 are equivalent, as are states 1 and 5. On the right is depicted the minimal automaton.

Proof. Let $\mathcal{A}$ be an Aho-Corasick automaton satisfying the statements of Lemmas 3 and 4. Since these properties are generic, it is sufficient to prove that such an automaton is minimized by the pseudo-minimization algorithm.

Let p and q be two Nerode-equivalent states of $\mathcal{A}$ such that $p \neq q$. For any letter $a \in A$, if $\delta(p, a)$ is a failure transition, then by Lemma 3, $\delta(p, a)$ ends at some state r with $|r| < \frac{\lambda}{2} \log n$. And by Lemma 4 the state r is not Nerode-equivalent to another state. Hence $\delta(q, a) = r$ too. If $\delta(p, a)$ and $\delta(q, a)$ are both tree transitions and p and q are in $\mathcal{D}_i$, then $p \cdot a$ and $q \cdot a$ are in $\mathcal{D}_j$ for some $j < i$, and have already been merged if they were Nerode-equivalent. Hence $\delta(p, a) = \delta(q, a)$ in this case too. □

4.2 Complexity

Lemma 6. *Algorithm 2 distinguishes r states in space and time $\mathcal{O}(|Q| + r)$.*

Proof. Lines 14-16 take at most the same number of steps as the loop in lines 11-13 (in the worst case, each state of L has a different outgoing transition labeled by a). Thus the time complexity is bounded by the number of times line 11 is executed and that is kr in the worst case. The additional time and space is used for the array T of size $|Q|$. □

The space complexity of Algorithm 2 is linear in $|Q|$ since, for any letter a, lists of states are split according to their outgoing transition labelled by a, which can reach any state of $\mathcal{AC}_X$. Hence, the array T is of size $|Q|$. This bound can be lowered by a renumbering of the states, using an idea of Revuz [12]: at step i, we use a counter to associate numbers to states that are adjacent to elements of $\mathcal{D}_i$. These numbers range from 1 to the number n_i of such states. Since the automaton is deterministic, $n_i \leq |\mathcal{D}_i| \times k$. We gain in complexity using these numbers as indices of T instead of elements of Q. To avoid conflicts when going to step $i+1$, we also store for each state at which step its number has been updated: at Line 12, when a state $q = \delta(p, a)$ is checked for some state p, either q has already been numbered at this step and we do not change it, or we associate q with the next value of the counter. In both cases, p is added at the corresponding index in T.

Theorem 2. *Algorithm 1 is linear in time and space.*

Proof. The distance function can be computed in linear time using a depth-first traversal of the tree automaton. Once it is done, the calculation of $\equiv$-equivalent states is done bottom-up, using a depth-first traversal of the tree automaton too. Using the renumbering technique mentioned above, we avoid the $|Q|$ multiplier at each level, which leads to an overall linear complexity, since $\sum_{i=0}^{d(\varepsilon)} |\mathcal{D}_i|$ is equal to the number of states of the Aho-Corasick automaton. $\square$

Remark: in practice, one can use closed hashing to assign a unique number to every $\equiv$-equivalent class or state label both viewed as strings. We see this closed hash table as an injective function from strings to positive integers. Start with adding the label of every state in the hash table. Then associate to any state q a string signature $\sigma(q)$ of the form $\sigma(q) =$ "$T|n_a|n_b|n_c$", for $A = \{a, b, c\}$, where T stands for "q is terminal", and where n_a is the number assigned by the hash table to the class of $\delta(q, a)$ if it is a tree transition, or the number assigned to state $\delta(q, a)$ if it is a failure transition. Doing this bottom-up, two states are $\equiv$-equivalent if and only if they have the same signature, the number associated to the signature being obtained by a search in the hash table. This leads to a linear algorithm in practice.

4.3 Experimental Results

We experimented our algorithm on different types of patterns. First we tested it on the genome of the *mycoplasma genitalium*, a parasitic bacterium. The genome consists of 482 protein encoding genes, for a total length of more than 580,000 base pairs. For this experiment, our algorithm did output the minimal automaton. There are no small words, as the genes' length range from around 100 to around 5500. It is therefore in the framework of our theoretical model, which explains why our algorithm behaves well, as shown in Table 1.

Table 1. For the mycoplasma genitalium genome, the pseudo-minimization algorithm actually computes the minimal automaton

Aho-Corasick automaton	Pseudo-minimal automaton	Minimal automaton
528 670 states	526 347 states	526 347 states

We also tried our algorithm on a French dictionary, which has almost 380,000 words. This result depicted in Table 2 has been anticipated: as a consequence of having both a lot of small words and high correlations, we are clearly out of our low correlation model, and the classical minimization is more efficient in reducing the space representation (though a lot of space is already gained using our linear algorithm).

Table 2. For the French dictionary, the pseudo-minimization does not compute the minimal automaton

Aho-Corasick automaton	Pseudo-minimal automaton	Minimal automaton
722 074 states	113 130 states	27 362 states

We also observed in several situations that though not computing the minimal automaton, the output of our algorithm differs from the minimal automaton by only a few states. This is typically the case when taking a set made of all the words that differ from a random word u by at most one letter.

5 Conclusion

We proposed a pseudo-minimization algorithm for Aho-Corasick automata and proved that, under natural probabilistic models, it outputs the minimal automaton with high probability. In order to obtain more general results we isolated the probabilistic properties that make the analysis work from the study of models of patterns that satisfy these properties: this way we obtained more general results, that can be reused for new models, provided one checks that the low correlation properties hold. Note that there is a trade off between Conditions (2) and (3): one can ask for a polynomial decrease in Condition (2) at the cost of a more restrictive size requirement in Condition (3). We choose this statement because most common sources for words satisfy the exponential decrease of Condition (2).

Experiments show that for probabilistic models with low correlation, the algorithm as expected outputs the minimal automaton. An interesting phenomenon can be observed for patterns that are clearly out of our probabilistic framework. Although not computing the minimal automaton, the algorithm still computes an automaton whose number of states is often close to that of the minimal automaton.

A natural continuation of this work is to quantify that kind of observations for relaxed probabilistic models.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
2. AitMous, O., Bassino, F., Nicaud, C.: Building the Minimal Automaton of A^*X in Linear Time, When X Is of Bounded Cardinality. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 275–287. Springer, Heidelberg (2010)
3. Baker, T.P.: A technique for extending rapid exact-match string matching to arrays of more than one dimension. SIAM J. Comput., 533–541 (1978)
4. Bassino, F., Giambruno, L., Nicaud, C.: The average state complexity of rational operations on finite languages. Int. J. Found. Comput. Sci. 21(4), 495–516 (2010)
5. Bird, R.S.: Two dimensional pattern matching. Inf. Process. Lett. 6(5), 168–170 (1977)
6. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. Cambridge University Press (2007)
7. Crochemore, M., Rytter, W.: Text Algorithms. Oxford Univ. Press (1994)
8. Hopcroft, J.E.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Theory of Machines and Computations, pp. 189–196. Academic Press (1971)
9. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
10. Lothaire, M.: Applied Combinatorics on Words. Cambridge University Press (2005)
11. Revuz, D.: Dictionnaires et lexiques: methodes et algorithmes. PhD thesis, Institut Blaise Pascal (1991)
12. Revuz, D.: Minimisation of acyclic deterministic automata in linear time. Theoret. Comput. Sci. 92(1), 181–189 (1992)

Efficient Two-Dimensional Pattern Matching with Scaling and Rotation and Higher-Order Interpolation

Christian Hundt and Florian Wendland

Institut für Informatik, Universität Rostock, Germany
{christian.hundt,florian.wendland}@uni-rostock.de

Abstract. Two-dimensional pattern matching with scaling and rotation for given pattern P and text T is the computational problem of finding a subtext in T such that a scaled and rotated transformation of P most accurately resembles the subtext. Applications of pattern matching are found, for instance, in computer vision, medical imaging, pattern recognition and watermarking. All known approaches to find a globally optimal matching depend on the basic nearest-neighbor interpolation. To use higher-order interpolations, current algorithms apply numerical techniques that provide only locally optimal solutions. This paper presents the first algorithm to find an optimal match under a large class of higher-order interpolation methods including bilinear and bicubic. The algorithm exploits a discrete characterization of the parameter space for scalings and rotations to achieve a polynomial time complexity.

Keywords: combinatorial pattern matching, discrete scalings and rotations, higher-order interpolation methods, discrete algorithms.

1 Introduction

For a two-dimensional pattern P and a text T, the objective of two-dimensional pattern matching, considered in this paper, is to determine a combination f of scaling and rotation such that the transformed pattern $f(P)$ most closely resembles some subtext T' of T. Although the problem originates from the challenge of recognizing patterns in strings, 2D-patterns and texts are usually interpreted as digital images.

A digital image consists of color values defined only at integer coordinates, also called pixels. Applying the transformation $f : \mathbb{R}^2 \to \mathbb{R}^2$ to pixels $(i, j)^T$, the coordinates $(x, y)^T = f(i, j)$ are probably not integral and a priori, they have no color values. In image processing, this difficulty is usually overcome by using interpolation methods to assign colors to all real points $(x, y)^T$ derived from the values of neighboring pixels. Certainly, the globally optimal matching depends crucially on the applied interpolation method which is subject to the particular application. Although many practical applications prefer higher-order interpolation methods, many approaches to 2D-pattern matching build on simple nearest neighbor interpolation.

This paper presents the first polynomial time 2D-pattern matching algorithm to find a globally optimal solution for combinations of scalings and rotations as well as a large class of higher-order interpolation methods including bilinear and bicubic interpolation.

Because of space limitations, all proofs have been removed.

Previous Work

Two-dimensional pattern matching has a wide range of applications, as for instance in computer vision [15], medical imaging [6, 18, 19], pattern recognition and digital watermarking [7].

Especially in medical imaging [19], the problem is solved by interpreting the pattern P and the text T as continuous mappings $p, t : \mathbb{R}^2 \to \mathbb{R}$ and subsequently minimizing $\int_{(x,y)\in\mathbb{R}^2} \|p(f(x,y)) - t(x,y)\| \, dx \, dy$ over all admissible transformations f. This approach, called image registration, finds locally optimal solutions but has difficulties to find a global optimum.

Feature based techniques to 2D-pattern matching, proposed for instance in [1, 16], extract salient features like points, lines and regions from P and T, and search for a transformation f that moves the geometrical objects of P closest to a subset of features from T. Feature matching, however, is also a difficult problem, even for points [13] and simple transformations such as rotation and translation [14].

Generalizing the search of patterns in strings, research in combinatorial pattern matching [2–5, 10, 11, 17] recently showed how to efficiently solve the 2D-pattern matching problem of finding all exact copies of a given pattern P and its transformations $f(P)$ in a text T if only nearest-neighbor interpolation is used. Their methods can be easily transferred to the case considered in this paper where the best match is sought.

The common idea is to find a subset $f_1, f_2, \ldots$ of transformations that are eligible to completely render $\mathcal{D}(P)$, the dictionary of all possible transformed patterns, i. e., $\mathcal{D}(P) = \{f_1(P), f_2(P), \ldots\}$. Then, 2D-pattern matching is conveniently solved by selecting a subtext T' that is most similar to one of the transformed patterns in $\mathcal{D}(P)$. Fredriksson and Ukkonen [11] were the first to apply this idea successfully for rotations and Amir et al. [3] found a similar discretization for scalings. Finally, Hundt et al. [12] developed a unique approach for combinations of scaling and rotation.

However, all these techniques build on nearest-neighbor interpolation. Although higher-order methods, such as bilinear or bicubic interpolation, are preferred in image processing applications, no analogous result has been known until recently. Wendland [22] demonstrated how to realize the same ideas for bilinear interpolation and introduced an algorithm to compute the dictionaries of all bilinear interpolated, scaled patterns in $O(mn^2 \log n)$ time and all bilinear interpolated, rotated patterns in $O(m^3 \log m)$ time.

Contribution of This Paper

This paper shows for a large class of interpolation methods $\mathcal{I}$ how to efficiently compute the dictionary $\mathcal{D}(P,\mathcal{I})$ that contains all $\mathcal{I}$-interpolated transformations of input pattern P obtained by simultaneously scaling by some factor $s \in \mathbb{R}$ and rotating by some angle $\alpha \in \mathbb{R}$.

For every $\mathcal{I}$, the dictionary $\mathcal{D}(P,\mathcal{I})$ is a finite set, whereas the set of all $(s,\alpha) \in \mathbb{R}^2$ is uncountable. Hence, $\mathbb{R}^2$ necessarily decomposes into equivalence classes where every class consists of all pairs (s,α) that transform P into the same element of $\mathcal{D}(P,\mathcal{I})$.

The main result of this paper is a precise description of these equivalence classes in dependence of P and $\mathcal{I}$. It shows how to efficiently compute the corresponding partition of $\mathbb{R}^2$ in two stages. The first stage subdivides $\mathbb{R}^2$ according to a set of lines that are defined just by the size of P. This works like the approach to 2D-pattern matching for nearest-neighbor interpolation [12]. The second stage refines the partition in a non-linear fashion that depends on $\mathcal{I}$ and the color values of P. The difficulties of this step are caused by the non-linear borders between the equivalence classes and the need to handle irrational numbers. Consequently, finding representative transformations $f_1, f_2, \ldots$ for all equivalence classes to compute $\mathcal{D}(P,\mathcal{I})$ represents the main challenge of this paper.

2 Technical Preliminaries

In this paper, a pattern P is a mapping $P : \mathbb{Z}^2 \to \mathbb{N} \cup \{\perp\}$ that assign a color value $P\langle i,j\rangle$ to every pixel $(i,j)^T \in \mathbb{Z}^2$. The size of P is denoted by $m \in \mathbb{N}$, which means that P has support $z(m) = \{(i,j)^T \mid -m \leq i,j \leq m\}$. For all pixels $(i,j)^T \in z(m)$, the color value $P\langle i,j\rangle$ is a natural number and for all $(i,j)^T \notin z(m)$ a special color $\perp$ marks the exterior of P, i. e., $P\langle i,j\rangle = \perp$. The largest color value in P is denoted by $c(P) = \max\{P\langle i,j\rangle \mid (i,j)^T \in z(m)\}$. The same definitions also apply to the text T, which has size $n \in \mathbb{N}$ and support $z(n) = \{(i,j)^T \mid -n \leq i,j \leq n\}$.

Interpolation methods $\mathcal{I}$ define color values at real points $(x,y)^T \in \mathbb{R}^2$ in patterns or texts. This requires arithmetics with colors which work like for natural numbers, except that operations fed with argument $\perp$ have result $\perp$. Every method $\mathcal{I}$ is defined as a set of mappings $\mathcal{I}_{di,dj} : [0,1)^2 \to [0,1]$ that describe how the color values of points $(dx,dy)^T \in [0,1)^2$ depend on the surrounding pixels $(di,dj)^T \in \mathbb{Z}^2$. The $\mathcal{I}$-interpolation of a given pattern P assigns a color value $P_{\mathcal{I}}\langle x,y\rangle$ to all $(x,y)^T \in \mathbb{R}^2$ as

$$P_{\mathcal{I}}\langle x,y\rangle = \left[\sum_{(di,dj)^T \in \mathbb{Z}^2} \mathcal{I}_{di,dj}(x - \lfloor x\rfloor, y - \lfloor y\rfloor) \cdot P\langle \lfloor x\rfloor + di, \lfloor y\rfloor + dj\rangle\right] \qquad (1)$$

where $\lfloor\cdot\rfloor$ denotes the floor function and $[\cdot]$ denotes rounding. For convenience, let also $\lfloor(x,y)^T\rfloor = (\lfloor x\rfloor, \lfloor y\rfloor)^T$ and $[(x,y)^T] = ([x],[y])^T$ for all vectors $(x,y)^T \in \mathbb{R}^2$. This notion allows, e. g., to define bilinear interpolation $\mathcal{I}^b$ by the mappings

$$\mathcal{I}^b_{di,dj}(dx,dy) = \begin{cases} (1-|dx-di|)\cdot(1-|dy-dj|) & : \text{ if } di,dj \in \{0,1\} \\ 0 & : \text{ otherwise,} \end{cases}$$

but also allows for more complex methods such as bicubic interpolation.

An interpolation method $\mathcal{I}$ is called feasible if (1) the mappings $\mathcal{I}_{di,dj}$ are polynomial functions for all $(di,dj)^T \in \mathbb{Z}^2$, (2) the degree of every polynomial $\mathcal{I}_{di,dj}$ is at most d for some fixed $d \in \mathbb{N}$, (3) every coefficient of any $\mathcal{I}_{di,dj}$ is a rational number with numerator and denominator encoded by b bits (plus one sign bit) for some fixed $b \in \mathbb{N}$ and (4) there is a fixed radius $r \in \mathbb{N}$ such that $\mathcal{I}_{di,dj} = 0$ for every $(di,dj)^T$ that fulfill $|di| \geq r$ or $|dj| \geq r$. In that case, $\mathcal{I}$ is called k-feasible for $k = \max\{d,b,r\}$. Many common higher methods, like bicubic interpolation, are feasible and, for example, $\mathcal{I}^b$ is 2-feasible.

The set of combined scaling and rotation contains exactly all functions

$$f(x,y) = \begin{pmatrix} s\cos\alpha & s\sin\alpha \\ -s\sin\alpha & s\cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{2}$$

given by some $s,\alpha \in \mathbb{R}$, where $s \neq 0$. Applying the transformation f to pattern P using interpolation method $\mathcal{I}$ yields a transformed pattern $f(P_\mathcal{I})$ of size $2n$. The color values of all pixels $(i,j)^T$ in the pattern $f(P_\mathcal{I})$ are defined using the inverse transformation f^{-1} which combines scaling by s^{-1} and rotation by $-\alpha$. This paper uses parameters $p = s^{-1}\cos\alpha$ and $q = -s^{-1}\sin\alpha$ to specify the inverse of f and this is denoted explicitly by $f^{-1}_{p,q}$. Then, the color values of all $(i,j)^T \in z(2n)$ are defined as $f(P_\mathcal{I})\langle i,j\rangle = P_\mathcal{I}\langle x,y\rangle$ with $(x,y)^T = f^{-1}_{p,q}(i,j)$. According to this, $\mathcal{D}(P,\mathcal{I})$ is the dictionary of all transformed patterns $f(P_\mathcal{I})$ with size $2n$ obtained by any combination f of scaling and rotation.

The distortion between the f-transformed pattern $f(P_\mathcal{I})$ and a subtext of T is given by $\Delta_{i,j}(f,P_\mathcal{I},T)$, a function computed in polynomial time $t(m,n)$, where $(i,j)^T$ selects a subtext of T. A possible choice is

$$\Delta_{i,j}(f,P_\mathcal{I},T) = \sum_{(i',j')^T \in \mathbb{Z}^2} |f(P_\mathcal{I})\langle i',j'\rangle - T\langle i'+i,j'+j\rangle|$$

where $|\bot| = 0$ to ignore $\bot$-pixels of $f(P_\mathcal{I})$ and T.

Finally, 2D-pattern matching with interpolation $\mathcal{I}$ and distortion Δ can be formalized as follows: For a given pattern P of size m and a text T of size n find a transformed pattern $f(P_\mathcal{I})$ in $\mathcal{D}(P,\mathcal{I})$ and select a subtext by $(i,j)^T \in z(n)$ such that the distortion $\Delta_{i,j}(f,P_\mathcal{I},T)$ is minimum over all possible choices.

3 Algebraic Parameter Space Characterization

Following the definitions of the previous section related to Equation 2, every inverse transformation f^{-1} determines one parameter $(p,q)^T \in \mathbb{R}^2$. This is in fact a one-to-one correspondence between the set of inverse transformations and the parameters, because every $(p,q)^T \in \mathbb{R}^2$ defines $f^{-1}_{p,q}$ with scaling $\sqrt{p^2+q^2}$ and rotation $\arccos(p(p^2+q^2)^{-2})$. The only exception occurs at point $p = q = 0$ which does not define an injective function $f^{-1}_{p,q}$ and which is therefore ignored in this paper. Consequently, it is meaningful to consider $\mathbb{R}^2$ as a parameter space for the set of inverse transformations.

The central tool developed in this paper is a subdivision of the parameter space $\mathbb{R}^2$ into a polynomial number of subspaces $\psi_1, \ldots, \psi_N$ such that any pair of transformations f_1, f_2 gives the same transformation $f_1(P_{\mathcal{I}}) = f_2(P_{\mathcal{I}})$ of a given pattern P if their inverse transformations $f^{-1}_{p_1,q_1}$ and $f^{-1}_{p_2,q_2}$ are represented by parameters $(p_1, q_1)^T$ and $(p_2, q_2)^T$ contained in the same subspace ψ_k for some $k \in \{1, \ldots, N\}$. This means that each of the N subspaces represents one transformed pattern in the dictionary $\mathcal{D}(P, \mathcal{I})$.

This algebraic characterization of the parameter space is used in the new algorithm to compute a polynomial number of transformations $f_1, \ldots, f_N$ that are eligible to render the complete dictionary $\mathcal{D}(P, \mathcal{I}) = \{f_1(P_{\mathcal{I}}), \ldots, f_N(P_{\mathcal{I}})\}$. Afterwards, the best match is found by the comparison of every pattern in $\mathcal{D}(P, \mathcal{I})$ and every subtext of T.

The parameter space subdivision is defined in two stages. The first one depends only on the size of P. It partitions $\mathbb{R}^2$ into convex subspaces $\phi_1, \ldots, \phi_M$. The benefit of this is that every subspace $\phi_w, w \in \{1, \ldots, M\}$ is associated with two mappings X_{ϕ_w} and Y_{ϕ_w} such that all parameters $(p, q)^T \in \phi_w$ fulfill for all pixels $(i, j)^T \in z(2n)$ that $\lfloor f^{-1}_{p,q}(i, j) \rfloor = (X_{\phi_w}(i, j), Y_{\phi_w}(i, j))^T$. This means that all transformations represented by ϕ_w have a similar behavior. Particularly, for all pixels $(i, j)^T \in z(2n)$ there is a unit square given by its lower left point $(X_{\phi_w}(i, j), Y_{\phi_w}(i, j))^T$ such that all inverse transformations $f^{-1}_{p,q}$ represented by parameters $(p, q)^T \in \phi_w$ transform $(i, j)^T$ to a point $(x, y)^T = f^{-1}_{p,q}(i, j)$ located in the given square, i. e., $\lfloor (x, y)^T \rfloor = (X_{\phi_w}(i, j), Y_{\phi_w}(i, j))^T$.

The basic idea of the second stage is to use the properties of feasible interpolation methods $\mathcal{I}$ and the mappings X_{ϕ_w} and Y_{ϕ_w} to refine the primal partition by subdividing every ϕ_w into subsubspaces $\psi_1, \ldots, \psi_{M_w}$. In fact, according to Equation 1, feasible interpolation of a color value $f(P_{\mathcal{I}})\langle i, j\rangle$ at the pixel $(i, j)^T$ is defined as a sum of $\mathcal{I}$-polynomials whose coefficients are multiplied by the color values of pixels around the point $(x, y)^T = f^{-1}_{p,q}(i, j)$. Considering the parameters $(p, q)^T$ of a single subspace ϕ_w the advantage of the mappings X_{ϕ_w} and Y_{ϕ_w} is that the approximate location of $(x, y)^T$, which is the unit square at $(X_{\phi_w}(i, j), Y_{\phi_w}(i, j))^T$, is already known. This fixes the pixels surrounding $(x, y)^T$ and has the effect that $f(P_{\mathcal{I}})\langle i, j\rangle$ is given by a unique sum of polynomials for all parameters from ϕ_w. Then, the sum's value depends only on the position of $(x, y)^T$ relative to the square's base point $(X_{\phi_w}(i, j), Y_{\phi_w}(i, j))^T$. Now, it is possible to subdivide the subspace ϕ_w exactly into the regions, where the interpolated value $f(P_{\mathcal{I}})\langle i, j\rangle$ is fixed to a specific color for all pixels $(i, j)^T \in z(2n)$.

The advantage of this two stage process is that the first one works on a global level which is independent of $\mathcal{I}$ and the actual color values in P. Although, it does not establish the complete partition, it helps to use the gained structure and proceed from this level. Subsequently, the refinement of the primal partition, which is much more technically involved, can be carried out locally and on its own for every subspace.

Convex Primal Partition

The partition of the parameter space $\mathbb{R}^2$ into the convex spaces $\phi_1, \dots, \phi_M$ is connected to the following proposition.

Proposition 1. *Let $(i,j)^T$ be a pixel in the support $z(2n)$, let $(i',j')^T$ be an integer pair and let $f_{p,q}^{-1}$ be an inverse combination of scaling and rotation represented by the parameter $(p,q)^T \in \mathbb{R}^2$. The transformed point $(x,y)^T = f_{p,q}^{-1}(i,j)$ fulfills $\lfloor x \rfloor \geq i'$ if and only if $ip + jq \geq i'$. Analogously, it is true that $\lfloor y \rfloor \geq j'$ if and only if $jp - iq \geq j'$.*

According to the proposition, it is possible to algebraically express the set of parameters $(p,q)^T$ which represent inverse transformations $f_{p,q}^{-1}$ that transform the pixel $(i,j)^T$ to a point $(x,y)^T = f_{p,q}^{-1}(i,j)$ belonging to a unit square with lower left point $(i',j')^T \in \mathbb{Z}^2$. In particular, it is true that $(x,y)^T$ belongs to the given square, i. e., $\lfloor (x,y)^T \rfloor = (i',j')^T$, if and only if (1) $ip + jq \geq i'$, (2) $ip + jq < i' + 1$, (3) $jp - iq \geq j'$ and (4) $jp - iq < j' + 1$. The set of parameters fulfilling these conditions forms a region in $\mathbb{R}^2$ that is bounded by lines.

The boundary lines of such regions can be described as linear polynomials. For a pixel $(i,j)^T \in z(2n)$ and any unit square with lower left point $(i',j')^T \in \mathbb{Z}^2$, let

$$e^{\mathtt{h}}_{i,j,i'}(p,q) = ip + jq - i' \quad \text{and} \quad e^{\mathtt{v}}_{i,j,j'}(p,q) = jp - iq - j'.$$

A line cuts the parameter space into two subspaces and to fix a notation, let

$$\begin{aligned} L^+(e^\star_{i,j,h}) &= \{(p,q)^T \mid e^\star_{i,j,h}(p,q) \geq 0\} \quad \text{and} \\ L^-(e^\star_{i,j,h}) &= \{(p,q)^T \mid e^\star_{i,j,h}(p,q) < 0\} \end{aligned}$$

denote these two subspaces of $\mathbb{R}^2$ for any $\star \in \{\mathtt{h}, \mathtt{v}\}$, any $(i,j)^T \in z(2n)$ and $h \in \mathbb{Z}$.

The partition of the parameter space is obtained by taking into account all combinations of pixels $(i,j)^T$, all unit squares with some lower left point $(i',j')^T$ and the k-feasibility of $\mathcal{I}$. Particularly, for given natural numbers k, m and n, let

$$\begin{aligned} \mathcal{L}(k,m,n) = \; & \left\{ e^{\mathtt{h}}_{i,j,i'} \mid (i,j)^T \in z(2n), i' \in \{-k-m-1, \dots, k+m+1\} \right\} \cup \\ & \left\{ e^{\mathtt{v}}_{i,j,j'} \mid (i,j)^T \in z(2n), j' \in \{-k-m-1, \dots, k+m+1\} \right\} \end{aligned}$$

be the set of primal polynomials and based on this notion, it is possible to describe the primal partition as follows: For given natural numbers k, m and n the primal partition $\Phi(k,m,n)$ of $\mathbb{R}^2$ is defined by the set $\mathcal{L}(k,m,n) = \{e_1, \dots, e_W\}$ of primal polynomials as

$$\Phi(k,m,n) = \left\{ \phi = \bigcap_{w=1}^{W} L^{s_w}(e_w) \;\middle|\; \exists s_1, \dots, \exists s_W \in \{+,-\}, \phi \neq \emptyset \right\}.$$

Theorem 1. *Let $k, m, n \in \mathbb{N}$. Then, any two parameters $(p_1, q_1)^T$ and $(p_2, q_2)^T$ represent transformations f_{p_1,q_1}^{-1} and f_{p_2,q_2}^{-1} such that for all pixels $(i, j)^T \in z(2n)$ and all $(i', j')^T \in \{-k-m-1, \ldots, k+m+1\}^2$ it is true: If there exists $\phi \in \Phi(k, m, n)$ such that $(p_1, q_1)^T \in \phi$ and $(p_2, q_2)^T \in \phi$ then*

$$\lfloor f_{p_1,q_1}^{-1}(i, j) \rfloor = (i', j')^T \iff \lfloor f_{p_2,q_2}^{-1}(i, j) \rfloor = (i', j')^T.$$

In the theorem, the coordinates of the lower left point $(i', j')^T$ of a square range only in $\{-k-m-1, \ldots, k+m+1\}^2$ instead of the complete set $\mathbb{Z}^2$. This is sufficient because the paper considers k-feasible interpolation methods $\mathcal{I}$ which operate on a limited support. Thus, if a transformed point $(x, y)^T = f_{p,q}^{-1}(i, j)$ falls left or below $-k-m-1$ or right or above $k+m+1$ then it has color value $P_{\mathcal{I}}\langle x, y\rangle = \perp$.

Finally, it is time to define the mappings $X_\phi, Y_\phi : z(2n) \to \mathbb{Z}$ for all subspaces $\phi \in \Phi(k, m, n)$. According to Theorem 1 it is true that for any given subspace ϕ there are integer pairs $(i', j')^T$ for all pixels $(i, j)^T \in z(2n)$ such that all parameters $(p, q)^T \in \phi$ represent $f_{p,q}^{-1}$ which transform (i, j) to the unit square with lower left at (i', j'), i. e., $\lfloor f_{p,q}^{-1}(i, j) \rfloor = (i', j')^T$. Consequently, it is straight forward to let $X_\phi(i, j) = i'$ and $Y_\phi(i, j) = j'$. Particularly, X_ϕ and Y_ϕ can be computed by selecting an arbitrary parameter $(p, q)^T$ from ϕ to set

$$X_\phi(i, j) = \lfloor x \rfloor \quad \text{and} \quad Y_\phi(i, j) = \lfloor y \rfloor$$

with $(x, y)^T = f_{p,q}^{-1}(i, j)$ for all $(i, j)^T \in z(2n)$.

Non-linear Refinement

The non-linear refinement of the parameter space partition builds on the properties of the primal partition and depends on the interpolation method $\mathcal{I}$ and the color values of the given pattern P. For every subspace $\phi_w \in \Phi(k, m, n)$, it defines a subpartition into subsubspaces $\psi_1, \ldots, \psi_{M_w}$ such that two combinations f_1, f_2 of scaling and rotation fulfill $f_1(P_{\mathcal{I}}) = f_2(P_{\mathcal{I}})$ if the inverse transformations f_{p_1,q_1}^{-1} and f_{p_2,q_2}^{-1} are represented by parameters $(p_1, q_1)^T, (p_2, q_2)^T \in \psi_{w'}$ with $w' \in \{1, \ldots, M_w\}$.

According to the previous section, it is known that both f_{p_1,q_1}^{-1} and f_{p_2,q_2}^{-1} transform every pixel (i, j) to the same unit square, the one identified by lower left point $(X_{\phi_w}(i, j), Y_{\phi_w}(i, j))^T$. But this does not yet mean that the color values $f_1(P_{\mathcal{I}})\langle i, j\rangle$ and $f_2(P_{\mathcal{I}})\langle i, j\rangle$ are equal.

But, looking at the points $(x_1, y_1)^T = f_{p_1,q_1}^{-1}(i, j)$ and $(x_2, y_2)^T = f_{p_2,q_2}^{-1}(i, j)$ it is true that a k-feasible method $\mathcal{I}$ interpolates $(x_1, y_1)^T$ and $(x_2, y_2)^T$ quite similar. In fact, the mappings X_{ϕ_w} and Y_{ϕ_w} can be used to define polynomial functions

$$e_{i,j}^{\phi_w, P_{\mathcal{I}}}(p, q) = \sum_{dj=-k}^{k} \sum_{di=-k}^{k} \mathcal{I}_{di,dj}(ip + jq - X_{\phi_w}(i, j), jp - iq - Y_{\phi_w}(i, j)) \cdot P\langle X_{\phi_w}(i, j) + di, Y_{\phi_w}(i, j) + dj\rangle$$

which express a relation between the argument parameters $(p,q)^T \in \phi_w$ and the color values $P_{\mathcal{I}}\langle x,y\rangle$ interpolated at the coordinates $(x,y)^T = f_{p,q}^{-1}(i,j)$. Particularly, from $f(P_{\mathcal{I}})\langle i,j\rangle = P_{\mathcal{I}}\langle x,y\rangle$ follows that $e_{i,j}^{\phi_w,P_{\mathcal{I}}}(p,q)$ describes the transformed pattern's color value at pixel $(i,j)^T$ in dependence of the argument parameter $(p,q)^T$, i. e., it is true that $f(P_{\mathcal{I}})\langle i,j\rangle = \left[e_{i,j}^{\phi_w,P_{\mathcal{I}}}(p,q)\right]$ for the parameter argument $(p,q)^T$ that represents the inverse $f_{p,q}^{-1}$ of f. The point is, that fixing a pattern P of size m, a natural number n, a k-feasible interpolation method $\mathcal{I}$, and a subspace $\phi_w \in \Phi(k,m,n)$ gives unique polynomials $e_{i,j}^{\phi_w,P_{\mathcal{I}}}$ for all pixels $(i,j)^T \in z(2n)$ to describe the corresponding color value in the transformed pattern. This definiteness allows the subdivision of ϕ_w into the subsubspaces $\psi_1,\ldots,\psi_{M_w}$ which is based on the following lemma:

Lemma 1. *Let k,m and n be natural numbers, let ϕ be a subspace in $\Phi(k,m,n)$, let P be a pattern of size m and let $\mathcal{I}$ be a k-feasible interpolation method. Moreover, let f be a combination of scaling and rotation with inverse $f_{p,q}^{-1}$ represented by a parameter $(p,q)^T \in \phi$, let $(i,j)^T$ be a pixel in the support $z(2n)$ and let $c \in \{0,\ldots,c(P)\}$ be a color value. Then $f(P_{\mathcal{I}})\langle i,j\rangle = P_{\mathcal{I}}\langle f_{p,q}^{-1}(i,j)\rangle \geq c$ if and only if $e_{i,j}^{\phi,P_{\mathcal{I}}}(p,q) \geq c - 0.5$.*

In compliance with the previous subsection, the subdivision of the parameter subspace ϕ is described by polynomials

$$e_{i,j,c}^{\phi,P_{\mathcal{I}}}(p,q) = e_{i,j}^{\phi,P_{\mathcal{I}}}(p,q) - c + 0.5$$

in the two variables p and q which are collected in the set

$$\mathcal{C}(k,m,n,\phi,P_{\mathcal{I}}) = \left\{ e_{i,j,c}^{\phi,P_{\mathcal{I}}} \;\middle|\; (i,j)^T \in z(2n), c \in \{0,\ldots,c(P)\} \right\}$$

of refinement polynomials for any given natural numbers k,m and n, subspace $\phi \in \Phi(k,m,n)$, pattern P of size m and k-feasible interpolation method $\mathcal{I}$. In fact, the subdivision given by a single polynomial $e_{i,j,c}^{\phi,P_{\mathcal{I}}}$ is again denoted as

$$L^+(e_{i,j,c}^{\phi,P_{\mathcal{I}}}) = \{(p,q)^T \in \phi \mid e_{i,j,c}^{\phi,P_{\mathcal{I}}}(p,q) \geq 0\} \quad \text{and}$$
$$L^-(e_{i,j,c}^{\phi,P_{\mathcal{I}}}) = \{(p,q)^T \in \phi \mid e_{i,j,c}^{\phi,P_{\mathcal{I}}}(p,q) < 0\}$$

and based on this, the complete partition of ϕ is defined by all refinement polynomials $\mathcal{C}(k,m,n,\phi,P_{\mathcal{I}}) = \{e_1,\ldots,e_W\}$ according to

$$\Psi(k,m,n,\phi,P_{\mathcal{I}}) = \left\{ \psi = \phi \cap \bigcap_{w=1}^{W} L^{s_w}(e_w) \;\middle|\; \exists s_1,\ldots,\exists s_W \in \{+,-\}, \psi \neq \emptyset \right\}.$$

However, this time the polynomials $e_{i,j,c}^{\phi,P_{\mathcal{I}}}$ are not necessarily linear, and thus, the subspaces in $\Psi(k,m,n,\phi,P_{\mathcal{I}})$ are separated by curves.

Corollary 1. *Let k,m,n be natural numbers, let P be a pattern of size m, let $\mathcal{I}$ be a k-feasible interpolation method and let $\phi \in \Phi(k,m,n)$. Then, two combinations f_1,f_2 of scaling and rotations fulfill $f_1(P_{\mathcal{I}}) = f_2(P_{\mathcal{I}})$ if their inverse transformations f_{p_1,q_1}^{-1} and f_{p_2,q_2}^{-1} are represented by parameters $(p_1,q_1)^T, (p_2,q_2)^T \in \phi$ that belong to the same subsubspace $\psi \in \Psi(k,m,n,\phi,P_{\mathcal{I}})$.*

Complete Partition

Theorem 1 and Corollary 1 together imply a complete characterization of the parameter space $\mathbb{R}^2$ for combinations of scaling and rotation: For given natural numbers k, m and n, pattern P of size m and k-feasible interpolation method $\mathcal{I}$ let

$$\Upsilon(k, m, n, P_{\mathcal{I}}) = \bigcup_{\phi \in \Phi(k,m,n)} \Psi(k, m, n, \phi, P_{\mathcal{I}})$$

denote the complete partition of $\mathbb{R}^2$.

Theorem 2. *Let k, m, n be natural numbers, let P be a pattern of size m and let $\mathcal{I}$ be a k-feasible interpolation method. Then, two combinations f_1, f_2 of scaling and rotations fulfill $f_1(P_{\mathcal{I}}) = f_2(P_{\mathcal{I}})$ if their inverse transformations $f^{-1}_{p_1,q_1}$ and $f^{-1}_{p_2,q_2}$ are represented by parameters $(p_1, q_1)^T \in \mathbb{R}^2$ and $(p_2, q_2)^T \in \mathbb{R}^2$ that belong to the same subsubspace $\psi \in \Upsilon(k, m, n, P_{\mathcal{I}})$.*

The theorem leads to the basic idea of computing $\mathcal{D}(P, \mathcal{I})$, namely, to retrieve a set $\{(p_1, q_1)^T, \ldots, (p_N, q_N)^T\}$ of points in parameter space $\mathbb{R}^2$, at least one from every subspace ψ in $\Upsilon(k, m, n, P_{\mathcal{I}})$, and then to find the represented inverse transformations $\{f^{-1}_{p_1,q_1}, \ldots, f^{-1}_{p_N,q_N}\}$ which enable the computation of the dictionary $\mathcal{D}(P, \mathcal{I}) = \{f_1(P_{\mathcal{I}}), \ldots, f_N(P_{\mathcal{I}})\}$.

4 A Polynomial Time Algorithm

The new algorithm works according to the idea introduced in the previous section to preprocess the dictionary $\mathcal{D}(P, \mathcal{I})$ before performing actual pattern matching. The algorithm starts by computing a data structure to enumerate $\Phi(k, m, n)$ and then samples for every subspace $\phi_w \in \Phi(k, m, n)$ a set of parameters $\{(p_1, q_1)^T, \ldots, (p_{M_w}, q_{M_w})^T\}$ to touch all subsubspaces in $\Psi(k, m, n, \phi, P_{\mathcal{I}})$. This solves the challenge to find a sampling set $\{(p_1, q_1)^T, \ldots, (p_N, q_N)^T\}$ of parameters in $\mathbb{R}^2$ suitable to cover every subsubspace $\psi \in \Upsilon(k, m, n, P_{\mathcal{I}})$ by at least one of them.

Enumerating $\boldsymbol{\Phi(k, m, n)}$

The proposed enumeration technique for the first stage is similar to the pattern matching algorithm introduced by Hundt et al. [12]. The new algorithm computes an incidence graph to be used as a data structure for $\Phi(k, m, n)$. Incidence graphs are a common notion from computational geometry that are analyzed for instance by Edelsbrunner [9]. They can be applied to describe the geometric structure and relationship of the subspaces in $\Phi(k, m, n)$. Basically, the incidence graph contains a node for every subspace in $\Phi(k, m, n)$ and two nodes are connected by an edge if the corresponding subspaces are neighbors in $\mathbb{R}^2$.

Theorem 3. *Let k, m and n be natural numbers. The set $\Phi(k, m, n)$ contains $O((k + m)^2 n^4)$ subspaces. The enumeration of all subspaces $\phi \in \Phi(k, m, n)$ and the computation of all mappings X_ϕ and Y_ϕ takes $O((k + m)^2 n^6)$ time.*

The next subsection describes how to hit every subsubspace in $\Psi(k, m, n, \phi, P_{\mathcal{I}})$ by a set of parameter points $\{(p_1, q_1)^T, (p_2, q_2)^T, \ldots\}$ from subspace ϕ. The procedure depends on the shape of ϕ. In fact, $\Phi(k, m, n)$ contains subspaces that are points, line segments or convex polygons in $\mathbb{R}^2$. The sampling of ϕ depends crucially on its shape and is most complex if ϕ is a polygon.

Sampling All Subspaces in $\Phi(k, m, n)$

For every enumerated subspace $\phi_w \in \Phi(k, m, n)$, the algorithm samples a set $\{(p_1, q_1)^T, \ldots, (p_{M_w}, q_{M_w})^T\}$ of parameters from ϕ_w such that every subsubspace $\psi \in \Psi(k, m, n, \phi_w, P_{\mathcal{I}})$ is touched by at least one of them.

The sampling approach depends crucially on the shape of ϕ_w. Firstly, if ϕ_w consists of a single parameter $(p, q)^T$, then sampling $(p, q)^T$ alone is sufficient. Secondly, if ϕ_w is a line segment then it is clearly the case that $\Psi(k, m, n, \phi_w, P_{\mathcal{I}})$ consists of smaller line subsegments which are bounded by the intersection points between the line segment ϕ_w and the curves that are given by the polynomials in $\mathcal{C}(k, m, n, \phi_w, P_{\mathcal{I}})$. Hence, the sampling set is built of two point types: in the first place, all intersection points between ϕ_w and the curves and secondly, one point between every pair of consecutive intersections. A numerical difficulty with this approach is that the intersection points can be irrational.

Finally, the most challenging case occurs if ϕ_w is a polygon. The basic idea of getting the sampling points in this situation works in three steps: Firstly, the algorithm determines the set of all intersection points between two curves described by polynomials in $\mathcal{C}(k, m, n, \phi_w, P_{\mathcal{I}})$. Secondly, the algorithm determines a circle of sufficiently small radius around every found intersection point. In this way, every subsubspace of ϕ_w should be intersected by at least one of these circles. The sampling set is built of two types of points on these circles where the first type consists of all intersection points between the circles and the curves and the second type is obtained from adding one point on every resulting arc.

The first challenge with this approach is to determine a suitable radius for the circles around the curve intersections. The difficulty in this task is that big circles may contain entire subsubspaces which are subsequently missed in the sampling process. Because nearly all subsubspaces have at least two curve intersections in their boundary, a convenient way to make sure that circles cannot contain entire subsubspaces, is to define a radius that prevents circles to contain more than one curve intersection. For that purpose, the following lemma estimates the minimum distance between two distinct curve intersections:

Lemma 2. *For given natural numbers k, m and n, subspace $\phi \in \Phi(k, m, n)$, pattern P of size m and k-feasible interpolation method $\mathcal{I}$, let $(p_1, q_1)^T$ be a point that fulfills $e_1(p_1, q_1) = 0$ and $e_2(p_1, q_1) = 0$ for two distinct refinement polynomials $e_1, e_2 \in \mathcal{C}(k, m, n, \phi, P_{\mathcal{I}})$ and let $(p_2, q_2)^T$ be a different point that fulfills $e_3(p_2, q_2) = 0$ and $e_4(p_2, q_2) = 0$ for two distinct refinement polynomials $e_3, e_4 \in \mathcal{C}(k, m, n, \phi, P_{\mathcal{I}})$ which may be the same as e_1 and e_2. Then, it holds that*

$$|p_1 - p_2| \geq \frac{2}{(216 \cdot c(P)k(k+m+n))^{16k^4}} \quad \textit{or}$$
$$|q_1 - q_2| \geq \frac{2}{(216 \cdot c(P)k(k+m+n))^{16k^4}}.$$

The argumentation for the correctness of the lemma applies the product between the q-resultant of e_1 and e_2 and the q-resultant of e_3 and e_4, which is a univariate polynomial that (1) eliminates the variable q, (2) has a root p for every $q \in \mathbb{R}$ with $e_1(p,q) = e_2(p,q) = 0$ or $e_3(p,q) = e_4(p,q) = 0$, (3) has degree at most $4k^2$, and (4) contains only integer coefficients with absolute values bounded polynomially in $c(P), m, n$ and exponentially in k^2. Rump [21] gives a lower bound for the root separation of such a polynomial which provides the first estimation in the lemma. The second estimation is found analogously using the p-resultants.

Based on the lemma, the radius of all circles has to be chosen less than $\sigma = 2(216 \cdot c(P)k(k+m+n))^{-16k^4}$. Although this number is very small, it can be represented exactly with a logarithmic amount of bits if k is a constant.

Assuming that the approach drafted above enables the sampling of all subsubspaces that have curve intersections in their boundary leaves the problem to hit subsubspaces ψ with an intersection-free boundary. In fact, those subsubspaces are caused by curves that are represented by refinement polynomials in $\mathcal{C}(k,m,n,\phi_w,P_{\mathcal{I}})$ which do not intersect any other curves. Now it may happen, that there is no nearby curve intersection where the corresponding surrounding circle intersects ψ and consequently, ψ would remain untouched by the sampling. To overcome this problem, the algorithm computes one additional circle for every polynomial in $\mathcal{C}(k,m,n,\phi_w,P_{\mathcal{I}})$ that surrounds a point on the represented curve.

Another difficulty is the exact computation of the curve intersections because they may have irrational coordinates. According to Lemma 2 the algorithm can compute every intersection point $(p,q)^T$ with an error of at most 0.25σ to obtain a point $(p',q')^T$ with a euclidean distance of less than 0.4σ. A circle around $(p',q')^T$ that has radius 0.5σ contains $(p,q)^T$ and surely no other intersection point between curves. The following lemma shows that all curve intersection points can be efficiently computed with sufficient precision:

Lemma 3. *For given natural numbers k, m and n, subspace $\phi \in \Phi(k,m,n)$, pattern P of size m and k-feasible interpolation method $\mathcal{I}$, the computation of a set $\mathcal{P}(k,m,n,\phi,P_{\mathcal{I}})$ of parameters such that*

1. *for every pair of distinct refinement polynomials e_1 and e_2 in $\mathcal{C}(k,m,n,\phi,P_{\mathcal{I}})$ and every parameter $(p,q)^T \in \phi$ that simultaneously fulfills $e_1(p,q) = 0$ and $e_2(p,q) = 0$, there exists a parameter $(p',q')^T \in \mathcal{P}(k,m,n,\phi,P_{\mathcal{I}})$ with* $\max\{|p-p'|, |q-q'|\} \leq 0.5(216 \cdot c(P)k(k+m+n))^{-16k^4}$*,*
2. *for every polynomial $e \in \mathcal{C}(k,m,n,\phi,P_{\mathcal{I}})$ there exists a parameter $(p',q')^T$ in $\mathcal{P}(k,m,n,\phi,P_{\mathcal{I}})$ with* $\max\{|p-p'|, |q-q'|\} \leq 0.5(216 \cdot c(P)k(k+m+n))^{-16k^4}$ *for some $(p,q)^T \in \phi$ with $e(p,q) = 0$ (if such $(p,q)^T$ exists),*

3. every $(p', q')^T \in \mathcal{P}(k, m, n, \phi, P_{\mathcal{I}})$ *fulfills* $|p'|, |q'| \le (40 \cdot c(P)k(k+m+n))^{2k^2}$ *and*
4. $|\mathcal{P}(k, m, n, \phi, P_{\mathcal{I}})| \in O(k^4 n^4 c(P)^2)$

takes $O(k^{13}mn^6c(P)^2\log^{13}\mathcal{N})$ *time where* $\mathcal{N} = \max\{k, m, n, c(P)\}$.

The basic idea behind the computation of $\mathcal{P}(k, m, n, \phi_w, P_{\mathcal{I}})$ is based on the algorithm of Diochnos et al. [8] that computes a set of disjoint rectangles for two given refinement polynomials e_1 and e_2, such that every rectangle contains exactly one common root of e_1 and e_2. To establish the upper bounds on the number of roots and the root spread the lemma applies the q- and p-resultants of e_1 and e_2 which are univariate polynomials that (1) describe the common roots of e_1 and e_2, (2) have degree $2k^2$ and (3) contain only integer coefficients $a_0, \ldots, a_{2k^2}$ with bounded absolute value. This implies directly that e_1 and e_2 have at most $4k^4$ common roots and all $O(n^2c(P))$ refinement polynomials together provide at most $O(k^4n^4c(P)^2)$ curve intersections. Moreover, Cauchy's upper bound $\sum_{i=0}^{2k^2}\left|\frac{a_i}{a_0}\right|$ on the absolute value of polynomial roots limits the root spread.

Using the parameters in $\mathcal{P}(k, m, n, \phi_w, P_{\mathcal{I}})$, the algorithm is able to determine the sampling set by following the idea of encircled curve intersections described above. Thus, it finds the set $\{f^{-1}_{p_1,q_1}, \ldots, f^{-1}_{p_{M_w},q_{M_w}}\}$ of inverse transformations to obtain $f_1(P_{\mathcal{I}}), \ldots, f_{M_w}(P_{\mathcal{I}})$. According to the following lemma all pattern transformations given by parameters in ϕ_w can be computed in polynomial time:

Lemma 4. *Let* k, m *and* n *be natural numbers, let* $\phi \in \Phi(k, m, n)$*, let* P *be a pattern of size* m *and let* $\mathcal{I}$ *be a* k*-feasible interpolation method. The computation of the set*

$$\mathcal{D}(P, \mathcal{I}, \phi) = \{f(P_{\mathcal{I}}) \mid \textit{the inverse } f^{-1}_{p,q} \textit{ of } f \textit{ is represented some } (p, q)^T \in \phi\}$$

takes $O(k^{13}mn^6c(P)^3\log^{13}\mathcal{N})$ *time.*

The lemma can be shown simply by following the presented ideas which provides a subroutine to compute $\mathcal{D}(P, \mathcal{I}, \phi)$. Finally, the new algorithm can compute the dictionary $\mathcal{D}(P, \mathcal{I})$ by merging the sets $\mathcal{D}(P, \mathcal{I}, \phi)$ for all $\phi \in \Phi(k, m, n)$. This is realized by enumerating all subspaces $\phi \in \Phi(k, m, n)$ and then calling the subroutine to get $\mathcal{D}(P, \mathcal{I}, \phi)$. The enumeration can be implemented by firstly computing the incidence graph of $\Phi(k, m, n)$ using Edelsbrunner's algorithm [9] and secondly traversing the graph with, e.g., depth first search. After preprocessing the entire dictionary $\mathcal{D}(P, \mathcal{I})$, performing 2D-pattern matching is solved by moving every pattern in $\mathcal{D}(P, \mathcal{I})$ over the text T. The following theorem specifies the complexity of the whole procedure:

Theorem 4. *Let* k, m *and* n *be natural numbers, let* P *be a pattern of size* m*, let* T *be a text of size* n *and let* $\mathcal{I}$ *be a* k*-feasible interpolation method. The computation of the set* $\mathcal{D}(P, \mathcal{I})$ *takes* $O(k^{15}m^3n^{10}c(P)^3\log^{13}\mathcal{N})$ *time.*

If the colors of P *and* T *originate from a fixed set* $\{0, \ldots, c\}$ *of colors, if* $\mathcal{I}$ *is fixed and if* Δ *is a fixed distortion measure which has a time complexity of*

$t(m,n) \in O(n^2)$, then 2D-pattern matching with scaling and rotation for given P and T can be solved in $O(m^3 n^{15})$ time.

These results show for the first time that globally optimal solutions for 2D-pattern matching with higher-order interpolation methods can be found in polynomial time.

5 Conclusions and Future Work

This paper presents the first polynomial time algorithm for 2D-pattern matching with scaling and rotation and higher-order interpolation methods. The announced time bound represents roughly a doubling of exponents with respect to the time complexity of the problem for nearest-neighbor interpolation. This remains true, if the quality of interpolation or the number of colors are increased. The following table compares the presented results with previous work:

	Nearest-Neighbor	Bilinear	Feasible
Scaling	$O(mn^2)$ [4]	$O(mn^4 \log n)$ [22]	open
Rotation	$O(m^3 n^2)$ [10]	$O(m^3 n^2 \log m)$ [22]	open
Scaling and Rotation	$O(m^2 n^6)$ [12]	$O(m^3 n^{15})$ (this paper)	$O(m^3 n^{15})$ (this paper)

Notice that [4] deals with exact matching, while the rest of the table deals with best matching. However, the computation of the dictionary dominates the time complexity in either case which makes the comparison reasonably fair.

The foundation of the proposed algorithm is a novel characterization of the parameter space for scaling and rotation. Although the time complexity does not suggest practical applicability of the algorithm, it nevertheless allows to evaluate current heuristic matching methods.

It remains future work to implement the new algorithm in the pattern matching framework developed by Nevries [20]. This framework is already capable of pattern matching with nearest-neighbor interpolation and consequently, it could be extended to compare the complexity and the matching quality for different interpolation methods.

Acknowledgment. The authors like to thank Maciej Liśkiewicz, Ragnar Nevries and the anonymous reviewers for their helpful suggestions.

References

1. Bovik, A. (ed.): Handbook of Image and Video Processing. Academic Press, San Diego (2000)
2. Amir, A., Butman, A., Crochemore, M., Landau, G., Schaps, M.: Two-dimensional pattern matching with rotations. Theor. Comput. Sci. 314(1-2), 173–187 (2004)

3. Amir, A., Butman, A., Lewenstein, M., Porat, E.: Real two dimensional scaled matching. Algorithmica 53(3), 314–336 (2009)
4. Amir, A., Chencinski, E.: Faster Two Dimensional Scaled Matching. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 200–210. Springer, Heidelberg (2006)
5. Amir, A., Kapah, O., Tsur, D.: Faster Two Dimensional Pattern Matching with Rotations. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 409–419. Springer, Heidelberg (2004)
6. Brown, L.G.: A survey of image registration techniques. ACM Computing Surveys 24(4), 325–376 (1992)
7. Cox, I.J., Bloom, J.A., Miller, M.L.: Digital Watermarking, Principles and Practice. Morgan Kaufmann, San Francisco (2001)
8. Diochnos, D.I., Emiris, I.Z., Tsigaridas, U.E.P.: On the asymptotic and practical complexity of solving bivariate systems over the reals. J. Sym. Comput. 44(7), 818–835 (2009)
9. Edelsbrunner, H., O'Rourke, J., Seidel, R.: Constructing arrangements of lines and hyperplanes with applications. SIAM J. Comput. 15, 341–363 (1986)
10. Fredriksson, K., Navarro, G., Ukkonen, E.: Sequential and indexed two-dimensional combinatorial template matching allowing rotations. Theor. Comput. Sci. 347(1-2), 239–275 (2005)
11. Fredriksson, K., Ukkonen, E.: A Rotation Invariant Filter for Two-Dimensional String Matching. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 118–125. Springer, Heidelberg (1998)
12. Hundt, C., Liśkiewicz, M., Nevries, R.: A combinatorial geometric approach to two-dimensional robustly pattern matching with scaling and rotation. Theor. Comput. Sci. 51(410), 5317–5333 (2009)
13. Indyk, P.: Algorithmic aspects of geometric embeddings. In: Proc. FOCS 2001, pp. 10–33 (2001)
14. Indyk, P., Motwani, R., Venkatasubramanian, S.: Geometric matching under noise: Combinatorial bounds and algorithms. In: Proc. SODA 1999, pp. 354–360 (1999)
15. Kasturi, R., Jain, R.C.: Computer Vision: Principles. IEEE Computer Society Press, Los Alamitos (1991)
16. Kropatsch, W.G., Bischof, H. (eds.): Digital Image Analysis - Selected Techniques and Applications. Springer, Berlin (2001)
17. Landau, G.M., Vishkin, U.: Pattern matching in a digitized image. Algorithmica 12(3/4), 375–408 (1994)
18. Maintz, J.B.A., Viergever, M.A.: A survey of medical image registration. Medical Image Analysis 2(1), 1–36 (1998)
19. Modersitzki, J.: Numerical Methods for Image Registration. Oxford University Press (2004)
20. Nevries, R.: Entwicklung und Analyse eines beschleunigten Image Matching-Algorithmus für natürliche Bilder, Diplomarbeit, Universität Rostock (2008)
21. Rump, S.M.: Polynomial minimum root separation. Math. Comp. 33(145), 327–336 (1979)
22. Wendland, F.: Exact polynomial algorithms for image matching with bilinear interpolation, Master Thesis, Universität Rostock (2011)

Hardness of Longest Common Subsequence for Sequences with Bounded Run-Lengths

Guillaume Blin[1], Laurent Bulteau[2], Minghui Jiang[3], Pedro J. Tejada[3], and Stéphane Vialette[1]

[1] Université Paris-Est, LIGM, UMR 8049, France
[2] Université de Nantes, LINA, UMR 6241, France
[3] Utah State University, Department of Computer Science, USA

Abstract. The longest common subsequence (LCS) problem is a classic and well-studied problem in computer science with extensive applications in diverse areas ranging from spelling error corrections to molecular biology. This paper focuses on LCS for fixed alphabet size and fixed run-lengths (*i.e.*, maximum number of consecutive occurrences of the same symbol). We show that LCS is **NP**-complete even when restricted to (i) alphabets of size 3 and run-length at most 1, and (ii) alphabets of size 2 and run-length at most 2 (both results are tight). For the latter case, we show that the problem is approximable within ratio 3/5.

1 Introduction

A *subsequence* of a string is obtained by deleting zero or more symbols of that string. Finding the longest string which is equal to a subsequence of two or more strings is a classic problem known as the *longest common subsequence* (LCS) problem. LCS yields extensive applications in diverse areas ranging from spelling error corrections to molecular biology, and has been extensively studied during the last 30 years. In particular the case where the number of sequences is 2 has been studied in detail, and LCS is well-known to be polynomial-time solvable by dynamic programming in this case (see [10] and references therein). Furthermore, there exist methods with lower complexity which often depend on the length of the lcs, the size of the alphabet, or both (the best general reference is [4]). More generally, the problem is solvable in polynomial-time by dynamic programming when the number of input sequences is constant. For the general case of an arbitrary number of input sequences, the problem is **NP**-complete [21]. The problem has also been studied in the framework of parameterized complexity [6,7,24]. LCS for unbounded alphabet size is $\mathbf{W}[t]$-hard for $t \geq 1$ when parameterized by the number of input sequences, and $\mathbf{W}[\mathbf{2}]$-hard when parameterized by the length of the sought common subsequence. For a fixed alphabet size, LCS is $\mathbf{W}[\mathbf{1}]$-hard when parameterized by the number of input sequences but is fixed-parameter tractable when parameterized by the length of the sought common subsequence.

Run-length encoding is a well-known method for compressing strings, and a whole line of research is devoted to studying LCS for run-length encoded strings.

A string s is run-length encoded if it is described as an ordered sequence of pairs (σ, i), often denoted σ^i, each consisting of an alphabet symbol σ and an integer i. Each pair corresponds to a *run* in s consisting of i consecutive occurrences of σ. For example, the string $s = bbbbbaaccc$ can be encoded as $b^5a^2c^3$. Two typical applications of run-length encoding can be found in image compression since many images contain large runs of identically-valued pixels, and in mini-satellites in biological sequences since these sequences contain a large number of tandem repeats. In this context, two lines of research are being explored. A first line of research has tried to improve the running time of the algorithms by using sparse dynamic programming to compute small subsets of the elements in the standard lcs table [9,11,20,2]. A second line of research has tried to find algorithms with running times depending only on the number of runs in the input strings, without computing individual elements of the standard lcs table [23,3]. It is worth mentioning that work has also been done on computing the similarity of two run-length encoded string in the affine gap penalty model [17], the string edit distance problem, the pairwise global alignment model, the pairwise local alignment model in the linear-gap model with arbitrary scoring matrices [14], and on computing the constrained lcs of run-length encoded strings [1]. Refer to [19,13,18,22,16] and references therein for more problems on run-length encoded strings.

This paper is devoted to studying LCS for the general case of an arbitrary number of input sequences for a fixed size alphabet with a special focus on fixed run-lengths. To shorten notations, for positive integers p and q, we let $\text{LCS}(p, q)$ stand for LCS where input sequences are defined over an alphabet of size at most p, and each input sequence has maximum run-length at most q. Abusing notation, we shall write $q = \infty$ to denote unbounded run-lengths.

The paper is organized as follows. In Section 2, we present a new simple proof for the hardness of LCS for binary alphabets and show that $\text{LCS}(3, 1)$ is **NP**-complete. Section 3 is devoted to proving hardness of $\text{LCS}(2, 2)$, and we consider in Section 4 approximation issues of this problem.

2 Preliminary Results and NP-Completeness of LCS$(3, 1)$

For an arbitrary number of sequences, Maier [21] showed that LCS is **NP**-complete even for an alphabet of size 2 (in our terms, $\text{LCS}(2, \infty)$ is **NP**-complete). This result serves as a classical example to demonstrate the limit of dynamic programming approaches to solving LCS for an arbitrary number of input sequences. However, Maier's proof is notoriously complicated and we propose here as a warm-up an alternate and simpler proof (we shall next adapt this proof to prove the **NP**-completeness of $\text{LCS}(3, 1)$).

Proposition 1. $\text{LCS}(2, \infty)$ *is* **NP**-*complete.*

Proof. By reduction from maximum independent set. Let $G = (V, E)$ be a graph with n vertices and m edges. Write $V = \{1, 2, \ldots, n\}$. We construct $m + 1$ sequences $S_0, S_1, \ldots, S_m$ over alphabet $\Sigma = \{0, 1\}$, each of length at most $(n+1)^2-2$ as follows. The sequence S_0 is defined to be $(0^n1)^n$. For each edge $e_j=\{u,v\}\in E$, $u<v$ and $1\leq j\leq m$, the sequence S_j is defined to be $(0^n1)^{u-1}\ 0^n\ (0^n1)^{v-u}\ 0^n\ (0^n1)^{n-v}$. For example, in a graph with 7 vertices the edge between vertices 2 and 4 is represented by the sequence $0^71\ 0^7\ (0^71)^2\ 0^7\ (0^71)^3$.

We claim that the graph G has an independent set of size k if and only if the sequences $S_0,S_1,\ldots,S_m$ have a common subsequence of length n^2+k.

Suppose first that G has an independent set I of size k. Consider the sequence $T=T_1\ T_2\ \ldots\ T_n$, where $T_i=0^n$ if $i\notin I$ and $T_i=0^n1$ if $i\in I$. Clearly, $|T|=n^2+k$, and T is a subsequence of S_0. Furthermore, since each edge has at least one vertex not in I, it can be seen (details omitted) that T is a common subsequence of $S_1,S_2,\ldots,S_m$.

We now prove the reverse implication. Suppose that $S_0,S_1,\ldots,S_m$ have a common subsequence of length n^2+k. Then any lcs of these $m+1$ input sequences has length at least n^2+k. Let $T_{\mathbf{opt}}$ be an lcs of $S_0,S_1,\ldots,S_m$, and consider a multiple alignment of $S_0,S_1,\ldots,S_m$ inducing $T_{\mathbf{opt}}$. We modify this multiple alignment by shifting 0s right to obtain another multiple alignment inducing $T_{\mathbf{opt}}$ where matched 0s cannot be shifted right anymore (by shifting a 0 right we mean modifying the alignment so that a 0 in the induced $T_{\mathbf{opt}}$ is aligned with a 0 of S_j to the right of the matched 0 of S_j in the original alignment, for at least one sequence S_j). For each sequence S_j, $0\leq j\leq m$, find the rightmost 0 included in the multiple alignment and shift it right until it is the rightmost 0 of S_j or it is just before a 1 included in the multiple alignment. Observe that each one of those 0s is the rightmost 0 of a 0^n1 or a 0^n substring, and hence all the other 0s in those 0^n1 or 0^n substrings of S_j, $0\leq j\leq m$, can be aligned by shifting other 0s included in the alignment right (otherwise, $T_{\mathbf{opt}}$ is not an lcs). Moreover, observe that at least one 0 from each 0^n1 substring of S_0 has to be included in the alignment since otherwise we obtain $|T_{\mathbf{opt}}|\leq n(n+1)-n=n^2<n^2+k$, for any $k\geq 1$, contradicting our assumption that the length of an lcs is at least n^2+k. Thus by repeating this shifting process for the substrings of S_j, $0\leq j\leq m$, to the left of the last 0^n1 or 0^n substrings considered, we can make sure that all the 0s in S_0 are included in the alignment, and moreover all the symbols that are included in the alignment from each 0^n1 substring of S_0 are aligned with symbols from the same 0^n1 or 0^n substring of S_j, $1\leq j\leq m$.

Therefore a longest common subsequence is obtained by including all the 0s in S_0 and maximizing the number of 1s that can be aligned. After the shifting process described above it can be seen that for each edge $e_j=\{u,v\}$ of G, at least one of the two 0^n1 substrings at positions u and v of S_0 must be aligned with a substring 0^n of S_j. To maximize the number of 1s in $T_{\mathbf{opt}}$, pick such vertices to create a minimum vertex cover. The remaining vertices form a maximum independent set and thus $|T_{\mathbf{opt}}|=n^2+\alpha(G)$. Hence, if we suppose that $S_0,S_1,\ldots,S_m$ has a common subsequence of length $n^2+k\leq n^2+\alpha(G)$, then G has an independent set of size $k\leq\alpha(G)$. □

Taking run-lengths into consideration, we observe that LCS(2, 1) is certainly solvable in polynomial-time since each input sequence is a binary string with alternate symbols. (We shall prove in Section 3 that LCS$(2, 2)$ is, however, already **NP**-complete.) The following negative result is thus tight.

Proposition 2. LCS$(3, 1)$ *is* **NP**-*complete.*

Proof. Let G be a graph with n vertices and m edges. We construct $m + 1$ sequences of length at most $(n+1)(2n+1) - 2$, over alphabet $\Sigma = \{a, b, c\}$. The proof uses the construction of Proposition 1 where we replace each 0 by ab and each 1 by c. For example, in a graph with 7 vertices the edge between vertices 2 and 4 is represented by the sequence $(ab)^7c\ (ab)^7\ ((ab)^7c)^2\ (ab)^7\ ((ab)^7c)^3$

We claim that the graph G has an independent set of size k if and only if the sequences $S_0, S_1, \ldots, S_m$ have a common subsequence of length $2n^2 + k$.

For the direct implication, the proof works as for the one of Proposition 1. For the reverse implication, a similar argument can be used to show that there is an alignment that induces an lcs including all the as and bs of S_0, and for which all the included symbols from each $(ab)^nc$ substring of S_0 are aligned with symbols from the same $(ab)^nc$ or $(ab)^n$ substring of S_j, $1 \leq j \leq m$: instead of finding the rightmost 0, simply find the rightmost b included in the alignment (if the rightmost symbol in $\{a, b\}$ included in the alignment is an a, the alignment does not induce an lcs) and shift it right until it is the rightmost b of S_j or it is next to a c included in the alignment; then align all the other as and bs in the same $(ab)^nc$ or $(ab)^n$ substrings of S_j, $0 \leq j \leq m$. Then again, an lcs is obtained by including all the as and bs in S_0 and maximizing the number of cs that can be aligned, and the rest of the proof works. □

3 NP-Completeness of LCS$(2, 2)$

In the light of Proposition 1 and Proposition 2, a natural question arises: is LCS$(2, 2)$ polynomial-time solvable? In other words, for an alphabet of size 2, does limiting the run-length to its minimal non-trivial value enough to guarantee tractability? We answer by the negative, and this section is devoted to proving hardness of LCS$(2, 2)$.

The following easy property will turn to be extremely useful for the rest of the discussion as it allows us to focus on common subsequences with run-lengths at most 2.

Proposition 3. *Let $\mathcal{S}$ be an arbitrary input instance of* LCS$(2, 2)$. *Then, there exists an lcs of $\mathcal{S}$ with maximum run-length* 2.

Proof. For any three consecutive identical symbols in a longest common subsequence, the second symbol can always be replaced. □

We, however, observe that the above proposition cannot be taken as a general rule. Indeed, the following two propositions rule out any extension of Proposition 3 to larger alphabets and/or run-lengths.

Proposition 4. *For any integer n, there exist an input instance $\mathcal{S}$ of* LCS$(3, 1)$ *such that every lcs of $\mathcal{S}$ has maximum run-length at least n.*

Proof. It is enough to observe that sequences $(01)^n$ and $(02)^n$ have exactly one lcs, which is 0^n. □

Proposition 5. *For any integer n, there exist an input instance $\mathcal{S}$ of* LCS(2, 3) *such that every lcs of $\mathcal{S}$ has maximum run-length at least n.*

Proof. Let L_n be the set of all sequences on alphabet $\Sigma = \{0, 1\}$ that start with 0, contain exactly n 0s, have run-length at most 3 for 0, and contain no two consecutive 1s (*i.e.*, run-lengths at most 1 for 1). Clearly, a common subsequence of L_n has length n since 0^n is a common subsequence of L_n. We show by induction on n that the only lcs of L_n is 0^n. The property is certainly valid for $1 \le n \le 3$ since $0^n \in L_n$. For $n \ge 4$, assume that the property holds up to $n - 1$, and suppose, aiming at a contradiction, that there exists a common subsequence T of L_n that contains at least one 1. Write $T = 0^p\,1\,R$, where $0 \le p \le n$ and $R \in \Sigma^*$. Define $S = (01)^{p-1}0001$ ($S = 001$ if $p = 0$). The sequence S contains $p + 2$ 0s, and we claim that $0 \le p \le n - 2$. Indeed, if $p > n - 2$ then the smallest prefix of S that contains n 0s belongs to L_n but is not a super-sequence of T, a contradiction. Now, define $L' = SL_{n-p-2}$. We have $L' \subseteq L_n$, and hence T is a common subsequence of L'. Furthermore, by construction of S, R is a common subsequence of L_{n-p-2}. Therefore, by the induction hypothesis, R has length at most $n - p - 2$, and hence T has length at most $p + 1 + (n - p - 2) = n - 1$. □

Before diving into the reduction, we need the following definitions. A 10-*sequence* is a sequence starting with 1 and ending with 0. If S and T are 10-sequences, we say that S is a 10-*tight* subsequence of T if S is a subsequence of T and neither $10S$ nor $S10$ is a subsequence of T. The following easy lemmas are used in upcoming Proposition 6.

Lemma 1. *Let S_1, T_1, S_2, and T_2 be* 10 *sequences. If S_1 is a* 10*-tight subsequence of T_1 and S_2 is a* 10*-tight subsequence of T_2, then S_1S_2 is a* 10*-tight subsequence of T_1T_2.*

Lemma 2. *Let S_1, S_2, S_3, T_1, T_2, T_3 be* 10*-sequences where S_1 is a* 10*-tight subsequence of T_1, and S_3 is a* 10*-tight subsequence of T_3. Then, $S_1S_2S_3$ is a subsequence of $T_1T_2T_3$ if and only if S_2 is a subsequence of T_2.*

Proposition 6. LCS(2, 2) *is* **NP***-complete.*

The proof is by a reduction from 3-SAT. Let an arbitrary instance of 3-SAT be given by a CNF formula Φ with n variables $\{v_1, v_2, \ldots, v_n\}$ and m clauses. For any variable v_i, we write $+v_i$ for positive literal v_i, $-v_i$ for the negative literal $\overline{v_i}$, and $\pm v_i$ for any literal of variable v_i. First, we define 9 basic substrings as follows (bold is used to emphasize some difference between the strings; A_0 vs B_0, $\{X, Y, Z\}_-$ vs $\{X, Y, Z\}_+$):

$$\begin{aligned}
A_0 &= 1011\,\mathbf{0011}\,0010\\
B_0 &= 1011\,\mathbf{0101}\,0010\\
D_0 &= 1100\,1100\,1100\\
X_- &= 1011\,\mathbf{001}\,00100\\
Y_- &= 11011\,\mathbf{001}\,00100\\
Z_- &= 11011\,\mathbf{001}\,0010\\
X_+ &= 1011\,\mathbf{011}\,00100\\
Y_+ &= 11011\,\mathbf{011}\,00100\\
Z_+ &= 11011\,\mathbf{011}\,0010.
\end{aligned}$$

We now define $m+2$ sequences $\{A, B, C_1, C_2, \ldots, C_m\}$ based on these 9 substrings. The first two sequences are simply defined to be $A = (A_0)^n$ and $B = (B_0)^n$. For each $1 \le j \le m$, write $x \vee y \vee z$ for the j-th clause of the formula, where literals x, y and z are $\pm v_a$, $\pm v_b$, and $\pm v_c$ respectively and $a < b < c$. Define

$$X_j = \begin{cases} X_- & \text{if } x = -v_a \\ X_+ & \text{if } x = +v_a \end{cases}$$

$$Y_j = \begin{cases} Y_- & \text{if } y = -v_b \\ Y_+ & \text{if } y = +v_b \end{cases}$$

$$Z_j = \begin{cases} Z_- & \text{if } z = -v_c \\ Z_+ & \text{if } z = +v_c \end{cases}$$

and

$$\begin{aligned}
\mathcal{L} &= (A_0)^{a-1}\, X_j\, (1\,0\,D_0)^{b-a-1}\\
\mathcal{R} &= (D_0\,1\,0)^{c-b-1}\, Z_j\, (A_0)^{n-c}.
\end{aligned}$$

The string C_j is defined to be $C_j = \mathcal{L}\,1\,0\,Y_j\,1\,0\,\mathcal{R}$.

After the construction step, we now turn to proving the correctness of the reduction. We need some technical lemmas. Let $P = 1011\,\mathbf{011}\,0010$ and $N = 1011\,\mathbf{001}\,0010$.

Lemma 3. *A and B have 2^n lcs, they are exactly $\{P, N\}^n$.*

Proof. Consider an alignment that induces an lcs of A and B. Note that B_0 is divided into a left block 101101 and a right block 010010. There are $2n$ such blocks in B. We show that the number of fully matched blocks is exactly $m = n$.

Every sequence of $\{P, N\}^n$ has length $11n$ and is a common subsequences of A and B. Since $|A| = |B| = 12n$, both A and B have n symbols that are not part of the common subsequence. Therefore, at most n blocks can be partially matched (otherwise we miss more than n symbols), and hence $m \geq n$. Furthermore, each block of B that is fully matched introduces at least one gap in A (since no block of B is a substring of A). Thus we also have $m \leq n$. Moreover, in A, the gap between two substrings matched to two consecutive blocks of B must be 0.

In the alignment, the $2n$ blocks of B specify $2n$ corresponding blocks of A, where each fully matched block of B corresponds to a partially matched block of A with one gap, and, each partially matched block of B with one gap corresponds to a completely matched block of A. Observe that a completely matched block (10110)1 of B introduces a single gap in A only if it corresponds to a partially matched block (10110)01 of A which is a prefix of A_0, and that a completely matched block 0(10010) of B introduces a single gap in A only if it corresponds to a partially matched block 01(10010) of A which is a suffix of A_0. Since the prefix (10110)01 and the suffix 01(10010) both overlap in each A_0, there must be at most one completely matched block (10110)1 or 0(10010) in each B_0. Hence each B_0 contains one fully matched block and one partially matched block, and, for each B_0, the lcs contains either (10110)1 (10010) $= P$ or (10110) 0(10010) $= N$. Therefore, the lcs is in $\{P, N\}^n$. □

According to Lemma 3, if sequences $\{A, B, C_1, C_2, \ldots, C_m\}$ have an lcs of length $11n$, then it is in $\{P, N\}^n$. A sequence $S \in \{P, N\}^n$ is easily mapped to a truth assignment ϕ_S as follows: $\phi_S(v_i) =$ true if the i-th block of S is P, and $\phi_S(v_i) =$ false otherwise.

Lemma 4. *For any j and any sequence $S \in \{P, N\}^n$, S is a subsequence of C_j if and only if $\phi(v_s)$ satisfies the j-th clause of Φ.*

Proof. We write S_i for the i-th block of S (such that for every i, $S_i \in \{P, N\}$). For $i \leq j$, we write $S_{i..j} = S_i S_{i+1} \ldots S_j$. If ϕ_S satisfies a boolean term t, we write $\phi_S \vDash t$, otherwise $\phi_S \nvDash t$. We first need to compare P and N with each basic substring of C_j, and we obtain the following important relations (the proof is tedious but easy).

– P is a 10-tight subsequence of A_0, $10D_0$, $D_0 10$, X_+, Z_+, $X_- 10$, $10Z_-$.
– N is a 10-tight subsequence of A_0, $10D_0$, $D_0 10$, X_-, Z_-, $X_+ 10$, $10Z_+$.
– P (but not N) is a subsequence of Y_+, and N (but not P) is a subsequence of Y_-. P and N are subsequences of $10Y_-$, $Y_- 10$, $10Y_+$ and $Y_+ 10$.

We prove the lemma by combining Lemma 1 with the above relations. The proof is divided into three parts.

Part 1: from 1 **to** $b - 1$. Each S_i, $1 \leq i < a$, is a 10-tight subsequence of A_0, and hence $S_{1..a-1}$ is a 10-tight subsequence of $(A_0)^{a-1}$. Furthermore, sequence S_a is a subsequence of X_j if and only if $\phi_S \vDash x$ (since P is a subsequence of X_+ but not of X_-, and N is a subsequence of X_- but not of X_+). Similarly, S_a is a 10-tight subsequence of $X_j 10$ if and only if $\phi_S \nvDash x$. Hence,

– if $\phi_S \vDash x$, $S_{1..a}$ is a subsequence of $(A_0)^{a-1} X_j$,
– if $\phi_S \nvDash x$, $S_{1..a}$ is a 10-tight subsequence of $(A_0)^{a-1} X_j 10$.

Moreover, each S_i, $a < i < b$, is a 10-tight subsequence of $10D_0$ and $D_0 10$, and hence

– if $\phi_S \vDash x$, $S_{1..b-1}$ is a subsequence of $(A_0)^{a-1} X_j (10D_0)^{b-a-1} = \mathcal{L}$,
– if $\phi_S \nvDash x$, $S_{1..b-1}$ is a 10-tight subsequence of $(A_0)^{a-1} X_j (10D_0)^{b-a-1} 10 = \mathcal{L}10$.

Part 2: from n down to $b+1$. Using a similar argument as in Part 1, reading sequences from right to left, with Z_j instead of X_j, we have:

– if $\phi_S \vDash z$, $S_{b+1..n}$ is a subsequence of $(D_0 10)^{c-b-1} Z_j (A_0)^{n-c} = \mathcal{R}$,
– if $\phi_S \nvDash z$, $S_{b+1..n}$ is a 10-tight subsequence of $10(D_0 10)^{c-b-1} Z_j (A_0)^{n-c} = 10\mathcal{R}$.

Part 3: junction. If $\phi_S \vDash x$, then $S_{1..b-1}$, S_b, $S_{b+1..n}$ are subsequences of $\mathcal{L}$, $10Y_j$, $10\mathcal{R}$, respectively, and hence S is a subsequence of $C_j = \mathcal{L}10Y_j10\mathcal{R}$. If $\phi_S \vDash y$, then $S_{1..b-1}$, S_b, $S_{b+1..n}$ are subsequences of $\mathcal{L}10$, Y_j, $10\mathcal{R}$, respectively, and hence S is a subsequence of $C_j = \mathcal{L}10Y_j10\mathcal{R}$. If $\phi_S \vDash z$, then $S_{1..b-1}$, S_b, $S_{b+1..n}$ are subsequences of $\mathcal{L}10$, Y_j10, $\mathcal{R}$, respectively, and hence S is a subsequence of $C_j = \mathcal{L}10Y_j10\mathcal{R}$.

If $\phi_S \nvDash x \vee y \vee z$, then S_b is not a subsequence of Y_j. Since $S_{1..b-1}$ and $S_{b+1..n}$ are 10-tight subsequences of $\mathcal{L}10$ and $10\mathcal{R}$, respectively, then, according to Lemma 2, $S = S_{1..b-1} S_b S_{b+1..n}$ is not a subsequence of $C_j = \mathcal{L}10\ Y_j\ 10\mathcal{R}$. □

Proof (Proof of Proposition 6.). Let S be an lcs of $\{A, B, C_1, \ldots, C_m\}$. If S has length at least $11n$, then by Lemma 3 it is in $\{P, N\}^n$, and by Lemma 4, ϕ_S satisfies every clause of Φ, and hence Φ is satisfiable. Conversely, if Φ is satisfiable, set any truth assignment, and create sequence $S = S_1\, S_2\, \ldots\, S_n$ such that $S_i = P$ if v_i is true, and $S_i = N$ otherwise. Then S is a common subsequence of $\{A, B, C_1, C_2, \ldots, C_m\}$ of size $11n$. Since the construction of sequences $A, B, C_1, \ldots, C_m$ can be carried on in polynomial-time, LCS$(2,2)$ is **NP**-complete. □

4 Approximation

In its most general setting (unbounded number of strings, etc.), LCS is approximable within ratio $O(m/\log(m))$, where m is the length of the shortest input string [12]. It is, however, not approximable within ratio n^{ε} for any constant $\varepsilon < 1$, where n is the length of the longest input string [15] (see also [5]), and it is **APX**-hard if the size of the alphabet is fixed [15]. Despite the discouraging results, it has been proved that LCS over a fixed alphabet can be indeed very well approximated on the average by using a simple algorithm called *Long Run* [15]. Bonizzoni et al. [8] developed an approximation algorithm for LCS called *Expansion Algorithm* (their algorithm first compresses sequences to streams by the same concept of run-length encoding, then progressively find a common sequence of all streams by the bottom-up tree merging technique).

We present here a 3/5-approximation algorithm for LCS$(2,2)$. The algorithm performs an exhaustive search for a limited number of common subsequences.

The approximation ratio analysis, which uses linear programming techniques, is based on the structure of optimal solutions of LCS$(2,2)$ obtained in Proposition 3.

Proposition 7. LCS$(2,2)$ *is approximable within ratio* $3/5$.

Proof. Let $S_1, S_2, \ldots, S_m$, be m input sequences over alphabet $\Sigma = \{0,1\}$ with maximum run-length 2. The approximation algorithm is as follows:

```
Input: Subsequences $S_1, S_2, \ldots, S_n$ over alphabet $\Sigma = \{0,1\}$
Output: A common subsequence of $S_1, S_2, \ldots, S_n$

  for all $P \in \{01, 001, 011\}$ do
    Let $T_P$ be the lcs of $S_1, S_2, \ldots, S_n$ in $\{P\}^*$
  end for
  return maximum length of $T_P$, $P \in \{01, 001, 011\}$
```

For the sake of clarity, write $A = 01$, $B = 001$, $C = 011$, $D = 0011$, and $X = \{A, B, C, D\}$. We focus on the case where each input sequence starts with 0 and terminates with 1 (the general case is easily deduced from this restriction).

Let $T_{\mathbf{opt}}$ be an lcs of $S_1, S_2, \ldots, S_m$, and $n_{\mathbf{opt}} = |T_{\mathbf{opt}}|$. According to Proposition 3, there is no loss of generality in assuming that $T_{\mathbf{opt}}$ has maximum run-length 2. First, it is easily seen that X is a code (*i.e.*, any $u \in \Sigma^*$ has at most one X-factorization). Furthermore, since each input sequence starts with 0 and terminates with 1, so does any lcs. Thus $T_{\mathbf{opt}}$ has maximum run-length 2, starts with 0 and terminates with 1, hence it has an X-factorization into k factors $T_{\mathbf{opt}} = t_1 t_2 \ldots t_k$. Let n_A (respectively n_B, n_C, and n_D) be the number of indices i, $1 \leq i \leq k$, such that $t_i = A$ (respectively, $t_i = B$, $t_i = C$, and $t_i = D$), and define $\overline{n_A} = n_A/n_{\mathbf{opt}}$, $\overline{n_B} = n_B/n_{\mathbf{opt}}$, $\overline{n_C} = n_C/n_{\mathbf{opt}}$, $\overline{n_D} = n_D/n_{\mathbf{opt}}$. Summing up the lengths of all factors, we have $n_{\mathbf{opt}} = 2n_A + 3n_B + 3n_C + 4n_D$, and

$$2\overline{n_A} + 3\overline{n_B} + 3\overline{n_C} + 4\overline{n_D} \geq 1 \tag{1}$$

(Notice that (1) is fundamentally an equality but only the "$\geq$" part is used in the rest of the proof.)

We now turn to relating $|T_P|$, $P \in \{01, 001, 011\}$, to n_A, n_B, n_C, and n_D. From optimality of $|T_P|$, $P \in \{01, 001, 011\}$, we obtain

$$|T_A| \geq 2n_A + 2n_B + 2n_C + 2n_D$$
$$|T_B| \geq 3n_B + 3n_D$$
$$|T_C| \geq 3n_C + 3n_D.$$

By definition, the approximation ratio r of our algorithm is

$$r = \max\{|T_A|, |T_B|, |T_C|\}/n_{\mathbf{opt}},$$

and hence

$$r \geq 2\overline{n_A} + 2\overline{n_B} + 2\overline{n_C} + 2\overline{n_D} \quad (2)$$

$$r \geq 3\overline{n_B} + 3\overline{n_D} \quad (3)$$

$$r \geq 3\overline{n_C} + 3\overline{n_D} \quad (4)$$

Inequalities (1) to (4) together with domain constraints $n_A \geq 0$, $n_B \geq 0$, $n_C \geq 0$, and $n_D \geq 0$ define a (minimization) linear program (LP) that can be solved as follows. Let r^* be the optimal solution of the defined LP (referred to as the primal problem). We consider the dual LP:

$$\begin{aligned} \text{maximize } & y_1 \\ \text{such that } & y_2 + y_3 + y_4 \leq 1 \\ & 2y_1 \leq 2y_2 \\ & 3y_1 \leq 2y_2 + 3y_3 \\ & 3y_1 \leq 2y_2 + 3y_4 \\ & 4y_1 \leq 2y_2 + 3y_3 + 3y_4 \end{aligned}$$

By the strong duality theorem, r^* is also the optimal solution for the dual LP. A lower bound for the dual LP is obtained by choosing: $y_1 = y_2 = 3/5$ and $y_3 = y_4 = 1/5$, and hence $r^* \geq 3/5$, thereby proving the proposition. □

Remark that the approximation ratio of the proposed algorithm is exactly 3/5. Indeed, for the sequence $(0110010011)^n$ none of the tested patterns in $\{01, 001, 011\}$ would return a subsequence of length greater than $3n/5$.

References

1. Ann, H.-Y., Yang, C.-B., Tseng, C.-T., Hor, C.-Y.: Fast algorithms for computing the constrained lcs of run-length encoded strings. In: Arabnia, H.R., Yang, M.Q. (eds.) Proc. International Conference on Bioinformatics & Computational Biology (BIOCOMP), Las Vegas, USA, pp. 646–649. CSREA Press (2009)
2. Ann, H.-Y., Yang, C.-B., Tseng, C.-T., Hor, C.-Y.: A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings. Information Processing Letters 108, 360–364 (2008)
3. Apostolico, A., Landau, G.M., Skiena, S.: Matching for run-length encoded strings. Journal of Complexity 15(1), 4–16 (1999)
4. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Proc. of the 7th International Symposium on String Processing Information Retrieval (SPIRE), Coruña, Spain, pp. 39–48. IEEE Computer Society (2000)
5. Berman, P., Schnitger, G.: On the complexity of approximating the independent set problem. Information and Computation 96, 77–94 (1992)
6. Bodlaender, H.L., Downey, R.G., Fellows, M.R., Hallett, M.T., Wareham, H.T.: Parameterized complexity analysis in computational biology. Computer Applications in the Biosciences 11(1), 49–57 (1995)

7. Bodlaender, H.L., Downey, R.G., Fellows, M.R., Wareham, H.T.: The parameterized complexity of sequence alignment and consensus. Theoretical Computer Science 147, 31–54 (1994)
8. Bonizzoni, P., Della Vedova, G., Mauri, G.: Experimenting an approximation algorithm for the lcs. Discrete Applied Mathematics 110(1), 13–24 (2001)
9. Bunke, H., Csirik, J.: An improved algorithm for computing the edit distance of run-length coded strings. Information Processing Letters 54, 93–96 (1995)
10. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings, Cambridge (2007)
11. Freschi, V., Bogliolo, A.: Longest common subsequence between run-length-encoded strings: a new algorithm with improved parallelism. Information Processing Letters 90, 167–173 (2004)
12. Halldórsson, M.M.: Approximation via partitioning. Technical report, School of Information Science, Japan Advanced Institute of Science and Technology, Hokuriku (1995)
13. Hsu, P.-H., Chen, K.-Y., Chao, K.-M.: Finding All Approximate Gapped Palindromes. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1084–1093. Springer, Heidelberg (2009)
14. Huang, G.S., Liu, J.J., Wang, Y.L.: Sequence Alignment Algorithms for Run-Length-Encoded Strings. In: Hu, X., Wang, J. (eds.) COCOON 2008. LNCS, vol. 5092, pp. 319–330. Springer, Heidelberg (2008)
15. Jiang, T., Li, M.: On the approximation of shortest common supersequences and longest common subsequences. SIAM Journal on Computing 24, 1122–1139 (1995)
16. Hsu, P.-H., Chen, K.-Y., Chao, K.-M.: Finding All Approximate Gapped Palindromes. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1084–1093. Springer, Heidelberg (2009)
17. Kim, J.W., Amir, A., Landau, G.M., Park, K.: Computing similarity of run-length encoded strings with affine gap penalty. Theoretical Computer Science 395, 268–282 (2008)
18. Liu, J.J., Huang, G.S., Wang, Y.L.: A fast algorithm for finding the positions of all squares in a run-length encoded string. Theoretical Computer Science 410, 3942–3948 (2009)
19. Liu, J.J., Huang, G.S., Wang, Y.L., Lee, R.C.T.: Edit distance for a run-length-encoded string and an uncompressed string. Information Processing Letters 105, 12–16 (2007)
20. Liu, J.J., Wang, Y.L., Lee, R.C.T.: Finding a longest common subsequence between a run-length-encoded string and an uncompressed string. Journal of Complexity 24, 173–184 (2008)
21. Maier, D.: The complexity of some problems on subsequences and supersequences. Journal of the ACM 25(2), 322–336 (1978)
22. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Efficient algorithms to compute compressed longest common substrings and compressed palindromes. Theoretical Computer Science 410, 900–913 (2009)
23. Mitchell, J.S.B.: A geometric shortest path problem, with application to computing a longest common subsequence in run-length encoded strings. Technical report, Department of Applied Mathematics, SUNY Stony Brook (1997)
24. Pietrzak, K.: On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. J. of Computer and System Sciences 67(4), 757–771 (2003); Special issue on Parameterized computation and complexity

Near Linear Time Construction of an Approximate Index for All Maximum Consecutive Sub-sums of a Sequence

Ferdinando Cicalese[1], Eduardo Laber[2], Oren Weimann[3], and Raphael Yuster[4]

[1] Department of Computer Science, University of Salerno, Italy
cicalese@dia.unisa.it

[2] Department of Informatics, PUC-Rio, Rio de Janeiro, Brazil
laber@inf.puc-rio.br

[3] Department of Computer Science, University of Haifa, Israel
oren@cs.haifa.ac.il

[4] Department of Mathematics, University of Haifa, Israel
raphy@math.haifa.ac.il

Abstract. We present a novel approach for computing all maximum consecutive subsums in a sequence of positive integers in near linear time. Solutions for this problem over binary sequences can be used for reporting existence (and possibly one occurrence) of Parikh vectors in a bit string. Recently, several attempts have been tried to build indexes for all Parikh vectors of a binary string in subquadratic time. However, to the best of our knowledge, no algorithm is know to date which can beat by more than a polylogarithmic factor the natural $\Theta(n^2)$ exhaustive procedure. Our result implies an approximate construction of an index for all Parikh vectors of a binary string in $O(n^{1+\eta})$ time, for any constant $\eta > 0$. Such index is approximate, in the sense that it leaves a small chance for false positives, i.e., Parikh vectors might be reported which are not actually present in the string. No false negative is possible. However, we can tune the parameters of the algorithm so that we can strictly control such a chance of error while still guaranteeing strong sub-quadratic running time.

Keywords: Parikh vectors, maximum subsequence sum, approximate pattern matching, approximation algorithms.

1 Introduction

Let $\mathbf{s} = s_1, \ldots, s_n$ be a sequence of non-negative integers. For each $\ell = 1, \ldots, n$, we denote with m_ℓ the maximum sum over a consecutive subsequence of $\mathbf{s}$ of size ℓ, in formulae:

$$m_\ell = \max_{i=1,\ldots,n-\ell+1} \sum_{j=i}^{i+\ell-1} s_j.$$

The Maximum Consecutive Subsums Problem (MCSP) asks for computing m_ℓ for each $\ell = 1, \ldots, n$.

Since an obvious implementation of the above formula allows to compute m_ℓ for a single value of ℓ in $O(n)$ time, it follows that there exists a trivial $O(n^2)$ procedure to accomplish the above task. The interesting question is then about subquadratic procedure for solving MCSP. Notwithstanding quite some effort which has been recently devoted to the problem, surprisingly enough, no algorithm is known which is significantly better than the natural $\Theta(n^2)$.

In this paper we show that we can closely approximate the values m_ℓ (within an approximation factor as close to 1 as desired) with a procedure whose running time can be as close to linear as desired. More precisely, our main result is as follows.

Main Theorem. *For any $\epsilon, \eta > 0$, there exists an algorithm that computes values $\tilde{m}_1, \ldots, \tilde{m}_n$ such that $1 \leq \tilde{m}_j / m_j \leq 1 + \epsilon$, for each $j = 1, \ldots, n$, in time $O(k_{\epsilon,\eta} \cdot n^{1+\eta})$, where $k_{\epsilon,\eta}$ is a constant only depending on ϵ and η.*

MCSP arises in several scenarios of both theoretical and practical interest. Our main motivation for studying MCSP comes from the following Parikh vector matching problem.

An Index for Constant Time Parikh Vector Membership Queries. Given a string t over an alphabet $[\sigma] = \{1, 2, \ldots, \sigma\}$, for each $c \in [\sigma]$, let x_c be the number of occurrences of character c in t. The vector $(x_1, \ldots, x_\sigma)$ is called the Parikh vector of t.

In the Parikh vector matching problem, given a string t (the text) over the alphabet $[\sigma]$, together with a vector of non-negative integers $p = (x_1, \ldots, x_\sigma)$ (the Parikh vector pattern) we ask for (all/one/existence) of occurrences of substrings s of t such that the Parikh vector of s is p.

Equivalently, we are asking for all the *jumbled* occurrences of a string s (the pattern) in a string t (the text), i.e., any occurrence of some permutation of s in t. Therefore, the Parikh vector matching problem is a type of approximate string matching [16,10].

Typical applications of such model come from interpretation of mass spectrometry data analysis [5]. More generally, Parikh vector matching applies in scenarios where, notwithstanding the linearity (mono-dimensional) of the structure in which we perform the pattern search, it is not important the order in which what we are searching for, actually occurs. A typical example might be testing for bio-chemical characteristics of (part) of a macromolecule which only depends upon the presence of some substructures and their occurrence within a relatively short distance, whilst their relative order is not significant [1,3,15,18].

Given an alphabet $[\sigma]$ and a string s it is not hard to see that the maximum number of Parikh vectors occurring in s is $O(n^2)$, since each Parikh vector is associated to at least one of the $\Theta(n^2)$ substrings of s. On the other hand, for any given length n there exist $\Omega(n^\sigma)$ distinct Parikh vectors.[1] This also motivates

[1] The number of Parikh vector of length n is equal to the number of ways we can split n elements into σ parts, which is exactly $\binom{n+\sigma-1}{\sigma-1}$.

filtering algorithms based on executing membership queries before looking for the actual occurrences of the Parikh vector query.

For binary string t, knowing for each $\ell = 1, \ldots, n$ the minimum and maximum number of 1's found in a substring of size ℓ of t, we can answer membership queries in constant time. More precisely, the connection between MCSP and Parikh vector membership query problem is given by the following.

Lemma 1. *[9] Let s be a binary string and for each $i = 1, \ldots, n$ let $\mu_\ell^{\min}$ (resp. $\mu_\ell^{\max}$) denote the minimum (resp. maximum) number of ones in a substring of s of length ℓ. Then for any Parikh vector $p = (x_0, x_1)$ we have that there exists a substring in s with Parikh vector p if and only if $\mu_{x_0+x_1}^{\min} \leq x_1 \leq \mu_{x_0+x_1}^{\max}$.*

It follows that, after constructing the tables of $\mu^{\min}$'s and $\mu^{\max}$'s—which is equivalent to solving two instances of the MCSP—we can answer in constant time Parikh vector membership queries, i.e., questions asking: "Is there an occurrence of the Parikh vector (x_0, x_1) in s?" This is achieved by simply checking the condition in the previous Lemma.

Therefore, there has been significant interest in trying to solve the MCSP on binary sequences in subquadratic time.

To the best of our knowledge the best known constructions are as follows: Burcsi *et al.* in [9,8] showed a $O(n^2/\log n)$-time algorithm which is based on the $O(n^2/\log n)$ algorithm of Bremner *et al.* [7,12] for computing $(\min, +)$-convolution; Moosa and Rahman in [17] obtained the same result by a different use of $(\min, +)$-convolution, moreover, they show that an $O(n^2/\log^2 n)$ construction can be obtained assuming word-RAM operations.

Another interesting application of MCSP is in the following problem in the analysis of statistical data.

Finding Large Empty Regions in Data Sets. Bergkvist and Damaschke in [4] used the index of all maximum consecutive subsums of a sequence of numbers for speeding up heuristics for the following problem: Given a sequence of positive real numbers $x_1, \ldots, x_n$, called items, and integers $s \geq 1$ and $p \geq 0$, find s pairwise disjoint intervals with total of $s + p$ items and maximum total length. Here an interval is a set of consecutive items $x_i, x_{i+1}, \ldots, x_j$ $(i \leq j)$ and its length is $x_i + x_{i+1} + \ldots x_j$. This problem, aka DIMaxL (Disjoint Intervals of Maximum Length) and its density variant where we are interested in interval of maximum density [12], rather than absolute length, are motivated by the problem of finding large empty regions (big holes) in data sets. Several other motivating applications can be found in [4] and [12].

By employing a geometric argument, in [4] a heuristic procedure is presented for the MCSP which can solve the problem in $O(n^{3/2})$ time in the best case, but whose worst case remains quadratic.

2 The Approximate Index

Let $\mathbf{s} = s_1, \ldots, s_n$ be a sequence of non-negative integers. Our aim is to compute $m_\ell = \max_{i=1,\ldots,n-\ell+1} \sum_{j=i}^{i+\ell-1} s_j$ for each $\ell = 1, \ldots, n$. In this section we will prove the following result

Theorem 1. *For any constant $\epsilon, \eta \in (0,1)$, let k be the minimum positive integer such that the positive real solution $\tilde{\alpha}$ of the equation $\alpha = 1 + 1/\alpha^k$, satisfies $1 + \epsilon > \tilde{\alpha}$. Let $t = \lceil 1/\eta \rceil$.*

There exists an algorithm which in time $O(k^{t-1} \cdot n^{1+\eta})$ computes values $\tilde{m}_1, \ldots, \tilde{m}_n$ such that $1 \leq \tilde{m}_\ell / m_\ell \leq 1 + \epsilon$, for each $\ell = 1, \ldots, n$.

We shall start describing our approach by first presenting an algorithm which computes a $(\frac{1+\sqrt{5}}{2})$-approximation of the values m_ℓ's in $O(n^{3/2})$ time. Then we will show a variant of the approach which can be parametrized to achieve the desired $(1+\epsilon)$-approximation in time $O(n^{1.5})$. Finally, we will show how, by recursively applying such a strategy we can achieve the same approximation in time $O(n^{1+\eta})$, for any constant $\eta > 0$.

We will use the following simple facts.

Fact 1. *For each $1 \leq i < j \leq n$ it holds that $m_i \leq m_j$.*

Fact 2. *For each $\ell \in [n]$ and positive integers i, j such that $i + j = \ell$, it holds that $m_\ell \leq m_i + m_j$.*

Fact 1 directly follows by the non-negativity of the elements in the sequence. Fact 2 is a consequence of the following easy observation. For a fixed ℓ, let $s_r, \ldots, s_{r+\ell-1}$ be a subsequence achieving $s_r + \cdots + s_{r+\ell-1} = m_\ell$. By definition we have that, for any i, j such that $i + j = \ell$ it holds that $s_r + \cdots + s_{r+i-1} \leq m_i$ and $s_{r+i} + \cdots + s_{r+\ell-1} \leq m_j$, from which we obtain the desired inequality.

For ease of presentation, we shall usually neglect rounding necessary to preserve the obvious integrality constraints. The reader can assume that, when necessary, numbers are rounded to the closest integer. It will always be clear that these inaccuracies do not affect the asymptotic results. On the other hand, this way, we gain in terms of much lighter expressions.

2.1 Warm-Up: A Golden Ratio Approximation in $O(n^{3/2})$

Let $\alpha = \frac{1+\sqrt{5}}{2}$. The value α defines the approximation of our solutions. Fix an integer $t \geq 2$ and set $g = n^{1/2}$.

The basic idea is to compute via exhaustive search the value of $m_{j \times g}$ for each $j = 1, \ldots, n/g$ and then to use these values for approximating all the others.

For each $j = 1, \ldots, n/g$, we set $\tilde{m}_{j \times g} = m_{j \times g}$, i.e., our approximate index will contain the exact value.

Let ℓ be such that $j \times g < \ell < (j+1) \times g$ for some $j = 1, \ldots, n/g - 1$. By Fact 1 we have that $m_{j \times g} \leq m_\ell \leq m_{(j+1) \times g}$. Therefore, if $m_{(j+1) \times g} / m_{j \times g} \leq \alpha$, by setting $\tilde{m}_\ell = m_{(j+1)\times g}$ we also have that $\tilde{m}_\ell$ is an α-approximation of the real value m_ℓ.

What happens if the gap between $m_{(j+1)\times g}$ and $m_{j\times g}$ is large?

If $m_{(j+1)\times g}/m_{j\times g} > \alpha$, our idea is to compute exhaustively m_ℓ for each $\ell = j \times g + 1, \dots, (j+1) \times g - 1$. The critical point here is that the above "large" gap can only happen once! This is formalized in the following argument: Let $j > 0$ be the minimum integer such that $m_{(j+1)\times g}/m_{j\times g} > \alpha$. Therefore, by Fact 1 we also have

$$m_{(j+1)\times g}/m_g > \alpha. \tag{1}$$

Now, for each $i > j$, we have

$$\frac{m_{(i+1)\times g}}{m_{i\times g}} \leq \frac{m_{i\times g} + m_g}{m_{i\times g}} = 1 + \frac{m_g}{m_{i\times g}} \leq 1 + \frac{m_g}{m_{(j+1)\times g}} < 1 + 1/\alpha = \alpha, \tag{2}$$

where the first inequality follows by Fact 2; the second inequality follows by Fact 1 together with $i \geq (j+1)$, respectively; the third inequality follows from (1) and the last equality because $\alpha = (1+\sqrt{5})/2$.

Therefore, after encountering the first large gap between $m_{j\times g}$ and $m_{(j+1)\times g}$ all the following gaps will be "small", hence we can safely set $\tilde{m}_\ell = m_{(i+1)\times g}$ for each $i > j$ and $i \times g < \ell \leq (i+1) \times g$. In fact, using again Fact 1 we have $\tilde{m}_\ell/m_\ell \leq m_{(i+1)\times g}/m_{i\times g}$ and then by (2) we are guaranteed that $\tilde{m}_\ell = m_{(i+1)\times g}$ is indeed an α-approximation of the exact m_ℓ.

2.2 As Close to 1 as Wished

In this section we prove our main result which follows by refining the algorithm described in the previous section. Here we use the expression "*compute m_ℓ exhaustively*" (for some fixed ℓ) to indicate the linear time computation attained by scanning the sequence **s** from left to right and computing the sum of all consecutive subsequences of size ℓ. This exhaustive computation for a single ℓ can be clearly achieved in $\Theta(n)$ time.

Let k be the minimum positive integer such that $\alpha \leq 1 + \epsilon$, where α is the positive real solution of the equation $\alpha = 1 + 1/\alpha^k$. In an explicit way, $k = \lceil -\frac{\ln(\epsilon)}{\ln(1+\epsilon)} \rceil$.

The value α defines the approximation of our solutions. Let us also set $g = n^{1/2}$.

We partition the list of values m_ℓ ($\ell = 1, \dots, n$) into n/g intervals, each of them with g consecutive values. Then, we proceed as follows:

1. Compute exhaustively the exact value for each m_ℓ in the first interval, i.e., for m_ℓ such that $\ell = 1, \dots, g$.
2. Compute exhaustively the exact value for the extremes of all intervals (i.e., for m_ℓ such that $\ell = 2g, 3g, \dots$).
3. Let $m_g, m_{2g}, \dots$ be the extremes of the n/g intervals. We say that an extreme $m_{i\times g}$ is relevant if $m_{i\times g}/m_{(i-1)\times g} > \alpha$. Compute exhaustively m_ℓ for every ℓ such that the rightmost extreme of its interval is among the first k relevant extremes.

4. The remaining points are approximated by the value of the rightmost point in the interval where they lie (i.e., for $\ell \in \{jg+1,\ldots,(j+1)g-1\}$ such that $m_{(j+1)g}/m_{jg} \leq \alpha$ we set $\tilde{m}_\ell = m_{(j+1)g}$).

We have to prove that the values $\tilde{m}$'s satisfy the desired $(1+\epsilon)$-approximation.

Let $j_1,\ldots,j_r$, with $r \leq k$, be the values of j for which the algorithm verified $m_{j\times g}/m_{(j-1)\times g} > \alpha$ and hence computed exhaustively m_ℓ in the interval $\ell = (j-1)g,\ldots,jg$.

It is easy to see that $\tilde{m}_\ell/m_\ell \leq \alpha$ for each $\ell \leq j_r \times g$.

Moreover, if $r < k$, it also means that $m_{j\times g}/m_{(j-1)\times g} \leq \alpha$ for each $j > j_r$. Hence, we also have that $\tilde{m}_\ell/m_\ell \leq \alpha$ for each $\ell > j_r \times g$.

Assume now that $r \geq k$. Let us write j^* for j_r. Since for k times we had that $m_{j\times g}/m_{(j-1)\times g} > \alpha$, we have in particular that

$$\frac{m_{j^*\times g}}{m_g} > \alpha^k. \tag{3}$$

Now, let us consider an integer ℓ such that $i \times g < \ell < (i+1)\times g - 1$ for some $i \geq j^*$. In such case we have that $\tilde{m}_\ell = m_{(i+1)\times g}$. Therefore, for our purposes, it is enough to show that $\frac{m_{(i+1)\times g}}{m_\ell} \leq \alpha$.

Indeed we have

$$\frac{m_{(i+1)\times g}}{m_\ell} \leq \frac{m_{(i+1)\times g}}{m_{i\times g}} \leq \frac{m_{i\times g}+m_g}{m_{i\times g}} = 1 + \frac{m_g}{m_{i\times g}} \leq 1 + \frac{m_g}{m_{j^*\times g}} < 1 + \frac{1}{\alpha^k} = \alpha.$$

The first inequality follows by Fact 1, since $\ell > i \times g$. The second inequality follows by Fact 2 yielding $m_\ell \leq m_{i\times g} + m_{\ell - i\times g}$ together with $m_{\ell-i\times g} \leq m_g$ (by Fact 1 and $\ell - i \times g < g$).

The third inequality follows by $m_{i\times g} \geq m_{j^*\times g}$ ($i \geq j^*$ and Fact 1). The fourth inequality follows from (3) and the last equality by the definition of α.

This concludes the proof that for each ℓ the values $\tilde{m}_\ell$ is indeed an α-approximation (and hence a $(1+\epsilon)$-approximation) of m_ℓ.

For the time bound, we observe that the number of times in the above procedure we compute with the exhaustive linear procedure some value m_ℓ is at most $(k+1)\times n^{1/2}$ for the full intervals of size g and $n^{1-1/2}$ for the extremes of the intervals. Therefore the algorithm runs in time $O(k \times n^{3/2})$.

2.3 The Last Piece: A Recursive Argument

In the above procedure, for $g = n^{1/2}$ we first compute $m_g, m_{2g}, m_{3g}, \ldots, m_n$ in time $O(n^{3/2})$. Then, we identify (up to) $k+1$ relevant intervals. The first interval is $[m_1, m_g]$ and the other k intervals are all the ones of the form $[m_{jg}, m_{(j+1)g}]$ where $m_{(j+1)g}/m_{jg} > \alpha$. For each one of the $k+1$ relevant intervals we compute *exhaustively* all m_ℓ values inside the interval, hence in total requiring time $O(k \times n^{3/2})$.

In order to reduce the time complexity, instead of computing all the values exhaustively in the relevant intervals we can recursively use in each interval the argument we use for the full "interval" $m_1, \ldots, m_n$. In particular, this means subdividing each relevant interval into subintervals and computing exhaustively the m_ℓ values only at the subintervals' boundaries, but for $k+1$ relevant subintervals, where we recurse again.

Let us first consider an example giving a $(1+\epsilon)$-approximation in time $O(n^{1+1/3})$. For this we choose $g = n^{2/3}$ and compute exhaustively $m_g, m_{2g}, \ldots,$ m_n, so spending in total $O(n^{1+1/3})$ time. Then, we identify $k+1$ relevant intervals just as before. Suppose that $[m_{jg}, m_{(j+1)g}]$ is one of the $k+1$ relevant intervals, i.e., $m_{(j+1)g}/m_{jg} > \alpha$. We partition this interval into $n^{1/3}$ sub-intervals each of size $n^{1/3}$. Now, we first compute m_ℓ for every sub-interval endpoint, which requires $n^{1/3} \times n = O(n^{1+1/3})$ time. Then, we compute m_ℓ exhaustively for each ℓ in the $k+1$ relevant sub-intervals, i.e., the first sub-interval and the ones for which the ratio of the values m_ℓ at the boundaries is greater than α. As done in the previous subsection, it can be easily shown that at most k sub-intervals can exist for which such condition is verified.

Overall, the number of values of ℓ for which we compute m_ℓ exhaustively are:

– the $n^{1/3}$ boundaries $g, 2g, \ldots$. This requires $O(n^{1/3})$ exhaustive computations of some m_ℓ.
– the $n^{1/3}$ boundaries of the sub-interval in the $k+1$ relevant intervals, i.e., in total $O(kn^{1/3})$ exhaustive computations of some m_ℓ.
– for each one of the $k+1$ relevant interval, we may have up to $k+1$ relevant sub-intervals and each relevant sub interval requires to compute $n^{1/3}$. In total this gives additional $O(k^2 n^{1/3})$ exhaustive computations of some m_ℓ.

Therefore, we have that the time complexity becomes $O(k^2 n^{1+1/3})$.

If we want to get $O(n^{1+1/4})$ running time we can choose $g = n^{3/4}$ and add one more level of recursion, i.e., partition the sub-intervals to sub-sub-intervals. This way the running time becomes $O(k^3 n^{1+1/4})$.

In general, given any fixed $\eta > 0$ we can set $t = \lceil 1/\eta \rceil$ and $g = n^{1-1/t}$. By using $t-1$ levels of recursion we then get a $(1+\epsilon)$-approximation in $O(k^t n^{1+1/t}) = O(k_{\epsilon,\eta} n^{1+\eta})$ time, where $k_{\epsilon,\eta}$ is a constant depending only on ϵ and η. This provides the proof of our main theorem. The algorithm is described in the pseudocode below.

3 Applying the Index to the Parikh Vector Matching Problem

We can now analyze the consequences of our result with respect to the connection between MCSP and Parikh vector membership query problem.

An immediate corollary of Theorem 1 is the following.

Corollary 1. *Let s be a binary string and for each $i = 1, \ldots, n$ let $\mu_\ell^{\min}$ (resp. $\mu_\ell^{\max}$) denote the minimum (resp. maximum) number of ones in a substring of s of lenght ℓ. For any $\epsilon, \eta \in (0,1)$, we can compute in $O(n^{1+\eta})$ approximate values $\tilde{\mu}_\ell^{\min}$ (resp. $\tilde{\mu}_\ell^{\max}$) such that*

$$\mu_\ell^{\min} \geq \tilde{\mu}_\ell^{\min} \geq (1-\epsilon)\mu_\ell^{\min} \qquad \mu_\ell^{\max} \leq \tilde{\mu}_\ell^{\max} \leq (1+\epsilon)\mu_\ell^{\max}$$

Algorithm 1. Approximate index for all maximum consecutive subsums

```
Input: A string $s$, an approximation value $\epsilon$ and a time threshold $\eta$ s.t. $\epsilon, \eta \in (0,1)$.
Output: $\tilde{m}_1, \ldots, \tilde{m}_n$ such that $m_j/\tilde{m}_j \leq \epsilon$, for each $j \in [n]$.
 1: Set $k$ to the minimum integer s.t. $1+\epsilon$ is not smaller than the positive real solution
    of $\alpha = 1 + 1/\alpha^k$.
 2: Set $t = \lceil 1/\eta \rceil$
 3: **Recursive-Approx**$(1, n, 1)$
 4: **return** $\tilde{m}_1, \ldots, \tilde{m}_n$.

**Recursive-Approx**$(start, end, depth)$
 1: if $depth = t - 1$ then
 2:    for all $\ell = start, \ldots, end$ do
 3:       compute $m_\ell$ exhaustively and set $\tilde{m}_\ell = m_\ell$
 4:    end for
 5: else
 6:    Set $size = (end - start + 1)$ and $g = size/n^{1/t}$.
 7:    Recursive-Approx$(start, start + g, depth + 1)$
 8:    Set $j = 2$ and $\kappa = 0$.
 9:    while $j \leq n^{1/t}$ and $\kappa < k$ do
10:       compute $m_{start+jg}$ exhaustively and set $\tilde{m}_{start+jg} = m_{start+jg}$
11:       if $m_{start+j\times g} > \alpha\, m_{start+(j-1)\times g}$ then
12:          Recursive-Approx$(start + (j-1)g, start + jg, depth + 1)$
13:          $\kappa = \kappa + 1$
14:       else
15:          for all $\ell = start + (j-1)g + 1, \ldots, start + jg - 1$ do
16:             Set $\tilde{m}_\ell = m_{start+jg}$
17:          end for
18:       end if
19:       $j = j + 1$
20:    end while
21:    for all $i = j, \ldots, n^{1/t}$ do
22:       compute $m_{start+ig}$ exhaustively and set $\tilde{m}_{start+ig} = m_{start+ig}$
23:       for all $\ell = start + (i-1)g + 1, \ldots, start + ig - 1$ do
24:          Set $\tilde{m}_\ell = m_{start+ig}$
25:       end for
26:    end for
27: end if
```

Let $\mathbf{s}$ be a binary string of length n. Fix a tolerance threshold $\epsilon > 0$, and let $\tilde{\mu}_\ell^{\min}$ and $\tilde{\mu}_\ell^{\max}$ $(\ell = 1, \ldots, n)$ be as in Corollary 1. In terms of Parikh vector membership queries, we have the following necessary condition for the occurrence of a Parikh vector in the string $\mathbf{s}$.

Corollary 2. *For any $p = (x_0, x_1)$ such that there exists a substring of* $\mathbf{s}$ *whose Parikh vector equals p, we have that*

$$\tilde{\mu}_{x_0+x_1}^{\min} \leq x_1 \leq \tilde{\mu}_{x_0+x_1}^{\max}.$$

We also have an "almost matching" sufficient condition, which also follows from Lemma 1 and Corollary 1.

Corollary 3. *Fix a Parikh vector* $p = (x_0, x_1)$. *If*

$$\frac{\tilde{\mu}^{\min}_{x_0+x_1}}{1-\epsilon} \leq x_1 \leq \frac{\tilde{\mu}^{\max}_{x_0+x_1}}{1+\epsilon}$$

then p *occurs in* **s**.

As a consequence, if we use the values $\tilde{\mu}^{\min}$ and $\tilde{\mu}^{\max}$ we can answer correctly to any membership query involving a Parikh vector which occurs in s. Moreover, we can also answer correctly any membership query involving a Parikh vector satisfying the condition in Corollary 3. In contrast, it might be that the approximate index makes us report false positives, when the membership query is about a Parikh vectors $p = (x_0, x_1)$ such that

$$\tilde{\mu}^{\min}_{x_0+x_1} \leq x_1 \leq \frac{\tilde{\mu}^{\min}_{x_0+x_1}}{1-\epsilon} \quad \text{or} \quad \frac{\tilde{\mu}^{\max}_{x_0+x_1}}{1+\epsilon} \leq x_1 \leq \tilde{\mu}^{\max}_{x_0+x_1}.$$

4 Some Final Observations and Open Problems

We presented a novel approach to approximating all maximum consecutive subsums of a sequence of non-negative integers in time $O(n^{1+\eta})$, for any constant $\eta > 0$. This can be directly used for obtaining a linear size index for binary string, which allows to answer in constant time Parikh vector membership queries. The existence of a $o(n^2)$ time solution for the exact case remains open. Some observations are in order regarding our approach.

A first observation is that we do not need to store $\tilde{m}_j$ for each $j = 1, \ldots, n$ as it is sufficient to store only the distinct values computed exhaustively. This brings down also the space complexity of our approximate index to $O(n^\epsilon)$.

The computation of the approximate index is extremely easy and fast to implement, and, in contrast to the previous *exact* solution present in the literature [9,8,17], it does not require any tabulation or convolution computation. In order to get the flavor of the quantities involved, notice that for $k = 30$ we can already guarantee an approximation ratio $\alpha = 1.085$ and with $k = 500$, we get $\alpha = 1.009$.

With respect to the Parikh vector indexing problem, one can clearly modify the algorithm in order to store together with the value $\tilde{m}_k$ also the position where the corresponding maximizing substring occurs. This way the algorithm can report a substring whose Parikh vector is an $(1+\epsilon)$-approximation of the desired Parikh vector. Moreover, this can also serve as a starting point for examining, in time proportional to ϵk the surrounding part of s in order to check whether the reporting position actually leads to an exact match. Along this line, an alternative is to associate to each $\tilde{m}_s$ both $m_{(j-1)\times g}$ and $m_{j\times g}$ and the corresponding positions.

It would be interesting to investigate whether it is possible to obtain Las Vegas algorithms for the Parikh vector problem based on our approximation perspective. Another direction for future investigation regards the extension of our approach to cover the case of Parikh vector membership queries in strings over non-binary alphabets.

References

1. Amir, A., Apostolico, A., Landau, G.M., Satta, G.: Efficient text fingerprinting via Parikh mapping. J. Discrete Algorithms 1(5-6), 409–421 (2003)
2. Babai, L., Felzenszwalb, P.F.: Computing rank-convolutions with a mask. ACM Trans. Algorithms 6(1), 1–13 (2009)
3. Benson, G.: Composition Alignment. In: Benson, G., Page, R.D.M. (eds.) WABI 2003. LNCS (LNBI), vol. 2812, pp. 447–461. Springer, Heidelberg (2003)
4. Bergkvist, A., Damaschke, P.: Fast algorithms for finding disjoint subsequences with extremal densities. Pattern Recognition 39, 2281–2292 (2006)
5. Böcker, S.: Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt. Journal of Computational Biology 11(6), 1110–1134 (2004)
6. Böcker, S., Lipták, Z.: A fast and simple algorithm for the Money Changing Problem. Algorithmica 48(4), 413–432 (2007)
7. Bremner, D., Chan, T.M., Demaine, E.D., Erickson, J., Hurtado, F., Iacono, J., Langerman, S., Taslakian, P.: Necklaces, Convolutions, and $X + Y$. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 160–171. Springer, Heidelberg (2006)
8. Burcsi, P., Cicalese, F., Fici, G., Lipták, Z.: On Approximate Jumbled Pattern Matching. Theory of Computing Systems 50(1), 35–51 (2012)
9. Burcsi, P., Cicalese, F., Fici, G., Lipták, Z.: On Table Arrangements, Scrabble Freaks, and Jumbled Pattern Matching. In: Boldi, P. (ed.) FUN 2010. LNCS, vol. 6099, pp. 89–101. Springer, Heidelberg (2010)
10. Butman, A., Eres, R., Landau, G.M.: Scaled and permuted string matching. Inf. Process. Lett. 92(6), 293–297 (2004)
11. Chan, T.M.: All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. Algorithmica 50(2), 236–243 (2008)
12. Chen, Y.H., Lu, H.I., Tang, C.Y.: Disjoint Segments with Maximum Density. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2005. LNCS, vol. 3515, pp. 845–850. Springer, Heidelberg (2005)
13. Cicalese, F., Fici, G., Lipták, Z.: Searching for jumbled patterns in strings. In: Proc. of the Prague Stringology Conference 2009 (PSC 2009), pp. 105–117 (2009)
14. Cieliebak, M., Erlebach, T., Lipták, Z., Stoye, J., Welzl, E.: Algorithmic complexity of protein identification: Combinatorics of weighted strings. Discrete Applied Mathematics 137(1), 27–46 (2004)
15. Eres, R., Landau, G.M., Parida, L.: Permutation pattern discovery in biosequences. Journal of Computational Biology 11(6), 1050–1060 (2004)
16. Jokinen, P., Tarhio, J., Ukkonen, E.: A comparison of approximate string matching algorithms. Software Practice and Experience 26(12), 1439–1458 (1996)
17. Moosa, T.M., Rahman, M.S.: Sub-quadratic time and linear size data structures for permutation matching in binary strings. J. Discrete Algorithms 10(1), 5–9 (2012)
18. Parida, L.: Gapped Permutation Patterns for Comparative Genomics. In: Bücher, P., Moret, B.M.E. (eds.) WABI 2006. LNCS (LNBI), vol. 4175, pp. 376–387. Springer, Heidelberg (2006)

The Complexity of String Partitioning

Anne Condon[1], Ján Maňuch[1,2], and Chris Thachuk[1]

[1] Dept. of Computer Science, University of British Columbia, Vancouver BC, Canada
{condon,jmanuch,cthachuk}@cs.ubc.ca

[2] Dept. of Mathematics, Simon Fraser University, Burnaby BC, Canada

Abstract. Given a string w over a finite alphabet Σ and an integer K, can w be partitioned into strings of length at most K, such that there are no *collisions*? We refer to this question as the *string partition* problem and show it is **NP**-complete for various definitions of collision and for a number of interesting restrictions including $|\Sigma| = 2$. This establishes the hardness of an important problem in contemporary synthetic biology, namely, oligo design for gene synthesis.

1 Introduction

Many problems in genomics have been solved by the application of elegant polynomial-time string algorithms, while others amount to solving known **NP**-complete problems; for instance, sequence assembly amounts to solving *shortest common superstring* [11], and genome rearrangement to *sorting strings by reversals and transpositions* [2]. The hardness of these problems has motivated extensive research into heuristic algorithms as well as polynomial-time algorithms for useful restrictions [6,10,19,9,8,14,16]. In a similar vein, we establish the hardness of the following fundamental question: can a string be partitioned into factors (*i.e.* substrings), of bounded length, such that no two *collide*? We refer to this as the *string partition* problem and study it under various restrictions and definitions of what it means for two factors to *collide*.

The study of string partitioning is motivated by an increasingly important problem arising in contemporary synthetic biology, namely gene synthesis. This technology is emerging as an important tool for a number of purposes including the determination of RNAi targeting specificity of a particular gene [12], design of novel proteins [5] and the construction of complete bacterial genomes [7]. There have been numerous studies utilizing synthetic genes to determine the potential of gene vaccines [13,3,17,1]. Despite the tremendous need for synthetic genes for both interrogative studies and for therapeutics, construction of genes, or any long DNA or RNA sequence, is not a trivial matter. Current technology can only produce short oligonucleotides (oligos) accurately. As such, a common approach is to design a set of oligos that could assemble into the desired sequence [18].

To understand the connection between string partitioning and gene synthesis, consider the following. A DNA *oligo*, or *strand* is a string over the four letter alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. The *reverse complement* F' of an oligo F is determined from F by replacing each A with a T and vice versa, each C with a G and vice versa, and reversing the resulting string. Two DNA oligos F and F' are said to *hybridize* if a sufficiently long factor of F is the reverse complement of a factor of F' (see Figure 1). A DNA *duplex* consists of a positive strand and its reverse complement, the negative strand. The *collision-aware oligo design for gene synthesis* (CA-ODGS) problem is to determine cut points in the positive and negative strands, which demarcate the oligos to be synthesized, such that the resulting design will successfully self-assemble. For the oligos to self-assemble correctly, they should 1) alternate between the positive and negative strands, with some overlap between successive oligos, and 2) only hybridize to the oligos they overlap with by design. Since there is variability in the length of the selected oligos, there are exponentially many designs.

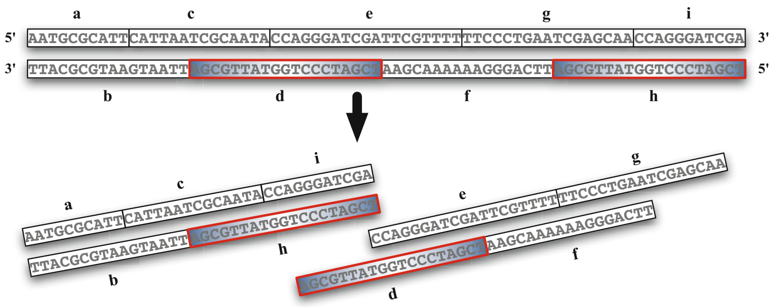


Fig. 1. An intended self-assembly (top) of a set of oligos for a desired DNA duplex. A foiled self-assembly (bottom) of the same oligos due to d and h being identical.

In previous work [4], the authors provided some evidence that the CA-ODGS problem may be hard by showing that partitioning a string into factors, of bounded length, such that no two are equal is **NP**-complete, even for strings over a quaternary alphabet. See Figure 1 for an example design that assembles incorrectly into two fragments, with the wrong ordering of oligos and therefore primary sequence, due to identical oligos. In this work, we study the underlying string partition problem in much greater detail. We show that partitioning strings such that no selected string is a copy/factor/prefix/suffix of another is **NP**-complete. We begin by showing that the more general problem of partitioning a set of strings is hard and then we show how those instances can be reduced to single string instances, for each respective definition of collision. See Figure 2 for an example of a single string instance (left) and set of strings instance (right). In all cases, we demonstrate the problems remain hard even when restricted to binary strings.

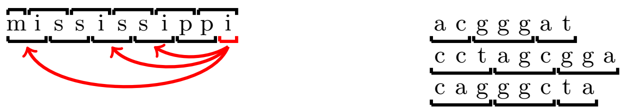


Fig. 2. (Left) Two partitions are shown for the string *mississippi*. The selected strings in both partitions have maximum length 2. The partition shown above the string is factor-free: no selected string is a factor of another; however, the partition shown below the string is not factor-free. (Right) A valid factor-free multiple string partition of a set of three strings into selected strings of maximum length 3.

2 Preliminaries

A *string* w is a sequence of letters over an alphabet Σ. Let $|w|$ denote the length of w, w^{R} a mirror image (reversal) of w, and let $(w)^i$ denote the string w repeated i times. The empty string is denoted as ε. String x is a *factor* of w if $w = \alpha x \beta$, for some (possibly empty) strings α and β. Similarly, x is a *prefix* (*suffix*) of w if $w = x\beta$ ($w = \alpha x$) for some (possibly empty) strings α and β. The prefix (suffix) of length k of w will be denoted as $\text{prefix}_k(w)$ ($\text{suffix}_k(w)$).

A *K-partition* of w is a sequence $P = p_1, p_2, \ldots, p_l$, for some l, where each p_i is a string over Σ of length at most K and $w = p_1 p_2 \ldots p_l$. We say that strings $p_1, \ldots, p_l$ are *selected* in the K-partition and that strings $p_i \ldots p_j$, $1 \le i \le j \le l$, are *super-selected*, with respect to the selected strings. We say P is *equality-free*, *prefix-free*, *suffix-free*, or *factor-free* if for all i, j, $1 \le i \ne j \le l$, neither p_i nor p_j is a copy, prefix, suffix, or factor, respectively, of the other. We say such partitions are *valid* (for the problem in question); otherwise, we say the partition contains a *collision*. We generalize the notion of a K-partition to a set of strings $\mathcal{W}$ to mean a K-partition for each string in $\mathcal{W}$. The length of $\mathcal{W}$ is the combined length of the strings in the set and will be denoted by $||\mathcal{W}||$. A K-partition for a set of strings is valid if no two elements in any, possibly different, partition collide. Finally, we will refer to the boundaries of a partition of string w as *cut points*, where the first cut point 0 and the last cut point $|w|$ are called trivial. For instance, the first partition of mississippi in Figure 2 has the following non-trivial cut points $1, 3, 5, 7$ and 9.

In what follows we will prove **NP**-completeness of various string partitioning problems by showing a polynomial reduction from an arbitrary instance of 3SAT(3), a problem shown to be **NP**-complete by Papadimitriou [15].

Problem 1 (3SAT(3)).
Instance: A formula ϕ with a set C of clauses over a set X of variables in conjunctive normal form such that:

1. every clause contains two or three literals,
2. each variable occurs in exactly three clauses, once negated and twice positive.

Question: Is ϕ satisfiable?

3 The String Partition Problems

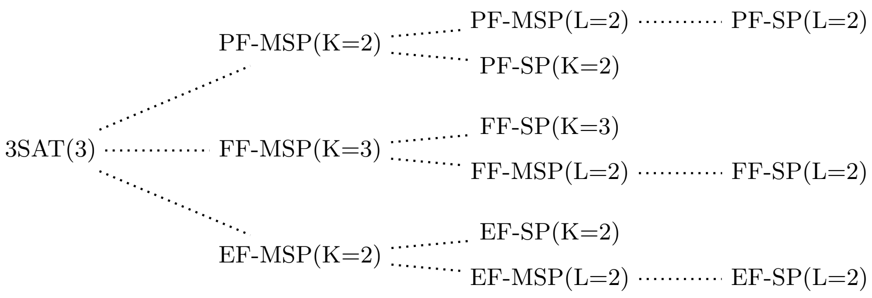


Fig. 3. Chain of reductions for different string partition variations from original 3SAT(3) problem. K is maximum selected string size and L is maximum alphabet size. Parameters are unbounded if not shown. EF, FF and PF are equality-free, factor-free, and prefix(suffix)-free, respectively.

For each $\mathcal{X}$ in $\{\text{equality}, \text{prefix}, \text{suffix}, \text{factor}\}$, we will consider two string partition problems.

Problem 2 ($\mathcal{X}$-Free Multiple String Partition ($\mathcal{X}$-MSP) Problem).
Instance: Finite alphabet Σ of size L, a positive integer K, and a set of strings $\mathcal{W}$ over Σ^*.
Question: Is there an $\mathcal{X}$-free, K-partition P of $\mathcal{W}$?

Problem 3 ($\mathcal{X}$-Free String Partition ($\mathcal{X}$-SP) Problem).
Instance: Finite alphabet Σ of size L, a positive integer K, and a string w over Σ^*.
Question: Is there an $\mathcal{X}$-free, K-partition P of w?

We will show **NP**-completeness of all these problems even when restricted to the constant size of the partition ($K = 2, 3$), or to the binary alphabet ($L = 2$). See Figure 3 showing the chain of reductions used to prove the complexity of the three variations and related restrictions of the problem.

4 Equality-Free String Partition Problems

4.1 Equality-Free Multiple String Partition with Unbounded Alphabet

We now describe a polynomial reduction from 3SAT(3) to EF-MSP with $K = 2$ and unbounded alphabet. Let ϕ be an instance of 3SAT(3), with set $C = \{c_1, \ldots, c_m\}$ of clauses, and set $X = x_1, \ldots, x_n$ of variables. We shall define an alphabet Σ and construct a set of strings $\mathcal{W}$ over Σ^*, such that $\mathcal{W}$ has a collision-free 2-partition if and only if ϕ is satisfiable. Let $|c_i|$ denote the number of literals contained in the clause c_i and let $c_i^1, \ldots, c_i^{|c_i|}$ be the literals of clause c_i.

We construct $\mathcal{W}$ to be a union of three types of strings: clause strings ($\mathcal{C}$), enforcer strings ($\mathcal{E}$) and forbidden strings ($\mathcal{F}$). First, for each clause of ϕ, we create a clause string C such that an equality-free 2-partition of $\mathcal{C}$ unambiguously selects exactly one literal from C. We refer to the selected strings corresponding to literals as *selected literals*. Intuitively, the selected literals of the clause strings are intended to be a satisfying truth assignment for the variables of ϕ. Second, for each variable we create an enforcer string to ensure that selected literals are *consistent*. Specifically, the enforcer strings ensure that a positive and a negative literal for the same variable cannot be simultaneously selected. Finally, we find it helpful to create so called forbidden strings that ensure certain strings cannot be selected in the clause and enforcer strings.

We construct an alphabet Σ, formally defined below, which includes a letter for each literal occurrence in the clauses, one letter for each variable, and the letters $\boxminus$ and $\boxplus$ used as delimiters.

$$\Sigma = \{\hat{x}_i;\ x_i \in X\} \cup \{\hat{c}_i^j;\ c_i \in C \wedge 1 \leq j \leq |c_i|\} \cup \{\boxminus, \boxplus\}$$

Note that $|\Sigma|$ is linear in the size of the 3SAT(3) problem ϕ (at most $n+3m+2$).

Construction of forbidden strings: To ensure that certain strings cannot be selected in $\mathcal{C}$ or $\mathcal{E}$, we will use the following set of forbidden strings $\mathcal{F} = \{\boxminus, \boxplus\}$.

Observation 1. *No string from the forbidden set $\mathcal{F}$ can be selected in $\mathcal{C}$ or $\mathcal{E}$.*

Construction of clause strings: For each clause $c_i \in C$, construct the *i-th clause string* to be $\hat{c}_i^1 \boxminus \hat{c}_i^2$ if $|c_i| = 2$, and $\hat{c}_i^1 \boxminus \hat{c}_i^2 \boxminus \hat{c}_i^3$ if $|c_i| = 3$.

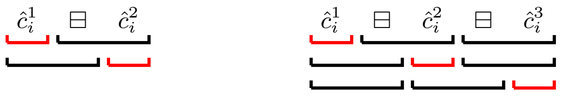

$\hat{c}_i^1$ $\boxminus$ $\hat{c}_i^2$
$\hat{c}_i^1$ $\boxminus$ $\hat{c}_i^2$ $\boxminus$ $\hat{c}_i^3$

Fig. 4. The 2-literal clause string (left) and 3-literal clause string (right) used in the reduction from 3SAT(3) to EF-MSP. Shown below each string are all valid 2-partitions. *Selected* literals of a partition are shown in red.

Lemma 1. *Given that no string from the forbidden set $\mathcal{F}$ is selected in $\mathcal{C}$, exactly one literal letter must be selected for each clause string in any equality-free 2-partition of $\mathcal{C}$.*

Proof. Consider the clause string for clause c_i. Whether c_i has two or three literals, the forbidden substring $\boxminus$ cannot be selected alone. Therefore, each $\boxminus$ must be selected with an adjacent literal letter. This leaves exactly one other literal letter which must be selected (see Figure 4). $\square$

Construction of enforcer strings: We must now ensure that no literal of ϕ that is selected in $\mathcal{C}$ is the negation of another selected literal. By definition of 3SAT(3), each variable appears exactly three times: twice positive and once negated. Let c_i^p and c_j^q be the two positive and c_k^r the negated occurrences of a variable x_v. Then construct the *enforcer string* for this variable as follows $\hat{c}_i^p \boxplus \hat{c}_k^r \hat{x}_v \hat{c}_k^r \hat{x}_v \hat{c}_k^r \boxplus \hat{c}_j^q$.

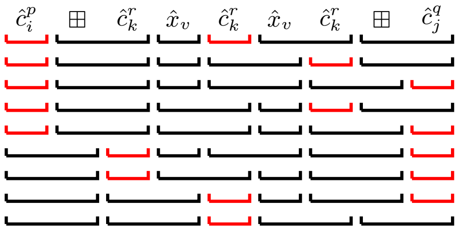


Fig. 5. All possible 2-partitions are shown for the enforcer string of a variable x_v having two positive literals c_i^p and c_j^q, and one negative literal c_k^r. In each partition, either $\hat{c}_k^r$ is selected or both $\hat{c}_i^p$ and $\hat{c}_j^q$ are which guarantees that letters for positive and negated literals of x_v cannot be simultaneously selected in $\mathcal{C}$.

Lemma 2. *Given that no string from the forbidden set $\mathcal{F}$ is selected in $\mathcal{C} \cup \mathcal{E}$, any equality-free 2-partition of $\mathcal{C} \cup \mathcal{E}$ must be consistent. In addition, for any consistent choice of selecting letters for literals in $\mathcal{C}$, there is an equality-free 2-partition of $\mathcal{C} \cup \mathcal{E} \cup \mathcal{F}$.*

Proof. Consider the enforcer string for variable x_v with positive literals $c_i^p = c_j^q = x_v$, and the negated literal $c_k^r = \neg x_v$. Figure 5 shows all 9 possible 2-partitions of the enforcer string (since $\boxplus$ is a forbidden string, each $\boxplus$ must be selected with an adjacent letter). It follows that in each of them either $\hat{c}_k^r$ is selected or both $\hat{c}_i^p$ and $\hat{c}_j^q$ are. In the first case, $\hat{c}_k^r$ cannot be selected in $\mathcal{C}$ and thus satisfied literals are chosen consistently for x_v. In the second case, letters for neither of the positive occurrences of x_v can be selected in $\mathcal{C}$.

To show the second part of the claim, observe that there is a 2-partition of the enforcer string compatible with any of four valid combinations of selecting letters for the corresponding literals in $\mathcal{C}$ (for example, by choosing the fifth or the last 2-partitions in Figure 5). Since enforcer strings share only one letter in common, namely, $\boxplus$, which is never selected in the enforcer strings, there are no collisions between 2-partitions of all enforcer strings. Furthermore, there are no collisions between strings selected in $\mathcal{C}$ and in $\mathcal{E}$: strings of length two selected in $\mathcal{C}$ contain the letter $\boxminus$, which does not appear in the enforcer strings; strings of length one are literals and the partitioning of enforcer strings was chosen in a way that literals (in $\mathcal{C}$) cannot be selected again in $\mathcal{E}$. □

This completes the reduction. Notice that the reduction is polynomial as the combined length of the constructed set of strings $\mathcal{W} = \mathcal{C} \cup \mathcal{E} \cup \mathcal{F}$ is at most $5m + 9n + 2$.

Theorem 1. *Equality-Free Multiple String Partition (EF-MSP) is* ***NP****-complete for $K = 2$.*

Proof. It is easy to see that EF-MSP Problem is in **NP**: a nondeterministic algorithm need only guess a partition P where $|p_i| \leq K$ for all p_i in P and check in polynomial time that no two strings in P are equal. Furthermore, it is clear that an arbitrary instance ϕ of 3SAT(3) can be reduced to an instance of EF-MSP, specified by a set of strings $\mathcal{W} = \mathcal{C} \cup \mathcal{E} \cup \mathcal{F}$, in polynomial time and space by the reduction detailed above.

Now suppose there is a satisfying truth assignment for ϕ. Simply select one corresponding true literal per clause in $\mathcal{C}$. The construction of clause strings guarantees that a 2-partition of the rest of each clause string is possible. Also, since a satisfying truth assignment for ϕ cannot assign truth values to opposite literals, then Lemma 2 guarantees that a valid partition of the enforcer strings is possible which does not conflict with the clause strings. Therefore, there exists an equality-free multiple string partition of $\mathcal{W}$.

Likewise, consider an equality-free multiple string partition of $\mathcal{W}$. Lemma 1 ensures that at least one literal per clause is selected. Furthermore, Lemma 2 guarantees that if there is no collision, then no two selected variables in the clauses are negations of each other. Therefore, this must correspond to a satisfying truth assignment for ϕ (if none of the three literals of a variable is selected in the partition of $\mathcal{C}$ then this variable can have arbitrary value in the truth assignment without affecting satisfiability of ϕ). □

4.2 Equality-Free String Partition with Unbounded Alphabet

Theorem 2. *Equality-Free String Partition (EF-SP) is* ***NP****-complete for $K = 2$.*

Proof. To show that EF-SP Problem for $K = 2$ is **NP**-complete, we will reduce EF-MSP Problem for $K = 2$ to it. Consider an arbitrary instance I of EF-MSP having a set of strings $\mathcal{W} = \{w_1, w_2, \ldots, w_\ell\}$ over alphabet Σ, and maximum partition size $K = 2$. We construct an instance $\bar{I}$ of EF-SP as follows. Let $\widehat{\Sigma} = \{\boxdot\} \cup \{d_i, \text{ for } 1 \leq i < \ell\}$, where $\widehat{\Sigma} \cap \Sigma = \emptyset$. Set the alphabet of $\bar{I}$ to $\bar{\Sigma} = \Sigma \cup \widehat{\Sigma}$ and the maximum partition size to $\bar{K} = 2$. Note that $|\bar{\Sigma}| = |\Sigma| + \ell$. Finally, construct the string

$$\bar{w} = \boxdot\ \boxdot\ \boxdot\ \boxdot\ \boxminus w_1 d_1 \boxdot\ \boxdot d_1 w_2 d_2 \boxdot\ \boxdot d_2 \ldots d_{\ell-1} \boxdot\ \boxdot d_{\ell-1} w_\ell\,.$$

The prefix of $\bar{w}$ of length five can be partitioned in two different ways each selecting $\boxdot$. Consequently, in any 2-partition of $\bar{w}$, remaining occurrences of $\boxdot$ must be selected together with an adjacent letter different from $\boxdot$, i.e., all strings $d_i\boxdot$ and $\boxdot d_i$ must be selected. Therefore, any 2-partition of $\bar{w}$ contains a 2-partition of $\mathcal{W}$ and the strings $\mathcal{D} = \{\boxdot, \boxdot\boxdot, \boxdot\boxminus, d_1\boxdot, \boxdot d_1, \ldots, d_{\ell-1}\boxdot, \boxdot d_{\ell-1}\}$. On the other hand, since all strings in $\mathcal{D}$ contain $\boxdot \notin \Sigma$, any 2-partition of $\bar{w}$ together with $\mathcal{D}$ forms a 2-partition of $\mathcal{W}$. It follows that there is a 2-partition of $\mathcal{W}$ if and only if there is a 2-partition of $\bar{w}$. The reduction is in polynomial time and space as $|\bar{w}| = ||\mathcal{W}|| + 4\ell + 1$. □

4.3 Equality-Free Multiple String Partition with Binary Alphabet

Theorem 3. *The EF-MSP with maximum partition size $K = 2$ can be polynomially reduced to the EF-MSP Problem with the alphabet size $L = 2$. Consequently, the EF-MSP is* ***NP****-complete for binary alphabet. In addition, this reduction satisfies the following property: for any set C containing n distinct strings of length δ, where n is the size of the alphabet of the EF-MSP with maximum partition size $K = 2$ and $\delta \geq \log_2 n$, every selected word in a valid partition (if it exists) of the EF-MSP with the binary alphabet is a prefix of a string in C^2, and its maximum partition size is $\bar{K} = 2\delta$.*

Proof. We will show a reduction from the EF-MSP with maximum partition size $K = 2$. Consider an arbitrary instance I of EF-MSP having a set of strings $\mathcal{W} = \{w_1, w_2, \ldots, w_\ell\}$ over alphabet $\Sigma = \{a_1, \ldots, a_n\}$, and maximum partition size $K = 2$. We will construct an instance $\bar{I}$ of EF-MSP over binary alphabet $\bar{\Sigma} = \{0, 1\}$. Let δ be any number greater or equal to $\log_2 n$. Let $C = \{c_1, \ldots, c_n\}$ be a set of any distinct binary codewords of length δ. We set $\bar{K}$ to 2δ. Let h be a homomorphism from Σ to C such that $h(a_i) = c_i$, for every $i = 1, \ldots, n$. The set of strings of $\bar{I}$ will contain $h(\mathcal{W})$, i.e., the original strings in $\mathcal{W}$ mapped by h to the binary alphabet $\bar{\Sigma}$. However, we need to guarantee that the partition of strings in $h(\mathcal{W})$ does not contain fragments of codewords. For this reason, we also add to $\bar{\mathcal{W}}$ the following strings:

$$\widehat{\mathcal{W}} = \{\text{prefix}_i(c);\ c \in C, i = 1, \ldots, \delta - 1\} \cup \{\text{prefix}_i(cd);\ c, d \in C, i = \delta + 1, \ldots, 2\delta - 1\}$$

We set $\bar{\mathcal{W}} = h(\mathcal{W}) \cup \widehat{\mathcal{W}}$.

First, consider a valid 2-partition P of $\mathcal{W}$. We construct a $\bar{K}$-partition $\bar{P}$ of $\bar{\mathcal{W}}$ as follows. For each string s selected in P, we select the corresponding $h(s)$ in $\bar{P}$. For each string $t \in \widehat{\mathcal{W}}$, we select t entirely. Note that strings selected from $h(\mathcal{W})$ have length either δ or 2δ, while strings selected from $\widehat{\mathcal{W}}$ have lengths different from δ and 2δ. Therefore, there cannot be any collisions between these two groups of selected strings. Furthermore, there are no collisions in the first group, since there were no collisions in P. Obviously, there are no collisions in the second group of selected strings. It follows that $\bar{P}$ is a valid $\bar{K}$-partition of $\bar{\mathcal{W}}$.

Conversely, consider a valid $\bar{K}$-partition $\bar{P}$ of $\bar{\mathcal{W}}$. First, we will show that all strings in $\widehat{\mathcal{W}}$ are selected without non-trivial cut points. We will prove that by induction on the length i of strings. The base case, $i = 1$, is trivially true, as one-letter strings cannot be partitioned into shorter strings. Now, assume the claim is true for all strings in $\widehat{\mathcal{W}}$ of lengths smaller than $i < 2\delta$ and different from δ. Consider a word $u \in \widehat{\mathcal{W}}$ of length i. Assume that u is partitioned into strings $u_1, \ldots, u_t$, where $t \geq 2$. Note that the length of u_1 is smaller than i. If the length of u_1 is different from δ, we have a collision, as $u_1 \in \widehat{\mathcal{W}}$ and by the induction hypothesis, it was selected without non-trivial cut points. Assume that the length of u_1 is δ. Then u_2 is a prefix of a codeword of length smaller than

$\min\{\delta, i\}$, and we have a collision again as in the previous case. It follows that $t = 1$, i.e., u is selected without non-trivial cut points in $\bar{P}$. Second, we show that all strings selected in the partition of strings in $h(\mathcal{W})$ have lengths either δ or 2δ. Assume that this is not the case for some string $s \in h(\mathcal{W})$. Note that $s = c_{i_1}c_{i_2}\dots c_{i_p}$, for some indices $i_1, \dots, i_p$. Let $s = s_1 \dots s_q$ be the partition of s and let j be the smallest j such that the length of s_j is not δ or 2δ. Then $s_1 \dots s_{j-1} = c_{i_1} \dots c_{i_r}$, for some $r < p$. Consequently, s_j is a prefix of $c_{i_{r+1}}c_{i_{r+2}}$, i.e., $s_j \in \widehat{\mathcal{W}}$, and we have a collision, since s_j was already selected in partition of $\widehat{\mathcal{W}}$. Hence, each string in $h(\mathcal{W})$ is partitioned into strings of lengths either δ or 2δ, which can be easily mapped to a valid 2-partition of $\mathcal{W}$.

It follows that there is a 2-partition of $\mathcal{W}$ if and only if there is $\bar{K}$-partition of $\bar{\mathcal{W}}$ and that the reduction satisfies the property described in the claim.

Finally, let us check that the reduction is polynomial. The size of $h(\mathcal{W})$ is $|\mathcal{W}|$ and the length of $h(\mathcal{W})$ is $\delta||\mathcal{W}||$. The size of $\widehat{\mathcal{W}}$—the set of all unique prefixes for codewords of length less than δ, and all unique prefixes of pairs of adjacent codewords with length greater than δ and less than 2δ—is at most $(n^2 + n)(\delta - 1)$ as there are n codewords in total. Therefore, the length of $\widehat{\mathcal{W}}$ is at most $n \cdot (1 + \dots + \delta - 1) + n^2 \cdot (\delta + 1 + \delta + 2 + \dots + 2\delta - 1) = (3n^2 + n)(\delta - 1)\delta/2$. Since δ can be chosen to be $\Theta(\log n)$, the size of $\bar{\mathcal{W}}$ is polynomial in the size of $\mathcal{W}$ and the size of the original alphabet Σ. □

4.4 Equality-Free String Partition with Binary Alphabet

Theorem 4. *Equality-Free String Partition (EF-SP) Problem is* ***NP****-complete for binary alphabet ($L = 2$).*

Proof. We will show a reduction from the EF-MSP Problem with the binary alphabet ($L = 2$) satisfying properties listed in Theorem 3. Consider an instance I of EF-MSP having a set of strings $\mathcal{W} = \{w_1, w_2, \dots, w_\ell\}$ over alphabet $\Sigma = \{0, 1\}$, and maximum partition size $K = 2\delta$ such that all selected words in any valid K-partition are prefixes of the elements of a set C^2, where C contains n distinct strings of length δ each starting with 0, $\ell \leq (n^2 + n)(\delta - 1)$, and $\delta \geq \max(9, 3\log_2(n + 1))$. By Theorem 3, this instance can be polynomially reduced to an instance of the EF-MSP with maximum partition size $K = 2$. We will construct an instance $\bar{I}$ of EF-SP over binary alphabet $\bar{\Sigma} = \{0, 1\}$ with the same partition size $K = 2\delta$. We will show that the size of $\bar{I}$ is polynomial in the size of I, and hence, it will follow by Theorems 1 and 3, that the EF-SP Problem is **NP**-complete.

To construct the string $\bar{w}$ we will interleave strings $w_1, \dots, w_\ell$ with delimiters $d_1, \dots, d_{\ell-1}$ defined in a moment as follows:

$$\bar{w} = w_1 d_1 w_2 d_2 w_3 \dots d_{\ell-1} w_\ell \, .$$

To define the delimiter strings, we will need the following functions. Let bin : $\mathbb{N} \to \{0, 1\}^*$ be a function mapping a positive integer to its standard binary representation without the leading one. For example bin$(1) = \varepsilon$, bin$(2) = 0$ and

$\mathrm{bin}(10) = 010$. Next, the functions $\mathrm{pad}_i : \{0,1\}^* \to \{0,1\}^*$ will pad a given string with $i-1$ ones and one zero on the left, i.e., $\mathrm{pad}_i(s) = (1)^{i-1}0s$. We will refer to strings returned by this functions as *padded strings*. The function $\mathrm{chain} : \{0,1\}^* \to \{0,1\}^*$ maps a string s with i trailing zeros, i.e., $s = s'(0)^i$, where s' is either the empty string or a string ending with 1, to the following concatenation of padded strings and mirror images (reversals) of padded strings:

$$\mathrm{chain}(s) = \mathrm{pad}^{\mathrm{R}}_{K-|s|}(s)\,\mathrm{pad}_{K-|s|}(s')\,\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s')\,\mathrm{pad}_{K-|s|}(s'0)\,\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s'0)$$
$$\dots\mathrm{pad}_{K-|s|}(s'(0)^{i-1})\,\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s'(0)^{i-1})\,\mathrm{pad}_{K-|s|}(s)\,.$$

Finally, we set the delimiter d_j to $\mathrm{chain}(\mathrm{bin}(j))$, for every $j > 1$. For $j = 1$, we set d_1 to $0(1)^{K-1}(1)^{K(K-1)/2}(1)^{K-1}0$. To illustrate this definition, let us list the first five delimiter strings:

$$\begin{aligned}
d_1 &= 0(1)^{K-1}(1)^{K(K-1)/2}(1)^{K-1}0\\
d_2 &= \mathrm{chain}(0) = 00(1)^{K-2}(1)^{K-2}00(1)^{K-2}(1)^{K-2}00\\
d_3 &= \mathrm{chain}(1) = 10(1)^{K-2}(1)^{K-2}01\\
d_4 &= \mathrm{chain}(00) = 000(1)^{K-3}(1)^{K-3}00(1)^{K-3}(1)^{K-3}0000(1)^{K-3}(1)^{K-3}000\\
d_5 &= \mathrm{chain}(01) = 100(1)^{K-3}(1)^{K-3}001
\end{aligned}$$

Now, consider a valid K-partition P of $\mathcal{W}$. We construct a K-partition $\bar{P}$ of $\bar{w}$ as follows. Each substring w_j is partitioned in the same way as in P. Each delimiter d_j, where $j > 1$, is partitioned to its padded strings and mirror images of padded strings. In addition, the delimiter d_1 is partitioned into one mirror image of a padded string, strings $(1), (1)^2, \dots, (1)^K$ in any order, and one padded string. Note that all strings selected in w_j's are prefixes of C^2, and since each $c \in C$ has length $\delta = K/2$ and starts with 0, all these selected strings start with 0 and the longest run of 1 they contain has length at most $\delta - 1$. Hence, they cannot collide with strings $(1), (1)^2, \dots, (1)^K$ and with padded strings which all start with 1. To show they do not collide with mirror images of padded strings, we will show that each padded string (or its mirror image) contains a run of at least δ ones. By the definition of functions pad_i, each padded string or its mirror image selected in a delimiter d_j contains a substring $(1)^{K-|\mathrm{bin}(d_j)|-1}$, i.e., a run of $K - (\lceil\log_2 j\rceil - 1) - 1 = K - \lceil\log_2 j\rceil$ ones. Since $j < \ell \le (n^2+n)(\delta-1)$, it is enough to show that $\log_2[(n^2+n)(\delta-1)] \le \delta$. This follows from the fact that $\delta \ge 2\log_2(n+1) + \delta/3$ and $\delta/3 \ge \log_2(\delta-1)$ for $\delta \ge 9$. Finally, we need to show that all selected padded strings and their mirror images are distinct. Note that each selected padded string starts with at least δ ones and contains at least one zero, hence, it cannot be equal to a selected mirror image of padded string. Hence, it is enough to show that two delimiter d_j and $d_{j'}$, where $j, j' < \ell$ do not contain the same padded string or its mirror image. Without loss of generality, let us only consider the padded strings. If $\mathrm{bin}(j)$ and $\mathrm{bin}(j')$ have different lengths then the padded strings of d_j and $d_{j'}$ start with $(1)^{K-|\mathrm{bin}(j)|-1}0$ and $(1)^{K-|\mathrm{bin}(j')|-1}0$, hence they cannot be equal. Therefore, assume they have the same length. Let s (respectively, s') be the prefix of $\mathrm{bin}(j)$ (respectively,

$\mathrm{bin}(j')$) without the trailing zeros. Clearly, $s \neq s'$. Now, the padded strings from d_j and $d_{j'}$ are same only if $s0^i = s'0^{i'}$ for some i and i'. However, since both s and s' end with one or one of them is the empty string, we must have $i = i'$, and hence also $s = s'$, a contradiction. Since the K-partition P of $\mathcal{W}$ was valid, it follows that the K-partition of $\bar{w}$ is also valid.

Conversely, consider a valid K-partition $\bar{P}$ of $\bar{w}$. It is enough to show that $\bar{P}$ super-selects each delimiter in $\bar{w}$. We will show by induction on j that delimiters $d_1, \ldots, d_j$ are super-selected and furthermore, that each of these delimiters is partitioned into its padded strings and mirror images of padded strings. For the base case $j = 1$, it is easy to see that $\bar{P}$ must select string $0(1)^{K-1}$, then strings $(1)^1, \ldots, (1)^K$ in any order and string $(1)^{K-1}0$, and thus d_1 is super-selected in $\bar{P}$ and its padded string and its mirror image of a padded string are selected. Next, assume that the induction hypothesis is satisfied for delimiters $d_1, \ldots, d_{j-1}$. Consider delimiter string d_j. First, we will show that d_j contains cut points in $\bar{P}$ shown by $\cdot$'s below:

$$\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s) \cdot \mathrm{pad}_{K-|s|}(s')\,\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s') \cdot \mathrm{pad}_{K-|s|}(s'0)\,\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s'0)\cdot \\ \ldots \cdot \mathrm{pad}_{K-|s|}(s'(0)^{i-1})\,\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s'(0)^{i-1}) \cdot \mathrm{pad}_{K-|s|}(s)\,,$$

where $s = \mathrm{bin}(j)$ and s' is the prefix of s without the trailing zeros and i is the number of trailing zeros. Note that each letter "$\cdot$" is preceded and followed by $K - |\mathrm{bin}(j)| - 1$ ones. Since $|\mathrm{bin}(j)| \leq \delta - 1$, we have a run of at least $K = 2\delta$ ones, thus this run must contain a cut point. By contradiction assume that there is a cut point before the letter "$\cdot$" in this run of ones. Then the selected string starting at this cut point is in the form $(1)^{K-i-1}0u$, where $i < |\mathrm{bin}(j)|$ and $|u| \leq i$. Note that u might be the empty string and the selected string must contain the zero preceding u since all strings consisting only of ones are already selected in d_1. Let $v = u(0)^{i-|u|}$. Since $|v| = i < |\mathrm{bin}(j)|$, we have $v = \mathrm{bin}(j')$, where $j' < j$. The delimiter string $d_{j'}$ contains $\mathrm{pad}_{K-i}(u) = (1)^{K-i-1}0u$, which by the induction hypothesis has been already selected. Analogously, we arrive into a contradiction, if there is a cut point after "$\cdot$" in the run of ones surrounding the letter "$\cdot$". It follows that there is a cut point at each letter "$\cdot$" above in $\bar{P}$.

Next, we show that each of super-selected strings of d_j:

$$\mathrm{pad}_{K-|s|}(s')\,\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s'), \ldots, \mathrm{pad}_{K-|s|}(s'(0)^{i-1})\,\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s'(0)^{i-1})\,,$$

has a cut point exactly in the middle. The length of each padded string or of its mirror image is at least $K - |s|$ and since $|\mathrm{bin}(j)| \leq \delta - 1$, this length is at least $\delta + 1$. Hence, there has to be at least one cut point in each of the above super-selected strings in $\bar{P}$. We will first prove the claim for the first super-selected string $\mathrm{pad}_{K-|s|}(s')\,\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s')$. By contradiction, and without loss of generality, assume that there is a cut point inside $\mathrm{pad}_{K-|s|}(s') = (1)^{K-|s|-1}0s'$. Thus a string in the form $(1)^{K-|s|-1}0u$, where u is a proper prefix of s', is selected in $\bar{P}$. Consider string $v = u(0)^{|s|-|u|}$. Obviously, $|v| = |s|$ and v is lexicographically smaller than s, and thus $\mathrm{bin}(j') = v$ for some $j' < j$. By the induction hypothesis, string $\mathrm{pad}_{K-|v|}(u) = (1)^{K-|s|-1}0u$ has been already selected in $d_{j'}$, a contradiction. It follows by straightforward induction on i that the remaining super-selected strings are partitioned exactly in the middle. Finally, observe that if there is a cut point inside $\mathrm{pad}_{K-|s|}(s)$ then either one the padded strings of $d_{j'}$ or one of the padded strings of d_j described above is selected again . Similarly, there cannot be any cut point inside $\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s)$. Since the length of these two strings is exactly K, there has to be a cut point just after $\mathrm{pad}_{K-|s|}(s)$ and just before $\mathrm{pad}^{\mathrm{R}}_{K-|s|}(s)$, i.e., d_j is super-selected. This completes the induction proof, and we have that all delimiter strings in $\bar{w}$ are super-selected by $\bar{P}$, and thus $\bar{P}$ gives us also a partition of the set $\mathcal{W}$.

It follows that there is a K-partition of $\mathcal{W}$ if and only if there is K-partition of $\bar{w}$. Finally, let us check that the reduction is polynomial. The length of each padded string or its mirror image is at most K. The length of d_1 is $K(K+3)/2 < K^2$. String $\mathrm{bin}(j)$ for $1 < j < \ell$ has length at most $\delta - 1$, and hence each d_j contains at most $2\delta = K$ padded strings and mirror images of padded strings. Hence, $|d_j| \leq K^2$. Thus, the total length of $\bar{w}$ is at most $||\mathcal{W}|| + \ell K^2$. □

5 Factor-, Prefix- and Suffix-Free String Partition Problems

Here, we summarize the results for these partition problems. Due to space limitations, their proof can be found in the full version of this paper.

Theorem 5. *Both Factor-Free Multiple String Partition (FF-MSP) and Factor-Free String Partition (FF-SP) are* ***NP****-complete in the following two cases: (a) when the maximum partition size is* 3*; and (b) when the alphabet is binary.*

Theorem 6. *Both Prefix(Suffix)-Free Multiple String Partition (PF-MSP) and Prefix(Suffix)-Free String Partition (PF-SP) are* ***NP****-complete in the following two cases: (a) when the maximum partition size is* 2*; and (b) when the alphabet is binary.*

6 Conclusion

We have established the complexity of the following fundamental question: given a string w over an alphabet Σ and an integer K, can w be partitioned into factors no longer than K such that no two *collide*? We have shown this problem is **NP**-complete for versions requiring that no string in the partition is a copy/factor/prefix/suffix of another. Furthermore, we have shown the problems remain hard even for binary strings. This resolves a number of open questions from previous work [4] and establishes the theoretical hardness of a practical problem in contemporary synthetic biology, specifically, the oligo design for gene synthesis problem.

References

1. Chlichlia, K., Schirrmacher, V., Sandaltzopoulos, R.: Cancer immunotherapy: Battling tumors with gene vaccines. Current Medicinal Chemistry - Anti-Inflammatory & Anti-Allergy Agents 4, 353–365 (2005)
2. Christie, D.A., Irving, R.W.: Sorting strings by reversals and by transpositions. SIAM Journal on Discrete Mathematics 14(2), 193–206 (2001)
3. Cid-Arregui, A., Juarez, V., Hausen, H.Z.: A Synthetic E7 Gene of Human Papillomavirus Type 16 That Yields Enhanced Expression of the Protein in Mammalian Cells and Is Useful for DNA Immunization Studies. Journal of Virology 77(8), 4928–4937 (2003)
4. Condon, A., Maňuch, J., Thachuk, C.: Complexity of a Collision-Aware String Partition Problem and Its Relation to Oligo Design for Gene Synthesis. In: Hu, X., Wang, J. (eds.) COCOON 2008. LNCS, vol. 5092, pp. 265–275. Springer, Heidelberg (2008)
5. Cox, J.C., Lape, J., Sayed, M.A., Hellinga, H.W.: Protein fabrication automation. Protein Science 16(3), 379–390 (2007)
6. Eriksen, N.: (1+ ε)-Approximation of sorting by reversals and transpositions. Theoretical Computer Science 289(1), 517–529 (2002)
7. Gibson, D., Benders, G., Andrews-Pfannkoch, C., Denisova, E., Baden-Tillson, H., Zaveri, J., Stockwell, T., Brownley, A., Thomas, D., Algire, M., et al.: Complete Chemical Synthesis, Assembly, and Cloning of a Mycoplasma genitalium Genome. Science 319(5867), 1215–1220 (2008)
8. Goldstein, A., Kolman, P., Zheng, J.: Minimum common string partition problem: Hardness and approximations. The Electronic Journal of Combinatorics 12(R50), 1 (2005)
9. Hannenhalli, S.: Polynomial algorithm for computing translocation distance between genomes. Discrete Applied Mathematics 71(1), 137–151 (1996)
10. Hartman, T.: A Simpler 1.5-Approximation Algorithm for Sorting by Transpositions. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 156–169. Springer, Heidelberg (2003)
11. Karp, R.M.: Mapping the genome: some combinatorial problems arising in molecular biology. In: STOC 1993: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, pp. 278–285. ACM, New York (1993)
12. Kumar, D., Gustafsson, C., Klessig, D.F.: Validation of RNAi silencing specificity using synthetic genes: salicylic acid-binding protein 2 is required for innate immunity in plants. Plant J. 45(5), 863–868 (2006)
13. Lin, C.T., Tsai, Y.C., He, L., Calizo, R., Chou, H.H., Chang, T.C., Soong, Y.K., Hung, C.F., Lai, C.H.: A DNA vaccine encoding a codon-optimized human papillomavirus type 16 E6 gene enhances CTL response and anti-tumor activity. J. Biomed. Sci. 13(4), 481–488 (2006)
14. Myers, E., Sutton, G., Delcher, A., Dew, I., Fasulo, D., Flanigan, M., Kravitz, S., Mobarry, C., Reinert, K., Remington, K., et al.: A whole-genome assembly of Drosophila. Science 287(5461), 2196 (2000)
15. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1994)
16. Pevzner, P., Tang, H., Waterman, M.: An Eulerian path approach to DNA fragment assembly. Proceedings of the National Academy of Sciences of the United States of America 98(17), 9748 (2001)

17. Roden, R., Wu, T.: Preventative and therapeutic vaccines for cervical cancer. Expert Review of Vaccines 2(4), 495–516 (2003)
18. Stemmer, W.P., Crameri, A., Ha, K.D., Brennan, T.M., Heyneker, H.L.: Single-step assembly of a gene and entire plasmid from large numbers of oligodeoxyribonucleotides. Gene 164(1), 49–53 (1995)
19. Yancopoulos, S., Attie, O., Friedberg, R.: Efficient sorting of genomic permutations by translocation, inversion and block interchange. Bioinformatics 21(16), 3340–3346 (2005)

Towards an Optimal Space-and-Query-Time Index for Top-k Document Retrieval*

Wing-Kai Hon[1], Rahul Shah[2], and Sharma V. Thankachan[2]

[1] Department of CS, National Tsing Hua University, Taiwan
`wkhon@cs.nthu.edu.tw`

[2] Department of CS, Louisiana State University, USA
`{rahul,thanks}@csc.lsu.edu`

Abstract. Let $\mathcal{D} = \{d_1, d_2, ...d_D\}$ be a given set of D string documents of total length n, our task is to index $\mathcal{D}$, such that the k most relevant documents for an online query pattern P of length p can be retrieved efficiently. We propose an index of size $|CSA| + n \log D(2 + o(1))$ bits and $O(t_s(p) + k \log\log n + poly \log\log n)$ query time for the basic relevance metric *term-frequency*, where $|CSA|$ is the size (in bits) of a compressed full text index of $\mathcal{D}$, with $O(t_s(p))$ time for searching a pattern of length p. We further reduce the space to $|CSA| + n \log D(1 + o(1))$ bits, however the query time will be $O(t_s(p) + k(\log \sigma \log\log n)^{1+\epsilon} + poly \log\log n)$, where σ is the alphabet size and $\epsilon > 0$ is any constant.

1 Introduction and Related Work

Document retrieval is a special type of pattern matching that is closely related to information retrieval and web searching. In this problem, the data consists of a collection of text documents, and given a query pattern P, we are required to report all the documents in which this pattern occurs (not all the occurrences). In addition, the notion of *relevance* is commonly applied to rank all the documents that satisfy the query, and only those documents with the highest relevance are returned. Such a concept of relevance has been central in the effectiveness and usability of present day search engines like Google, Bing, Yahoo, or Ask. When relevance is considered, the query has an additional input parameter k, and the task is to report the k documents with the highest relevance to the query pattern (in the decreasing order of relevance), instead of finding all the documents that contain the query pattern (as there may be too many). More formally, let $\mathcal{D} = \{d_1, d_2, ...d_D\}$ denote a given set of D string documents to be indexed, whose total lengths is n, and let P denote a query pattern of length p. Let *occ* be the number of occurrences of this pattern over the entire collection $\mathcal{D}$, and *ndoc* be the number of documents out of D in which the pattern P appears. One of the main issues is the fact that $k \ll ndoc \ll occ$. Thus, it is important to design indexes which do not have to go through all the occurrences or even all the documents in order to answer a query.

* This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W. Hon) and US NSF Grant CCF–1017623 (R. Shah).

The research in string document retrieval was introduced by Matias et al. [20], and Muthukrishnan [22] formalized it with the introduction of relevance metrics like *term-frequency (tf)* and *min-dist*[1], and proposed indexes with efficient query performance. Since then, this has been an active research area [29,30]. The top-k document retrieval problem was introduced in [12], where an $O(n \log n)$-word index is proposed with $O(p + k + \log n \log \log n)$ query time for the case when the relevance metric is *term-frequency*. A recent flurry of activities in this area [26,16,8,2,4,27,24,18,13,25] came with Hon et al.'s work [15] where they gave a linear-space index with $O(p + k \log k)$ query time, which works for a wide class of relevance metrics. The recent structure by Navarro and Nekrich [23] achieves optimal $O(p + k)$ query time using $O(n(\log \sigma + \log D + \log \log n))$ bits, which improves the results in [15] in both space and time. If the relevance metric is *term-frequency*, their index space can be further improved to $O(n(\log \sigma + \log D))$ bits. All these interesting results have contributed towards the goal of achieving an optimal query time index. However, the space is far from optimal, moreover the constants hidden in the space bound can restrict the use of these indexes in practice. On the other side, the succinct index proposed by Hon et al. [15] takes about $O(\log^4 n)$ time to report each document, which is likely to be impractical. This time bound has been further improved by [2,8], but still polylog(n) time is required per reported document. Another line of work is to derive indexes using about $n \log D$ bits additional space, and the best known index takes a per document report time of $O(\log k \log^{1+\epsilon} n)$ [2]. Efficient practical indexes are also known [4], but their query algorithms are heuristics with no worst-case bound. In this paper, we introduce two space efficient indexes with per document report time *poly*-log-logarithmic in n. The main results are summarized as follows.

Theorem 1. *There exists an index of size* $|CSA| + n \log D(2 + o(1))$ *bits with a query time of* $O(t_s(p) + k \log \log n + poly \log \log n)$ *for retrieving top-k documents with the highest term frequencies, where* $|CSA|$ *is the size (in bits) of a compressed full text index of* $\mathcal{D}$ *with* $O(t_s(p))$ *time for searching a pattern of length* p.

Theorem 2. *There exists an index of size* $|CSA| + n \log D(1 + o(1))$ *bits with a query time of* $O(t_s(p) + k(\log \sigma \log \log n)^{1+\epsilon} + poly \log \log n)$ *for retrieving top-k documents with the highest term frequencies, where* $|CSA|$ *is the size (in bits) of a compressed full text index of* $\mathcal{D}$ *with* $O(t_s(p))$ *time for searching a pattern of length* p*,* σ *is the alphabet size and* $\epsilon > 0$ *is a constant.*

Table 1 gives a summary of the major results in the top-k frequent document retrieval problem. The time complexities are simplified by assuming that we are using the full text index proposed by Belazzougui and Navarro, of size $|CSA| = nH_h + O(n) + o(n \log \sigma)$ bits and $t_s(p) = O(p)$, where H_h is the hth order empirical entropy of $\mathcal{D}$ [1]. We also assume $D < n^{\varepsilon}$ for some $\varepsilon < 1$ and $\epsilon > 0$ is any constant.

[1] $tf(P, d)$ is the number of occurrences of P in d and *min-dist*(P, d) is the minimum distance between two occurrences of P in d.

Table 1. Indexes for Top-k Frequent Document Retrieval

Source	**Index Space (in bits)**	**Time per reported document**
[12]	$O(n \log n + n \log^2 D)$	$O(1)$
[15]	$O(n \log n)$	$O(\log k)$
[4]	$\|CSA\| + n \log D(1 + o(1))$	Unbounded
[15]	$2\|CSA\| + o(n)$	$O(\log^{4+\epsilon} n)$
[2]	$2\|CSA\| + o(n)$	$O(\log k \log^{2+\epsilon} n)$
[8]	$\|CSA\| + O(\frac{n \log D}{\log \log D})$	$O(\log^{3+\epsilon} n)$
[2]	$\|CSA\| + O(\frac{n \log D}{\log \log D})$	$O(\log k \log^{2+\epsilon} n)$
[2]	$\|CSA\| + O(n \log \log \log D)$	$O(\log k \log^{2+\epsilon} n)$
[23]	$O(n \log \sigma + n \log D)$	$O(1)$
[8]	$\|CSA\| + n \log D + o(n)$	$O(\log^{2+\epsilon} n)$
[2]	$\|CSA\| + n \log D + o(n)$	$O(\log k \log^{1+\epsilon} n)$
Ours	$\|CSA\| + 2n \log D(1 + o(1))$	$O(\log \log n)$
Ours	$\|CSA\| + n \log D(1 + o(1))$	$O((\log \sigma \log \log n)^{1+\epsilon})$

2 Preliminaries

2.1 Top-k Using Range Maximum/Minimum Queries

One of the main tools in top-k retrieval is the *range maximum/minimum query structures* (RMQ) [6]. We summarize the results in the following lemmas (We defer the proofs to the full version [14]).

Lemma 1. *Let $A[1...n]$ be an array of n numbers. We can preprocess A in linear time and associate A with a $2n + o(n)$ bits RMQ data structure such that given a set of t non-overlapping ranges $[L_1, R_1], [L_2, R_2], \ldots, [L_t, R_t]$, we can find the largest (or smallest) k numbers in $A[L_1..R_1] \cup A[L_2..R_2] \cup \cdots \cup A[L_t..R_t]$ in unsorted order in $O(t + k)$ time.*

Lemma 2. *Let $A[1...n]$ be an array of n integers taken from the set $[1, \pi]$, and each number $A[i]$ is associated with a score (which may be stored separately and can be computed in t_{score} time). Then the array A can be maintained in $O(n \log \pi)$ bits, such that given two ranges $[x', x'']$, $[y', y'']$, and a parameter k, we can search among those entries $A[i]$ with $x' \leq i \leq x''$ and $y' \leq A[i] \leq y''$, and report the k highest scoring entries in unsorted order in $O((\log \pi + k)(\log \pi + t_{score}))$ time.*

3 A Brief Review of Hon et al.'s Index

In this section we give a brief description of Hon et al.'s index [15]. Let $T = d_1\#d_2\# \cdots \#d_D\#$ be a text obtained by concatenating all the documents in $\mathcal{D}$, separated by a special symbol # not appearing elsewhere inside any of the d_is.

Then the suffix tree [31,21,19] of T is called the *generalized suffix tree* GST of $\mathcal{D}$. Then any given substring $T[a...b]$ (which does not contain #) of T is a substring of some document $d_x \in \mathcal{D}$, and the value of x can be computed in $O(1)$ time by maintaining an $(n+D)(1+o(1))$-bit auxiliary data structure[2]. Each edge in GST is labeled by a character string and for any node u, the *path label* of u, denoted by $path(u)$ is the string formed by concatenating the edge labels from root to u. Note that the path label of the ith leftmost leaf in GST is exactly the ith lexicographically smallest suffix of T. For a pattern $P[1..p]$ that appears in T, the *locus node* of P is denoted by $locus(P)$, which is the unique node closest to the root such that P is a prefix of $path(locus(P))$, and can be determined in $O(p)$ time. We augment the following structures on GST.

N-structure: An N-structure entry is a triplet $(doc, score, parent)$ and is associated with some node in GST. If u is a leaf node with $path(u)$ is a suffix of document d, the an N-structure entry with $doc = d$ is stored at u. However, if it is an internal node, multiple N-structure entries may be stored at u as follows: an entry with $doc = d$ is stored if and only if at least two children of u contain (a suffix of) document d in their subtrees. The *score* field in an N-structure entry for a document d associated with a node u is $score(path(u), d)$: the relevance score of d with respect to the pattern $path(u)$[3]. The *parent* field stores (the preorder rank of) the lowest ancestor of u which has an entry for document d in its N-structure. In case there is no such ancestor, we assign a dummy node which is regarded as the parent of the root of GST.

I-structure: An I-structure entry is a triplet $(doc, score, origin)$ and is associated with some node in GST. If node u has an N-structure entry for document d and an N-structure entry of another node v is given by $(d, score(path(v), d), u)$, then u will have an I-structure entry $(d, score(path(v), d), v)$. An internal node may be associated with multiple I-structure entries, and these entries are maintained in an array, sorted by the *origin* field. In addition, a range maximum query (RMQ) structure is maintained over the array based on the *score* field.

3.1 Query Answering

To answer a top-k query, we first search for the query pattern P in GST and find its locus node $locus(P)$. We also find the rightmost leaf $locus_R(P)$ in the subtree of $locus(P)$. Now, our task is to find, among the documents whose suffixes appear in the subtree of $locus(P)$, which k of them have the highest occurrences of P. Hon et al. showed that this can be done by checking only the I-structure entries associated with the proper ancestors of $locus(P)$, and then retrieving those k entries which has the highest *score* values and whose *origin* is from the subtree of $locus(P)$ (inclusively). The number of ancestors of P is bounded by p and since the I-structure entries are sorted according to the origin values, the entries to be checked will occupy a contiguous region in the sorted array. The boundaries of

[2] Maintain a bit vector $\mathcal{B}[1...(n+D)]$, where $\mathcal{B}[i] = 1$ if and only if $T[i] = \#$, then $x = rank_{\mathcal{B}}(a) + 1$ and can be computed in $O(1)$ time using [28].

[3] The *score* is dependent only on d and the set of occurrences of $path(u)$ in d.

the contiguous region can be obtained by performing a binary search based on (the pre-order ranks of) $locus(P)$ and $locus_R(P)$. Once we get the boundaries of the contiguous region in each proper ancestors of $locus(P)$, we can apply RMQ queries repeatedly over *score* and retrieve the top-k scoring documents in sorted order in $O(p \log n + k \log k)$ time. The binary search step can be made faster by maintaining a predecessor structure [32] and the resulting time will become $O(p \log \log n + k \log k)$. This time has been further improved to $O(p + k \log k)$ by introducing two additional fields δ_f and δ_ℓ in each N-structure entry. The number of N-structure entries (hence I-structure entries) is $\leq 2n$. Therefore the index space is $O(n \log n)$ bits.

4 Our Linear-Space Index

In this section, we derive a modified version of Hon et al.'s linear index without δ fields and still achieve $O(p)$ term in query time. The main technique is by introducing a novel criterion that categorizes the I-structure entries as *near* and *far*. The *far* entries associated with certain nodes can be maintained together as a combined I-structure, which reduces the number of I-structure boundaries to be searched to $O(p/\pi + \pi)$, where π is a sampling factor. By choosing $\pi = \log \log n$, we shall use predecessor search structure (instead of δ fields) and can compute the I-structure boundaries in $O((p/\pi + \pi) \log \log n) = O(p + \log^2 \log n)$ time. We have the following result.

Theorem 3. *There exists an index of size $O(n \log n)$ bits for top-k document retrieval with $O(p + \log^2 \log n + k \log \log \log n + k \log k)$ query time.*

Proof. Firstly, we mark all nodes in GST whose node-depths are multiples of π (node-depth of root is 0). Thus, any unmarked node is at most π nodes away from its lowest marked ancestor. Also, the number of marked ancestors of any node $= \lceil(\text{number of ancestors})/\pi\rceil$. For any node w in GST, we define a value $\zeta(w) < \pi$, where $\zeta(w) = 0$ if w is marked, else it is the number of nodes in the path from w (exclusively) till its lowest marked ancestor (inclusively). In each I-structure entry (d, s, v) associated with a node w, we maintain a fourth component $\zeta(w)$. Next, we categorize the I-structure entries as *far* and *near* as follows:

> *An I-structure entry associated with a node w, with origin $= v$, is near if there exists no marked node in the path from v (inclusively) to w (exclusively), else it is far.*

We restructure the entries such that all *far* entries are maintained in a combined I-structure associated with some marked nodes as follows: if $(d, s, v, \zeta(w))$ is a *far* entry in the I-structure I_w associated with node w, then we remove this entry from I_w and move to a combined I-structure associated with the node u, where $u = w$ if w is marked, else u is the lowest marked ancestor of w (i.e., u is $\zeta(w)$ nodes above w). All the entries in the combined I-structure are maintained in the sorted order of *origin* values. A predecessor search structure over the *origin* field and RMQ structure over the *score* field is maintained over all I-structures. Next, to understand how to answer a query with our index, we introduce the following auxiliary lemma.

Lemma 3. *The top-k documents corresponding to a pattern P can be obtained by checking the following I-structure entries (with origins coming from the subtree of $locus(P)$):*
(i) near entries in the regular I-structures associated with the nodes in the path from $locus(P)$ (exclusively) till its lowest marked ancestor u (inclusively), and there are at most π such nodes;
(ii) far entries with $\zeta < \zeta(locus(P))$ in the combined I-structure of u, and
(iii) far entries in the combined I-structures associated with the marked proper (at most p/π) ancestors of u.

Proof. In the original index by Hon et al., we need to check the I-structure entries in all ancestors of $locus(P)$. We may categorize them as follows:

(a) *near* entries associated with a node in the subtree of u (inclusively);
(b) *far* entries associated with a node in the subtree of u (inclusively);
(c) *far* entries associated with an ancestor node of u;
(d) *near* entries associated with an ancestor node of u.

All entries in (a) belong to category (i) in the lemma. The valid entries in (b) belong to category (ii), where the inequality $\zeta < \zeta(locus(P))$ ensures that the all entries in category (ii) were originally from an ancestor of $locus(P)$. All those entries in (c), which may be a possible candidate for the top-k documents, belong to category (iii) in the lemma. None of the entries in (d) can be a valid output, as the origin of those entries are not coming from the subtree of u (from the definition of a *near* entry), hence not from the subtree of $locus(P)$. On the other hand, since we always check for the entries with origins coming from the subtree of $locus(P)$, these entries must be a subset of those checked in the original index by Hon et al. In conclusion, the entries checked in both indexes are exactly the same, and the lemma follows. □

Based on the above lemma, we may compute k candidate answers from each category and the actual top-k answers can be computed by comparing the score of these $3k$ documents. In category (i) we have at most π boundaries to be searched, which takes $O(\pi \log\log n)$ time, and then retrieve the k candidate answers in the unsorted order in $O(\pi + k)$ time using lemma 1. Similarly in category (iii), the number of I-structure boundaries to be searched is p/π and it takes total $O((p/\pi)\log\log n + k)$ time. However, for category (ii), we have an additional constraint on ζ value of the entries. To facilitate the process, the ζ components are maintained by the data structure in Lemma 2 in $O(n \log \pi)$ bits, so that the desired answers can be reported in $O((\log \pi + k)(\log \pi + O(1)))$ time. The $O(k \log k)$ is for sorting the answers. The time for initial pattern search is $O(p)$. Putting all together with $\pi = \log\log n$, we obtain Theorem 3. □

5 Space-Efficient Encoding of Our Index

In this section, we derive a space-efficient index for the relevance metric *term-frequency.* The major contribution is that, instead of using $O(\log n)$ bits for an I-structure entry, we design some novel encodings so that each entry requires only $\log D + \log \pi + O(1)$ bits. The GST will be replaced by a compressed full text index CSA of size $|CSA|$ bits [11,5,10,1] along with the tree encoding of GST in $4n + o(n)$ bits [17][4]. Thus $locus(P)$ can be computed in $O(p)$ time by taking the LCA (lowest common ancestor) of leftmost and rightmost leaf in the suffix range of P.

A core component of our index is the document array D_A, where $D_A[i]$ stores the id of document to which the ith smallest suffix in GST belongs to. The D_A can be maintained in $n \log D + O(\frac{n \log D}{\log \log D})$ bits and can answer the following queries in $O(\log \log D)$ time [9]. (i) *access*(i): returns $D_A[i]$; (ii) *rank*(d, i): returns the number of occurrences of document d in $D_A[1...i]$; (iii) *select*(d, j): is -1 if $j > |d|$, else i where $D_A[i] = d$ and $rank(d, i) = j$. Now we show how to use D_A for efficient encoding and decoding of different components in an I-structure entry.

Term-frequency Encoding: Given an I-structure entry with $origin = v$ and $doc = d$, the corresponding *term-frequency* score is exactly the number of occurrences of d in $D_A[i...j]$, where i and j are the leftmost leaf and the rightmost leaf of v, respectively. Thus, given the values v and d, we can find i and j in constant time based on the tree encodings of the GST, and then compute *term-frequency* in $O(\log \log D)$ time based on two rank queries on D_A. Thus, we will discard the *score* field completely for all I-structure entries, but keeping only the RMQ structure over it.

Origin Encoding: Origin encoding is the most trickiest part, and is based on the following observation by Hon et. al [15]: for any document d and for any node v in GST, there is at most one ancestor of v that contains an I-structure entry with $doc = d$ and $origin$ from a node in the subtree of v (inclusively). We introduce two separate schemes for encoding *origin* fields in *near* and *far* entries. This reduces the *origin* array space from $O(n \log n)$ bits to $O(n)$ bits and decoding takes $O(\log \log D)$ time.

Encoding near entries: Let I_w be a regular I-structure (with only *near* entries) associated with a node w and let w_q represents the pre-order rank of q^{th} child of w. Then from the definition of I-structures, for a given document d, there exists at most one entry in I_w with $doc = d$ and origin from the sub-tree of w_q (inclusively). Thus, for a given document d and an internal node w, an entry in I_w can be associated to a unique child node w_q of w (where w_q represent

[4] Any n-node ordered tree can be represented in $2n + o(n)$ bits, such that if each node is labeled by its pre-order rank in the tree, any of the following operations can be supported in constant time [17]: *parent*(i), which returns the parent of node i; *child*(i, q), which returns the q-th child of node i; *child-rank*(i), which returns the number of siblings to the left of node i; *lca*(i, j), which returns the lowest common ancestor of two nodes i and j; and *lmost-leaf*(i)/*rmost-leaf*(i), which returns the leftmost/rightmost leaf of node i.

the qth child of w from left, $1 \leq q \leq degree(w)$, and pre-order rank of w_q can be computed in constant time [17]), such that *origin* is in the subtree of w_q. Moreover, this *origin* must be the node, closest to root, in the subtree of w_q which has an N-structure entry for d. From the definition of N-structure, this *origin* node must be the lowest common ancestor (LCA) of the leaves corresponding to the first and last suffixes of d in the subtree of w_q, which can be computed using the tree encoding of GST and a constant number of rank/select operations on D_A in total $O(\log\log D)$ time. Therefore, by maintaining the information about w_q (*origin-child* $= q$) for each I-structure entry, the corresponding *origin* value can be decoded in $O(\log\log D)$ time. Thus, the *origin* array can be replaced completely by the *origin-child* array. Recall that each node maintains the I-structure entries in sorted order of the origins, so that the corresponding *origin-child* array will be monotonic increasing. In addition, the value of each entry is between 1 and $degree(w)$, so that the array can be encoded using a bit vector of length $|I_w| + degree(w)$[5]. The total size of the bit vectors associated with all nodes can be bounded by $\sum_{w \in GST}(|I_w| + degree(w)) = O(n)$ bits. The $O(n \log n)$ bits predecessor search structure over *origin* array is replaced by a structure of $o(n)$ bits space and $O(\log\log n)$ search time[6].

Encoding far entries: In order to encode the *origin* values in *far* entries, we introduce the following notions. Let w^* be a marked node, then another node w_q^* is called its qth marked child, if w_q^* is the qth smallest (in terms of pre-order rank) marked node with w^* as its lowest marked ancestor. Given the pre-order rank of w^*, the pre-order rank of w_q^* can be computed in constant time by maintaining an additional $O(n)$ bits structure[7]. Let I_{w^*} represents the combined I-structure (with only *far* entries) associated with a marked node w^*. The origin value of any far entry in I_{w^*} is always a node in the subtree of some marked child w_q^* of w^*, and is always unique for a given q and $doc = d$. Thus by maintaining the information about w_q^* (*origin-child*$^* = q$), we can decode the corresponding *origin* value for a particular document d. i.e. *origin* is the LCA of the leaves corresponding to the first and last suffix of d in the

[5] A monotonic increasing sequence $S = 1333445$ can be encoded as $B = 101100010010$ in $|B|(1 + o(1))$ bits, where $S[i] = rank_1(select_0(i))$ on B, and can be computed in constant time [28].

[6] Construct a new array by sampling every $\log^2 n$th element in the original array, and maintain predecessor search structure over it. Now, when we perform the query, we can first query on this sampled structure to get an approximate answer, and the exact answer can be obtained by performing binary search on a smaller range of only $\log^2 n$ elements in the original array. The search time still remains $O(\log\log n)$.

[7] Let GST* be a tree induced by the marked nodes in GST, so that w^* is the lowest marked ancestor of w_q^* in GST if and only if the node corresponding to w^* in GST* (say, w) is the parent of node corresponding to w_q^* (say w_q) in GST*. Moreover, w_q^* is said to be the qth marked child of node w^* in GST, if w_q is the qth child of q in GST*. Given the pre-order rank of any marked node in GST, its pre-order rank in GST* (and vice versa) can be computed in constant time by maintaining an additional bit vectors of size $2n + o(n)$ which maintain the information if a node is marked or not.

sub-tree of w_q^*, which can be computed using the tree encoding of GST and a constant number of rank/select operations on D_A in total $O(\log\log D)$ time. Now *origin* array can be replaced by *origin-child** array, which can be encoded in $\sum_{w^* \in GST^*}(|I_{w^*}| + degree(w^*)) = O(n)$ bits (using the similar scheme for encoding *origin-child* array for *near* entries). The predecessor search structure is replaced by $o(n)$ bits sampled predecessor search structure.

***Query Answering*:** Query answering algorithm remains the same as that in our linear index, except the fact that decoding *origin* and *term-frequency* takes $O(\log\log D)$ time. Then the time complexities for the steps in Lemma 3 are as follows: Step (i) $O((\pi \log\log n + k)\log\log D)$, Step (ii) $O((\log \pi + k)(\log \pi + \log\log D))$ and Step (iii) $(((p/\pi)\log\log n + k)\log\log D)$. Since the *term-frequencies* are positive integers $\leq n$, we shall use a y-fast trie [32] to get the sorted answer in $O(k \log\log n)$ time. By choosing $\pi = \log^2 \log n$, the query time can be bounded by $O(t_s(p) + p + \log^4 \log n + k \log\log n)$, which gives the query time in Theorem 1. Here $t_s(p)$ is the time for initial pattern searching in CSA, and is $\Omega(p)$ for space-optimal CSA's [5,1].

***Space Analysis*:** The index consists of a full text index of $|CSA|$ bits, D_A of $n \log D(1 + o(1))$ bits, I-structures of total $2n(\log D + O(\log \pi) + O(1))$ bits, tree encodings, RMQ structures and sampled predecessor search structures (together $O(n)$ bits). By choosing $\pi = \log^2 \log n$, the index space can be bounded by $|CSA| + n \log D(3 + o(1)) + O(n \log\log\log n)$ bits. In order to obtain the space bounds in Theorem 1, we may categorize D into the following two cases.

1. When $\log D/\log\log D > \log\log\log n$, the $O(n \log\log\log n)$ term can be absorbed in $o(n \log D)$. The space can be further reduced by $n \log D$ bits from the following observation that the *term-frequency* is 1 for those I-structure entries with *origin = a leaf in GST*, and there are n such entries. Therefore all such entries can be deleted and in case if such a document is within top-k, that can be reported using document listing. For that we shall use Muthukrishnan's chain array idea [22]. The chain array $C[1...n]$ is defined as follows: $C[i] = j$, where $j < i$ is the largest number with $D_A[i] = D_A[j]$ and can be simulated using D_A as $j = select(D_A[i], rank(D_A[i], i) - 1)$ in $O(\log\log D)$ time. Thus we do not maintain chain array, instead an $2n + o(n) = o(n \log D)$ bits RMQ structure [6] over it. Let $[L, R]$ be the suffix range of P in the full text index, then document listing can be performed (in $O(\log\log D)$ time per document) by reporting all those documents $D_A[i]$ such that $L \leq i \leq R$ and $C[i] < L$ using repeated RMQ's. Although those documents with *frequency* > 1 will get retrieved again (but only once), it will not affect the overall time complexity.
2. When $\log D/\log\log D \leq \log\log\log n$, we shall use the index described in Theorem 4. Thus the space-query bounds will be $|CSA| + n \log D(1 + o(1))$ bits and $O(t_s(p) + \log\log n + k \log D \log^2 \log D) = O(t_s(p) + k \log\log n)$ respectively.

By combining the above case, we get the result in Theorem 1. □

Theorem 4. *There exists an index of size $|CSA| + n \log D(1 + o(1))$ bits with a query time of $O(t_s(p) + \log\log n + k \log D \log^2 \log D)$ for retrieving top-k documents with the highest term frequencies for a query pattern P of length p.*

Proof. See the full version [14].

6 Saving More Space

The most space-efficient version of our index (described in theorem 2) is proved in this section. First, we give the following auxiliary lemma (see the full version for proof [14]).

Lemma 4. *There exists an $O(n \log \sigma \log\log n)$ bits structure, which can answer access/rank/select queries on D_A in $O(\log^2 \log n)$ time, and can compute an entry $C[i]$ in the chain-array data structure (for document listing) in $O(\log\log n)$ time.*

To achieve space reduction, we categorize D into the following cases:

1. $\log D < (\log \sigma \log\log n)^{1+\epsilon/2}$: We shall use the index described in Theorem 4 and the query time will be $O(t_s(p) + k(\log \sigma \log\log n)^{1+\epsilon})$.
2. $\log D \geq (\log \sigma \log\log n)^{1+\epsilon/2}$: In this case D_A is replaced by a structure described in Lemma 4, which makes the index space $n \log D(1 + o(1))$ bits. Then by re-deriving the bounds with $\pi = \log^3 \log n$, our query time will be $O(t_s(p) + \log^6 \log n + k \log^2 \log n)$.

   The $O(k \log^2 \log n)$ term can be further improved to $O(k \log\log n)$ from the following observation that, once we get the I-structure boundaries, we do not need any information about the *origin* fields for further query processing. Thus the only value needed is the *term-frequency*, which can be computed as follows: a sampled document array D_A^s is maintained, such that $D_A[i] = d$ is stored if and only if $(rank_{D_A}(d, i)) mod\ \rho = 0$, for an integer $\rho = \Theta(\log D)$, else we store a NIL value, where $rank_{D_A}(d, j)$ is the number of occurrences of d in $D_A[1...j]$. Then D_A^s can be maintained in $O(n \log D/\alpha) = O(n)$ bits and can compute an approximate rank. That is $\rho\ rank_{D_A^2}(d, j) \leq rank_{D_A}(d, j) \leq \rho\ rank_{D_A^2}(d, j) + \rho$. Thus associated with each I-structure entry, we shall store this error ($= \Theta(\log D)$), which is equal to actual *term-frequency* minus approximate *term-frequency* (computed using D_A^s). Thus by storing this error corresponding to each I-structure entry in total $O(n \log \rho) = O(n \log\log D) = o(n \log D)$ bits space, the *term-frequency* can be obtained in $O(\log\log D) = O(\log\log n)$ time by first computing the approximate *term-frequency* using D_A^s and then by adding this stored value. Note that for the initial I-structure boundary searches, the origin decoding is performed using the structure in Lemma 4. Moreover, this structure can compute chain array values in $O(\log\log n)$ time, which can be used for document listing in $O(\log\log n)$ time per report (when the I-structure entries with *term-frequency* $= 1$ are deleted from the index, and later such a document is an answer for a query).

By combining the above cases, we obtain an $|CSA| + n \log D(1 + o(1))$ bits index with query time $O(t_s(p) + k(\log \sigma \log\log n)^{1+\epsilon} + \log^6 \log n)$, which completes the proof of Theorem 2. □

References

1. Belazzougui, D., Navarro, G.: Alphabet-Independent Compressed Text Indexing. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 748–759. Springer, Heidelberg (2011)
2. Belazzougui, D., Navarro, G.: Improved Compressed Indexes for Full-Text Document Retrieval. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 386–397. Springer, Heidelberg (2011)
3. Blum, M., Floyd, R.W., Pratt, V., Rivest, R., Tarjan, R.: Time Bounds for Selection. Journal of Computer and System Sciences 7(4), 448–481 (1973)
4. Shane Culpepper, J., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k Ranked Document Search in General Text Databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. 3(2), art. 20 (2007)
6. Fischer, J.: Optimal Succinctness for Range Minimum Queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010)
7. Frederickson, G.N.: An Optimal Algorithm for Selection in a Min-Heap. Information and Computation 104(2), 197–214 (1993)
8. Gagie, T., Navarro, G., Puglisi, S.J.: Colored Range Queries and Document Retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 67–81. Springer, Heidelberg (2010)
9. Golynski, A., Munro, J.I., Rao, S.S.: Rank/Select Operations on Large Alphabets: A Tool for Text Indexing. In: SODA, pp. 368–373 (2006)
10. Grossi, R., Vitter, J.S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. SIAM Journal on Computing 35(2), 378–407 (2005)
11. Grossi, R., Gupta, A., Vitter, J.S.: High-Order Entropy-Compressed Text Indexes. In: SODA, pp. 841–850 (2003)
12. Hon, W.K., Patil, M., Shah, R., Wu, S.-B.: Efficient Index for Retrieving Top-k Most Frequent Documents. Journal of Discrete Algorithms 8(4), 402–417 (2010)
13. Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: String Retrieval for Multi-pattern Queries. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 55–66. Springer, Heidelberg (2010)
14. Hon, W.-K., Shah, R., Thankachan, S.V.: Towards an Optimal Space-and-Query-Time Index for Top-k Document Retrieval. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 173–184. Springer, Heidelberg (2012)
15. Hon, W.K., Shah, R., Vitter, J.S.: Space-Efficient Framework for Top-k String Retrieval Problems. In: FOCS, pp. 713–722 (2009)
16. Hon, W.-K., Shah, R., Vitter, J.S.: Compression, Indexing, and Retrieval for Massive String Data. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 260–274. Springer, Heidelberg (2010)
17. Jansson, J., Sadakane, K., Sung, W.K.: Ultra-succinct Representation of Ordered Trees. In: SODA, pp. 575–584 (2007)

18. Karpinski, M., Nekrich, Y.: Top-k Color Queries for Document Retrieval. In: SODA, pp. 401–411 (2011)
19. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing 22(5), 935–948 (1993)
20. Matias, Y., Muthukrishnan, S.M., Şahinalp, S.C., Ziv, J.: Augmenting Suffix Trees, with Applications. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 67–78. Springer, Heidelberg (1998)
21. McCreight, E.M.: A Space-Economical Suffix Tree Construction Algorithm. Journal of the ACM 23(2), 262–272 (1976)
22. Muthukrishnan, S.: Efficient Algorithms for Document Retrieval Problems. In: SODA, pp. 657–666 (2002)
23. Navarro, G., Nekrich, Y.: Top-k document retrieval in optimal time and linear space. In: SODA, pp. 1066–1077 (2012)
24. Navarro, G., Puglisi, S.J., Valenzuela, D.: Practical Compressed Document Retrieval. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 193–205. Springer, Heidelberg (2011)
25. Navarro, G., Valenzuela, D.: Space-Efficient Top-k Document Retrieval. To appear in SEA (2012)
26. Navarro, G., Puglisi, S.J.: Dual-Sorted Inverted Lists. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 309–321. Springer, Heidelberg (2010)
27. Patil, M., Thankachan, S.V., Shah, R., Hon, W.K., Vitter, J.S., Chandrasekaran, S.: Inverted Indexes for Phrases and Strings. In: SIGIR, pp. 555–564 (2011)
28. Raman, R., Raman, V., Rao, S.: Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. ACM Transactions on Algorithms 3(4) (2007)
29. Sadakane, K.: Succinct Data Structures for Flexible Text Retrieval Systems. Journal of Discrete Algorithms 5(1), 12–22 (2007)
30. Välimäki, N., Mäkinen, V.: Space-Efficient Algorithms for Document Retrieval. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 205–215. Springer, Heidelberg (2007)
31. Weiner, P.: Linear Pattern Matching Algorithms. In: SWAT (1973)
32. Willard, D.E.: Log-logarithmic Worst-Case Range Queries Are Possible in Space $\Theta(N)$. Information Processing Letters 17(2), 81–84 (1983)

Document Listing for Queries with Excluded Pattern[⋆]

Wing-Kai Hon[1], Rahul Shah[2], Sharma V. Thankachan[2], and Jeffrey Scott Vitter[3]

[1] National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw
[2] Louisiana State University, USA
{rahul,thanks}@csc.lsu.edu
[3] The University of Kansas, USA
jsv@ku.edu

Abstract. Let $\mathcal{D} = \{d_1, d_2, ..., d_D\}$ be a given collection of D string documents of total length n. We consider the problem of indexing $\mathcal{D}$ such that, whenever two patterns P^+ and P^- comes as an online query, we can list all those documents containing P^+ *but not* P^-. Let t represent the number of such documents. An index proposed by Fischer et al. (LATIN, 2012) can answer this query in $O(|P^+| + |P^-| + t + \sqrt{n})$ time. However, its space requirement is $O(n^{3/2})$ bits. We propose the first linear-space index for this problem with a worst case query time of $O(|P^+| + |P^-| + \sqrt{n} \log\log n + \sqrt{nt} \log^{2.5} n)$.

1 Introduction and Related Work

Document retrieval is a fundamental problem in information retrieval, where the task is to index a collection of documents, such that whenever a pattern (or a set of patterns) comes as an online query, we can efficiently retrieve those documents which are relevant to the query. An occurrence of a query pattern in a document makes it relevant to the query. However, query with excluded patterns is a problem orthogonal to this. That is, the occurrence of an excluded pattern in a document makes it less relevant to the query. Such queries are fundamental and important in web-search applications. For example, the search results from Google for a pattern ”jaguar” consists of many webpages related to “jaguar car”, but one may be interested in jaguar as a big cat, not as a car. Whereas the search results for the query “jaguar -car” will be those documents which are related to “jaguar”, but not to “car”. Here the “-” symbol before the pattern“car” indicates that it is an excluded pattern.

More formally, we shall define the document listing problem for excluded pattern queries as follows: given a collection $\mathcal{D}$ of D documents $\{d_1, d_2, ..., d_D\}$ of total length n, and the query consisting of two patterns P^+ (called included

⋆ This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W. Hon) and US NSF Grant CCF–1017623 (R. Shah).

pattern) and P^- (called excluded pattern), our task is to list the set of documents containing P^+ *but not* P^-. Traditionally, the documents are split into terms (or words) and then an inverted index is built over such terms. However, in the case of genome data or some Asian texts, there may be no natural word demarcation (we may call such documents as strings), so that the inverted index may provide only limited searching capabilities or may require too much space. To the best of our knowledge, the only known index which supports this kind of queries for string documents is by Fischer et al. [8], which takes $O(n^{3/2})$ bits of space and has $O(|P^+| + |P^-| + t + \sqrt{n})$ query time, where t is the number of documents containing P^+ *but not* P^-. We propose the first linear-space solution for this problem, and our main result is captured in the following theorem.

Theorem 1. *Given a collection of D string documents of total length n, there exists an $O(n)$-word data structure that supports listing documents with P^+ but not P^- in $O(|P^+| + |P^-| + \sqrt{n} \log\log n + \sqrt{nt} \log^{2.5} n)$ time, where P^+ and P^- are two online query patterns and t represents the number of such documents.*

On a related note, string document retrieval problem for queries with a single (included) pattern is a well studied problem [21,26,27,20] with many interesting results. Another fundamental problem which has received a lot of attention recently is the top-k document retrieval [20,22,18,1,5,9,12,23,24,15,14]. Muthukrishnan [21] has studied the problem where the query consists of an excluded pattern alone, and has given an optimal-query-time solution. Document listing for queries with two included-patterns (P_1 and P_2) is another harder problem, and the following are the space-time tradeoffs of the known indexes (here t represents the number of documents containing both P_1 and P_2):

- $\tilde{O}(n^{3/2})$-space[1] and $O(|P_1| + |P_2| + \sqrt{n} + t)$ query time [7].
- $O(n \log n)$ words and $O(|P_1| + |P_2| + \sqrt{n(t+1)} \log^{2.5} n)$ query time [4].
- $O(n)$ words and $O(|P_1| + |P_2| + \sqrt{n(t+1)} \log^{1.5} n)$ query time [16].

Fischer et al. [8] showed that document listing problem for two-included-pattern queries is much harder than the one with single-included-pattern using reduction techniques via Geometric Burrows-Wheeler Transform (GBWT) [3].

2 Preliminaries

2.1 Suffix Trees and Suffix Arrays

Suffix Tree: Given a text $T[1...n]$, a substring $T[i...n]$ with $1 \leq i \leq n$ is called a suffix of T. The lexicographic arrangement of all n suffixes of T in a compact trie is called the *suffix tree* of T [28], where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge in the suffix tree is labeled by a

[1] The notation $\tilde{O}$ ignores poly-logarithmic factors. Precisely, $\tilde{O}(f(n)) \equiv O(f(n) \log^{O(1)} n)$.

character string and for any node u, path(u) is the string formed by concatenating the edge labels from root to u. For any leaf v, path(v) is exactly the suffix corresponding to v. For a given pattern P, a node u is defined as the *locus node* of P if it is the closest node to the root such that P is a prefix of path(u); such a node can be determined in $O(|P|)$ time.

Suffix Array: Suffix array $SA[1...n]$ of a text T is an array such that $SA[i]$ stores the starting position of the ith lexicographically smallest suffix of T [19]. In SA the starting positions of all suffixes with a common prefix are always stored in contiguous range. The suffix range of a pattern P is defined as the maximal range $[\ell, r]$ such that for all $j \in [\ell, r]$, P is a common prefix of the suffix which starts at $SA[j]$.

Generalized Suffix Tree: Given a collection $\mathcal{D}$ of strings, the *generalized suffix tree* (GST) of $\mathcal{D}$ is a compact trie which stores all suffixes of all strings in $\mathcal{D}$. For the purpose of our index, we define an extra array D_A called *document array*, such that $D_A[i] = j$ if and only if the $i$the lexicographically smallest suffix is from document d_j.

2.2 Wavelet Tree

Let $A[1...n]$ be an array of length n, where each element $A[i]$ is a symbol drawn from a set Σ of size σ. The *wavelet tree* (WT) [11] for A is an ordered balanced binary tree on Σ, where each leaf is labeled with a symbol in Σ, and the leaves are sorted alphabetically from left to right. Each internal node W_k represents an alphabet set Σ_k, and is associated with a bit-vector B_k. In particular, the alphabet set of the root is Σ, and the alphabet set of a leaf is the singleton set containing its corresponding symbol. Each node partitions its alphabet set among the two children (almost) equally, such that all symbols represented by the left child are lexicographically (or numerically) smaller than those represented by the right child. For the node W_k, let A_k be a subsequence of A by retaining only those symbols that are in Σ_k. Then B_k is a bit-vector of length $|A_k|$, such that $B_k[i] = 0$ if and only if $A_k[i]$ is a symbol represented by the left child of W_k. Indeed, the subtree from W_k itself forms a wavelet tree of A_k. To reduce space requirement, the array A is not stored explicitly in the wavelet tree. Instead, we only store the bit-vectors B_k, each of which is augmented with Raman et al.'s scheme [25] to support constant-time bit-rank and bit-select operations. WT takes $n \log \sigma (1 + o(1))$ bits space and can answer the following queries in $O(\log \sigma)$ time.

$rank_c(i)$ = number of occurrences of $c \in \Sigma$ in $A[1...i]$
$select_c(i) = -1$ if $rank_c(n) < i$, else return j, where $A[j] = c$ and $rank_c(j) = i$.

Note that by using the $n \log \sigma + O(n \log \sigma / \log \log \sigma)$ bits index by [10], $rank_c$ and $select_c$ can be performed in $O(\log \log \sigma)$ time.

2.3 Weight-Balanced Wavelet Tree

Weight-balanced wavelet tree (WBT) is a modified version of WT proposed by Hon et al. [16]. Here the number of 0's and 1's in any bit-vector B_k is made almost equal, which ensures the following property.

Lemma 1. *Let W_k be a node in WBT at depth δ_k, and B_k denote its associated bit-vector. Let $n_k = |B_k|$. Then we have $n_k \leq 4n/2^{\delta_k}$.*

WBT on an array $A[1...n]$ takes $n(\log \sigma + 2)(1 + o(1))$ bits of space. The tree depth of WBT can be of $O(\log n)$, so that the worst case query time (for $rank_c(i)$ and $select_c(i)$ for any $c \in \Sigma$) is $O(\log n)$. See Appendix A and B for more details of WBT.

3 Data Structures for Document Counting

Here we describe an index which can count the number of documents containing P^+ *but not* P^-. We capture the result in the following theorem.

Theorem 2. *There exists an $O(n)$-word index that supports counting the number of documents with P^+ but not P^- in $O(|P^+| + |P^-| + \sqrt{n} \log \log n)$ time, where P^+ and P^- are two online query patterns.*

***Index Construction*:** The following shows the main components of the document counting index.

- GST/GSA, the generalized suffix tree/array of $\mathcal{D}$.
- Document array D_A, where $D_A[i] = j$ if the ith lexicographically smallest suffix belongs to document d_j.
- An $2n + o(n)$ bits structure, which can compute *document-frequency* $df(P)$ of a pattern P in $O(1)$ time from the suffix range of P [26].[2]
- COUNT matrix, to be defined below.

First, starting from left in GST, we combine every g (called group size, to be determined later) contiguous leaves together to form a group. Thus, the first group consists of $\ell_1, ..., \ell_g$, the next group consists of $\ell_{g+1}, ..., \ell_{2g}$, and so on, where ℓ_j denotes the jth leftmost leaf in GST. Consequently, we have a total of $O(n/g)$ groups, and for each group we mark the least common ancestor (LCA) of its first and its last leaves. Moreover, if two nodes are marked, we mark their LCA as well. The total number of marked nodes by this scheme can be bounded by $O(n/g)$ [13]. Now suppose for any node u in GST, with its subtree containing the leaves $\ell_x, \ell_{x+1}, \ldots, \ell_y$, then the range $[x, y]$ is referred to as the *suffix range* corresponding to u.

Lemma 2. *[13] The suffix range $[L, R]$ of any pattern P can be split into a suffix range $[L', R']$ corresponding to some marked node u^*, and two other suffix ranges $[L, L' - 1]$ and $[R' + 1, R]$ with $L' - L < g$ and $R - R' < g$.*

[2] $df(P)$ = the number of distinct documents in $\mathcal{D}$ which has at least one occurrence of P.

Proof. By setting $L' = g\lceil L/g \rceil + 1$ and $R' = g\lfloor R/g \rfloor$, we have $L' - L < g$ and $R - R' < g$, and the LCA of $\ell_{L'}$ and $\ell_{R'}$ gives the desired marked node u^*. $\square$

Essentially, the suffix range $[L, R]$ of a pattern P corresponds to the leaves $\ell_L, \ell_{L+1}, \ldots, \ell_R$ in the GST. This set of leaves can be partitioned into three groups: those which are under the subtree of u^* $(\ell_{L'}, \ell_{L'+1}, \ldots, \ell_{R'})$, and the remaining two with those on the left of $\ell_{L'}$ and those on the right of $\ell_{R'}$. We shall refer to the latter two groups of leaves $(\ell_L, \ell_{L+1}, \ldots, \ell_{L'-1}$ and $\ell_{R'+1}, \ell_{R'+2}, \ldots, \ell_R)$ as *fringe leaves*, each such group contains fewer than g leaves.

Let d be a document in $\mathcal{D}$, and u and v be two nodes in GST. Then we define the following functions:

– $F(d, u, v) = 1$, if d contains the pattern $path(u)$ *but not* the pattern $path(v)$, else 0.

– $COUNT(u, v) = \sum_{d \in \mathcal{D}} F(d, u, v)$, which is the number of documents containing the pattern $path(u)$ *but not* the pattern $path(v)$.

Lemma 3. *The function $F(d, u, v)$ can be evaluated in $O(\psi)$ time, where ψ denotes the time for a $rank_d$ query on D_A.*

Proof. Using the tree encoding of GST, the suffix ranges $[L_u, R_u]$ and $[L_v, R_v]$ corresponding to u and v can be computed in constant time. Then, the number of occurrences of $path(u)$ in d, called *term-frequency* (denoted by $tf(path(u), d)$) can be computed as follows: $tf(path(u), d) = rank_d(R_u) - rank_d(L_u - 1)$. Similarly $tf(path(v), d) = rank_d(R_v) - rank_d(L_v - 1)$. If $tf(path(u), d) \geq 1$ and $tf(path(v), d) = 0$, then $F(d, u, v) = 1$, else 0. Therefore, the time for computing F can be bounded by $O(\psi)$, where ψ denotes the time for a $rank_d$ query on D_A. $\square$

COUNT matrix is simply an $O(n/g) \times O(n/g)$ matrix (of size $O(n^2 \log D/g^2)$ bits), which stores $COUNT(u^*, v^*)$ between all pairs of marked nodes u^* and v^* in GST.

***Query Answering*:** The first step is to obtain the locus nodes u and v (and the corresponding suffix ranges $[L_u, R_u]$ and $[L_v, R_v]$) of P^+ and P^-, respectively. Then, we compute the suffix ranges $[L'_u, R'_u]$ and $[L'_v, R'_v]$ (as described in Lemma 2), and the corresponding marked LCA nodes u^* and v^*. Our objective is to compute $COUNT(u, v)$, where as $COUNT(u^*, v^*)$ is precomputed and is stored in the *COUNT matrix*. We have the following lemma on these values.

Lemma 4. *Given $COUNT(u^*, v^*)$, the value $COUNT(u, v)$ can be computed in $O(g\psi)$ time, where g is the group size and ψ is the time for a $rank_d$ query on D_A.*

Proof: Let $S(u, v)$ represent the set of all documents containing the pattern $path(u)$ *but not* the pattern $path(v)$, hence $COUNT(u, v) = |S(u, v)|$. Note that for those documents d_j, with none of its suffix corresponding to a fringe leaf (i.e., $D_A[i] \neq d_j$ for all $i \in [L_u, L'_u - 1] \cup [R'_u + 1, R_u] \cup [L_v, L'_v - 1] \cup [R'_v + 1, R_v]$), $d_j \in S(u^*, v^*)$ if and only if $d_j \in S(u, v)$. From this observation, $COUNT(u, v)$ can be computed from $COUNT(u^*, v^*)$ by recomputing the membership of only those documents with suffixes corresponding to a fringe leaf, and the number of such documents is bounded by $4g$. Note that we may not be able to find the set $S(u,v)$ efficiently as we have not stored $S(u^*,v^*)$, however what we are interested is in $|S(v,v)|$, which can be computed from $|S(u^*,v^*)|$ as follows:

```
$COUNT(u,v) \leftarrow COUNT(u^*,v^*)$
**for** all distinct documents $d$ corresponding to a fringe
leaf **do**
  **if** $F(d,u,v)=1$ and $F(d,u^*,v^*)=0$ **then**
    $COUNT(u,v) \leftarrow COUNT(u,v)+1$
  **else if** $F(d,u,v)=0$ and $F(d,u^*,v^*)=1$ **then**
    $COUNT(u,v) \leftarrow COUNT(u,v)-1$
  **end if**
**end for**
**return** $COUNT(u,v)$
```

The time for evaluating F is $O(\psi)$, and the number of such distinct documents is bounded by $4g$. This completes the proof of the lemma. □

Therefore document counting query can in general be answered in $O(|P^+|+|P^-|+g\psi)$ time. However, we need to handle the following two special cases as well.

1. When $R_u - L_u + 1 < 2g$, the marked node u^* may not exist. Then we shall retrieve all (at most g) distinct documents corresponding to the suffixes in $[L_u, R_u]$, and eliminate those documents which has a suffix in $[L_v, R_v]$ as well. This can be verified in $O(\psi)$ time per document, hence the total time can be bounded by $O(|P^+|+|P^-|+g\psi)$.
2. When $R_v - L_v + 1 < 2g$ the marked node v^* may not exist.. Therefore, we first retrieve all (at most g) distinct documents corresponding to the suffixes in $[L_v, R_v]$. These are the documents (say excluded documents) which do not contribute to $COUNT(u,v)$. Now, we compute the number of excluded documents which have an occurrence of P^+ as well using D_A in $O(g\psi)$ time. By subtracting this number from $df(P^+)$ (the number of distinct documents where P^+ occurs), we get $COUNT(u,v)$, and the total time can be bounded by $O(|P^+|+|P^-|+g\psi)$.

The space and time bounds in Theorem 2 can simultaneously be achieved by choosing $g=\sqrt{n}$, and by maintaining D_A using the data structure in [10], where $\psi = O(\log\log D) = O(\log\log n)$.

4 Data Structures for Document Listing

Our index supporting document listing consists of the following components:

- GST of $\mathcal{D}$.
- *Weight-balanced wavelet tree* (WBT) over document array D_A.
- Let W_k represent an internal node in WBT, $\mathcal{D}_k$ be the set of distinct documents represented by the leaf nodes in the sub-tree of W_k and $n_k = \sum_{d_j \in \mathcal{D}_k} |d_j|$. At every internal node W_k, we maintain the index (from Section 3) for answering document counting query for the corresponding document collection $\mathcal{D}_k$. However, to save space, we do not maintain the generalized suffix tree GST_k of $\mathcal{D}_k$; instead, we maintain only its tree encoding[3] along with marked nodes information and the $2n_k + o(n_k)$ bits data structure for finding *document-frequency.* Moveover we do not need to maintain separate document array for this collection, since the subtree of W_k in WBT is a *weight-balanced wavelet tree* (WBT_k) on $\mathcal{D}_k$. We choose the group size $g_k = \sqrt{n_k \log n}$ and since we are using WBT, the time for a $rank_d$ query on D_A is $\psi = O(\log n)$.

***Index Space*:** The total index space can be computed as follows: GST takes $O(n \log n)$ bits, WBT takes $O(n \log D)$ bits. The bit vector B_k associated with the node W_k is of length n_k. Therefore the tree encoding (along with the marked nodes information and the data structure for computing $df(P)$) of GST_k takes $O(n_k)$ bits space. The $COUNT$ matrix associated with data structure in node W_k takes $O(n_k^2 \log D/g_k^2) = O(n_k)$ bits by choosing $g_k = \sqrt{n_k \log n}$. Note that $\sum_k |n_k|$ is the size of WBT (in bits). Thus the total space is $O(n \log n)$ bits $= O(n)$ words.

***Query Answering*:** Query answering is performed as follows: After computing the locus nodes of P^+ and P^- in GST, we perform a document counting query on $\mathcal{D}$. This is performed using the count structure associated with the root node in WBT. If the count is non-zero, we do a multi-way search in both child nodes, which correspond to searching two partitions of $\mathcal{D}$. This procedure is continued recursively until we reach a leaf node in the binary tree, thus the document corresponding to that leaf can be listed as an output. At any node, if the count is zero, we do not need to continue the recursive step further in its subtree.

Let $[L, R]$ be the suffix range of a pattern P in GST. Then, the suffix range of P in GST_k can be computed in $O(\psi)$ time by translating the range $[L, R]$ to the node W_k by navigating the WBT. Once we get the suffix range of a pattern, its locus node (and the corresponding marked node) in GST_k can be computed in constant time using the tree encoding [17]. Therefore, we need to perform the pattern searching only once (in GST), and the count queries at each internal node W_k of the WBT can be performed in $O(g_k \psi)$ time, instead of $O(|P^+| + |P^-| + g_k \psi)$ time. The overall query time consists of the following components and can be analyzed as follows:

[3] Any n-node ordered tree can be represented in $2n + o(n)$ bits, such that if each node is labeled by its pre-order rank in the tree, any of the following operations can be supported in constant time [17]: *parent*(i), which returns the parent of node i; *lca*(i, j), which returns the lowest common ancestor of two nodes i and j; and *lmost-leaf*(i)/*rmost-leaf*(i), which returns the leftmost/rightmost leaf of node i.

– *Count Queries*: The count query at an internal node W_k takes $O(g_k\psi) = O(\sqrt{n_k \log n}\log n)$ time. Since WBT ensures that $n_k \leq 4n/2^{\delta_k}$, where δ_k is the depth of W_k, so the overall time for count queries will be bounded by:

$$\begin{aligned}
&O\left(\sum_{W_k \in WBT_{visited}} \sqrt{n_k \log n}\log n\right) \\
&= O\left(\sqrt{n}\log^{3/2} n \sum_{W_k \in WBT_{visited}} 2^{-\delta_k/2}\right) \\
&= O\left(\sqrt{n}\log^{3/2} n \sqrt{\sum_{W_k \in WBT_{visited}} 1^2}\sqrt{\sum_{W_k \in WBT_{visited}} 2^{-\delta_k}}\right) \qquad (1)\\
&= O\left(\sqrt{n}\log^{3/2} n \sqrt{t \log n}\sqrt{\log(1 + \#\text{ of nodes in } WBT_{visited})}\right) \qquad (2)\\
&= O\left(\sqrt{nt}\log^{2.5} n\right),
\end{aligned}$$

where Equation (1) is by Cauchy-Schwarz's inequality,[4] while Equation (2) is by the following fact: In a binary tree T with a total of z nodes, and the depth of a node $u \in T$ is given by δ_u, then $\sum_{u\in T} 2^{-\delta_u} \leq \log(1+z)$[5].

– *Initial pattern matching*: This is the time for searching P^+ and P^- in GST and computing the locus nodes u and v, respectively, which can be bounded by $O(|P^+| + |P^-|)$.

– *WBT tree traversal*: Let $t = COUNT(u, v)$ be the number of outputs. Now, consider a binary tree structure $WBT_{visited}$, which is a subtree of WBT with only those nodes visited when we answer the query. Since each internal node in $WBT_{visited}$ must be on the path from the root to some document in the output set, and since the height of WBT is $O(\log n)$, the number of internal nodes in $WBT_{visited}$ is bounded by $O(t \log n)$. As $WBT_{visited}$ is a binary tree, the total number of nodes (i.e., leaves and internal nodes) is bounded by $O(t \log n)$. Thus, the tree traversal time can be bounded by $O(t \log n)$, since it takes only constant time to traverse from a node to its child node.

Note that even if $t = 0$, we need to spend $O(\sqrt{n}\log^{3/2} n)$ time for count query at the root note of WBT. Putting all things together, we get a query time of $O(|P^+| + |P^-| + \sqrt{n}\log^{3/2} n + t\log n + \sqrt{nt}\log^{2.5} n) = O(|P^+| + |P^-| + \sqrt{n}\log^{3/2} n + \sqrt{nt}\log^{2.5} n)$. The $O(\sqrt{n}\log^{3/2} n)$ term can be improved to $O(\sqrt{n}\log\log n)$ by maintaining an additional $O(n)$-word data structure (described in Theorem 2) for performing the first count query, just in case $t = 0$. This completes the proof of Theorem 1.

[4] $\sum_{i=1}^{n} x_i y_i \leq \sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}$.

[5] This fact can be proven by induction: When T contains a single node this is trivially valid ($\delta_{root} = 0$). And for a general tree, we can split T as the root, and two binary trees T_1 and T_2 of z_1 and z_2 nodes respectively, where $z = 1 + z_1 + z_2$. Then $\sum_{u\in T} 2^{-\delta_u} = 1 + \frac{1}{2}(\sum_{u\in T_1} 2^{-\delta_u} + \sum_{u\in T_2} 2^{-\delta_u}) \leq 1 + \frac{1}{2}(\log(1+z_1) + \log(1+z_2)) \leq \log(2\sqrt{(1+z_1)(1+z_2)}) \leq \log(1 + z_1 + 1 + z_2) = \log(1 + z)$.

5 Concluding Remarks

In this paper, we give the first linear space index for two-pattern queries with one included pattern and one excluded pattern. The technique used in this paper is similar to that in [16], where we define a different COUNT matrix for solving the two-pattern queries problem with positive patterns only. However, there are some subtle differences. In particular, the handling of the fringe leaves, and the analysis of the query time in document listing, are much trickier. For further work, we hope to extend the study to the top-k version of this problem, though we suspect that it may not be easily solved with the existing techniques in the literature for positive patterns. Finally, the authors wish to thank Travis Gagie for providing his manuscript [8] on the first solution to this problem.

References

1. Belazzougui, D., Navarro, G.: Improved Compressed Indexes for Full-Text Document Retrieval. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 386–397. Springer, Heidelberg (2011)
2. Bender, M.A., Farach-Colton, M.: The LCA Problem Revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
3. Chien, Y.-F., Hon, W.-K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In: DCC, pp. 252–261 (2008)
4. Cohen, H., Porat, E.: Fast Set Intersection and Two Patterns Matching. Theor. Comput. Sci. 411(40-42), 3795–3800 (2010)
5. Shane Culpepper, J., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k Ranked Document Search in General Text Databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
6. Ferragina, P., Giancarlo, R., Manzini, G.: The Myriad Virtues of Wavelet Trees. Inf. and Comp. 207(8), 849–866 (2009)
7. Ferragina, P., Koudas, N., Muthukrishnan, S., Srivastava, D.: Two-dimensional substring indexing. J. Comput. Syst. Sci. 66(4), 763–774 (2003)
8. Fischer, J., Gagie, T., Kopelowitz, T., Lewenstein, M., Mäkinen, V., Salmela, L., Välimäki, N.: Forbidden Patterns. In: Fernández-Baca, D. (ed.) LATIN 2012. LNCS, vol. 7256, pp. 327–337. Springer, Heidelberg (2012)
9. Gagie, T., Navarro, G., Puglisi, S.J.: Colored Range Queries and Document Retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 67–81. Springer, Heidelberg (2010)
10. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: SODA, pp. 368–373 (2006)
11. Grossi, R., Gupta, A., Vitter, J.S.: High-Order Entropy-Compressed Text Indexes. In: SODA, pp. 841–850 (2003)
12. Hon, W.K., Patil, M., Shah, R., Wu, S.-B.: Efficient Index for Retrieving Top-k Most Frequent Documents. Journal of Discrete Algorithms 8(4), 402–417 (2010)
13. Hon, W.K., Shah, R., Vitter, J.S.: Space-Efficient Framework for Top-k String Retrival Problems. In: FOCS, pp. 713–722 (2009)
14. Hon, W.-K., Shah, R., Vitter, J.S.: Compression, Indexing, and Retrieval for Massive String Data. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 260–274. Springer, Heidelberg (2010)

15. Hon, W.-K., Shah, R., Thankachan, S.V.: Towards an Optimal Space-and-Query-Time Index for Top-k Document Retrieval. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp.173–184. Springer, Heidelberg (2012)
16. Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: String Retrieval for Multi-pattern Queries. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 55–66. Springer, Heidelberg (2010)
17. Jansson, J., Sadakane, K., Sung, W.K.: Ultra-succinct Representation of Ordered Trees. In: SODA, pp. 575–584 (2007)
18. Karpinski, M., Nekrich, Y.: Top-K Color Queries for Document Retrieval. In: SODA, pp. 401–411 (2011)
19. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SICOMP 22(5), 935–948 (1993)
20. Matias, Y., Muthukrishnan, S.M., Şahinalp, S.C., Ziv, J.: Augmenting Suffix Trees, with Applications. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 67–78. Springer, Heidelberg (1998)
21. Muthukrishnan, S.: Efficient Algorithms for Document Retrieval Problems. In: SODA, pp. 657–666 (2002)
22. Navarro, G., Nekrich, Y.: Top-k document retrieval in optimal time and linear space. In: SODA, pp. 1066–1077 (2012)
23. Navarro, G., Puglisi, S.J.: Dual-Sorted Inverted Lists. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 309–321. Springer, Heidelberg (2010)
24. Patil, M., Thankachan, S.V., Shah, R., Hon, W.K., Vitter, J.S., Chandrasekaran, S.: Inverted Indexes for Phrases and Strings. In: SIGIR, pp. 555–564 (2011)
25. Raman, R., Raman, V., Rao, S.S.: Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. TALG 3(4) (2007)
26. Sadakane, K.: Succinct Data Structures for Flexible Text Retrieval Systems. JDA 5(1), 12–22 (2007)
27. Välimäki, N., Mäkinen, V.: Space-Efficient Algorithms for Document Retrieval. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 205–215. Springer, Heidelberg (2007)
28. Weiner, P.: Linear Pattern Matching Algorithms. In: Proc. Switching and Automata Theory, pp. 1–11 (1973)

A Proof of Lemma 1

Let W_ℓ and W_r denote the left and the right children of W_k, respectively. Let B_ℓ and B_r be their corresponding bit-vectors, and n_ℓ and n_r be their lengths. Thus $n_k = n_\ell + n_r$. Let L_k denote the number of occurrences of the least frequent symbol σ' (i.e., σ_1) represented by W_k, and similarly L_ℓ and L_r. By the partitioning property, we can easily show that $n_r - L_r \leq n_l$ and $n_\ell - L_\ell \leq n_r$. Also, $L_k \leq L_\ell$ (resp., L_r). Combining these, $2(n_r - L_r) \leq n_\ell + n_r - L_r \leq n_k - L_k$ (similar is true for $n_\ell - L_\ell$). Thus, we get that the quantity $n_k - L_k$ goes down by at least the factor of half as we go to a child node.

Thus, $n_k - L_k \leq n/2^{\delta_k}$. Now for any node which has at least two leaves in its subtree, $L_k \leq (1/2)n_k$ and thus $n_k \leq 2n/2^{\delta_k}$. Taking leaf nodes into account, we get $n_k \leq 4n/2^{\delta_k}$. This completes the proof of Lemma 1.

B Space of a WBT

Lemma 5. *The space of a weight-balanced wavelet tree of an array A of size n is $n(H_0(A)+2)(1+o(1))$ bits, where $H_0(A)$ is the 0th-order empirical entropy of A.*

Proof. Let the depth of a leaf corresponding to the symbol σ_i be δ_i. Then σ_i contributes f_i bits in each bit-vector corresponding to the nodes from root to this leaf (excluding the leaf). Hence the contribution of σ_i towards the total space is $f_i \cdot \delta_i$. By Lemma 1, $\delta_i \leq \log(4n/f_i)$. Therefore, the total size of a weight-balanced wavelet tree is at most $(1+o(1)) \sum f_i \cdot (\log(n/f_i)+2) = n(H_0(A)+2)(1+o(1))$ bits. This completes the proof of Lemma 5. $\square$

Cross-Document Pattern Matching

Gregory Kucherov[1], Yakov Nekrich[2], and Tatiana Starikovskaya[3,1]

[1] Laboratoire d'Informatique Gaspard Monge, Université Paris-Est & CNRS, Marne-la-Vallée, Paris, France
Gregory.Kucherov@univ-mlv.fr

[2] Department of Computer Science, University of Chile, Santiago, Chile
yakov.nekrich@googlemail.com

[3] Lomonosov Moscow State University, Moscow, Russia
tat.starikovskaya@gmail.com

Abstract. We study a new variant of the string matching problem called *cross-document string matching*, which is the problem of indexing a collection of documents to support an efficient search for a pattern in a selected document, where the pattern itself is a substring of another document. Several variants of this problem are considered, and efficient linear-space solutions are proposed with query time bounds that either do not depend at all on the pattern size or depend on it in a very limited way (doubly logarithmic). As a side result, we propose an improved solution to the *weighted level ancestor* problem.

1 Introduction

In this paper we study the following variant of the string matching problem that we call *cross-document string matching*: given a collection of strings (documents) stored in a "database", we want to be able to efficiently search for a pattern in a given document, where the pattern itself is a substring of another document. More formally, assuming we have a set of documents $T_1, \ldots, T_m$, we want to answer queries about the occurrences of a substring $T_k[i..j]$ in a document T_ℓ.

This scenario may occur in various situations when we have to search for a pattern in a text stored in a database, and the pattern is itself drawn from a string from the same database. In bioinformatics, for example, a typical project deals with a selection of genomic sequences, such as a family of genomes of evolutionary related species. A common repetitive task consists then in looking for genomic elements belonging to one of the sequences in some other sequences. These elements may correspond to genes, exons, mobile elements of any kind, regulatory patterns, etc., and their location (i.e. start and end positions) in the sequence of origin is usually known from a genome annotation provided by a sequence data repository (such as GenBank or any other). A similar scenario may occur in other application fields, such as the bibliographic search for example.

In this paper, we study different versions of the cross-document string matching problem. First, we distinguish between counting and reporting queries, asking respectively about the number of occurrences of $T_k[i..j]$ in T_ℓ or about the occurrences themselves. The two query types lead to slightly different solutions.

In particular, the counting problem uses the *weighted level ancestor* problem [1,2] to which we propose a new solution with an improved complexity bound.

We further consider different variants of the two problems. The first one is the dynamic version where new documents can be added to the database. In another variant, called *document counting and reporting*, we only need to respectively count or report the documents containing the pattern, rather than counting or reporting pattern occurrences within a given document. This version is very close to the *document retrieval problem* previously studied (see [3] and later papers referring to it), with the difference that in our case the pattern is itself selected from the documents stored in the database. Finally, we also consider *succinct* data structures for the above problems, where we keep involved index data structure in compressed form.

Let m be the number of stored strings and n the total length of all strings. Our results are summarized below:

(i) for the counting problem, we propose a solution with query time $O(t + \log\log m)$, where $t = \min(\sqrt{\log \mathrm{occ}/\log\log \mathrm{occ}}, \log\log|P|)$, $P = T_k[i..j]$ is the searched substring and occ is the number of its occurrences in T_ℓ,

(ii) for the reporting problem, our solution outputs all the occurrences in time $O(\log\log m + \mathrm{occ})$,

(iii) in the dynamic case, when new documents can be dynamically added to the database, we are able to answer counting queries in time $O(\log n)$ and reporting queries in time $O(\log n + \mathrm{occ})$, whereas the updates take time $O(\log n)$ per character,

(iv) for the document counting and document reporting problems, our algorithms run in time $O(\log n)$ and $O(t + \mathrm{ndocs})$ respectively, where t is as above and ndocs is the number of reported documents,

(v) finally, we also present succinct data structures that support counting, reporting, and document reporting queries in cross-document scenario (see Theorems 6 and 7 in Section 4.3).

For problems (i)-(iv), the involved data structures occupy $O(n)$ space. Interestingly, in the cross-document scenario, the query times either do not depend at all on the pattern length or depend on it in a very limited (doubly logarithmic) way.

Throughout the paper positions in strings are numbered from 1. Notation $T[i..j]$ stands for the substrings $T[i]T[i+1]\dots T[j]$ of T, and $T[i..]$ denotes the suffix of T starting at position i.

2 Preliminaries

2.1 Basic Data Structures

We assume a basic knowledge of suffix trees and suffix arrays.

Besides using suffix trees for individual strings T_i, we will also be using the *generalized suffix tree* for a set of strings $T_1, T_2, \dots, T_m$ that can be viewed as the suffix tree for the string $T_1\$_1T_2\$_2\ldots T_m\$_m$. A leaf in a suffix tree for T_i is associated with a distinct suffix of T_i, and a leaf in the generalized suffix tree is associated with a suffix of some document T_i together with the index i of this document. We assume that for each node v of a suffix tree, the number n_v of leaves in the subtree rooted at v, as well as its string depth $d(v)$ can be recovered in constant time. Recall that the string depth $d(v)$ is the total length of strings labelling the edges along the path from the root to v.

We will also use the suffix arrays for individual documents as well as the *generalized suffix array* for strings $T_1, T_2, \ldots, T_m$. Each entry of the suffix array for T_i is associated with a distinct suffix of T_i and each entry of the generalized suffix array for $T_1, \ldots, T_m$ is associated with a suffix of some document T_i and the index i of the document the suffix comes from. We store these document indices in a separate array D, called *document array*, such that $D[i] = k$ if the i-th entry of the generalized suffix array for $T_1, \ldots, T_m$ corresponds to a suffix coming from T_k.

For each considered suffix array, we assume available, when needed, two auxiliary arrays: an inverted suffix array and another array, called the LCP-array, of longest common prefixes between each suffix and the preceding one in the lexicographic order.

2.2 Weighted Level Ancestor Problem

The *weighted level ancestor* problem, defined in [1], is a generalization of the level ancestor problem [4,5] for the case when tree edges are assigned positive weights.

Consider a rooted tree $\mathcal{T}$ whose edges are assigned positive integer weights. For a node w, let $weight(w)$ denote the total weight of the edges on the path from the root to w; $depth(w)$ denotes the usual tree depth of w.

A weighted level ancestor query $\mathrm{wla}(v, q)$ asks, given a node v and a positive integer q, for the ancestor w of v of minimal depth such that $weight(w) \geq q$ ($\mathrm{wla}(v, q)$ is undefined if there is no such node w).

Two previously known solutions [1,2] for weighted level ancestors problem achieve $O(\log\log W)$ query time using linear space, where W is the total weight of all tree edges. Our data structure also uses $O(n)$ space, but achieves a faster query time in many special cases. We prove the following result.

Theorem 1. *There exists an $O(n)$ space data structure that answers weighted ancestor query $\mathrm{wla}(v, q)$ in $O(\min(\sqrt{\log g/\log\log g}, \log\log q))$ time, where $g = \min(depth(\mathrm{wla}(v, q)), depth(v) - depth(\mathrm{wla}(v, q)))$.*

If every internal node is a branching node, we obtain the following corollary.

Corollary 1. *Suppose that every internal node in $\mathcal{T}$ has at least two children. There exists an $O(n)$ space data structure that finds $w = \mathrm{wla}(v, q)$ in $O(\sqrt{\log n_w/\log\log n_w})$ time, where n_w is the number of leaves in the subtree of w.*

Our approach combines the heavy path decomposition technique of [2] with efficient data structures for finger searching in a set of integers. The proof is given in the Appendix.

3 Cross-Document Pattern Counting and Reporting

3.1 Counting

In this section we consider the problem of counting occurrences of a pattern $T_k[i..j]$ in a document T_ℓ.

Our data structure consists of the generalized suffix array GSA for documents $T_1, \ldots, T_m$ and individual suffix trees $\mathcal{T}_i$ for every document T_i.

For every suffix tree $\mathcal{T}_\ell$ we store a data structure of Theorem 1 supporting weighted level ancestor queries on $\mathcal{T}_\ell$. We also augment the document array D with an $O(n)$-space data structure that answers queries $\mathit{rank}(k, i)$ (number of entries storing k before position i in D) and $\mathit{select}(k, i)$ (i-th entry from the left storing k). Using the result of [6], we can support such rank and select queries in $O(\log\log m)$ and $O(1)$ time respectively. Moreover, we construct a data structure that answers range minima queries (RMQ) on the LCP array: for any $1 \le r_1 \le r_2 \le n$, find the minimum among $LCP[r_1], \ldots LCP[r_2]$. There exists a linear space RMQ data structure that supports queries in constant time, see e.g., [7]. An RMQ query on the LCP array computes the length of the longest common prefix of two suffixes $GSA[r_1]$ and $GSA[r_2]$, denoted $LCP(r_1, r_2)$.

Our counting algorithm consists of two stages. First, using GSA, we identify a position p of T_ℓ at which the query pattern $T_k[i..j]$ occurs, or determine that no such p exists. Then we find the locus of $T_k[i..j]$ in the suffix tree $\mathcal{T}_\ell$ using a weighted ancestor query.

Let r be the position of $T_k[i..]$ in the GSA. We find indexes $r_1 = \mathit{select}(\ell, \mathit{rank}(r, \ell))$ and $r_2 = \mathit{select}(\ell, \mathit{rank}(r, \ell) + 1)$ in $O(\log\log m)$ time. $GSA[r_1]$ (resp. $GSA[r_2]$) is the closest suffix from document T_ℓ that precedes (resp. follows) $T_k[i..]$ in the lexicographic order of suffixes. Observe now that $T_k[i..j]$ occurs in T_ℓ if and only if either $LCP(r_1, r)$ or $LCP(r, r_2)$ (or both) is no less than $j - i + 1$. If this holds, then the starting position p of $GSA[r_1]$ (respectively, starting position of $GSA[r_2]$) is the position of $T_k[i..j]$ in T_ℓ. Once such a position p is found, we jump to the leaf v of $\mathcal{T}_\ell$ that contains the suffix $T_\ell[p..]$.

The weighted level ancestor $u = \mathrm{wla}(v, (j - i + 1))$ is the locus of $T_k[i..j]$ in $\mathcal{T}_\ell$. This is because $T_\ell[p..p + j - i] = T_k[i..j]$. By Corollary 1, we can find node u in $O(\sqrt{\log n_u / \log\log n_u})$ time, where n_u is the number of leaf descendants of u. Since u is the locus node of $T_k[i..j]$, n_u is the number of occurrences of $T_k[i..j]$ in T_ℓ. By Theorem 1, we can find u in $O(\log\log(j - i + 1))$ time.

Summing up, we obtain the following Theorem.

Theorem 2. *For any* $1 \le k, \ell \le m$ *and* $1 \le i \le j \le |T_k|$*, we can count the number of occurrences of* $T_k[i..j]$ *in* T_ℓ *in* $O(t + \log\log m)$ *time, where* $t = \min(\sqrt{\log \mathrm{occ} / \log\log \mathrm{occ}}, \log\log(j - i + 1))$ *and* occ *is the number of occurrences.*

The underlying indexing structure takes $O(n)$ space and can be constructed in $O(n)$ time.

3.2 Reporting

A reporting query asks for all occurrences of a substring $T_k[i..j]$ in T_ℓ.

Compared to counting queries, we make a slight change in the data structures: instead of using suffix trees for individual documents T_i, we use suffix arrays. The rest of the data structures is unchanged.

We first find an occurrence of $T_k[i..j]$ in T_ℓ (if there is one) with the method described in Section 3.1. Let p be the position of this occurrence in T_ℓ. We then jump to the corresponding entry r of the suffix array SA_ℓ for the document T_ℓ. Let LCP_ℓ be the LCP-array of SA_ℓ. Starting with entry r, we visit adjacent entries t of SA_ℓ moving both to the left and to the right as long as $LCP_\ell[t] \geq j-i+1$. While this holds, we report $SA_\ell[t]$ as an occurrence of $T_k[i..j]$. It is easy to observe that the procedure is correct and that no occurrence is missing. As a result, we obtain the following theorem.

Theorem 3. *All the occurrences of $T_k[i..j]$ in T_ℓ can be reported in $O(\log\log m + \text{occ})$ time, where occ is the number of occurrences. The underlying indexing structure takes $O(n)$ space and can be constructed in $O(n)$ time.*

4 Variants of the Problem

4.1 Dynamic Counting and Reporting

In this section we focus on a *dynamic version* of counting and reporting problems, where the only dynamic operation consists in *adding a document to the database*[1].

Recall that in the static case, counting occurrences of $T_k[i..j]$ in T_ℓ is done through the following two steps (Section 3.1):

1. compute position p of some occurrence of $T_k[i..j]$ in T_ℓ,
2. in the suffix tree of T_ℓ, find the locus of string $T_\ell[p..p+j-i]$, and retrieve the number of leaves in the subtree rooted at u.

For reporting queries (Section 3.2), Step 1 is basically the same, while Step 2 is different and uses an individual suffix array for T_ℓ.

In the dynamic framework, we follow the same general two-step scenario. Note first that since Step 2, for both counting and reporting, uses data structures for individual documents only, it trivially applies to the dynamic case without changes. However, Step 1 requires serious modifications that we describe below.

Since the suffix array is not well-suited for dynamic updates, at Step 1 we will use the generalized suffix tree for $T_1, T_2, \ldots, T_m$ hereafter denoted GST. For

[1] Document deletions are also possible to support but require some additional constructions that are left to the extended version of this paper.

each suffix of $T_1, T_2, \ldots, T_m$ we store a pointer to the leaf of GST corresponding to this suffix.

We maintain a dynamic doubly-linked list EL corresponding to the Euler tour of the current GST. Each internal node of GST is stored in two copies in EL, corresponding respectively to the first and last visits of the node during the Euler tour. Leaves of GST are kept in one copy. Observe that the leaves of GST appear in EL in the "left-to-right" order, although not consecutively.

On EL, we maintain the data structure of [8] which allows, given two list elements, to determine their order in the list in $O(1)$ time (see also [9]). Insertions of elements in the list are supported in $O(1)$ time too.

Furthermore, we maintain a balanced tree, denoted BT, whose leaves are elements of EL. Note that the size of EL is bounded by $2n$ (n is the size of GST) and the height of BT is $O(\log n)$. Since the leaves of GST are a subset of the leaves of BT, we call them *suffix leaves* to avoid the ambiguity.

Each internal node u of BT stores two kinds of information: (i) the rightmost and leftmost suffix leaves in the subtree of BT rooted at u, (ii) minimal LCP value among all suffix leaves in the subtree of BT rooted at u.

Finally, we will also need an individual suffix array for each inserted document T_i.

We are now in position to describe the algorithm of Step 1. Like in the static case, we first retrieve the leaf of GST corresponding to suffix $T_k[i..]$. To identify a position of an occurrence of $T_k[i..j]$ in T_ℓ, we have to examine the two closest elements in the list of leaves of GST, one from right and from left, corresponding to suffixes of T_ℓ. To find these two suffixes, we perform a binary search on the suffix array for T_ℓ using order queries of [8] on EL. This step takes $O(\log |T_\ell|)$ time.

We then check if at least one of these two suffixes corresponds to an occurrence of $T_k[i..j]$ in T_ℓ. In a similar way to Section 3, we have to compute the longest common prefix between each of these two suffixes and $T_k[i..]$, and compare this value with $(j-i+1)$. This amounts to computing the minimal LCP value among all the suffixes of the corresponding range.

This can be done in $O(\log n)$ time by using a standard range trees approach [10]: for any sublist of EL we can retrieve $O(\log n)$ nodes v_i that cover it. The least among all minimal LCP values stored in nodes v_i is the minimal LCP value for the specified range of suffixes.

The query time bounds are summarized in the following lemma.

Lemma 1. *Using the above data structures, counting and reporting all occurrences of $T_k[i..j]$ in T_ℓ can be done respectively in time $O(\log n)$ and time $O(\log n + \text{occ})$, where* occ *is the number of reported occurrences.*

We now explain how the involved data structures are updated. Suppose that we add a new document T_{m+1}. Extending the generalized suffix tree by T_{m+1} is done in time $O(|T_{m+1}|)$ by McCreight's or Ukkonen's algorithm, i.e. in $O(1)$ amortized time per symbol.

When a new node v is added to a suffix tree, the following updates should be done (in order):

(i) insert v at the right place of the list EL (in two copies if v is an internal node),
(ii) rebalance the tree BT if needed,
(iii) if v is a leaf of GST (i.e. a suffix leaf of BT), update LCP values and rightmost/leftmost suffix leaf information in BT,

To see how update (i) works, we have to recall how suffix tree is updated when a new document is inserted. Two possible updates are creation of a new internal node v by splitting an edge into two (edge subdivision) and creating a new leaf u as a child of an existing node. In the first case, we insert the first copy of v right after the first copy of its parent, and the second copy right before the second copy of its parent. In the second case, the parent of u has already at least one child, and we insert u either right after the second (or the only) copy of its left sibling, or right before the first (or the only) copy of its right sibling.

Rebalancing the tree BT (update (ii)) is done using standard methods. Observe that during the rebalancing we may have to adjust the LCP and rightmost/leftmost suffix leaf information for internal nodes, but this is easy to do as only a constant number of local modifications is done at each level.

Update (iii) is triggered when a new leaf u is created in GST and added to EL. First of all, we have to compute the LCP value for u and possibly to update the LCP value of the next suffix leaf u' to the right of u in EL. This is done in $O(1)$ time as follows. At the moment when u is created, we memorize the string depth of its parent $D = d(parent(u))$. Recall that the parent of u already has at least one child before u is created. If u is neither the leftmost nor the rightmost child of its parent, then we set $LCP(u) = D$ and $LCP(u')$ remains unchanged (actually it also equals D). If u is the leftmost child of its parent, then we set $LCP(u) = LCP(u')$ and then $LCP(u') = D$. Finally, if u is the rightmost child, then $LCP(u) = D$ and $LCP(u')$ remains unchanged.

We then have to follow the path in BT from the new leaf u to the root and possibly update the LCP and rightmost/leftmost suffix leaf information for all nodes on this path. These updates are straightforward. Furthermore, during this traversal we also identify suffix leaf u' (as the leftmost child of the first right sibling encountered during the traversal), update its LCP value and, if necessary, the LCP values on the path from u' to the root of BT. All these steps take time $O(\log n)$.

Thus, updates of all involved data structures take $O(\log n)$ time per symbol. The following theorem summarizes the results of this section.

Theorem 4. *In the case when documents can be added dynamically, the number of occurrences of $T_k[i..j]$ in T_ℓ can be computed in time $O(\log n)$ and reporting these occurrences can be done in time $O(\log n + occ)$, where occ is their number. The underlying data structure occupies $O(n)$ space and an update takes $O(\log n)$ time per character.*

4.2 Document Counting and Reporting

Consider a static collection of documents $T_1, \ldots, T_m$. In this section we focus on document reporting and counting queries: report or count the documents which contain at least one occurrence of $T_k[i..j]$, for some $1 \leq k \leq m$ and $i \leq j$.

For both counting and reporting, we use the generalized suffix tree, generalized suffix array and the document array D for $T_1, T_2, \ldots, T_m$. We first retrieve the leaf of the generalized suffix tree labelled by $T_k[i..]$ and compute its highest ancestor u of string depth at least $j - i + 1$, using the weighted level ancestor technique of Section 2.2. The suffixes of $T_1, T_2, \ldots, T_m$ starting with $T_k[i..j]$ (i.e. occurrences of $T_k[i..j]$) correspond then to the leaves of the subtree rooted at u, and vice versa. As shown in Section 3.1, this step takes $O(t)$ time, where $t = \min(\sqrt{\log \mathrm{occ} / \log\log \mathrm{occ}}, \log\log(j-i+1))$ and occ is the number of occurrences of $T_k[i..j]$ (this time in all documents).

Once u has been computed, we retrieve the interval $[left(u)..right(u)]$ of ranks of all the leaves under interest. We are then left with the problem of counting/reporting distinct values in $D[left(u)..right(u)]$. This problem is exactly the same as the color counting/ color reporting problem that has been studied extensively (see e.g., [11] and references therein).

For color reporting queries, we can use the solution of [3] based on an $O(n)$-space data structure for RMQ, applied to (a transform of) the document array D. The pre-processing time is $O(n)$. Each document is then reported in $O(1)$ time, i.e. all relevant documents are reported in $O(\mathrm{ndocs})$ time, where ndocs is their number. The whole reporting query then takes time $O(t + \mathrm{ndocs})$ for t defined above.

For counting, we use the solution described in [12]. The data structure requires $O(n)$ space and a color counting query takes $O(\log n)$ time. The following theorem presents a summary.

Theorem 5. *We can store a collection of documents $T_1, \ldots, T_m$ in a linear space data structure, so that for any pattern $P = T_k[i..j]$ all documents that contain P can be reported and counted in $O(t + \mathrm{ndocs})$ and $O(\log n)$ time respectively. Here $t = \min(\sqrt{\log \mathrm{occ} / \log\log \mathrm{occ}}, \log\log|P|)$, ndocs is the number of documents that contain P and occ is the number of occurrences of P in all documents.*

4.3 Compact Counting, Reporting and Document Reporting

In this section, we show how our reporting and counting problems can be solved on *succinct* data structures [13].

Reporting and Counting. Our compact solution is based on compressed suffix arrays [14]. A compressed suffix array for a text T uses $|CSA|$ bits of space and enables us to retrieve the position of the suffix of rank r, the rank of a suffix $T[i..]$, and the character $T[i]$ in time *Lookup*(n). Different trade-offs between space usage and query time can be achieved (see [13] for a survey).

Our data structure consists of a compressed generalized suffix array CSA for $T_1, \ldots, T_m$ and compressed suffix arrays CSA_i for each document T_i. In [15] it was shown that using $O(n)$ extra bits, the length of the longest common prefix of any two suffixes can be computed in $O(\mathit{Lookup}(n))$ time. Besides, the ranks of any two suffixes $T_k[s..]$ and $T_\ell[p..]$ can be compared in $O(\mathit{Lookup}(n))$ time: it suffices to compare $T_\ell[p+f]$ with $T_k[s+f]$ for $f = LCP(T_k[s..], T_\ell[p..])$.

Note that ranks of the suffixes of T_ℓ starting with $T_k[i..j]$ form an interval $[r_1..r_2]$. We use a binary search on the compressed suffix array of T_ℓ to find r_1 and r_2. At each step of the binary search we compare a suffix of T_ℓ with $T_k[i..]$. Therefore $[r_1..r_2]$ can be found in $O(\mathit{Lookup}(n) \cdot \log n)$ time. Obviously, the number of occurrences of $T_k[i..j]$ in T_ℓ is $r_2 - r_1$. To report the occurrences, we compute the suffixes of T_ℓ with ranks in interval $[r_1..r_2]$.

Theorem 6. *All occurrences of $T_k[i..j]$ in T_ℓ can be counted in $O(\mathit{Lookup}(n) \cdot \log n)$ time and reported in $O((\log n + \mathrm{occ})\mathit{Lookup}(n))$ time, where* occ *is the number of those. The underlying indexing structure takes $2|CSA| + O(n + m \log \frac{n}{m})$ bits of memory.*

Document Reporting. Again, we use a binary search on the generalized suffix array to find the rank interval $[r_1..r_2]$ of suffixes that start with $T_k[i..j]$. This can be done in $O(\mathit{Lookup}(n) \cdot \log n)$ time.

In [16], it was shown how to report, for any $1 \le r_1 \le r_2 \le n$, all distinct documents T_f such that at least one suffix of T_f occurs at position r, $r_1 \le r \le r_2$, of the generalized suffix array. The construction uses $O(n + m \log \frac{n}{m})$ additional bits, and all relevant documents are reported in $O(\mathit{Lookup}(n) \cdot \mathrm{ndocs})$ time, where ndocs is the number of documents that contain $T_k[i..j]$. Summing up, we obtain the following result.

Theorem 7. *All documents containing $T_k[i..j]$ can be reported in $O((\log n + \mathrm{ndocs})\mathit{Lookup}(n))$ time, where* ndocs *is the number of those. The underlying indexing structure takes $2|CSA| + O(n + m \log \frac{n}{m})$ bits of space.*

Acknowledgments. T.Starikovskaya has been supported by the mobility grant funded by the French Ministry of Foreign Affairs through the EGIDE agency and by a grant 10-01-93109-CNRS-a of the Russian Foundation for Basic Research. Part of this work has been done during a one-month stay of Y.Nekrich at the Marne-la-Vallée University supported by the BEZOUT grant of the French government. The authors also would like to thank Tsvi Kopelowitz for fruitful discussions and valuable comments.

References

1. Farach, M., Muthukrishnan, S.: Perfect Hashing for Strings: Formalization and Algorithms. In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 130–140. Springer, Heidelberg (1996)
2. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic text and static pattern matching. ACM Trans. Algorithms 3 (2007)

3. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002. Society for Industrial and Applied Mathematics, Philadelphia (2002)
4. Berkman, O., Vishkin, U.: Finding level-ancestors in trees. J. Comput. Syst. Sci. 48(2), 214–230 (1994)
5. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. 321(1), 5–12 (2004)
6. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 368–373. ACM Press (2006)
7. Bender, M.A., Farach-Colton, M.: The LCA Problem Revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
8. Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two Simplified Algorithms for Maintaining Order in a List. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 152–164. Springer, Heidelberg (2002)
9. Dietz, P., Sleator, D.: Two algorithms for maintaining order in a list. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC 1987, pp. 365–372. ACM, New York (1987)
10. Bentley, J.L.: Multidimensional divide-and-conquer. Commun. ACM 23(4), 214–229 (1980)
11. Gagie, T., Navarro, G., Puglisi, S.J.: Colored Range Queries and Document Retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 67–81. Springer, Heidelberg (2010)
12. Bozanis, P., Kitsios, N., Makris, C., Tsakalidis, A.: New Upper Bounds for Generalized Intersection Searching Problems. In: Fülöp, Z., Gécseg, F. (eds.) ICALP 1995. LNCS, vol. 944, pp. 464–474. Springer, Heidelberg (1995)
13. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. 39 (2007)
14. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, STOC 2000, pp. 397–406. ACM, New York (2000)
15. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. 41, 589–607 (2007)
16. Sadakane, K.: Succinct data structures for flexible text retrieval systems. J. of Discrete Algorithms 5, 12–22 (2007)
17. Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. J. Comput. Syst. Sci. 48(3), 533–551 (1994)
18. Andersson, A., Thorup, M.: Dynamic ordered sets with exponential search trees. J. ACM 54(3), 13 (2007)

Appendix: Proof of Theorem 1

Here we prove Theorem 1. We use the heavy path decomposition technique of [2].

A path π in $\mathcal{T}$ is heavy if every node u on π has at most twice as many nodes in its subtree as its child v on π. A tree $\mathcal{T}$ can be decomposed into paths using the following procedure: we find the longest heavy path π_r that starts at the root of $\mathcal{T}$ and remove all edges of π_r from $\mathcal{T}$. All remaining vertices of $\mathcal{T}$ belong to a forest; we recursively repeat the same procedure in every tree of that forest.

We can represent the decomposition into heavy paths using a tree $\mathbb{T}$. Each node $\mathbb{v}_j$ in $\mathbb{T}$ corresponds to a heavy path π_j in $\mathcal{T}$. A node $\mathbb{v}_j$ is a child of a node $\mathbb{v}_i$ in $\mathbb{T}$ if the head of π_j (i.e., the highest node in π_j) is a child of some node $u \in \pi_i$. Some node in π_i has at least twice as many descendants as each node in π_j; hence, $\mathbb{T}$ has height $O(\log n)$.

$O(n \log n)$-Space Solution. Let $\mathbb{p}_j$ denote a root-to-leaf path in $\mathbb{T}$. For a node $\mathbb{v}$ in $\mathbb{T}$ let $weight(\mathbb{v})$ denote the weight of the head of π, where π is the heavy path represented by $\mathbb{v}$ in $\mathbb{T}$. We store a data structure $D(\mathbb{p}_j)$ that contains the values of $weight(\mathbb{v})$ for all nodes $\mathbb{v} \in \mathbb{p}_j$. $D(\mathbb{p}_j)$ contains $O(\log n)$ elements; hence, we can find the highest node $\mathbb{v} \in \mathbb{p}_j$ such that $weight(\mathbb{v}) \geq q$ in $O(1)$ time. This can be achieved by storing the weights of all nodes from $\mathbb{p}_j$ in the q-heap [17].

For every heavy path π_j, we store the data structure $E(\pi_j)$ from [18] that contains the weights of all nodes $u \in \pi_j$ and supports the following queries: for an integer q, find the lightest node $u \in \pi_j$ such that $weight(u) \geq q$. Using Theorem 1.5 in [18], we can find such a node $u \in \pi_j$ in $O(\sqrt{\log n' / \log\log n'})$ time where $n' = \min(n_h, n_l)$, $n_h = |\{ v \in p_j \,|\, weight(v) > weight(u) \}|$, and $n_l = |\{ v \in p_j \,|\, weight(v) < weight(u) \}|$. Moreover, we can also find the node u in $O(\log\log q)$ time; we will show how this can be done in the full version of this paper. Thus $E(\pi_j)$ can be modified to support queries in $O(\min(\sqrt{\log n' / \log\log n'}, \log\log q))$ time.

For each node $u \in \mathcal{T}$ we store a pointer to the heavy path π that contains u and to the corresponding node $\mathbb{v} \in \mathbb{T}$.

A query $\text{wla}(v, q)$ can be answered as follows. Let $\mathbb{v}$ denote the node in $\mathbb{T}$ that corresponds to the heavy path containing v. Let $\mathbb{p}_j$ be an arbitrary root-to-leaf path in $\mathbb{T}$ that also contains $\mathbb{v}$. Using $D(\mathbb{p}_j)$ we can find the highest node $\mathbb{u} \in \mathbb{p}_j$, such that $weight(\mathbb{u}) \geq q$ in $O(1)$ time. Let π_t denote the heavy path in $\mathcal{T}$ that corresponds to the parent of $\mathbb{u}$, and π_s denote the path that corresponds to $\mathbb{u}$. If the weighted ancestor $\text{wla}(v, q)$ is not the head of π_s, then $\text{wla}(v, q)$ belongs to the path π_t. Using $E(\pi_t)$, we can find $u = \text{wla}(v, q)$ in $O(\min(\sqrt{\log n' / \log\log n'}, \log\log q))$ time where $n' = \min(n_h, n_l)$, $n_h = |\{ v \in \pi_t \,|\, weight(v) > weight(u) \}|$, and $n_l = |\{ v \in \pi_t \,|\, weight(v) < weight(u) \}|$.

All data structures $E(\pi_i)$ use linear space. Since there are $O(n)$ leaves in $\mathbb{T}$ and each path $\mathbb{p}_i$ contains $O(\log n)$ nodes, all $D(\mathbb{p}_i)$ use $O(n \log n)$ space.

Lemma 2. *There exists an $O(n \log n)$ space data structure that finds the weighted level ancestor u in $O(\min(\sqrt{\log n' / \log\log n'}, \log\log q))$ time.*

$O(n)$-Space Solution. We can reduce the space from $O(n \log n)$ to $O(n)$ using a micro-macro tree decomposition. Let $\mathcal{T}_0$ be a tree induced by the nodes of $\mathcal{T}$ that have at least $\log n/8$ descendants. The tree $\mathcal{T}_0$ has at most $O(n/\log n)$ leaves. We construct the data structure described above for $\mathcal{T}_0$; since $\mathcal{T}_0$ has $O(n/\log n)$ leaves, its heavy-path tree $\mathbb{T}_0$ also has $O(n/\log n)$ leaves. Therefore all structures $D(\mathbb{p}_j)$ use $O(n)$ words of space. All $E(\pi_i)$ also use $O(n)$ words of

space. If we remove all nodes of $\mathcal{T}_0$ from $\mathcal{T}$, the remaining forest $\mathcal{F}$ consists of $O(n)$ nodes. Every tree $\mathcal{T}_i$, $i \geq 1$, in $\mathcal{F}$ consists of $O(\log n)$ nodes. Nodes of $\mathcal{T}_i$ are stored in a data structure that uses linear space and answers weighted ancestor queries in $O(1)$ time. This data structure will be described later in this section.

Suppose that a weighted ancestor $\mathrm{wla}(v, q)$ should be found. If $v \in \mathcal{T}_0$, we answer the query using the data structure for $\mathcal{T}_0$. If v belongs to some $\mathcal{T}_i$ for $i \geq 1$, we check the weight w_r of $root(\mathcal{T}_i)$. If $w_r \leq q$, we search for $\mathrm{wla}(v, q)$ in $\mathcal{T}_i$. Otherwise we identify the parent v_1 of $root(\mathcal{T}_i)$ and find $\mathrm{wla}(v_1, q)$ in $\mathcal{T}_0$. If $\mathrm{wla}(v_1, q)$ in $\mathcal{T}_0$ is undefined, then $\mathrm{wla}(v, q) = root(\mathcal{T}_i)$.

Data Structure for a Small Tree. It remains to describe the data structure for a tree $\mathcal{T}_i$, $i \geq 1$. Since $\mathcal{T}_i$ contains a small number of nodes, we can answer weighted level ancestor queries on $\mathcal{T}_i$ using a look-up table V. V contains information about any tree with up to $\log n/8$ nodes, such that node weights are positive integers bounded by $\log n/8$. For any such tree $\widetilde{\mathcal{T}}$, for any node v of $\widetilde{\mathcal{T}}$, and for any integer $q \in [1, \log n/8]$, we store the pointer to $\mathrm{wla}(v, q)$ in $\widetilde{\mathcal{T}}$. There are $O(2^{\log n/4})$ different trees $\widetilde{\mathcal{T}}$ (see e.g., [5] for a simple proof); for any $\widetilde{\mathcal{T}}$, we can assign weights to nodes in less than $(\log n/8)!$ ways. For any weighted tree $\widetilde{\mathcal{T}}$ there are at most $(\log n)^2/64$ different pairs v, q. Hence, the table V contains $O(2^{\log n/4}(\log n)^2(\log n/8)!) = o(n)$ entries. We need only one look-up table V for all mini-trees $\mathcal{T}_i$.

We can now answer a weighted level ancestor query on $\mathcal{T}_i$ using reduction to rank space. The *rank* of a node u in a tree $\mathcal{T}$ is defined as $\mathrm{rank}(u, \mathcal{T}) = |\{\, v \in \mathcal{T} \mid weight(v) \leq weight(u) \,\}|$. The successor of an integer q in a tree $\mathcal{T}$ is the lightest node $u \in \mathcal{T}$ such that $weight(u) \geq q$. The rank $\mathrm{rank}(q, \mathcal{T})$ of an integer q is defined as the rank of its successor. Let $\mathrm{rank}(\mathcal{T})$ denote the tree $\mathcal{T}$ in which the weight of every node is replaced with its rank. The weight of a node $u \in \mathcal{T}$ is not smaller than q if an only if $\mathrm{rank}(u, \mathcal{T}) \geq \mathrm{rank}(q, \mathcal{T})$. Therefore we can find $\mathrm{wla}(v, q)$ in a small tree $\mathcal{T}_i$, $i \geq 1$, as follows. For every $\mathcal{T}_i$ we store a pointer to $\widetilde{\mathcal{T}}_i = \mathrm{rank}(\mathcal{T}_i)$. Given a query $\mathrm{wla}(v, q)$, we find $\mathrm{rank}(q, \mathcal{T}_i)$ in $O(1)$ time using a q-heap [17]. Let v' be the node in $\widetilde{\mathcal{T}}_i$ that corresponds to the node v. We find $u' = \mathrm{wla}(v', \mathrm{rank}(q, \mathcal{T}_i))$ in $\widetilde{\mathcal{T}}_i$ using the table V. Then the node u in $\mathcal{T}_i$ that corresponds to u' is the weighted level ancestor of v.

FEMTO: Fast Search of Large Sequence Collections

Michael P. Ferguson

Laboratory for Telecommunications Sciences, College Park, Maryland
mferguson@ltsnet.net

Abstract. We present FEMTO, a new system for indexing and searching large collections of sequence data. We used FEMTO to index and search three large collections, including one 182 GB collection. We compare the performance of FEMTO indexing and search with Bowtie and with Lucene, and we compare performance with indexes stored on hard disks and in flash memory. To our knowledge, we report on the first compressed suffix array storing more than 100 GB. Even for the largest collection, most searches completed in under 10 seconds.

Keywords: FM-index, document retrieval, external memory, regular expression, compressed suffix array.

1 Introduction

The FM-index of Ferragina and Manzini [8–10] is a remarkable structure for string searching because it supports $O(m)$ search, where m is the length of the search pattern, while typically using less space than the original data. Furthermore, the FM-index for n bytes of data can be constructed in $O(n)$ time. However, in practice, an FM-index - as well as other compressed suffix arrays - must compete with more established techniques, such as Boyer-Moore [3] or *grep*, which perform an $O(n)$ scan of the data.

There are three problems with the typical compressed suffix array that make it a poor choice for many information retrieval problems. The first problem is that most implementations are not capable of handling very large collections. They assume that the index fits into main memory or that 4 bytes are sufficient to store offsets. Furthermore, a straightforward implementation of the FM-index for hard disks will have poor performance since each search step requires millisecond-long random access to the index.

The second problem is that, when compared to a tool like *grep*, compressed suffix arrays offer limited flexibility. The implementations only return counts or offsets. At the same time, many uses of an index only need a list of matching documents. In addition, compressed suffix arrays only allow string search; there is no support for regular expressions.

The third problem is that the indexing procedure is extremely slow for large collections. Collections that cannot be indexed in main memory must be indexed on disk. Dementiev et al. demonstrated external-memory suffix array construction at 9.88 microseconds/byte, or about 100 KB/second [6]. Compare that rate with the speed of scanning from disk - easily about 70 MiB/second [1].

We believe that we have made progress in solving these three problems. We have created a system called FEMTO: the **F**M-index for **E**xternal **M**emory with **T**hroughput **O**ptimizations. It is an FM-index with some improvements to make external memory operation faster. Our goal is to create an FM-index that can compete better with the scanning approach for large, archival collections. FEMTO minimizes seeks to perform well in external memory, supports more flexible usage including document retrieval and regular expression search, and constructs indexes in parallel and in external memory at rates exceeding 1 MiB/second with 4 machines.

2 Background

We summarize the operation of an FM-index as in [10]. First, to index any number of documents, concatenate the documents together to create an n-byte text string T. The next step in the indexing process is to perform the Burrows-Wheeler transform (BWT) on T [4]. Conceptually, form an n by n matrix from all the rotations of the n-byte input string, and sort the rows of this matrix lexicographically. This sorted matrix is called M. The BWT of the original n-byte string is the last column of M, which is called L. The original input T can be reconstructed from the L column using the *last-to-first column mapping* [4]. This mapping, denoted LF, provides a mechanism for stepping backwards in the text. That is, if row r begins with $T[i..]$, $LF(r)$ will give the index of the row beginning with $T[i-1..]$. The LF mapping can be computed entirely from information in the L column. In particular, $LF(i) = C[L[i]] + Occ(L[i], i) - 1$, where C stores, for each character, the number of characters less than that character in the entire input, and $Occ(ch, i)$ is the number of times the character ch appears at or before row i in the L column.

Note that in practice, the BWT is computed by first building the suffix array of the input string. It is straightforward to compute the L column from the suffix array [4].

An FM-index stores the L column, the C array, information to help compute $Occ(L[i], i)$, and - for some fraction of *marked* rows - offset information. The FM-index uses these structures to support two kinds of searches: *count* and *locate*. Given a pattern of m bytes, the *count* search works in $O(m)$ steps to find the rows in the conceptual matrix M that begin with that pattern. The *count* search simply returns the number of matching rows. A *locate* search finds the offset for each matching row. Both of these methods find the range of rows beginning with a particular pattern with the *backward_search* method of Ferragina and Manzini [10], which goes through the pattern backwards, maintaining the range of matching rows, and uses a variant on the LF-mapping. To find the offset for

[1] In this paper, we use the IEC binary prefixes: MB and GB mean 10^6 and 10^9 bytes; MiB and GiB mean 2^{20} and 2^{30} bytes.

a particular row, an FM-index applies the LF-mapping until a marked row is reached [10]. We call each application of the LF-mapping a *backward step.*

3 The FEMTO Index

The FEMTO index is based on a combination of methods, including the FM-index [8, 10] and the Compressed Suffix Array [13]. Like other indexes, the FEMTO index is made up of blocks. A FEMTO index has a single header block and many data blocks. The header block stores the C array and the number of occurrences of each character before the start of each data block. Each data block contains buckets storing the L column and marking information as well as the number of times each character appears before each bucket in the block. The L column data is stored in a wavelet tree inside each bucket. This wavelet tree structure provides a fast way of computing $Occ(ch, i)$. We use our own implementation of the wavelet tree described by Grossi, Gupta, and Vitter [13]. FEMTO marks character offsets i with $i \pmod{k} = 0$, for some parameter k known as the *mark period.* FEMTO uses a succinct dictionary [13] to record which rows are marked and stores the offsets for marked rows in a separate array. Lastly, for each chunk of h rows, FEMTO stores the corresponding set of document numbers.

3.1 High Throughput with External Memory

We assume that at any given time, FEMTO is processing several queries in parallel. Each of these queries can be expressed as a sequence of *requests.* These requests find a small unit of information - such as $Occ(L[i], i)$. In order to service one of these requests, the system must read the appropriate bucket from disk. Once the bucket is loaded, the system can service any number of requests for that bucket with no extra I/O cost. The search system keeps the requests in a tree structure sorted by row. That way, it visits requests in row order in order to avoid needing to re-read parts of the index. In addition, FEMTO processes the sorted requests in multiple threads using a work-stealing strategy.

This method is new as suggested; compressed suffix array implementations typically assume that they will process a single query at a time. Two implementations use related techniques to reduce the search time for a single query, as in Grossi and Vitter [14] for reporting occurrences and in the FM-index version 2 for doing count queries with the mark character inserted [9].

Sorting requests provides an enormous reduction in the number of block cache misses. After enabling request sorting during development, we observed about a 5000-fold reduction in the amount of I/O to perform 100,000 queries.

3.2 Bi-directional Locate

During a locate operation, FEMTO steps both forward and backward at the same time to find a marked row. Backward step works in the usual manner, but forward step uses binary search on the header block and on a data block. These forward and backward steps are continued independently until a marked row is located. As a row range is stepped forward or backward, it becomes more and more spread across the index. By stepping both forwards and backwards, FEMTO reduces this spreading, leading to better cache performance. Note that the complexity of this locate procedure remains $O(k)$ index operations, assuming that every k^{th} character is marked. Early experiments showed that bi-directional locate incurs about half the I/O of backward steps.

3.3 Improving Document Search Time

FEMTO employs a new scheme to improve the performance of queries for matching documents, which we call *locate_documents* queries. These queries do not need to return the locations of matches within the documents - only the set of matching documents. To improve these searches, FEMTO divides the index into chunks of h rows. For each chunk, it stores a list of documents present in that range of rows. When reporting the documents contained in an arbitrary range of rows, FEMTO does the normal locate procedure for the results in the partial first and last chunks. For each full chunk, it reads the set of matching documents directly. The list of document numbers stored in each chunk is compressed using standard information retrieval methods as in [21].

Assuming that each chunk can be read in a single block operation, the number of block operations to locate the documents represented by r matching rows is $O(r/h + hk)$, where h is the chunk size and k is the mark period. This is an improvement over the original $O(rk)$ search time, and in practice it can mean 10-200x speedup when returning a large number of results.

These measures to improve document search time, especially when reporting a very large number of documents, are important for applications that require the entire list of matching documents, such as Boolean query processing. An alternative approach would be to store the document numbers for each row in a wavelet tree as described in [12]. In addition, Culpepper et al. extend that result in order to directly compute the top k results, ranked by term frequency [5]. Although we were not able to perform a direct comparison, we believe that the wavelet tree approaches create larger indexes - on the order of $3n$ bytes - while the FEMTO indexes range from $n/2$ to $2n$ bytes.

3.4 Regular Expression Search

We have developed a regular expression search method for the FM-index which is an analogue of the trie automaton search of Baeza-Yates et al. [1]. First, take a regular expression and reverse it - e.g. *ab(c*d |ef)g* becomes *g(dc* |fe)ba*. Then, compile it to a nondeterministic finite automaton (NFA) using Thompson's method [20]. Next, simulate the NFA on the compressed suffix array. The simulation operates on a mapping from ranges of rows to automaton states. Each step of the algorithm proceeds in a similar manner to the *backward_search*; take a backwards step for each character on a transition from a current NFA state and then add the resulting range of rows and state to the mapping.

```
simulate_nfa :
  add the entry ([0, n-1] -> the set of start states) to mapping
  while( the mapping is not empty ) :
    pop an entry ([first, last] -> nfa_states) from the mapping
    for every character ch reachable from nfa_states :
      new_first = C[ch] + Occ(ch, first - 1)
      new_last = C[ch] + Occ(ch, last) - 1
      new_states=states reachable from nfa_states after reading ch
      add ([new_first, new_last] -> new_states) to the mapping,
         reporting a match if a final state is set
```

This algorithm has some desirable properties. First, exact string search proceeds as in *backward_search*. Second, this algorithm is able to re-use a partial match; for example when searching for $(abc|def)xyz$, the algorithm only computes the range of rows matching xyz once. Note that this algorithm performs at least z index operations, where z is the length of the longest match; thus it is $\Omega(z)$. Furthermore, some regular expressions, such as the infinite wildcard $.*$, will cause the algorithm to visit every row in the index. Although this algorithm is not appropriate for all regular expressions, it performs well with simple regular expressions. Lastly, it is possible to modify this algorithm to support approximate queries by simulating approximate NFA search, although space does not allow us to describe it in detail here. Our implementation of approximate NFA search is based upon the approximate regular expression search of Wu and Manber [22], but interacts with the index just like *simulate_nfa* above.

3.5 Parallel External Memory Index Construction

We developed a parallel external-memory suffix array construction system, based upon the difference cover algorithm [16]. Our independent implementation combines the ideas of [17] and [6]. We observed linear performance indexing the English corpus from the Pizza & Chili website [11]. In a 10-gigabit Ethernet cluster using 3 disks per node, we measured 0.45 MiB/s on a single machine, 1.61 MiB/s on four machines, and 2.7 MiB/s on eight machines.

There are many notable alternative ways to construct an FM-index. First, there are direct index construction schemes, such as [15] and [19], that require memory space for the resulting compressed index. Also, Ferragina et al. present in [7] a many-pass method to compute the BWT in external memory. Since this many-pass method has I/O complexity $O(n^2/(MB \log n))$, it is only appropriate for collections that are not much larger than main memory. Lastly, [2] shows how to efficiently index many small documents, but their method works with fixed-size documents and is quadratic in the document size - and so not applicable to a large, varied collection. Nonetheless, we believe that future work along these lines may lead to a generally useful direct-to-BWT external memory scheme.

4 Experimental Methods

We report experiments with three different large data sets. The first is a collection of bacterial sequence data from the National Center for Biotechnology Information (NCBI) Genomes collection - totaling 3.8 GB. The second collection is 43 GB containing all textual documents from Project Gutenberg collection. The final collection is all 182 GB of sequence data from the NCBI Genomes collection. To our knowledge, this is the first time that compressed suffix arrays have been constructed for such large collections.

To construct our Gutenberg dataset, we created a local mirror of the Project Gutenberg collection (http://www.gutenberg.org) in November 2011 and then we selected only documents that the *file* command reported as text. We indexed this collection with Apache Lucene Core 3.5.0 (http://lucene.apache.org) for comparison. We modified the Lucene searching demonstration to give performance timing and to only ever return 10 results. When searching for a pattern containing multiple words, we used the phrase query mechanism in Lucene.

To construct the Genomes collection, we downloaded all .fna, .fa, .fasta, .seq, .fsa, .ffn, .faa, .frn, and gzip compressed versions of those same files, in November 2011, from ftp://ftp.ncbi.nih.gov/genomes. We then removed redundant compressed files as well as symbolic links and ARCHIVE directories and uncompressed the compressed files.

We constructed the Bacteria collection by taking files from the Bacteria subdirectory of our Genomes collection, up to 3.76 GB. We limited the size in order to compare with Bowtie2 (available at http://bowtie-bio.sourceforge.net) [18]. Bowtie2 cannot currently create an index for more than 4 GiB of input data. We used Bowtie 2.0.0-beta3. When searching with Bowtie, we used the arguments *–mm –sam-nosq –end-to-end -M 0 -N 0 -L 500 -i L,0,500* in addition to the *-x* argument to specify the index and the *-c* argument to specify a search pattern. We used these arguments in order to request exact matches and to request that the index be memory mapped instead of read in at program start.

FEMTO indexes were constructed with a block size of 128 MiB, a bucket size of 1MiB, a document chunk size of $h = 2048$, and a mark period of $k = 20$.

Because we are measuring external memory performance, we flushed the Linux page cache between each experiment and performed an operation to clear out hardware RAID and disk buffers. Then we performed a search for an unrelated term in order to make sure that any index headers or program images were in cache. Finally, we performed the measured search. Thus, these search performance numbers represent the time to get results when the search program is cached but almost all of the index is not.

We performed experiments on two different systems. The first has an 8-disk hardware RAID-6 array, 8-cores of Intel Xeon X5355 at 2.66 GHz, and 16 GiB of memory. The second system has 8 flash memory devices configured in a hardware RAID-5 array, 24 cores of Xeon X5670 at 2.93 GHz, and 24 GiB of memory.

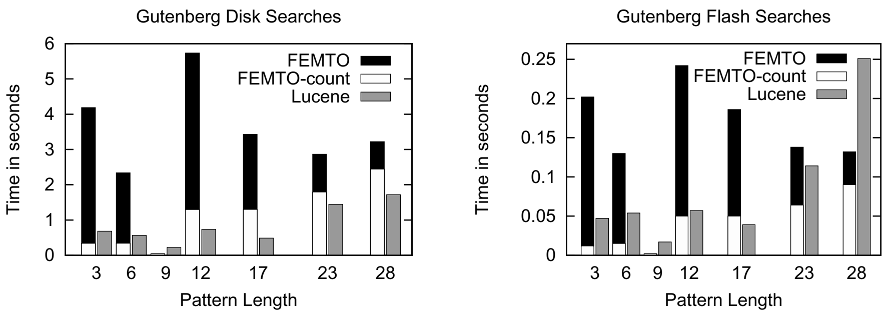


Fig. 1. Gutenberg (42.94 GB) indexed search speeds. Note that the time to search with *grep* is 559 s on disk, or 221 s on flash memory. Each query is limited to 10 results.

4.1 Results

FEMTO, Lucene, and Bowtie2 all create indexes for completely different purposes: FEMTO is meant to support large, archival indexing and search of any sequence data; Lucene indexes word data; and Bowtie2 was created for sequence alignment. Nonetheless, it is useful to compare these systems to understand, as a practical matter, in what situations a system such as FEMTO should be used. Table 1 summarize index size and indexing time.

Table 1. Data set and index construction information. Bacteria was indexed with Bowtie2, Gutenberg with Lucene. Genomes indexed with 7 machines. FEMTO indexes sizes show optional document chunk structure; see Section 4.3.

Corpus Information			Indexing time and index size				
	Size	# Docs	Bowtie2/Lucene		FEMTO		
Bacteria	3.76 GB	1896	16940 s	5.26 GB	7572 s	1.91 GB -	2.34 GB
Gutenberg	42.93 GB	94471	5133 s	14.84 GB	108025 s	19.83 GB -	40.87 GB
Genomes	182.43 GB	23242528	-		158434 s*	87.62 GB -	296.68 GB

Figures 1-3 show external memory search performance. Figure 1 shows that FEMTO is not as fast as Lucene, but it is not far behind. Most patterns were one word and returned exactly 10 matches. Note that the pattern of length 9 had no matches, the pattern of length 28 had only 3 matches, and that the 23-long pattern is two words and the 28-long pattern is three words. Lucene search time depends on the number of words in the search - hence it increases for the last two - while a FEMTO search depends more directly on the number of characters. Observe that the time to identify matching rows - shown in the figure in the white area labeled FEMTO-count - increases as the pattern size grows. Searches backed by flash memory are noticeably faster for both systems, and benefit FEMTO more than Lucene, since FEMTO is limited by disk seek speed. Lastly, both systems are 100-1000x faster on disk than *grep* for this collection.

As Figure 2 shows, FEMTO is competitive with Bowtie2 for single-query exact-match searches. Bowtie2 was not designed for this kind of external memory operation, but it is a BWT-based index like FEMTO. Bowtie2 performs exact search about as quickly as FEMTO. For most of these searches, FEMTO is slightly faster. Note that for both of these systems, the search time increases as the pattern length grows. Lastly, for a collection of only 4GB, a scan of the collection with *grep* is competitive with the time to do a search with either of these indexes if the index is stored on disk; we measured *grep* to take 11.5 s which is about the time it takes to run a 256-character query.

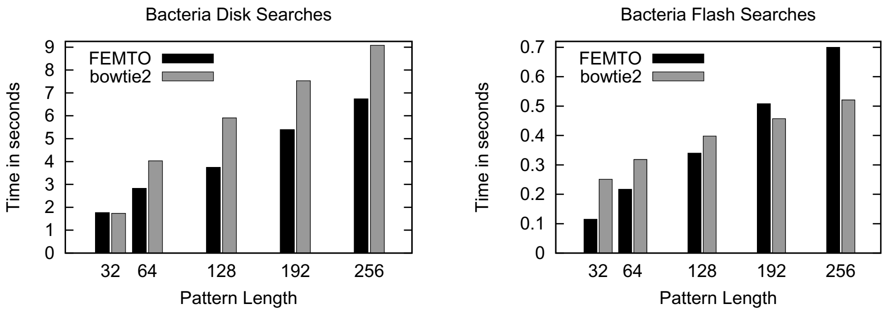


Fig. 2. Bacteria (3.76 GB) indexed search speeds. Note that the time to search with *grep* is 11.5 s on disk, or 6.12 s with flash memory. Each query returned only one result.

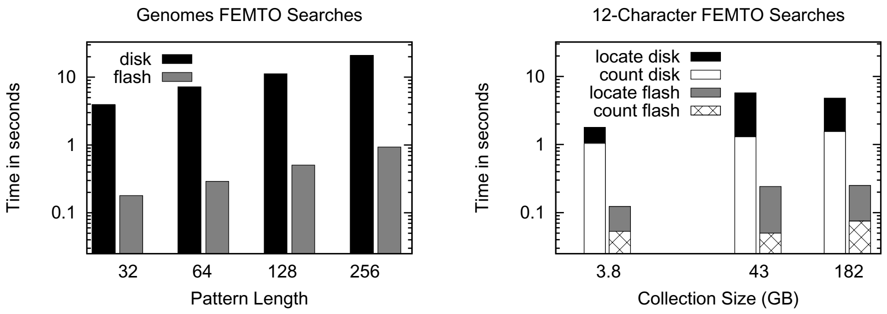


Fig. 3. Genomes (182.43 GB) indexed search speeds. Note that the time to search this collection with *grep* is 1246 s on disk, or an estimated 493 s with flash memory. Each query returned one or two results. Note that these plots are on log-log scale.

FEMTO scales to very large index sizes while still offering fast searches. Only FEMTO was able to construct a sequence index for the Genomes collection. As the left-hand side of Figure 3 shows, sequence queries still complete in seconds and take time proportional to the length of the query. Observe on the right-hand side of Figure 3 that FEMTO search performance is approximately constant over the range of collection sizes measured here - from 3.76 GB to 182 GB.

To demonstrate the capabilities of sorting requests, we also measured the performance of serial and parallel count queries over randomly selected length-12 patterns in the Gutenberg data set. See Figure 4. The request sorting mechanism offers better throughput on disk when processing many count queries simultaneously, but does not offer much of an advantage when the index is in memory.

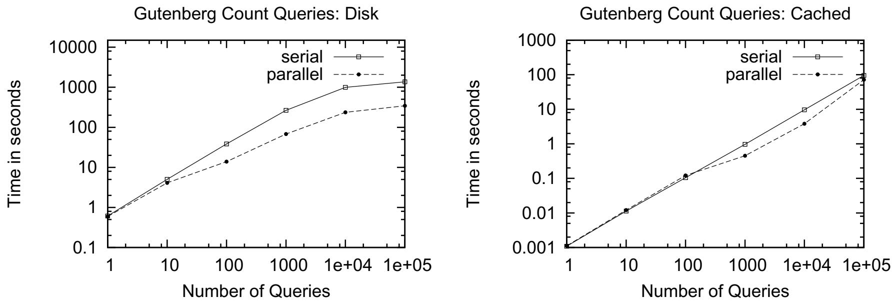


Fig. 4. Gutenberg (42.94 GB) indexed search speeds for many count queries. Note that these plots are on log-log scale.

4.2 Approximate Search

The FEMTO query "APPROX constitutional" searches for all sequences within edit distance 1 of "constitutional". That query took 37.6 s on the disk system and 2.41 s on the flash system. The analogous Lucene query, "constitutional~" took 103 s on the disk system and 96 s on the flash memory system. Thus, FEMTO is quite a bit faster than Lucene at approximate search. Approximate search is still fast even for our largest index - a 32 character search against the Genomes index took 53.2 s with the disk system and 3.34 s with flash memory.

4.3 FEMTO Index Space

Table 2 shows how the space of each FEMTO index was used. Each column shows index structures that support different features. Using only the data in the *BWT & Occs*, the index supports finding the range of matching rows for a pattern. Adding in the *Offsets* allows the index to report back the offsets of matching rows, and the mark period (20) represents a time-space trade-off for that operation. Finally, adding in the *Doc Chunks* allows the index to rapidly

Table 2. FEMTO index statistics. Percentages show proportion of collection size.

Collection	BWT & Occs	Offsets	Doc Chunks	Total
Bacteria - 3.76	0.96 (25%)	0.94 (25%)	0.44 (11%)	2.34 (62%)
Gutenberg - 42.93	7.74 (18%)	12.00 (28%)	21.04 (49%)	40.87 (95%)
Gutenberg' - 42.93	7.74 (18%)	12.00 (28%)	2.4594 (6%)	22.21 (52%)
Genomes - 182.43	32.97 (18%)	51.13 (28%)	209.06 (115%)	296.68 (163%)

report a large number of matching documents for large ranges of rows. This kind of operation is important for searches which require follow-on processing, such as Boolean queries. Note that without the document chunk information, each of these indexes would be about 50% the size of the original collection. The document chunks did not compress well in the larger collections, which have orders of magnitude more documents than the chunk size of $h = 2048$. To investigate further, we created a Gutenberg index with a $h = 131072$, listed as Gutenberg' in the table, that achieves much better compression. Thus, for consistent compression, h should depend on the number of documents.

4.4 Reporting Results

We measured the speed of finding the offset or document for a varying number of rows in the Gutenberg index. These queries really show the ability for FEMTO to minimize I/O operations. See Figure 5. With flash memory, we observed locate speeds up to 7000 results/second and document locate speeds up to 90,000 results/second. On disk, locate performed at up to 800 results/second and document locate operated at up to 70,000 results/second. We believe that the decrease in locate performance around 10,000 results comes from the working set no longer fitting into main memory. Also, note that for 10 million results, the query processing data structures occupy 8 GiB of memory.

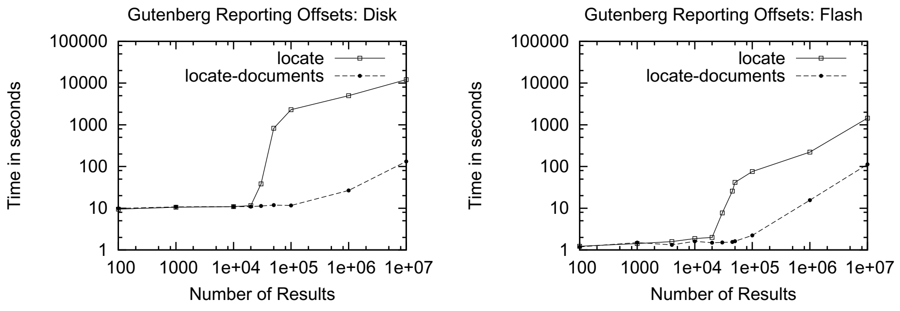


Fig. 5. Gutenberg (42.93 GB) locate speeds. Note that these plots are on log-log scale.

5 Conclusion

We have demonstrated the feasibility of using an improved FM-index for large sequence and document retrieval problems. We have demonstrated interactive search times even with our largest collection. This index structure is very quick to search - especially with flash memory - and supports regular expression and sequence queries.

Availability. The FEMTO software is not yet released. Please contact the author regarding distribution and development.

Acknowledgments. We would like to thank Aaron Marcus from Virginia Tech and Katherine Gibson from the University of Maryland Baltimore County for their help implementing regular expression search; and John Dorband, Yaacov Yesha, and Nancy Walia from the University of Maryland Baltimore County for their help with parallel suffix array construction.

References

1. Baeza-Yates, R.A., Gonnet, G.H.: Fast text searching for regular expressions or automaton searching on tries. J. ACM 43(6), 915–936 (1996)
2. Bauer, M.J., Cox, A.J., Rosone, G.: Lightweight BWT Construction for Very Large String Collections. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 219–231. Springer, Heidelberg (2011)
3. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10), 762–772 (1977)
4. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
5. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k Ranked Document Search in General Text Databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
6. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. In: ALENEX/ANALCO, pp. 86–97. SIAM (2005)
7. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica 63(3), 707–730 (2012)
8. Ferragina, P., Manzini, G.: An experimental study of a compressed index. Information Sciences 135(1-2), 13–28 (2001)
9. Ferragina, P., Manzini, G.: Fm-index version 2 web page (2005), `http://roquefort.di.unipi.it/~ferrax/fmindexV2/index.html`
10. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52(4), 552–581 (2005)
11. Ferragina, P., Navarro, G.: Pizza & chili website (2006), `http://pizzachili.dcc.uchile.cl` or `http://pizzachili.di.unipi.it`
12. Gagie, T., Puglisi, S.J., Turpin, A.: Range Quantile Queries: Another Virtue of Wavelet Trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
13. Grossi, R., Gupta, A., Vitter, J.S.: When indexing equals compression: experiments with compressing suffix arrays and applications. In: SODA 2004: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (2004)
14. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: STOC 2000: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (2000)
15. Hon, W.-K., Lam, T.-W., Sadakane, K., Sung, W.-K., Yiu, S.-M.: A space and time efficient algorithm for constructing compressed suffix arrays. Algorithmica 48(1), 23–36 (2007)
16. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)

17. Kulla, F., Sanders, P.: Scalable parallel suffix array construction. Parallel Comput. 33(9), 605–612 (2007)
18. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.: Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biology 10(3), R25 (2009)
19. Sirén, J.: Compressed Suffix Arrays for Massive Data. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 63–74. Springer, Heidelberg (2009)
20. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM 11(6), 419–422 (1968)
21. Witten, I., Moffat, A., Bell, T.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann (1999)
22. Wu, S., Manber, U.: Fast text searching: allowing errors. Commun. ACM 35, 83–91 (1992)

Speeding Up q-Gram Mining on Grammar-Based Compressed Texts

Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda

Department of Informatics, Kyushu University
{keisuke.gotou,bannai,inenaga,takeda}@inf.kyushu-u.ac.jp

Abstract. We present an efficient algorithm for calculating q-gram frequencies on strings represented in compressed form, namely, as a straight line program (SLP). Given an SLP $\mathcal{T}$ of size n that represents string T, the algorithm computes the occurrence frequencies of *all* q-grams in T, by reducing the problem to the weighted q-gram frequencies problem on a trie-like structure of size $m = |T| - dup(q, \mathcal{T})$, where $dup(q, \mathcal{T})$ is a quantity that represents the amount of redundancy that the SLP captures with respect to q-grams. The reduced problem can be solved in linear time. Since $m = O(qn)$, the running time of our algorithm is $O(\min\{|T| - dup(q, \mathcal{T}), qn\})$, improving our previous $O(qn)$ algorithm when $q = \Omega(|T|/n)$.

1 Introduction

Many large string data sets are usually first compressed and stored, while they are decompressed afterwards in order to be used and analyzed. Compressed string processing (CSP) is an approach that has been gaining attention in the string processing community. Assuming that the input is given in compressed form, the aim is to develop methods where the string is processed or analyzed without explicitly decompressing the entire string, leading to algorithms with time and space complexities that depend on the compressed size rather than the whole uncompressed size. Since compression algorithms inherently capture regularities of the original string, clever CSP algorithms can be theoretically [13,4,10,7], and even practically [19,9], faster than algorithms which process the uncompressed string.

In this paper, we assume that the input string is represented as a Straight Line Program (SLP), which is a context free grammar in Chomsky normal form that derives a single string. SLPs are a useful tool when considering CSP algorithms, since it is known that outputs of various grammar based compression algorithms [17,16], as well as dictionary compression algorithms [24,22,23,21] can be modeled efficiently by SLPs [18]. We consider the q-gram frequencies problem on compressed text represented as SLPs. q-gram frequencies have profound applications in the field of string mining and classification. The problem was first considered for the CSP setting in [11], where an $O(|\Sigma|^2 n^2)$-time $O(n^2)$-space algorithm for finding the *most frequent* 2-gram from an SLP of size n representing text T over alphabet Σ was presented. In [3], it is claimed that the most frequent 2-gram can be found in $O(|\Sigma|^2 n \log n)$-time and $O(n \log |T|)$-space, if the SLP is pre-processed and a self-index is built. A much simpler and efficient $O(qn)$ time and space algorithm for general $q \geq 2$ was recently developed [9].

Remarkably, computational experiments on various data sets showed that the $O(qn)$ algorithm is actually faster than calculations on uncompressed strings, when q is small [9]. However, the algorithm slows down considerably compared to the uncompressed approach when q increases. This is because the algorithm reduces the q-gram frequencies problem on an SLP of size n, to the weighted q-gram frequencies problem on a weighted string of size at most $2(q-1)n$. As q increases, the length of the string becomes longer than the uncompressed string T. Theoretically, q can be as large as $O(|T|)$, hence in such a case the algorithm requires $O(|T|n)$ time, which is worse than a trivial $O(|T|)$ solution that first decompresses the given SLP and runs a linear time algorithm for q-gram frequencies computation on T.

In this paper, we solve this problem, and improve the previous $O(qn)$ algorithm both theoretically and practically. We introduce a q-gram neighbor relation on SLP variables, in order to reduce the redundancy in the partial decompression of the string which is performed in the previous algorithm. Based on this idea, we are able to convert the problem to a weighted q-gram frequencies problem on a weighted trie, whose size is at most $|T| - \mathit{dup}(q, \mathcal{T})$. Here, $\mathit{dup}(q, \mathcal{T})$ is a quantity that represents the amount of redundancy that the SLP captures with respect to q-grams. Since the size of the trie is also bounded by $O(qn)$, the time complexity of our new algorithm is $O(\min\{qn, |T| - \mathit{dup}(q, \mathcal{T})\})$, improving on our previous $O(qn)$ algorithm when $q = \Omega(|T|/n)$. Preliminary computational experiments show that our new approach achieves a practical speed up as well, for all values of q.

2 Preliminaries

2.1 Intervals, Strings, and Occurrences

For integers $i \leq j$, let $[i : j]$ denote the interval of integers $\{i, \ldots, j\}$. For an interval $[i : j]$ and integer $q > 0$, let $\mathit{pre}([i : j], q)$ and $\mathit{suf}([i : j], q)$ represent respectively, the length-q prefix and suffix interval, that is, $\mathit{pre}([i : j], q) = [i : \min(i + q - 1, j)]$ and $\mathit{suf}([i : j], q) = [\max(i, j - q + 1) : j]$.

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. For any integer $q > 0$, an element of Σ^q is called a *q-gram*. The length of a string T is denoted by $|T|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $T = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. The i-th character of a string T is denoted by $T[i]$, where $1 \leq i \leq |T|$. For a string T and interval $[i : j] (1 \leq i \leq j \leq |T|)$, let $T([i : j])$ denote the substring of T that begins at position i and ends at position j. For convenience, let $T([i : j]) = \varepsilon$ if $j < i$. For a string T and integer $q \geq 0$, let $\mathit{pre}(T, q)$ and $\mathit{suf}(T, q)$ represent respectively, the length-q prefix and suffix of T, that is, $\mathit{pre}(T, q) = T(\mathit{pre}([1 : |T|], q))$ and $\mathit{suf}(T, q) = T(\mathit{suf}([1 : |T|], q))$.

For any strings T and P, let $Occ(T,P)$ be the set of occurrences of P in T, i.e., $Occ(T,P) = \{k > 0 \mid T[k : k + |P| - 1] = P\}$. The number of elements $|Occ(T,P)|$ is called the *occurrence frequency* of P in T.

2.2 Straight Line Programs

A *straight line program* (*SLP*) is a set of assignments $\mathcal{T} = \{X_1 \to expr_1, X_2 \to expr_2, \ldots, X_n \to expr_n\}$, where each X_i is a variable and each $expr_i$ is an expression, where $expr_i = a$ ($a \in \Sigma$), or $expr_i = X_{\ell(i)}X_{r(i)}$ ($i > \ell(i), r(i)$). It is essentially a context free grammar in the Chomsky normal form, that derives a single string. Let $val(X_i)$ represent the string derived from variable X_i. To ease notation, we sometimes associate $val(X_i)$ with X_i and denote $|val(X_i)|$ as $|X_i|$, and $val(X_i)([u : v])$ as $X_i([u : v])$ for any interval $[u : v]$. An SLP $\mathcal{T}$ *represents* the string $T = val(X_n)$. The *size* of the program $\mathcal{T}$ is the number n of assignments in $\mathcal{T}$. Note that $|T|$ can be as large as $\Theta(2^n)$. However, we assume as in various previous work on SLP, that the computer word size is at least $\log |T|$, and hence, values representing lengths and positions of T in our algorithms can be manipulated in constant time.

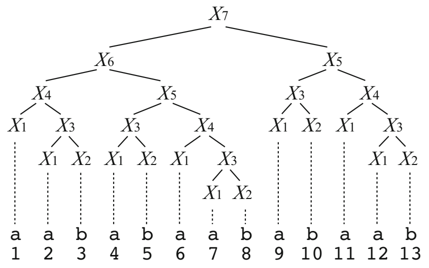


Fig. 1. The derivation tree of SLP $\mathcal{T} = \{X_1 \to \mathtt{a},\ X_2 \to \mathtt{b},\ X_3 \to X_1X_2,\ X_4 \to X_1X_3,\ X_5 \to X_3X_4,\ X_6 \to X_4X_5,\ X_7 \to X_6X_5\}$, representing string $T = val(X_7) = \mathtt{aababaababaab}$.

The derivation tree of SLP $\mathcal{T}$ is a labeled ordered binary tree where each internal node is labeled with a non-terminal variable in $\{X_1, \ldots, X_n\}$, and each leaf is labeled with a terminal character in Σ. The root node has label X_n. Let $\mathcal{V}$ denote the set of internal nodes in the derivation tree. For any internal node $v \in \mathcal{V}$, let $\langle v \rangle$ denote the index of its label $X_{\langle v \rangle}$. Node v has a single child which is a leaf labeled with c when $(X_{\langle v \rangle} \to c) \in \mathcal{T}$ for some $c \in \Sigma$, or v has a left-child and right-child respectively denoted $\ell(v)$ and $r(v)$, when $(X_{\langle v \rangle} \to X_{\langle \ell(v) \rangle}X_{\langle r(v) \rangle}) \in \mathcal{T}$. Each node v of the tree derives $val(X_{\langle v \rangle})$, a substring of T, whose corresponding interval $itv(v)$, with $T(itv(v)) = val(X_{\langle v \rangle})$, can be defined recursively as follows. If v is the root node, then $itv(v) = [1 : |T|]$. Otherwise, if $(X_{\langle v \rangle} \to X_{\langle \ell(v) \rangle}X_{\langle r(v) \rangle}) \in \mathcal{T}$, then, $itv(\ell(v)) = [b_v : b_v + |X_{\langle \ell(v) \rangle}| - 1]$ and $itv(r(v)) = [b_v + |X_{\langle \ell(v) \rangle}| : e_v]$, where $[b_v : e_v] = itv(v)$. Let $vOcc(X_i)$ denote the number of times a variable X_i occurs in the derivation tree, i.e., $vOcc(X_i) = |\{v \mid X_{\langle v \rangle} = X_i\}|$. We assume that any variable X_i is used at least once, that is $vOcc(X_i) > 0$.

For any interval $[b : e]$ of $T (1 \leq b \leq e \leq |T|)$, let $\xi_{\mathcal{T}}(b, e)$ denote the deepest node v in the derivation tree, which derives an interval containing $[b : e]$, that is, $itv(v) \supseteq [b : e]$, and no proper descendant of v satisfies this condition. We say that node v *stabs* interval $[b : e]$, and $X_{\langle v \rangle}$ is called the variable that stabs the interval. If $b = e$, we have that $(X_{\langle v \rangle} \to c) \in \mathcal{T}$ for some $c \in \Sigma$, and $itv(v) = b = e$. If $b < e$, then we have $(X_{\langle v \rangle} \to X_{\langle \ell(v) \rangle}X_{\langle r(v) \rangle}) \in \mathcal{T}$, $b \in itv(\ell(v))$, and $e \in itv(r(v))$. When it is not confusing, we will sometimes use $\xi_{\mathcal{T}}(b, e)$ to denote the variable $X_{\langle \xi_{\mathcal{T}}(b,e) \rangle}$.

SLPs can be efficiently pre-processed to hold various information. $|X_i|$ and $vOcc(X_i)$ can be computed for all variables $X_i (1 \leq i \leq n)$ in a total of $O(n)$ time by a simple dynamic programming algorithm. Also, the following Lemma is useful for partial decompression of a prefix of a variable.

Lemma 1 ([8]). *Given an SLP $\mathcal{T} = \{X_i \to expr_i\}_{i=1}^{n}$, it is possible to pre-process $\mathcal{T}$ in $O(n)$ time and space, so that for any variable X_i and $1 \leq j \leq |X_i|$, $X_i([1 : j])$ can be computed in $O(j)$ time.*

The formal statement of the problem we solve is:

Problem 1 (q-gram frequencies on SLP). Given integer $q \geq 1$ and an SLP $\mathcal{T}$ of size n that represents string T, output $(i, |Occ(T, P)|)$ for all $P \in \Sigma^q$ where $Occ(T, P) \neq \emptyset$, and some $i \in Occ(T, P)$.

Since the problem is very simple for $q = 1$, we shall only consider the case for $q \geq 2$ for the rest of the paper. Note that although the number of distinct q-grams in T is bounded by $O(qn)$, we would require an extra multiplicative $O(q)$ factor for the output if we output each q-gram explicitly as a string. In our algorithms to follow, we compute a compact, $O(qn)$-size representation of the output, from which each q-gram can be easily obtained in $O(q)$ time.

3 $O(qn)$ Algorithm [9]

In this section, we briefly describe the $O(qn)$ algorithm presented in [9]. The idea is to count occurrences of q-grams with respect to the variable that stabs its occurrence. The algorithm reduces Problem 1 to calculating the frequencies of all q-grams in a weighted set of strings, whose total length is $O(qn)$. Lemma 2 shows the key idea of the algorithm.

Lemma 2. *For any SLP $\mathcal{T} = \{X_i \to expr_i\}_{i=1}^{n}$ that represents string T, integer $q \geq 2$, and $P \in \Sigma^q$, $|Occ(T, P)| = \sum_{i=1}^{n} vOcc(X_i) \cdot |Occ(t_i, P)|$, where $t_i = suf(val(X_{\ell(i)}), q-1)pre(val(X_{r(i)}), q-1)$.*

Proof. For any $q \geq 2$, v stabs the interval $[u : u+q-1]$ if and only if $[u : u+q-1] \subseteq [s_v : f_v] = suf(itv(\ell(v)), q-1) \cup pre(itv(r(v)), q-1)$. (See Fig. 2.) Also, since an occurrence of X_i in the derivation tree always derives the same string $val(X_i)$, $t_i = T([s_v : f_v])$ for any node v such that $X_{\langle v \rangle} = X_i$. For any node v and $u > 0$, let $[b_v, e_v] = itv(v)$ and $j_v = u - b_v + 1$. We have $|Occ(T, P)| = |\{u > 0 \mid T([u : u+q-1]) = P\}| = \sum_{v \in \mathcal{V}} |\{u > 0 \mid \xi_{\mathcal{T}}(u, u+q-1) = v, X_{\langle v \rangle}([j_v : j_v + q - 1]) = P\}| = \sum_{i=1}^{n} \sum_{v \in \mathcal{V} : X_{\langle v \rangle} = X_i} |\{u > 0 \mid \xi_{\mathcal{T}}(u, u+q-1) = v, X_{\langle v \rangle}([j_v : j_v + q - 1]) = P\}| = \sum_{i=1}^{n} \sum_{v \in \mathcal{V} : X_{\langle v \rangle} = X_i} Occ(T([s_v : f_v]), P) = \sum_{i=1}^{n} vOcc(X_i) \cdot Occ(t_i, P)$. □

From Lemma 2, we have that occurrence frequencies in T are equivalent to occurrence frequencies in t_i weighted by $vOcc(X_i)$. Therefore, the q-gram frequencies problem can be regarded as obtaining the *weighted* frequencies of all q-grams in the set of strings $\{t_1, \ldots, t_n\}$, where each occurrence of a q-gram in t_i is weighted by $vOcc(X_i)$.

This can be further reduced to a weighted q-gram frequency problem for a single string z, where each position of z holds a weight associated with the q-gram that starts at that position. String z is constructed by concatenating all t_i's with length at least q. The weights of positions corresponding to the first $|t_i|-(q-1)$ characters of t_i will be $vOcc(X_i)$, while the last $(q-1)$ positions will be 0 so that superfluous q-grams generated by the concatenation are not counted. The remaining is a simple linear time algorithm using suffix and lcp arrays (e.g. [12,14]) on the weighted string, thus solving the problem in $O(qn)$ time and space.

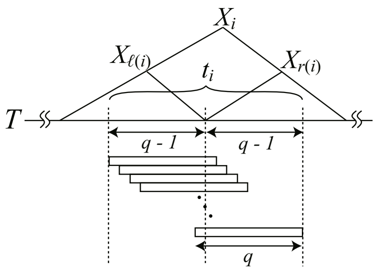


Fig. 2. Length-q intervals where $X_{\langle \xi_{\mathcal{T}}(u,u+q-1)\rangle} = X_i$, and $(X_i \to X_{\ell(i)}X_{r(i)}) \in \mathcal{T}$

4 New Algorithm

We now describe our new algorithm which solves the q-gram frequencies problem on SLPs. The new algorithm basically follows the previous $O(qn)$ algorithm, but is an elegant refinement. The reduction for the previous $O(qn)$ algorithm leads to a fairly large amount of redundantly decompressed regions of the text as q increases. This is due to the fact that the t_i's are considered independently for each variable X_i, while *neighboring* q-grams that are stabbed by different variables actually share $q-1$ characters. The key idea of our new algorithm is to exploit this redundancy. (See Fig. 3.) In what follows, we introduce the concept of q-gram neighbors, and reduce the q-gram frequencies problem on SLP to a weighted q-gram frequencies problem on a weighted tree.

4.1 *q*-Gram Neighbor Graph

We say that X_j is a *right q-gram neighbor* of X_i $(i \neq j)$, or equivalently, X_i is a *left q-gram neighbor* of X_j, if for some integer $u \in [1 : |T| - q]$, $X_{\langle \xi_{\mathcal{T}}(u,u+q-1)\rangle} = X_i$ and $X_{\langle \xi_{\mathcal{T}}(u+1,u+q)\rangle} = X_j$. Notice that $|X_i|$ and $|X_j|$ are both at least q if X_i and X_j are right or left q-gram neighbors of each other.

Definition 1. *For $q \geq 2$, the right q-gram neighbor graph of SLP $\mathcal{T} = \{X_i \to expr_i\}_{i=1}^n$ is the directed graph $G_q = (V, E_r)$, where*

$$V = \{X_i \mid i \in \{1, \ldots, n\}, |X_i| \geq q\}$$
$$E_r = \{(X_i, X_j) \mid X_j \textit{ is a right q-gram neighbor of } X_i \}$$

Note that there can be multiple right q-gram neighbors for a given variable. However, the total number of edges in the neighbor graph is bounded by $2n$, as will be shown below.

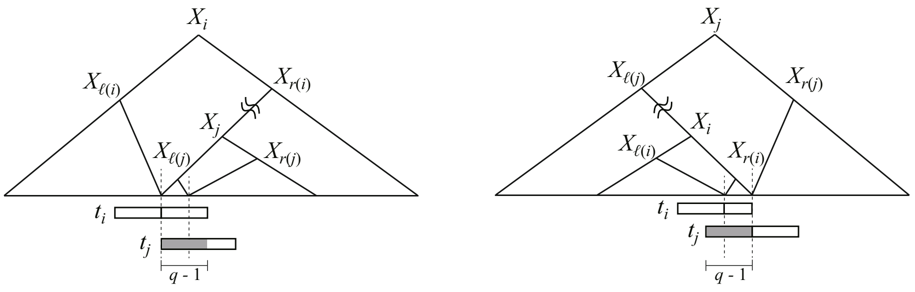


Fig. 3. q-gram neighbors and redundancies. (Left) X_j is a right q-gram neighbor of X_i, and X_i is *a* left q-gram neighbor of X_j. Note that the right q-gram neighbor of X_i is uniquely determined since $|X_{r(i)}| \geq q$ and it must be a descendant on the left most path rooted at $X_{r(i)}$. However, X_j may have other left q-gram neighbors, since $|X_{\ell(j)}| < q$, and they must be ancestors of X_j. t_i (resp. t_j) represents the string corresponding to the union of intervals $[u : u{+}q{-}1]$ where $X_{\langle \xi_{\mathcal{T}}(u,u+q-1)\rangle} = X_i$ (resp. $X_{\langle \xi_{\mathcal{T}}(u,u+q-1)\rangle} = X_j$). The shaded region depicts the string which is redundantly decompressed, if both t_i and t_j are considered independently. (Right) Shows the reverse case, when $|X_{r(i)}| < q$.

Lemma 3. *Let X_j be a right q-gram neighbor of X_i. If, $|X_{r(i)}| \geq q$, then X_j is the label of the deepest node on the left-most path of the derivation tree rooted at a node labeled $X_{r(i)}$ whose length is at least q. Otherwise, if $|X_{r(i)}| < q$, then X_i is the label of the deepest node on the right-most path rooted at a node labeled $X_{\ell(j)}$ whose length is at least q.*

Proof. Suppose $|X_{r(i)}| \geq q$. Let u be a position, where $X_{\langle \xi_{\mathcal{T}}(u,u+q-1)\rangle} = X_i$ and $X_{\langle \xi_{\mathcal{T}}(u+1,u+q)\rangle} = X_j$. Then, since the interval $[u{+}1 : u{+}q]$ is a prefix of $itv(X_{r(i)})$, X_j must be on the left most path rooted at $X_{r(i)}$. Since $X_j = X_{\langle \xi_{\mathcal{T}}(u+1,u+q)\rangle}$, the lemma follows from the definition of $\xi_{\mathcal{T}}$. The case for $|X_{r(i)}| < q$ is symmetrical and can be shown similarly. □

Lemma 4. *For an arbitrary SLP $\mathcal{T} = \{X_i \to expr_i\}_{i=1}^n$ and integer $q \geq 2$, the number of edges in the right q-gram neighbor graph G_q of $\mathcal{T}$ is at most $2n$.*

Proof. Suppose X_j is a right q-gram neighbor of X_i. From Lemma 3, we have that if $|X_{r(i)}| \geq q$, the right q-gram neighbor of X_i is uniquely determined and that $|X_{\ell(j)}| < q$. Similarly, if $|X_{r(i)}| < q$, $|X_{\ell(j)}| \geq q$ and the left q-gram neighbor of X_j is uniquely X_i. Therefore, $\sum_{i=1}^n |\{(X_i, X_j) \in E_r \mid |X_{r(i)}| \geq q\}| + \sum_{i=1}^n |\{(X_i, X_j) \in E_r \mid |X_{r(i)}| < q\}| = \sum_{i=1}^n |\{(X_i, X_j) \in E_r \mid |X_{r(i)}| \geq q\}| + \sum_{i=1}^n |\{(X_i, X_j) \in E_r \mid |X_{\ell(j)}| \geq q\}| \leq 2n$. □

Lemma 5. *For an arbitrary SLP $\mathcal{T} = \{X_i \to expr_i\}_{i=1}^n$ and integer $q \geq 2$, the right q-gram neighbor graph G_q of $\mathcal{T}$ can be constructed in $O(n)$ time.*

Proof. For any variable X_i, let $lm_q(X_i)$ and $rm_q(X_i)$ respectively represent the index of the label of the deepest node with length at least q on the left-most and right-most path in the derivation tree rooted at X_i, or *null* if $|X_i| < q$. These values can be computed for all variables in a total of $O(n)$ time based on the following recursion: If $(X_i \to a) \in \mathcal{T}$ for some $a \in \Sigma$, then $lm_q(X_i) = rm_q(X_i) = null$. For $(X_i \to X_{\ell(i)} X_{r(i)}) \in \mathcal{T}$,

$$lm_q(X_i) = \begin{cases} null & \text{if } |X_i| < q, \\ i & \text{if } |X_i| \geq q \text{ and } |X_{\ell(i)}| < q, \\ lm_q(X_{\ell(i)}) & \text{otherwise.} \end{cases}$$

$rm_q(X_i)$ can be computed similarly. Finally,

$$\begin{aligned} E_r = & \{(X_i, X_{lm_q(X_{r(i)})}) \mid lm_q(X_{r(i)}) \neq null, i = 1, \ldots, n\} \\ & \cup \{(X_{rm_q(X_{\ell(i)})}, X_i) \mid rm_q(X_{\ell(i)}) \neq null, i = 1, \ldots, n\}. \end{aligned}$$

□

Lemma 6. *Let $G_q = (V, E_r)$ be the right q-gram neighbor graph of SLP $\mathcal{T} = \{X_i = expr_i\}_{i=1}^n$ representing string T, and let $X_{i_1} = X_{\langle \xi_{\mathcal{T}}(1,q) \rangle}$. Any variable $X_j \in V (i_1 \neq j)$ is reachable from X_{i_1}, that is, there exists a directed path from X_{i_1} to X_j in G_q.*

Proof. Straightforward, since any q-gram of T except for the left most $T([1 : q])$ has a q-gram on its left. □

4.2 Weighted q-Gram Frequencies over a Trie

From Lemma 6, we have that the right q-gram neighbor graph is connected. Consider an arbitrary directed spanning tree rooted at $X_{i_1} = X_{\langle \xi_{\mathcal{T}}(1,q) \rangle}$ which can be obtained in linear time by a depth first traversal on G_q from X_{i_1}. We define the label $label(X_i)$ of each node X_i of the q-gram neighbor graph, by

$$label(X_i) = t_i[q : |t_i|]$$

where $t_i = suf(val(X_{\ell(i)}), q-1) pre(val(X_{r(i)}), q-1)$ as before. For convenience, let X_{i_0} be a dummy variable such that $label(X_{i_0}) = T([1 : q-1])$, and $X_{r(i_0)} = X_{i_1}$ (and so $(X_{i_0}, X_{i_1}) \in E_r$).

Lemma 7. *Fix a directed spanning tree on the right q-gram neighbor graph of SLP $\mathcal{T}$, rooted at X_{i_0}. Consider a directed path $X_{i_0}, \ldots, X_{i_m}$ on the spanning tree. The weighted q-gram frequencies on the string obtained by the concatenation $label(X_{i_0}) label(X_{i_1}) \cdots label(X_{i_m})$, where each occurrence of a q-gram that ends in a position in $label(X_{i_j})$ is weighted by $vOcc(X_{i_j})$, is equivalent to the weighted q-gram frequencies of strings $\{t_{i_1}, \ldots t_{i_m}\}$ where each q-gram in t_{i_j} is weighted by $vOcc(X_{i_j})$.*

Proof. Proof by induction: for $m = 1$, we have that $label(X_{i_0}) label(X_{i_1}) = t_{i_1}$. All q-grams in t_{i_1} end in t_{i_1} and so are weighted by $vOcc(X_{i_1})$. When $label(X_{i_j})$ is added to $label(X_{i_0}) \cdots label(X_{i_{j-1}})$, $|label(X_{i_j})|$ new q-grams are formed, which correspond to q-grams in t_{i_j}, i.e. $|t_{i_j}| = q - 1 + |label(X_{i_j})|$, and t_{i_j} is a suffix of $label(X_{i_{j-1}}) label(X_{i_j})$. All the new q-grams end in $label(X_{i_j})$ and are thus weighted by $vOcc(X_{i_j})$. □

Algorithm 1. Constructing weighted trie from SLP

```
1 Construct right $q$-gram neighbor graph $G = (V, E_r)$;
2 Calculate $vOcc(X_i)$ and $|label(X_i)|$ for $i = 1, \ldots, n$;
3 for $i = 0, \ldots, n$ do visited$[i]$ = false;
4 $X_{i_1} = X_{\langle \xi_{\mathcal{T}}(1,q) \rangle} = lm_q(X_n)$;
5 Define $X_{i_0}$ so that $X_{r(i_0)} = X_{i_1}$ and $|label(X_{i_0})| = q - 1$;
6 $root \leftarrow$ new node; // root of resulting trie
7 BuildDepthFirst($i_0$, $root$);
8 return $root$
```

Procedure. BuildDepthFirst(i, $trieNode$)

```
  // add prefix of $r(i)$ to trieNode while right neighbors are unique
1 $l \leftarrow 0$; $k \leftarrow i$;
2 while true do
3     $l \leftarrow l + |label(X_k)|$;
4     visited$[k] \leftarrow$ true;
      // exit loop if right neighbor might be non-unique or is visited
5     if $|X_{r(k)}| < q$ or visited$[lm_q(X_{r(k)})]$ = true then break;
6     $k \leftarrow lm_q(X_{r(k)})$;
7 add new branch from $trieNode$ with string $X_{r(i)}([1 : l])$;
8 let end of new branch be $newTrieNode$;
  // If $|X_{r(k)}| < q$, there may be multiple right neighbors.
  // If $|X_{r(k)}| \geq q$, nothing is done because it was already visited.
9 for $X_c \in \{X_j \mid (X_k, X_j) \in E_r\}$ do
10    if visited$[c]$ = false then BuildDepthFirst ($X_c$, $newTrieNode$);
```

From Lemma 7, we can construct a weighted trie Υ based on a directed spanning tree of G_q and $label()$, where the weighted q-grams in Υ (represented as length-q paths) correspond to the occurrence frequencies of q-grams in T[1].

Lemma 8. *Υ can be constructed in time linear in its size.*

Proof. See Algorithm 1. Let G be the q-gram neighbor graph. We construct Υ in a depth first manner starting at X_{i_0}. The crux of the algorithm is that rather than computing $label()$ separately for each variable, we are able to aggregate the $label()$s and limit all partial decompressions of variables to prefixes of variables, so that Lemma 1 can be used.

Any directed acyclic path on G starting at X_{i_0} can be segmented into multiple sequences of variables, where each sequence $X_{i_j}, \ldots, X_{i_k}$ is such that j is the only integer in $[j : k]$ such that $j = 0$ or $|X_{r(i_{j-1})}| < q$. From Lemma 3, we have that $X_{i_{j+1}}, \ldots, X_{i_k}$ are uniquely determined. If $j > 0$, $label(X_{i_j})$ is a prefix of $val(X_{r(i_j)})$ since $|X_{r(i_{j-1})}| < q$ (see Fig. 3 Right), and if $j = 0$,

[1] A minor technicality is that a node in Υ may have multiple children with the same character label, but this does not affect the time complexities of the algorithm.

$label(X_{i_0})$ is again a prefix of $val(X_{r(i_0)}) = val(X_{i_1})$. It is not difficult to see that $label(X_{i_j}) \cdots label(X_{i_k})$ is also a prefix of $X_{r(i_j)}$ since $X_{i_{j+1}}, \ldots, X_{i_k}$ are all descendants of $X_{r(i_j)}$, and each $label()$ extends the partially decompressed string to consider consecutive q-grams in $X_{r(i_j)}$. Since prefixes of variables of SLPs can be decompressed in time proportional to the output size with linear time pre-processing (Lemma 1), the lemma follows. □

We only illustrate how the character labels are determined in the pseudo-code of Algorithm 1. It is straightforward to assign a weight $vOcc(X_k)$ to each node of Υ that corresponds to $label(X_k)$.

Lemma 9. *The number of edges in* Υ *is* $(q-1) + \sum\{|t_i| - (q-1) \mid |X_i| \geq q, i = 1, \ldots, n\} = |T| - dup(q, \mathcal{T})$ *where*

$$dup(q, \mathcal{T}) = \sum\{(vOcc(X_i) - 1) \cdot (|t_i| - (q-1)) \mid |X_i| \geq q, i = 1, \ldots, n\}$$

Proof. $(q-1) + \sum\{|t_i| - (q-1) \mid |X_i| \geq q, i = 1, \ldots, n\}$ is straight forward from the definition of $label(X_i)$ and the construction of Υ. Concerning dup, each variable X_i occurs $vOcc(X_i)$ times in the derivation tree, but only once in the directed spanning tree. This means that for each occurrence after the first, the size of Υ is reduced by $|label(X_i)| = |t_i| - (q-1)$ compared to T. Therefore, the lemma follows. □

To efficiently count the weighted q-gram frequencies on Υ, we can use suffix trees. A suffix tree for a trie is defined as a generalized suffix tree for the set of strings represented in the trie as leaf to root paths[2]. The following is known.

Lemma 10 ([20]). *Given a trie of size* m*, the suffix tree for the trie can be constructed in* $O(m)$ *time and space.*

With a suffix tree, it is a simple exercise to solve the weighted q-gram frequencies problem on Υ in linear time. In fact, it is known that the suffix array for the common suffix trie can also be constructed in linear time [6], as well as its longest common prefix array [15], which can also be used to solve the problem in linear time.

Corollary 1. *The weighted q-gram frequencies problem on a trie of size* m *can be solved in* $O(m)$ *time and space.*

From the above arguments, the theorem follows.

Theorem 1. *The q-gram frequencies problem on an SLP* $\mathcal{T}$ *of size* n*, representing string* T *can be solved in* $O(\min\{qn, |T| - dup(q, \mathcal{T})\})$ *time and space.*

Note that since each $q \leq |t_i| \leq 2(q-1)$, and $|label(X_i)| = |t_i| - (q-1)$, the total length of decompressions made by the algorithm, i.e. the size of the reduced problem, is at least halved and can be as small as $1/q$ (e.g. when all $|t_i| = q$), compared to the previous $O(qn)$ algorithm.

[2] When considering leaf to root paths on Υ, the direction of the string is the reverse of what is in T. However, this is merely a matter of representation of the output.

5 Preliminary Experiments

We first evaluate the size of the trie Υ induced from the right q-gram neighbor graph, on which the running time of the new algorithm of Section 4 is dependent. We used data sets obtained from Pizza & Chili Corpus, and constructed SLPs using the RE-PAIR [16] compression algorithm. Each data is of size 200MB. Table 1 shows the sizes of Υ for different values of q, in comparison with the total length of strings t_i, on which the previous $O(qn)$-time algorithm of Section 3 works. We cumulated the lengths of all t_i's only for those satisfying $|t_i| \geq q$, since no q-gram can occur in t_i's with $|t_i| < q$. Observe that for all values of q and for all data sets, the size of Υ (i.e., the total number of characters in Υ) is smaller than those of t_i's and the original string.

Table 1. A comparison of the size of Υ and the total length of strings t_i for SLPs that represent textual data from Pizza & Chili Corpus. The length of the original text is 209,715,200. The SLPs were constructed by RE-PAIR [16].

	XML		DNA		ENGLISH		PROTEINS	
q	$\sum \lvert t_i \rvert$	size of Υ	$\sum \lvert t_i \rvert$	size of Υ	$\sum \lvert t_i \rvert$	size of Υ	$\sum \lvert t_i \rvert$	size of Υ
2	19,082,988	9,541,495	46,342,894	23,171,448	37,889,802	18,944,902	64,751,926	32,375,964
3	37,966,315	18,889,991	92,684,656	46,341,894	75,611,002	37,728,884	129,449,835	64,698,833
4	55,983,397	27,443,734	139,011,475	69,497,812	112,835,471	56,066,348	191,045,216	93,940,205
5	72,878,965	35,108,101	185,200,662	92,516,690	148,938,576	73,434,080	243,692,809	114,655,697
6	88,786,480	42,095,985	230,769,162	114,916,322	183,493,406	89,491,371	280,408,504	123,786,699
7	103,862,589	48,533,013	274,845,524	135,829,862	215,975,218	103,840,108	301,810,933	127,510,939
8	118,214,023	54,500,142	315,811,932	153,659,844	246,127,485	116,339,295	311,863,817	129,618,754
9	131,868,777	60,045,009	352,780,338	167,598,570	273,622,444	126,884,532	318,432,611	131,240,299
10	144,946,389	65,201,880	385,636,192	177,808,192	298,303,942	135,549,310	325,028,658	132,658,662
15	204,193,702	86,915,492	477,568,585	196,448,347	379,441,314	157,558,436	347,993,213	138,182,717
20	255,371,699	104,476,074	497,607,690	200,561,823	409,295,884	162,738,812	364,230,234	142,213,239
50	424,505,759	157,069,100	530,329,749	206,796,322	429,380,290	165,882,006	416,966,397	156,257,977
100	537,677,786	192,816,929	536,349,226	207,838,417	435,843,895	167,313,028	463,766,667	168,544,608

The construction of the suffix tree or array for a trie, as well as the algorithm for Lemma 1, require various tools such as level ancestor queries [5,2,1] for which we did not have an efficient implementation. Therefore, we try to assess the practical impact of the reduced problem size using a simplified version of our new algorithm. We compared three algorithms (NSA, SSA, STSA) that count the occurrence frequencies of all q-grams in a text given as an SLP. NSA is the $O(|T|)$-time algorithm which works on the uncompressed text, using suffix and LCP arrays. SSA is our previous $O(qn)$-time algorithm [9], and STSA is a simplified version of our new algorithm. STSA further reduces the weighted q-gram frequencies problem on Υ, to a weighted q-gram frequencies problem on a single string as follows: instead of constructing Υ, each branch of Υ (on line 7 of BuildDepthFirst) is appended into a single string. The q-grams that are represented in the branching edges of Υ can be represented in the single string, by redundantly adding $suf(X_{r(i)}([1:l]), q-1)$ in front of the string corresponding to the next branch. This leads to some duplicate partial decompression, but the resulting string is still always shorter than the string produced by our previous algorithm [9]. The partial decompression of $X_{r(i)}([1:l])$ is implemented using

Table 2. Running time in seconds for SLPs that represent textual data from Pizza & Chili Corpus. The SLPs were constructed by RE-PAIR [16]. Bold numbers represent the fastest time for each data and q. STSA is faster than SSA whenever $q > 3$.

	XML			DNA			ENGLISH			PROTEINS		
q	NSA	SSA	STSA	NSA	SSA	STSA	NSA	SSA	STSA	NSA	SSA	STSA
2	41.67	**6.53**	7.63	61.28	**19.27**	22.73	56.77	**16.31**	19.23	60.16	**27.13**	30.71
3	41.46	10.96	**10.92**	61.28	**29.14**	31.07	56.77	25.58	**25.57**	60.53	**47.53**	50.65
4	41.87	16.27	**14.5**	61.65	42.22	**41.69**	56.77	37.48	**34.95**	**60.86**	74.89	73.51
5	41.85	21.33	**17.42**	61.57	56.26	**54.21**	57.09	49.83	**45.21**	**60.53**	101.64	79.1
6	41.9	25.77	**20.07**	**60.91**	73.11	68.63	57.11	62.91	**55.28**	**61.18**	123.74	75.83
7	41.73	30.14	**21.94**	**60.89**	90.88	82.85	**56.64**	75.69	63.35	**61.14**	136.12	72.62
8	41.92	34.22	**23.97**	**61.57**	110.3	93.46	**57.27**	87.9	69.7	**61.39**	142.29	71.08
9	41.92	37.9	**25.08**	**61.26**	127.29	96.07	**57.09**	100.24	73.63	**61.36**	148.12	69.88
10	41.76	41.28	**26.45**	**60.94**	143.31	96.26	**57.43**	110.85	75.68	**61.42**	149.73	69.34
15	41.95	58.21	**32.21**	**61.72**	190.88	84.86	**57.31**	146.89	70.63	**60.42**	160.58	66.57
20	41.82	74.61	**39.62**	**61.36**	203.03	83.13	**57.65**	161.12	64.8	**61.01**	165.03	66.09
50	**42.07**	134.38	53.98	**61.73**	216.6	78.0	**57.02**	166.67	57.89	**61.05**	181.14	66.36
100	**41.81**	181.23	60.18	**61.46**	217.05	75.91	57.3	166.67	**56.86**	**60.69**	197.33	69.9

a simple $O(h + l)$ algorithm, where h is the height of the SLP which can be as large as $O(n)$.

All computations were conducted on a Mac Pro (Mid 2010) with MacOS X Lion 10.7.2, and 2 x 2.93GHz 6-Core Xeon processors and 64GB Memory, only utilizing a single process/thread at once. The program was compiled using the GNU C++ compiler (`g++`) 4.6.2 with the `-Ofast` option for optimization. The running times were measured in seconds, after reading the uncompressed text into memory for NSA, and after reading the SLP that represents the text into memory for SSA and STSA. Each computation was repeated at least 3 times, and the average was taken.

Table 2 summarizes the running times of the three algorithms. SSA and STSA computed weighted q-gram frequencies on t_i and Υ, respectively. Since the difference between the total length of t_i and the size of Υ becomes larger as q increases, STSA outperforms SSA when the value of q is not small. In fact, in Table 2 STSA was faster than SSA for all values of $q > 3$. STSA was even faster than NSA on the XML data whenever $q \leq 20$. What is interesting is that STSA outperformed NSA on the ENGLISH data when $q = 100$.

References

1. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. 321(1), 5–12 (2004)
2. Berkman, O., Vishkin, U.: Finding level-ancestors in trees. J. Comput. System Sci. 48(2), 214–230 (1994)
3. Claude, F., Navarro, G.: Self-indexed grammar-based compression. Fundamenta Informaticae 111(3), 313–337 (2011)
4. Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A subquadratic sequence alignment algorithm for unrestricted scoring matrices. SIAM J. Comput. 32(6), 1654–1673 (2003)

5. Dietz, P.: Finding Level-Ancestors in Dynamic Trees. In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) WADS 1991. LNCS, vol. 519, pp. 32–40. Springer, Heidelberg (1991)
6. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM 57(1) (2009)
7. Gawrychowski, P.: Pattern Matching in Lempel-Ziv Compressed Strings: Fast, Simple, and Deterministic. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 421–432. Springer, Heidelberg (2011)
8. Gąsieniec, L., Kolpakov, R., Potapov, I., Sant, P.: Real-time traversal in grammar-based compressed files. In: Proc. DCC 2005, p. 458 (2005)
9. Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Fast q-gram Mining on SLP Compressed Strings. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 278–289. Springer, Heidelberg (2011)
10. Hermelin, D., Landau, G.M., Landau, S., Weimann, O.: A unified algorithm for accelerating edit-distance computation via text-compression. In: Proc. STACS 2009, pp. 529–540 (2009)
11. Inenaga, S., Bannai, H.: Finding characteristic substrings from compressed texts. International Journal of Foundations of Computer Science 23(2), 261–280 (2012); a preliminary version appeared in PSC 2009
12. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. Journal of the ACM 53(6), 918–936 (2006)
13. Karpinski, M., Rytter, W., Shinohara, A.: An efficient pattern-matching algorithm for strings with short descriptions. Nordic Journal of Computing 4, 172–186 (1997)
14. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
15. Kimura, D., Kashima, H.: A linear time subpath kernel for trees. IEICE Technical Report, IBISML2011-85, pp. 291–298 (2011)
16. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. In: Proc. DCC 1999, pp. 296–305. IEEE Computer Society (1999)
17. Nevill-Manning, C.G., Witten, I.H., Maulsby, D.L.: Compression by induction of hierarchical grammars. In: Proc. DCC 1994, pp. 244–253 (1994)
18. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. 302(1-3), 211–222 (2003)
19. Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T., Arikawa, S.: Speeding Up Pattern Matching by Text Compression. In: Bongiovanni, G., Petreschi, R., Gambosi, G. (eds.) CIAC 2000. LNCS, vol. 1767, pp. 306–315. Springer, Heidelberg (2000)
20. Shibuya, T.: Constructing the suffix tree of a tree with a large alphabet. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E86-A(5), 1061–1066 (2003)
21. Storer, J., Szymanski, T.: Data compression via textual substitution. Journal of the ACM 29(4), 928–951 (1982)
22. Welch, T.A.: A technique for high performance data compression. IEEE Computer 17, 8–19 (1984)
23. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory IT-23(3), 337–349 (1977)
24. Ziv, J., Lempel, A.: Compression of individual sequences via variable-length coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)

Simple and Efficient LZW-Compressed Multiple Pattern Matching

Paweł Gawrychowski[⋆]

Institute of Computer Science, University of Wrocław, Poland
Max-Planck-Institute für Informatik, Saarbrücken, Germany
gawry@cs.uni.wroc.pl

Abstract. We consider a natural variant of the classical multiple pattern matching problem: given a Lempel-Ziv-Welch representation of a string $t[1\,..\,N]$ and a collection of (uncompressed) patterns $p_1, p_2, \ldots, p_\ell$ with $\sum_i |p_i| = M$, does any of p_i occur in t? As shown by Kida *et al.* [12], extending the single pattern algorithm of Amir, Benson and Farach [2] gives a running time of $\mathcal{O}(n + M^2)$ for the more general case. We prove that in fact it is possible to achieve $\mathcal{O}(n \log M + M)$ or $\mathcal{O}(n + M^{1+\epsilon})$ complexity. While not linear, running time of our solution matches the single pattern bounds achieved by [2] and [14] in a more structured and unified manner, and without using a lot of combinatorics on words. The only nontrivial components are the suffix array, constant time range minimum queries, and any balanced binary search trees.

Keywords: multiple pattern matching, compression, Lempel-Ziv-Welch.

1 Introduction

Pattern matching is the most natural problem concerning processing text data. It has been thoroughly studied, and many different linear time solutions are known, starting from the well-known Knuth-Morris-Pratt algorithm [13]. While it might seem that the existence of a linear time [5,13,15], constant space [9], and constant delay [8] solutions means that the question is completely solved, this is not quite the case. Whenever we must store a lot of text data, we store it in a compressed representation. This suggests a natural research direction: could we process this compressed representation without wasting time (and space) to uncompress it? Or, in other words, can we use the high compression ratio to accelerate the computation? In turns out that for pattern matching and some compression methods, the answer is yes. For the case of Lempel-Ziv-Welch [16] compressed text, there are two algorithms given by Amir, Benson, and Farach [2]: one with a $\mathcal{O}(n + m^2)$ running time, and one with $\mathcal{O}(n \log m + m)$, where m is the length of the pattern and n the size of the compressed representation of a text $t[1\,..\,N]$. Farach and Thorup [7] considered the more general case of

[⋆] Supported by MNiSW grant number N N206 492638, 2010–2012 and START scholarship from FNP.

Lempel-Ziv compression, and developed a (randomized) $\mathcal{O}(n \log^2 \frac{N}{n} + m)$ time algorithm. When the compression used is Lempel-Ziv-Welch, their complexity reduces to $\mathcal{O}(n \log \frac{N}{n} + m)$. In a recent paper we proved that in fact it is possible to achieve a (deterministic) linear running time for this case [10], even if both the pattern and the text are compressed [11]. A natural research direction is to consider *multiple pattern matching*, where instead of just one pattern we are given their collection $p_1, p_2, \ldots, p_\ell$ (which, for example, can be a set of forbidden words from a dictionary), and we should check if any of them occurs in the text. It is known that extending one of the algorithms given by Amir *et al.* results in a $\mathcal{O}(n + M^2)$ running time for multiple Lempel-Ziv-Welch-compressed pattern matching, where $M = \sum_i |p_i|$ [12]. It seems realistic that the set of the patterns is very large, and hence M^2 addend in the running time might be substantially larger than n. In this paper we prove that in fact it is possible to achieve $\mathcal{O}(n \log M + M)$ or $\mathcal{O}(n + M^{1+\epsilon})$ complexity for this problem, with the space usage being $\mathcal{O}(n + M)$ and $\mathcal{O}(n + M^{1+\epsilon})$, respectively. While such running time is not linear, it matches the best bounds for the single pattern case developed by [2] and [14] in a unified and structured manner, and is achieved using rather simple means. The main tool in our algorithms is reducing the problem to simple-to-state purely geometrical questions on an integer grid, which allows us to avoid using a lot of nontrivial combinatorics on words. Hence we believe they could be good candidates in real-life applications, and "simple" in the title is justified.

2 Preliminaries

Let $M = \sum_{i=1}^{\ell} |p_i|$ be the total size of all patterns. We assume that the alphabet Σ is either constant (which is the simple case) or consists of integers which can be sorted in linear time (in other words, polynomial in n and M). In the latter case we renumber the letters and assume the alphabet to be $\{0, 1, \ldots, M-1\}$). We consider strings of length N over Σ given in a Lempel-Ziv-Welch compressed form which are represented as a sequence of n *codewords* where a codeword is either a single letter, or a previously occurring codeword concatenated with a single character (using both n and N might be confusing, but it is enough to remember that big N stands for the big original size, while small n refers to the hopefully small compressed size; similarly, big M denotes the big original size of all patterns). This additional character is not given explicitly: we define it as the first character of the next codeword, and initialize the set of codewords to contain all single characters in the very beginning (this is a technical detail which is not important to us). The resulting compression method enjoys a particularly simple encoding/decoding process, but unfortunately requires outputting at least $\Omega(\sqrt{N})$ codewords. Still, its simplicity and good compression ratio achieved on real life instances make it an interesting model to work with. For the rest of the paper we will use LZW when referring to Lempel-Ziv-Welch compression.

We use the following notion for a string w: prefix(w) is the longest prefix of w which is a suffix of some pattern, suffix(w) is the longest suffix of w which is a prefix of some pattern, and w^r is its reversal.

To prove the main theorem we need to design a few data structures. To simplify the exposition we use the notion of a $\langle f(M), g(M)\rangle$ *structure* meaning that after a $f(M)$ time preprocessing we are able to execute one query in $g(M)$ time. If such structure is *offline*, we are able execute a sequence of t queries in total $f(M)+tg(M)$ time. Similarly, a $\langle f(M), g(M)\rangle$ *dynamic structure* allows updates in $f(M)$ time and queries in $g(M)$ time. It is *persistent* if updating creates a new copy instead of modifying the original data. The notion of persistence is well-studied, see for example [6].

We will extensively use the suffix tree T and the suffix array built for concatenation of all patterns separated by a special character \$ (which does not occur in either the text or any pattern, and is smaller than any original letter) which we call A:

$$A = p_1\$p_2\$\ldots\$p_{\ell-1}\$p_\ell$$

Similarly, T^r is the suffix tree built for the reversed concatenation A^r:

$$A^r = p_\ell^r\$p_{\ell-1}^r\$\ldots\$p_2^r\$p_1^r$$

Both suffix arrays are enriched with range minimum query structures enabling us to compute the longest common prefix and suffix of any two substrings in constant time.

Lemma 1 (see [3]). *Given an array $t[1\,..\,n]$ we can build in linear time and space a range minimum/maximum query structure* RMQ(t) *which allows computing the minimum/maximum $t[k]$ over all $k \in \{i, i+1, \ldots, j\}$ for a given i, j in constant time.*

Lemma 2 (see [3]). *A can be preprocessed in linear time so that given any two fragments $A[i\,..\,i+k]$ and $A[j\,..\,j+k]$ we can find their longest common prefix (suffix) in constant time.*

A snippet is any substring of any pattern $p_i[j\,..\,k]$. We represent it as a triple (i, j, k). Given such triple, we would like to retrieve the corresponding (explicit or implicit) node in the suffix tree (or reversed suffix tree) efficiently. $\langle f(M), g(M)\rangle$ *locator* allows $g(M)$ time retrieval after a $f(M)$ time preprocessing.

Lemma 3. *$\langle \mathcal{O}(M), \mathcal{O}(\log M)\rangle$ locator exists.*

Proof. $\langle \mathcal{O}(M \log M), \mathcal{O}(\log M)\rangle$ is very simple to implement: for each vertex of the suffix tree we construct a balanced search tree containing all its ancestors sorted according to their depths. Constructing the tree for a vertex requires inserting just one new element into its parent tree (note that most standard balanced binary search trees can be made persistent so that inserting a new number creates a new copy and does not destroy the old one) and so the whole construction takes $\mathcal{O}(M \log M)$ time. This is too much by a factor of $\log M$, though. We use the standard micro-macro tree decomposition (see [4]) to remove it. The suffix tree is partitioned into small subtrees by choosing at most $\frac{M}{\log M}$ macro nodes such that after removing them we get a collection of connected components of at most logarithmic size. Such partition can be easily found in linear time. Then for each macro node we construct a binary search tree containing all its macro ancestors sorted according to their depths. There are just $\frac{M}{\log M}$ macro nodes so the whole preprocessing is linear. To find the ancestor v at depth d we first retrieve the lowest macro ancestor u of v by following at most $\log M$ edges up from v. If none of the traversed vertices is the answer, we find the macro ancestor of u of largest depth not smaller than d using the binary search tree in $\mathcal{O}(\log M)$ time. Then retrieving the answer requires following at most $\log M$ edges up from u. □

To improve the query time in the above lemma we need to replace the balanced search tree. $\langle f(M), g(M)\rangle$ *dynamic dictionary* stores a subset S of $\{0, \ldots, M-1\}$ so that we can add or remove elements in $\mathcal{O}(M^{\epsilon})$ time, and check if a given x belongs to S (and if so, retrieve its associated information) or find its successor and predecessor in $\mathcal{O}(1)$ time.

Lemma 4. *$\langle \mathcal{O}(M^{\epsilon}), \mathcal{O}(1)\rangle$ persistent dynamic dictionary exists for any $\epsilon > 0$.*

Proof. Choose an integer $k \geq \frac{1}{\epsilon}$. The idea is to represent the numbers in base $B = M^{\frac{1}{k}}$ and store them in a trie of depth k. At each vertex we maintain a table child$[0\,..\,B-1]$ with the i-th element containing the pointer to the corresponding child, if any. This allows efficient checking if a given x belongs to the current set (we just inspect at most k vertices and at each of them use the table to retrieve the next one in constant time). Note that we do not create a vertex if its corresponding tree is empty. To find the successor (or predecessor) efficiently, we maintain at each vertex two additional tables next$[0\,..\,B-1]$ and prev$[0\,..\,B-1]$ where next$[i]$ is the smallest $j \geq i$ such that child$[j]$ is defined and prev$[i]$ is the largest $j \leq i$ such that child$[j]$ is defined. Using those tables the running time becomes $\mathcal{O}(k) = \mathcal{O}(1)$. Whenever we add or remove an element, we must recalculate the tables at all vertices from the traversed path. Its length is k and each table is of size B so the updates require $\mathcal{O}(kB) = \mathcal{O}(M^{\epsilon})$ time. Note that the whole structure is easily made persistent as after each update we create a new copy of the traversed path and do not modify any other vertices. □

Lemma 5. *$\left\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1)\right\rangle$ locator exists for any $\epsilon > 0$.*

Proof. The idea is the same as in Lemma 3: for each vertex of the suffix tree we construct a structure containing all its ancestors sorted according to their depths. Note that the depth are smaller than M so we can apply Lemma 4. The total construction time is $\mathcal{O}(M \times M^{\epsilon}) = \mathcal{O}(M^{1+\epsilon})$ and answering a query reduces to one predecessor lookup. □

We assume the following preprocessing for both the suffix tree and the reversed suffix tree.

Lemma 6. *A suffix tree built for a text of length M can be preprocessed in linear time so that given an implicit or explicit vertex v we can retrieve its pre- and post-order numbers (*pre(v) *and* post(v)*, respectively) in the uncompressed version of the tree (i.e., in the suffix trie) in constant time.*

Proof. For each explicit vertex, we store its pre- and post-order numbers in the suffix trie. To compute the numbers for an implicit vertex, we use the data stored at its lowest explicit ancestor. □

3 Overview of the Algorithm

We are given a sequence of blocks, each block being either a single letter, or a previously defined block concatenated with a single letter. For each block we would like to check if the corresponding word occurs in any of the patterns, and if not, we would like to find its longest suffix (prefix) which is a prefix (suffix) of any of the patterns. First we consider all blocks at once and for each of them compute its longest prefix which occurs in some p_i.

Lemma 7. *Given a LZW compressed text we can compute for all blocks the corresponding snippet (if any) and the longest prefix which is a suffix of some pattern in total linear time.*

Proof. The idea is the same as in the single pattern case [10]: intersect the suffix tree and the trie defined by all blocks at once. □

To compute the longest suffix which is a prefix of some pattern, we would like to use the Aho-Corasick automaton built for all $p_1, p_2, \ldots, p_\ell$, which is a standard multiple pattern matching tool [1]. Recall that its state set consists of all unique prefixes $p_i[1 \mathinner{..} j]$ organized in a trie. Additionally, each v stores the so-called *failure link* failure(v), which points to the longest proper suffix of the corresponding word which occurs in the trie as well. If the alphabet is of constant time, we can afford to build and store the full transition function of such automaton. If the alphabet is $\{0, 1, \ldots, M-1\}$, it is not clear if we can afford to store the full transition function. Nevertheless, storing the trie and all failure links are enough to navigate in amortized constant time per letter. This is not enough for our purposes, though, as we need a worst case bound. We start with building the trie and computing the failure links. This is trivial to perform in linear time after constructing the reversed suffix tree: each state is a (implicit or explicit) node of the tree with an outgoing edge starting with \$. Its failure link is simply the lowest ancestor corresponding to such node as well. Then depending on the preprocessing allowed we get two time bounds.

Lemma 8. *Given a LZW compressed text we can compute for all blocks the longest suffix which is a prefix of some pattern in total time $\mathcal{O}(n + M^{1+\epsilon})$ for any $\epsilon > 0$.*

Proof. At each vertex we create a $\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \rangle$ persistent dynamic dictionary. To create the dictionary for v we take the dictionary stored at failure(v) and update it by inserting all edges outgoing from v. There are at most M updates to all dictionaries, each of them taking $\mathcal{O}(M^{\epsilon})$ time, and then any query is answered in constant time, resulting in the claimed bound. □

Lemma 9. *Given a LZW compressed text we can compute for all blocks the longest suffix which is a prefix of some pattern in total time $\mathcal{O}(n \log M + M)$.*

Proof. For a vertex v consider the sequence of its ancestors $\text{failure}(v)$, $\text{failure}^2(v)$, $\text{failure}^3(v), \ldots$. To retrieve the transition $\delta(v, c)$ we should find the first vertex in this sequence having an outgoing edge starting with c. For each different character c we build a separate structure $S(c)$ containing all intervals $[\text{pre}(v), \text{post}(v)]$ for v having an outgoing edge starting with c, where $\text{pre}(v)$ and $\text{post}(v)$ are the pre- and post-order numbers of v in a tree defined by the failure links (i.e., $\text{failure}(v)$ is the parent of v there). Then to to calculate $\delta(v, c)$ we should locate the smallest interval containing $\text{pre}(v)$ in $S(c)$. By implementing $S(c)$ as a balanced search tree we get the claimed bound. □

Hence we reduced the original problem to multiple pattern matching in a collection of sequences of snippets, with the total size of all collections linear in n. To solve the latter, we try to simulate the Knuth-Morris-Pratt algorithm on each of those sequences. Of course we cannot afford to process the snippets letter-by-letter, and hence must develop efficient procedures operating on whole snippets. A high level description of the algorithm is given in MULTIPLE-PATTERN-MATCHING. prefixer and detector are low-level procedures which will be developed in the next section.

Note that instead of constructing the set P we could call $\text{detector}(c, s_k)$ directly but then its implementation would have to be online, and that seems difficult to achieve in the $\langle \mathcal{O}(M), \mathcal{O}(\log M) \rangle$ variant.

Algorithm 1. MULTIPLE-PATTERN-MATCHING($s_1, s_2, \ldots, s_{n'}$)

```
1: $P \leftarrow \emptyset$
2: $c \leftarrow s_1$
3: for $k = 2, 3, \ldots, n'$ do
4:     add $(c, s_k)$ to $P$
5:     $c \leftarrow \text{prefixer}(c, s_k)$
6: end for
7: for all $(s, s') \in P$ do
8:     $\text{detector}(s, s')$
9: end for
```

4 Multiple Pattern Matching in a Sequence of Snippets

A $\langle f(M), g(M) \rangle$ *prefixer* is a data structure which preprocesses the collection of patterns in $f(M)$ time so that given any two snippets we can compute the longest suffix of their concatenation which is a prefix of some pattern in $g(M)$ time.

Lemma 10. *$\langle \mathcal{O}(M), \mathcal{O}(\log M) \rangle$ prefixer exists.*

Proof. Let the two snippets be s_1 and s_2. First note that using Lemma 2 we can compute the longest common prefix of $s_2^r s_1^r$ and a given suffix of A^r in constant time. Hence we can apply binary search to find the (lexicographically) largest suffix of A^r which either begins with $s_2^r s_1^r$ or is (lexicographically) smaller in $\mathcal{O}(\log M)$ time. Given this suffix $A^r[i\,..\,|A^r|]$ we compute $d = |\text{LCP}(|s_s^r s_1^r|, A^r[i\,..\,|A^r|])|$ and apply Lemma 3 to retrieve the ancestor v of $A^r[i\,..\,|A^r|]$ at depth d in $\mathcal{O}(\log M)$ time. The longest prefix we are looking for corresponds to an ancestor u of v which has at least one outgoing edge starting with \$. Observe that such u must be explicit as there are no \$ characters on the root-to-v path. This means that we can apply a simple linear time preprocessing to compute such u for each possible explicit v in linear time. Then given a (possibly implicit) v we use the preprocessing to compute the u corresponding to the longest prefix in constant time, giving a $\mathcal{O}(\log M)$ total query time. □

Lemma 11. $\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \rangle$ *prefixer exists for any* $\epsilon > 0$.

Proof. For each pattern p_i we consider all possibilities to cut it into two parts $p_i = p_i[1\,..\,j]p_i[j+1\,..\,|p_i|]$. For each cut we locate vertex u corresponding to $p_i[1\,..\,j]$ in the reversed suffix tree and v to $p_i[j+1\,..\,|p_i|]$ in the suffix tree. By Lemma 5 it takes constant time and by Lemma 6 we can then compute $\text{pre}(u)$, $\text{pre}(v)$ and $\text{post}(v)$. Then we add a horizontal segment $\{\text{pre}(u)\} \times [\text{pre}(v), \text{post}(v)]$ with weight j to the collection. Now consider a query consisting of two snippets s_1 and s_2. First locate the vertex u corresponding to s_1 in the reversed suffix tree and v to s_2 in the suffix tree. Then construct a vertical segment $[\text{pre}(u), \text{post}(u)] \times \{\text{pre}(v)\}$ and observe that the query reduces to finding the heaviest horizontal segment in the collection it intersects (if there is none, we retrieve the lowest ancestor of v which has an outgoing edge starting with \$, which can be precomputed in linear time), see Figure 1. Additionally, the horizontal segments are either disjoint or contained in each other. If the latter case, weight of the longer segment is bigger than weight of the shorter. To this end we prove that there exists a $\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \rangle$ structure for computing the heaviest horizontal segments intersected by a given vertical segments in such collection on a $M^2 \times M^2$ grid.

We sweep the grid from left to right maintaining a structure describing the currently active horizontal segments. The structure is based on the idea from Lemma 5 with $k \geq \frac{2}{\epsilon}$. Each leaf corresponds to a different y coordinate and stores all active horizontal segments with this coordinate on stack, with the most recently encountered segment on top (because weights of intersecting segments are monotone with their lengths, it is also the heaviest segment). Each inner vertex stores a table heaviest$[0\,..\,M^{\frac{2}{k}}]$ with the i-th element containing the maximum weight in the subtree corresponding to the i-th leaf, if any. Additionally, a range maximum query structure RMQ(heaviest) is stored so that given any two indices i, j we can compute the maximum heaviest$[k]$ over all $k \in \{i, i+1, \ldots, j\}$ in constant time. Adding or removing an active segment require locating the corresponding stack and either pushing a new element or removing the topmost element. Then we must update the tables at all ancestors of the corresponding

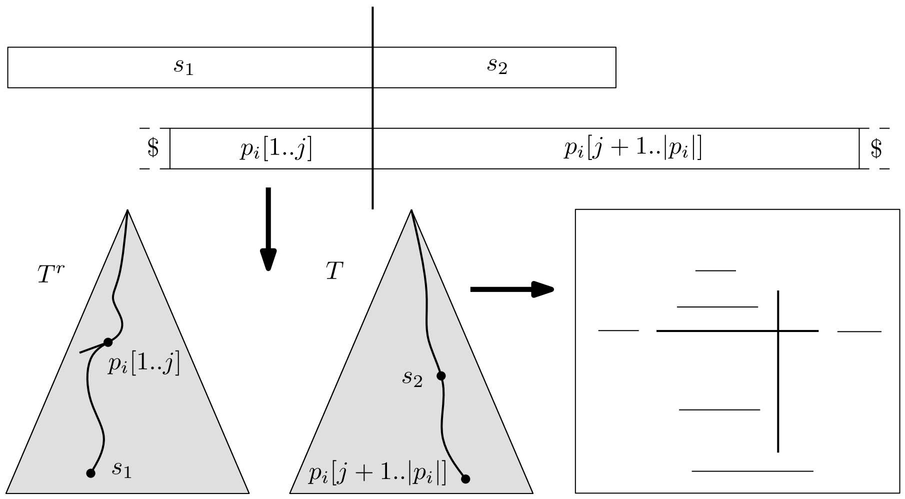


Fig. 1. Reducing prefixer queries to segments intersection

leaf, which by Lemma 1 takes $k\mathcal{O}(M^{\frac{2}{k}}) = \mathcal{O}(M^{1+\epsilon})$ time. Given a query, we first locate the appropriate version of the structure. Then we traverse the trie and find the heaviest intersected segment by asking at most $2k$ range maximum queries. $\square$

A $\langle f(M), g(M)\rangle$ *detector* is a data structure which preprocesses the collection of patterns in time $f(M)$ so that given any two snippets we can detect an occurrence of a pattern in a their concatenation in $g(M)$ time. Both implementation that we are going to develop are based on the same idea of reducing the problem to a purely geometric question on an integer grid, similar to the one from Lemma 11. For each pattern p_i we consider all possibilities to cut it into two parts $p_i = p_i[1..j]p_i[j+1..|p_i|]$. For each cut we locate in constant time vertex u corresponding to $p_i[1..j]$ in the reversed suffix tree and v to $p_i[j+1..|p_i|]$ in the suffix tree. If both u and v are explicit vertices, add a rectangle $[\text{pre}(u), \text{post}(u)] \times [\text{pre}(v), \text{post}(v)]$ to the collection. Then given two snippets s_1 and s_2 detecting an occurrence in their concatenation reduces in constant time to retrieving any rectangle containing $(\text{pre}(u), \text{pre}(v))$ where u is the vertex corresponding to s_1 in the reversed suffix tree and v to s_2 in the suffix tree, see Figure 2. Note that the x and y projections of any two rectangles in the collection are either disjoint or contained in each other. Assuming no pattern occurs in another, no two rectangles are contained in each other (if some p_i occurs in some p_j, which can be efficiently detected in the preprocessing stage, we can forget about p_j). We call a collection with such two properties *valid.*

Lemma 12. *$\langle \mathcal{O}(M), \mathcal{O}(\log M)\rangle$ offline detector exists.*

Proof. Recall that in the offline version we are given all queries in an advance. We sweep the grid from left to right maintaining a structure describing currently

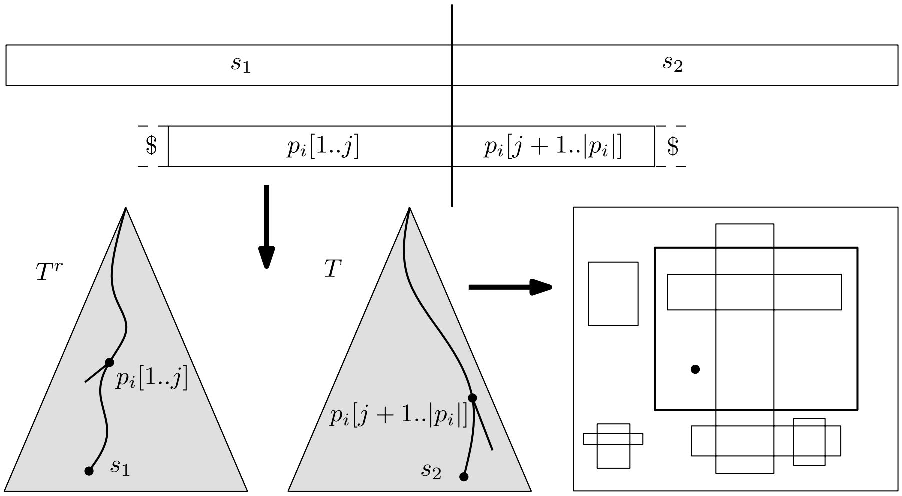


Fig. 2. Reducing detector queries to rectangles retrieval in a valid collection

intersected rectangles. At a high level the structure is just a full binary tree on M leaves corresponding to different y coordinates (and each inner vertex corresponding to an continuous interval of y coordinates). If we aim to achieve logarithmic time of both update and query, the implementation is rather straightforward. We want to achieve constant time update, though. Say that we encounter a new rectangle and need to insert an interval $[y_1, y_2]$ with $y_1 < y_2$ into the structure. We compute the lowest common ancestor v of the leaves corresponding to y_1 and y_2 in the tree (as the tree is full there exists a simple arithmetic formula for that) and call v *responsible* for $[y_1, y_2]$. v corresponds to an interval $\left[\alpha 2^\ell, (\alpha+2)2^\ell\right)$ such that $y_1 \in \left[\alpha 2^\ell, (\alpha+1)2^\ell\right)$ and $y_2 \in \left[(\alpha+1)2^\ell, (\alpha+2)2^\ell\right)$. For each inner vertex we store its *interval stack*. To insert $[y_1, y_2]$ we simply push it on the interval stack of the responsible vertex. Note that because the collection is valid, all intervals $I_1, I_2, \ldots, I_k$ stored on the same interval stack at a given moment are nested, i.e., $I_1 \subseteq I_2 \subseteq \ldots \subseteq I_k$. To remove an interval we locate the responsible vertex and pop the topmost element from its interval stack. The only nontrivial part is detecting an interval containing a given point x. First traverse the path starting at the corresponding leaf. This gives us a sequence of $\log M$ interval stacks. Observe that for a fixed interval stack it is enough to check if its top element (if any) contains x, hence $\mathcal{O}(\log M)$ query time follows. □

Lemma 13. *$\left\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \right\rangle$ detector exists for any $\epsilon > 0$.*

Proof. We sweep the grid from left to right maintaining a structure describing currently intersected rectangles. The structure should allow adding or removing intervals from $\{0, 1, \ldots, M-1\}$ and retrieving any interval containing a specified point. A straightforward use of the idea from Lemma 4 allows an efficient implementation of those operations in $\mathcal{O}(M^\epsilon)$ and $\mathcal{O}(1)$ time, respectively. □

By plugging either Lemma 10 and Lemma 12 or Lemma 11 and Lemma 13 into Multiple-pattern-matching we get the main theorem.

Theorem 1. *Multiple pattern matching in a sequence of n snippets can be performed in $\mathcal{O}(n \log M + M)$ or $\mathcal{O}(n + M^{1+\epsilon})$ time, where M is the combined size of all patterns.*

By adding Lemma 7 and either Lemma 9 or Lemma 8 we get the claimed total running time of the whole solution.

Theorem 2. *Multiple pattern matching in LZW compressed texts can be performed in $\mathcal{O}(n \log M + M)$ or $\mathcal{O}(n + M^{1+\epsilon})$ time, where M is the combined size of all patterns and n size of the compressed representation. The space used by the solutions is $\mathcal{O}(n + M)$ and $\mathcal{O}(n + M^{1+\epsilon})$, respectively.*

5 Conclusions

We presented two efficient solutions for detecting if some pattern occurs in a LZW-compressed text. A natural generalization is detecting **all** occurrences. If no pattern occurs in another, a straightforward generalizations of Lemma 12 and Lemma 13 allow generating all occ occurrences in $\mathcal{O}(n \log M + M + occ)$ and $\mathcal{O}(n + M^{1+\epsilon} + occ)$ time, respectively. Without such assumptions, there is one problem, though: we cannot assume that the collection of rectangles in Lemma 12 is valid, and $\mathcal{O}((n+M) \log M + occ)$ running time seems to be required. We leave dealing with this problem as future work. Additionally, we offer the following open problem: how quickly can we just count the number of occurrences?

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM 18, 333–340 (1975)
2. Amir, A., Benson, G., Farach, M.: Let sleeping files lie: pattern matching in z-compressed files. In: SODA 1994: Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, PA, USA, pp. 705–714. Society for Industrial and Applied Mathematics (1994)
3. Bender, M.A., Farach-Colton, M.: The LCA Problem Revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
4. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. 321(1), 5–12 (2004)
5. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10), 762–772 (1977)
6. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. 38(1), 86–124 (1989)
7. Farach, M., Thorup, M.: String matching in Lempel-Ziv compressed strings. In: STOC 1995: Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, pp. 703–712. ACM, New York (1995)

8. Galil, Z.: String matching in real time. J. ACM 28(1), 134–149 (1981)
9. Galil, Z., Seiferas, J.: Time-space-optimal string matching (preliminary report). In: STOC 1981: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, pp. 106–113. ACM, New York (1981)
10. Gawrychowski, P.: Optimal pattern matching in LZW compressed strings. In: Randall, D. (ed.) SODA, pp. 362–372. SIAM (2011)
11. Gawrychowski, P.: Tying up the loose ends in fully LZW-compressed pattern matching. In: Dürr, C., Wilke, T. (eds.) STACS. LIPIcs, vol. 14, pp. 624–635. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
12. Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., Arikawa, S.: Multiple pattern matching in LZW compressed text. In: Proceedings of Data Compression Conference, DCC 1998, pp. 103–112. IEEE (1998)
13. Knuth, D.E., Morris Jr., J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(2), 323–350 (1977)
14. Kosaraju, S.R.: Pattern Matching in Compressed Texts. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 349–362. Springer, Heidelberg (1995)
15. Morris Jr., J.H., Pratt, V.R.: A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley (1970)
16. Welch, T.A.: A technique for high-performance data compression. Computer 17(6), 8–19 (1984)

Computing the Burrows-Wheeler Transform of a String and Its Reverse

Enno Ohlebusch[1], Timo Beller[1], and Mohamed I. Abouelhoda[2,3]

[1] Institute of Theoretical Computer Science, University of Ulm, 89069 Ulm, Germany
{Enno.Ohlebusch,Timo.Beller}@uni-ulm.de
[2] Center for Informatics Sciences, Nile University, Giza, Egypt
mabouelhoda@yahoo.com
[3] Faculty of Engineering, Cairo University, Giza, Egypt

Abstract. The contribution of this paper is twofold. First, we provide new theoretical insights into the relationship between a string and its reverse: If the Burrows-Wheeler transform (BWT) of a string has been computed by sorting its suffixes, then the BWT and the longest common prefix array of the reverse string can be derived from it without suffix sorting. Furthermore, we show that the longest common prefix arrays of a string and its reverse are permutations of each other. Second, we provide a parallel algorithm that, given the BWT of a string, computes the BWT of its reverse much faster than all known (parallel) suffix sorting algorithms. Some bioinformatics applications will benefit from this.

1 Introduction

The Burrows-Wheeler transform [2] is used in many lossless data compression programs, of which the best known is Julian Seward's bzip2. Moreover, it is the basis of FM-indexes that support backward search [4]. In some bioinformatics applications, one needs both the Burrows-Wheeler transform BWT of a string S and the Burrows-Wheeler transform BWT^{rev} of the reverse string S^{rev}. For example, in the prediction of RNA-coding genes [15] or short read alignment [12], it is advantageous to be able to search in forward and backward direction. This bidirectional search requires BWT as support for backward search and BWT^{rev} as support for forward search. Another example is *de novo* sequence assembly based on pairwise overlaps between sequence reads. Simpson and Durbin [16] showed how an assembly string graph can be efficiently constructed using all pairs of exact suffix-prefix overlaps between reads. (Välimäki et al. [17] provide techniques to find all pairs of approximate suffix-prefix overlaps.) To compute overlaps between reverse complemented reads, they build an FM-index for the set of reads *and* an FM-index for the set of *reversed* reads.

The Burrows-Wheeler transform BWT of a string S is usually computed by sorting all suffixes of S (hence the suffix array of S is known). Of course, the Burrows-Wheeler transform BWT^{rev} of the reverse string S^{rev} can be obtained in the same fashion. However, because of the strong relationship between a string and its reverse, it is quite natural to ask whether BWT^{rev} can be directly derived from BWT—without sorting the suffixes of S^{rev}. In this paper, we prove that this is indeed the case. More precisely, we give an algorithm for this task that has $O(n \log \sigma)$ worst-case time complexity. (If needed, the suffix array SA^{rev} of S^{rev} can easily be obtained from BWT^{rev}; see e.g. [13].) Interestingly, essentially the same algorithm can be applied to obtain the Burrows-Wheeler transform of the reverse complement of a DNA sequence.

We further study the relationship between the lcp-array LCP of S and the lcp-array LCP^{rev} of S^{rev}. To be precise, we prove that LCP^{rev} is a permutation of LCP. Furthermore, we show that LCP^{rev} can also be directly computed: by just one additional statement, it is possible to compute all irreducible lcp-values of LCP^{rev} with the same algorithm, and the remaining reducible lcp-values of LCP^{rev} can easily be derived from them.

In contrast to suffix sorting, our new algorithm is easy to parallelize. Experiments show that it is faster than all known (parallel) suffix sorting algorithms.

2 Preliminaries

Let Σ be an ordered alphabet of size σ whose smallest element is the so-called sentinel character \$. In the following, S is a string of length n over Σ having the sentinel character at the end (and nowhere else). For $1 \leq i \leq n$, $S[i]$ denotes the *character at position* i in S. For $i \leq j$, $S[i..j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j. Furthermore, S_i denotes the ith suffix $S[i..n]$ of S.

The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of the string S, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$; see Fig. 1 for an example. We refer to the overview article [14] for construction algorithms (some of which have linear run time). In the following, ISA denotes the inverse of the permutation SA.

The *suffix tree* ST for S is a compact trie storing the suffixes of S: for any leaf i, the concatenation of the edge labels on the path from the root to leaf i exactly spells out the suffix S_i. In the following, we denote an internal node α in ST by $\overline{\omega}$, where ω is the concatenation of the edge labels on the path from the root to α. A pointer from an internal node $\overline{c\omega}$ to the internal node $\overline{\omega}$ is called a *suffix link*; see [8] for details.

The Burrows and Wheeler transform [2] converts a string S into the string $\mathsf{BWT}[1..n]$ defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i]-1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] =$ \$ otherwise; see Fig. 1. The permutation LF, defined by $LF(i) = \mathsf{ISA}[\mathsf{SA}[i]-1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $LF(i) = 1$ otherwise, is called LF-mapping. The LF-mapping can be implemented by $LF(i) = C[c] + Occ(c, i)$, where $c = \mathsf{BWT}[i]$, $C[c]$ is the overall number of characters in S which are strictly smaller than c, and $Occ(c, i)$ is the number of occurrences of the character c in $\mathsf{BWT}[1..i]$.

The *lcp-array* of S is an array LCP such that $\mathsf{LCP}[1] = -1$ and $\mathsf{LCP}[i] = |\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \leq i \leq n$, where $\mathsf{lcp}(u, v)$ denotes the longest common prefix between two strings u and v; see Fig. 1. It can be computed in linear time from the suffix array and its inverse; see [11,13,10,6]. A value $\mathsf{LCP}[i]$ is called *reducible* if $\mathsf{BWT}[i] = \mathsf{BWT}[i-1]$; otherwise it is *irreducible*.

i	SA	BWT	$S_{\mathsf{SA}[i]}$
1	10	g	$\$$
2	3	t	$aataatg\$$
3	6	t	$aatg\$$
4	4	a	$ataatg\$$
5	7	a	$atg\$$
6	1	$\$$	$ctaataatg\$$
7	9	t	$g\$$
8	2	c	$taataatg\$$
9	5	a	$taatg\$$
10	8	a	$tg\$$

i	LCP^{rev}	BWT^{rev}	$S^{rev}_{\mathsf{SA}^{rev}[i]}$
1	-1	c	$\$$
2	$\underline{0}$	t	$aataatc\$$
3	3	t	$aatc\$$
4	$\underline{1}$	a	$ataatc\$$
5	2	a	$atc\$$
6	$\underline{0}$	t	$c\$$
7	$\underline{0}$	$\$$	$gtaataatc\$$
8	$\underline{0}$	g	$taataatc\$$
9	$\underline{4}$	a	$taatc\$$
10	$\underline{1}$	a	$tc\$$

Fig. 1. Left-hand side: suffix array SA and BWT of the string $S = ctaataatg\$$ (the input). Right-hand side: Burrows-Wheeler transform BWT^{rev} of $S^{rev} = gtaataatc\$$ (the output). The computation of LCP^{rev} will be explained in Sect. 4.

Ferragina and Manzini [4] showed that it is possible to search a pattern backwards, character by character, in the suffix array SA of string S, without storing SA. Let $c \in \Sigma$ and ω be a substring of S. Given the ω-interval $[i..j]$ in the suffix array SA of S (i.e., ω is a prefix of $S_{\mathsf{SA}[k]}$ for all $i \leq k \leq j$, but ω is not a prefix of any other suffix of S), $backwardSearch(c, [i..j])$ returns the $c\omega$-interval $[C[c] + Occ(c, i-1) + 1 .. C[c] + Occ(c, j)]$. In our example of Fig. 1, $backwardSearch(a, [2..5])$ returns the aa-interval $[2..3]$.

The *wavelet tree* introduced by Grossi et al. [7] supports one backward search step in $O(\log \sigma)$ time. To explain this data structure, we may view the ordered alphabet Σ as an array of size σ so that the characters appear in ascending order in the array $\Sigma[1..\sigma]$, i.e., $\Sigma[1] = \$ < \Sigma[2] < \ldots < \Sigma[\sigma]$. We say that an interval $[l..r]$ is an *alphabet interval*, if it is a subinterval of $[1..\sigma]$. For an alphabet interval $[l..r]$, the string $\mathsf{BWT}^{[l..r]}$ is obtained from the Burrows-Wheeler transformed string BWT of S by deleting all characters in BWT that do not belong to the sub-alphabet $\Sigma[l..r]$ of $\Sigma[1..\sigma]$. The wavelet tree of the string BWT over the alphabet $\Sigma[1..\sigma]$ is a balanced binary search tree defined as follows. Each node v of the tree corresponds to a string $\mathsf{BWT}^{[l..r]}$, where $[l..r]$ is an alphabet interval. The root of the tree corresponds to the string $\mathsf{BWT} = \mathsf{BWT}^{[1..\sigma]}$. If $l = r$, then v has no children. Otherwise, v has two children: its left child corresponds to the string $\mathsf{BWT}^{[l..m]}$ and its right child corresponds to the string $\mathsf{BWT}^{[m+1..r]}$, where $m = \lfloor \frac{l+r}{2} \rfloor$. In this case, v stores a bit vector $B^{[l..r]}$ whose i-th entry is 0 if the i-th character in $\mathsf{BWT}^{[l..r]}$ belongs to the sub-alphabet $\Sigma[l..m]$ and 1 if it belongs to the sub-alphabet $\Sigma[m+1..r]$. To put it differently, an entry in the bit vector is 0 if the corresponding character belongs to the left subtree and 1 if it belongs to the right subtree. Moreover, each bit vector B in the tree is preprocessed so that the queries $rank_0(B, i)$ and $rank_1(B, i)$ can be answered in constant time, where $rank_b(B, i)$ is the number of occurrences of bit b in $B[1..i]$. Obviously, the wavelet tree has height $O(\log \sigma)$. Because in an actual implementation it suffices to store only the bit vectors, the wavelet tree requires only $n \log \sigma$ bits of space

Algorithm 1. For an ω-interval $[i..j]$, the function call *getIntervals*($[i..j]$) returns the list of all $c\omega$-intervals, and is defined as follows.

```
$getIntervals([i..j])$
  $list \leftarrow [\,]$
  $getIntervals'([i..j], [1..\sigma], list)$
  **return** $list$

$getIntervals'([i..j], [l..r], list)$
  **if** $l = r$ **then**
    $c \leftarrow \Sigma[l]$
    $add(list, [C[c] + i..C[c] + j])$
  **else**
    $(a_0, b_0) \leftarrow (rank_0(B^{[l..r]}, i - 1), rank_0(B^{[l..r]}, j))$
    $(a_1, b_1) \leftarrow (i - 1 - a_0, j - b_0)$
    $m = \lfloor \frac{l+r}{2} \rfloor$
    **if** $b_0 > a_0$ **then**
      $getIntervals'([a_0 + 1..b_0], [l..m], list)$
    **if** $b_1 > a_1$ **then**
      $getIntervals'([a_1 + 1..b_1], [m + 1..r], list)$
```

plus $o(n \log \sigma)$ bits for the data structures that support rank queries in constant time.

For an ω-interval $[i..j]$, the procedure *getIntervals*($[i..j]$) presented in Algorithm 1 returns the list of all $c\omega$-intervals; cf. [1,3]. More precisely, it starts with the ω-interval $[i..j]$ at the root and traverses the wavelet tree in a top down fashion as follows. At the current node v, it uses constant time rank queries to obtain the number $b_0 - a_0$ of zeros in the bit vector of v within the current interval. If $b_0 > a_0$, then there are characters in $\mathsf{BWT}[i..j]$ that belong to the left subtree of v, and the algorithm proceeds recursively with the left child of v. Furthermore, if the number of ones is positive (i.e. if $b_1 > a_1$), then it proceeds with the right child in an analogous fashion. Clearly, if a leaf corresponding to character c is reached with current interval $[p..q]$, then $[C[c]+p \,..\, C[c]+q]$ is the $c\omega$ interval. In this way, Algorithm 1 computes the list of all $c\omega$-intervals. This takes $O(k \log \sigma)$ time for a k-element list. In our example of Fig. 1, *getIntervals*($[2..5]$) returns the list $[[2..3], [8..9]]$, where $[2..3]$ is the aa-interval and $[8..9]$ is the ta-interval.

3 The Burrows-Wheeler Transform of the Reverse String

If we reverse the order of the characters in a string, we obtain its *reverse string*. For technical reasons, however, we assume that the sentinel symbol \$ occurs at the end of each string under consideration. For this reason, the reverse string S^{rev} of a string S that is terminated by \$ is obtained by deleting \$ from S, reversing the order of the characters, and appending \$. For example, the reverse string of $S = ctaataatg\$$ is $S^{rev} = gtaataatc\$$ (and not $\$gtaataatc$).

Algorithm 2. Procedure $bwtrev(k, [i..j], \ell)$ uses the wavelet tree of the BWT, the suffix array SA, and S. The call $bwtrev(1, [1..n], 0)$ computes BWT^{rev}.

```
$bwtrev(k, [i..j], \ell)$
  $list \leftarrow getIntervals([i..j])$        /* intervals in increasing lexicographic order */
  while $list$ not empty do
    $[lb..rb] \leftarrow head(list)$
    if $lb = rb$ then
      $pos \leftarrow \mathsf{SA}[lb] + \ell + 1$
      if $pos > n$ then
        $pos \leftarrow pos - n$
      $\mathsf{BWT}^{rev}[k] \leftarrow S[pos]$
      $k \leftarrow k + 1$
    else
      if $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[rb] + \ell + 1]$ then
        for $q \leftarrow k$ to $k + rb - lb$ do
          $\mathsf{BWT}^{rev}[q] \leftarrow S[\mathsf{SA}[lb] + \ell + 1]$
      else
        $bwtrev(k, [lb..rb], \ell + 1)$
      $k \leftarrow k + rb - lb + 1$
    if $list$ not empty then $\mathsf{LCP}^{rev}[k] \leftarrow \ell$        /* this will be explained in Sect. 4 */
```

Given the BWT of S, Algorithm 2 recursively computes the Burrows-Wheeler transformed string BWT^{rev} of S^{rev} by the procedure call $bwtrev(1, [1..n], 0)$. We exemplify the algorithm by applying it to $S = ctaataatg\$$. The procedure call $getIntervals([1..10])$ returns the list $[[1..1], [2..5], [6..6], [7..7], [8..10]]$, where $[1..1]$ is the \$-interval, $[2..5]$ is the a-interval, and so on; cf. Fig. 1. Then, the first interval $[lb..rb] = [1..1]$ is taken from the list (note that $head(list)$ removes the first element of $list$ and returns it). Because $lb = rb$, the algorithm computes $pos = \mathsf{SA}[1] + 0 + 1 = 11$. Furthermore, since $pos > n = 10$, it assigns the character $S[11 - 10] = S[1] = c$ to $\mathsf{BWT}^{rev}[1]$ and increments k (so the new value of k is 2). Now the interval $[lb..rb] = [2..5]$ is taken from the list. Because $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[2] + 0 + 1] = S[4] = a \neq t = S[8] = S[\mathsf{SA}[5] + 0 + 1] = S[\mathsf{SA}[rb] + \ell + 1]$, Algorithm 2 recursively calls $bwtrev(2, [2..5], 1)$. The procedure call $getIntervals([2..5])$ returns the list $[[2..3], [8..9]]$, where $[2..3]$ is the aa-interval and $[8..9]$ is the ta-interval. Then, the first interval $[lb..rb] = [2..3]$ is taken from the list. In this case, $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[2] + 1 + 1] = S[5] = t = t = S[8] = S[\mathsf{SA}[3] + 1 + 1] = S[\mathsf{SA}[rb] + \ell + 1]$. Thus, t is assigned to both $\mathsf{BWT}^{rev}[2]$ and $\mathsf{BWT}^{rev}[3]$. Now the algorithm continues with the new value $k = 2 + 3 - 2 + 1 = 4$. Fig. 2 shows the recursion tree of Algorithm 2.

In essence, the correctness of Algorithm 2 is a consequence of the following lemma. Moreover, the lemma allows us to parallelize the algorithm.

Lemma 1. *Let $[i..j]$ be the ω-interval for some substring ω of S, and let k be the left boundary of the ω^{rev}-interval in SA^{rev}. If $[lb_1..rb_1], \ldots, [lb_m..rb_m]$ are the intervals in $list = getIntervals([i..j])$ corresponding to the strings $c_1\omega, \ldots, c_m\omega$,*

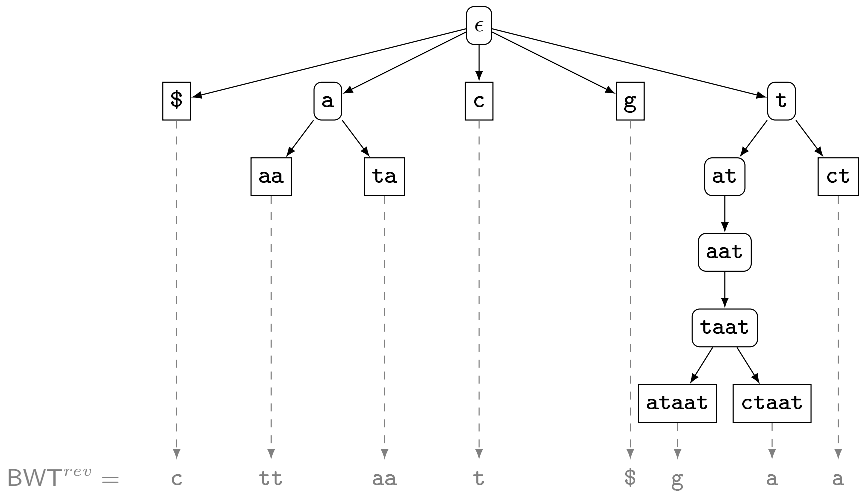


Fig. 2. The recursion tree of Algorithm 2 applied to the BWT of $S = ctaataatg\$$: for each internal node there is a recursive call. E.g., the recursive call with the aat-interval as parameter determines the characters of BWT^{rev} in the taa-interval of SA^{rev}.

where $c_1 < \ldots < c_m$*, then the intervals* $[s_1..e_1], \ldots, [s_m..e_m]$ *in* SA^{rev} *corresponding to the strings* $\omega^{rev}c_1, \ldots, \omega^{rev}c_m$ *satisfy* $s_q = k + \sum_{p=1}^{q-1}(rb_p - lb_p + 1)$ *and* $e_q = s_q + (rb_q - lb_q)$*, where* $1 \leq q \leq m$*.*

Proof. We prove the lemma by finite induction on q. In the base case $q = 1$. Because c_1 is the smallest character in Σ for which the $c_1\omega$-interval is non-empty, the suffixes of S^{rev} that have $\omega^{rev}c_1$ as a prefix are lexicographically smaller than those suffixes of S^{rev} that have $\omega^{rev}c_p$, $2 \leq p \leq m$, as a prefix. Hence $s_1 = k$. Moreover, it follows from the fact that the $\omega^{rev}c_1$-interval has size $rb_1 - lb_1 + 1$ that the $\omega^{rev}c_1$-interval is the interval $[k..k + (rb_1 - lb_1)]$. For the inductive step suppose that the $\omega^{rev}c_{q-1}$-interval $[s_{q-1}..e_{q-1}]$ satisfies $s_{q-1} = k + \sum_{p=1}^{q-2}(rb_p - lb_p + 1)$ and $e_{q-1} = s_{q-1} + (rb_{q-1} - lb_{q-1})$. Because c_q is the q-th smallest character in Σ for which the $c_q\omega$-interval is non-empty, the suffixes of S^{rev} that have $\omega^{rev}c_q$ as a prefix are lexicographically larger than the suffixes of S^{rev} that have $\omega^{rev}c_p$ as a prefix, where $1 \leq p \leq q-1$. It follows that the $\omega^{rev}c_q$-interval $[s_q..e_q]$ satisfies $s_q = e_{q-1} + 1 = k + \sum_{p=1}^{q-1}(rb_p - lb_p + 1)$ and $e_q = s_q + (rb_q - lb_q)$. □

Theorem 1. *Algorithm 2 correctly computes* BWT^{rev}*.*

Proof. We prove the theorem by induction on k. Suppose Algorithm 2 is applied to the ω-interval $[i..j]$ for some ℓ-length substring ω of S, and let the $c\omega$-interval $[lb..rb]$ be the interval dealt with in the current execution of the while-loop. According to the inductive hypothesis, $\mathsf{BWT}^{rev}[1..k-1]$ has been computed correctly. Moreover, by Lemma 1, $[k..k+rb-lb]$ is the $\omega^{rev}c$-interval in SA^{rev}. We further proceed by a case-by-case analysis.

– If $lb = rb$, then $c\omega$ occurs exactly once in S and it is the length $\ell+1$ prefix of suffix $S_{\mathsf{SA}[lb]}$. In this case, the suffix of S^{rev} that has $\omega^{rev}c$ as a prefix is the k-th lexicographically smallest suffix of S^{rev}. The character $\mathsf{BWT}^{rev}[k]$ is $S[\mathsf{SA}[lb]+\ell+1]$ because this is the character that immediately follows the prefix $S[\mathsf{SA}[lb]..\mathsf{SA}[lb]+\ell] = c\omega$ of suffix $S_{\mathsf{SA}[lb]}$.
– If $lb \neq rb$ and $S[\mathsf{SA}[lb]+\ell+1] = S[\mathsf{SA}[rb]+\ell+1]$, then each occurrence of $c\omega$ in S is followed by the same character $a = S[\mathsf{SA}[lb]+\ell+1]$. Thus, each suffix of S^{rev} in the $\omega^{rev}c$-interval $[k..k+rb-lb]$ is preceded by a. Consequently, $\mathsf{BWT}^{rev}[q] = a$ for every q with $k \leq q \leq k+rb-lb$. So in this case, it is not necessary to know the exact lexicographic order of the suffixes in the $\omega^{rev}c$-interval.
– If $lb \neq rb$ and $S[\mathsf{SA}[lb]+\ell+1] \neq S[\mathsf{SA}[rb]+\ell+1]$, then not all the characters in $\mathsf{BWT}^{rev}[k..k+rb-lb]$ are the same and thus the recursive call $bwtrev(k, [lb..rb], \ell+1)$ determines the lexicographic order of the suffixes in the $\omega^{rev}c$-interval $[k..k+rb-lb]$ as far as it is needed. □

Next, we analyse the worst-case time complexity of Algorithm 2.

Lemma 2. *The procedure bwtrev is executed with the parameters $(k, [i..j], \ell)$, where $[i..j]$ is the ω-interval for some substring ω of S, if and only if $\overline{\omega}$ is an internal node in the suffix tree ST of S.*

Proof. We use induction on ℓ. There is just one procedure call with $\ell = 0$, namely $bwtrev(1, [1..n], 0)$. The interval $[1..n]$ is the ε-interval, where ε denotes the empty string. Clearly, the node $\overline{\varepsilon}$ is the root node of the suffix tree ST, and the root is an internal node. According to the inductive hypothesis, the procedure $bwtrev$ is executed with the parameters $(k, [i..j], \ell)$, where $[i..j]$ is the ω-interval for some substring ω of S, if and only if $\overline{\omega}$ is an internal node in the suffix tree ST of S. For the inductive step, assume that $[lb..rb]$ is one of the intervals returned by the procedure call $getIntervals([i..j])$, say the $c\omega$-interval. We prove that there is a recursive procedure call $bwtrev(k', [lb..rb], \ell+1)$ if and only if $\overline{c\omega}$ is an internal node of ST. It is readily verified that $\overline{c\omega}$ is an internal node of ST if and only if the $c\omega$-interval contains two different suffixes of S, one having $c\omega a$ as a prefix and one having $c\omega b$ as a prefix, where a and b are different characters from Σ. Again, we proceed by case analysis:

– If $lb = rb$, then $c\omega$ occurs exactly once in S. Hence $\overline{c\omega}$ is not an internal node of ST. Note that Algorithm 2 does not invoke a recursive call to $bwtrev$.
– If $S[\mathsf{SA}[lb]+\ell+1] = S[\mathsf{SA}[rb]+\ell+1]$, then each occurrence of $c\omega$ in S is followed by the same character. Again, $\overline{c\omega}$ is not an internal node of ST and Algorithm 2 does not invoke a recursive call to $bwtrev$.
– If $a = S[\mathsf{SA}[lb]+\ell+1] \neq S[\mathsf{SA}[rb]+\ell+1] = b$, then $\overline{c\omega}$ is an internal node of ST and Algorithm 2 invokes the recursive call $bwtrev(k', [lb..rb], \ell+1)$. □

i	SA	BWT	$S_{\mathsf{SA}[i]}$
1	10	g	\$
2	3	t	$aataatg\$$
3	6	t	$aatg\$$
4	4	a	$ataatg\$$
5	7	a	$atg\$$
6	1	\$	$ctaataatg\$$
7	9	t	$g\$$
8	2	c	$taataatg\$$
9	5	a	$taatg\$$
10	8	a	$tg\$$

i	SA^{rev}	BWT^{rev}	$S^{rev}_{\mathsf{SA}^{rev}[i]}$
1	10	g	\$
2	8	t	$ag\$$
3	5	t	$attag\$$
4	2	c	$attattag\$$
5	1	\$	$cattattag\$$
6	9	a	$g\$$
7	7	t	$tag\$$
8	4	t	$tattag\$$
9	6	a	$ttag\$$
10	3	a	$ttattag\$$

Fig. 3. Left-hand side: suffix array SA and BWT of the string $S = ctaataatg\$$. Right-hand side: SA^{rev} and BWT^{rev} of the reverse complement $cattattag\$$ of S.

Lemma 2 implies that the recursion tree of Algorithm 2 coincides with the suffix link tree SLT of S (recall that the suffix link tree SLT has a node $\underline{\omega}$ for each internal node $\overline{\omega}$ of ST and, for each suffix link from $\overline{c\omega}$ to $\overline{\omega}$ in ST, there is an edge $\underline{\omega} \to \underline{c\omega}$ in SLT); see Fig. 2. This is can be seen as follows. If the execution of $bwtrev(k, [i..j], \ell)$ invokes the recursive call $bwtrev(k', [lb..rb], \ell + 1)$, where $[i..j]$ is the ω-interval and $[lb..rb]$ is the $c\omega$-interval, then there is a suffix link from node $\overline{c\omega}$ to node $\overline{\omega}$ because both are internal nodes in the suffix tree of S.

Theorem 2. *Algorithm 2 has a worst-case time complexity of $O(n \log \sigma)$.*

Proof. According to Lemma 2, there are as many recursive calls to the procedure *bwtrev* as there are internal nodes in the suffix tree ST of S. Because ST has n leaves and each internal node in ST is branching, the number of internal nodes is at most $n - 1$. We use an amortized analysis to show that the overall number of intervals returned by calls to the procedure *getIntervals* is bounded by $2n - 1$. Let L denote the concatenation of all lists returned by procedure calls to *getIntervals*. For each element $[lb..rb]$ of L, either at least one entry of BWT^{rev} is filled in, or there is a recursive call to the procedure *bwtrev*. It follows that L has at most $2n - 1$ elements because BWT^{rev} has n entries and there are at most $n - 1$ recursive calls to the procedure *bwtrev*. It is a consequence of this amortized analysis that the overall time taken by all procedure calls to *getIntervals* is $O(n \log \sigma)$ because a procedure call to *getIntervals* that returns a k-element list takes $O(k \log \sigma)$ time. Clearly, the theorem follows from this fact. $\square$

The Burrows-Wheeler transform of the reverse complement of a DNA-sequence can also be computed by Algorithm 2. One just has to change the order in which intervals are generated by the procedure *getIntervals*. Recall that the reverse complement of a DNA-sequence S is obtained by reversing S and then replacing each nucleotide by its Watson-Crick complement (a is replaced with t and vice versa; c is replaced with g and vice versa). For example, the reverse

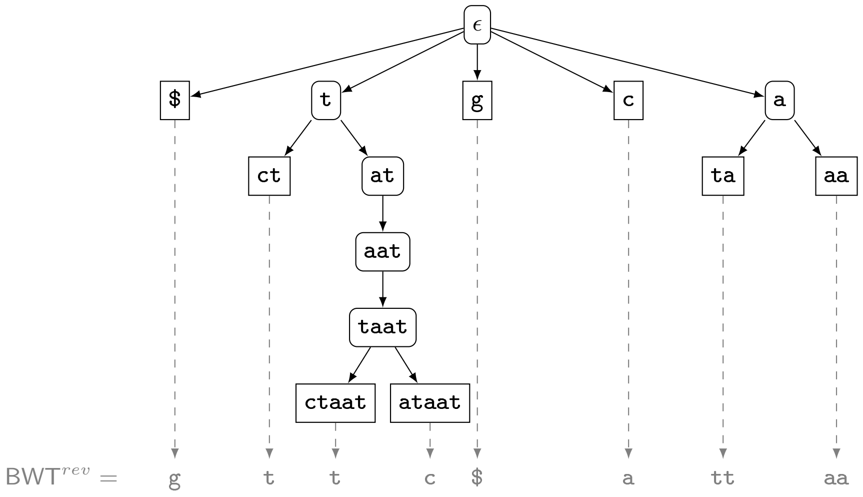


Fig. 4. The recursion tree of Algorithm 2 with modified procedure *getIntervals* applied to the BWT of the string $S = ctaataatg\$$, calculating the Burrows-Wheeler transform of the reverse complement $cattattag\$$.

complement of *ctaataatg* is *cattattag*; see Fig. 3 for the corresponding suffix arrays and Burrows-Wheeler transforms. Up to now, a ϕ-interval was generated before an ω-interval if and only if $\phi^{rev} <_{lex} \omega^{rev}$, where $<_{lex}$ is the lexicographic order induced by the order $\$ < a < c < g < t$ on the alphabet Σ. In the computation of the Burrows-Wheeler transform of the reverse complement of a DNA-sequence, a ϕ-interval must be generated before an ω-interval if and only if $\phi^{rev} <_{lex'} \omega^{rev}$, where $<_{lex'}$ is the lexicographic order induced by the order $\$ < t < g < c < a$. Moreover, instead of assigning the nucleotide $S[\mathsf{SA}[lb] + \ell + 1]$ to a position in BWT^{rev}, its Watson-Crick complement must be assigned. The recursion tree of our example can be found in Fig. 4.

4 The LCP-Array of the Reversed String

The next lemma proves a strong relationship between LCP and LCP^{rev}.

Lemma 3. *The longest common prefix array* LCP^{rev} *of* S^{rev} *is a permutation of the longest common prefix array* LCP *of* S.

Proof. We show that each lcp-value occurs as often in LCP as in LCP^{rev}. Let $\ell \in \{1, \ldots, n\}$ and define the set M_ℓ by $M_\ell = \{\omega \mid \omega \text{ is an } \ell\text{-length substring but not a suffix of } S\}$. We count how many entries in the array LCP are smaller than ℓ. There are ℓ proper suffixes of S having a length $\leq \ell$. For each such suffix $S_{\mathsf{SA}[k]}$ we have $\mathsf{LCP}[k] < \ell$. Any other suffix has a length greater than ℓ and hence its ℓ-length prefix belongs to M_ℓ. Let $\omega \in M_\ell$ and let $[i..j]$ be the ω-interval. Clearly, for all k with $i < k \leq j$, we have $\mathsf{LCP}[k] \geq \ell$ because the suffixes $S_{\mathsf{SA}[k-1]}$ and

i	LCP	$S_{\mathsf{SA}[i]}$
1	-1	\$
2	0	$atc\$$
3	2	$atgcatc\$$
4	0	$c\$$
5	1	$catc\$$
6	3	$catgcatc\$$
7	0	$gcatc\$$
8	4	$gcatgcatc\$$
9	0	$tc\$$
10	1	$tgcatc\$$

i	LCP^{rev}	$S^{rev}_{\mathsf{SA}^{rev}[i]}$
1	-1	\$
2	0	$acg\$$
3	3	$acgtacg\$$
4	0	$cg\$$
5	2	$cgtacg\$$
6	1	$ctacgtacg\$$
7	0	$g\$$
8	1	$gtacg\$$
9	0	$tacg\$$
10	4	$tacgtacg\$$

Fig. 5. The lcp-arrays of $S = gcatgcatc\$$ and $S^{rev} = ctacgtacg\$$

$S_{\mathsf{SA}[k]}$ share the prefix ω. By contrast, $\mathsf{LCP}[i] < \ell$ because ω is not a prefix of $S_{\mathsf{SA}[i-1]}$. Thus, there are $|M_\ell|$ many entries in the array LCP satisfying $\mathsf{LCP}[k] < \ell$ and $|S_{\mathsf{SA}[k]}| > \ell$. In total, the array LCP has $|M_\ell|+\ell$ many entries that are smaller than ℓ. Analogously, there are $|M_{\ell+1}|+\ell+1$ many entries in the array LCP that are smaller than $\ell+1$, where $\ell \in \{1,\dots,n-1\}$. Consequently, the lcp-value ℓ occurs $(|M_{\ell+1}|+\ell+1)-(|M_\ell|+\ell) = |M_{\ell+1}|-|M_\ell|+1$ times in the LCP-array. By the same argument, the lcp-value ℓ occurs $|M^{rev}_{\ell+1}|-|M^{rev}_\ell|+1$ many times in the array LCP^{rev}, where $M^{rev}_\ell = \{\omega \mid \omega$ is an ℓ-length substring but not a suffix of $S^{rev}\}$. Now the lemma follows from the equality $|M_\ell| = |M^{rev}_\ell|$, which is true because $\omega \in M_\ell$ if and only if $\omega^{rev} \in M^{rev}_\ell$.

Fig. 5 illustrates the proof of Lemma 3. The proper suffixes of S with length ≤ 2 occur at the indices 1 and 4 in the (conceptual) suffix array, so $\mathsf{LCP}[1]$ and $\mathsf{LCP}[4]$ are smaller than 2. Furthermore, we have $M_2 = \{at, ca, gc, tc, tg\}$ and the corresponding entries in the LCP-array at the indices 2, 5, 7, 9, and 10 are also smaller than 2. So there are $|M_2|+2=7$ entries of the LCP-array that are smaller than 2. Since $M_3 = \{atc, atg, cat, gca, tgc\}$, there are $|M_3|+3=8$ entries of the LCP-array that are smaller than 3. We conclude that the value 2 occurs $8-7=1$ times in the LCP-array. By the same argument, it occurs only once in LCP^{rev}. Note that $M^{rev}_2 = \{ac, cg, ct, gt, ta\}$ and $M^{rev}_3 = \{acg, cgt, cta, gta, tac\}$.

In fact, Algorithm 2 can also be used to compute LCP^{rev}. This can be seen as follows. Suppose Algorithm 2 is applied to the ω-interval $[i..j]$ for some ℓ-length substring ω of S, and let k be the left boundary of the ω^{rev}-interval in SA^{rev}. It is a consequence of Lemma 1 that the procedure call $bwtrev(k, [i..j], \ell)$ correctly computes the boundaries $[s_q..e_q]$ of the $\omega^{rev}c_q$-intervals in SA^{rev}, where $c_1,\dots,c_m$ are the characters for which $c_q\omega$ is a substring of S $(1 \leq q \leq m)$. By the conditional statement "**if** *list* not empty **then** $\mathsf{LCP}^{rev}[k] \leftarrow \ell$", Algorithm 2 assigns the value ℓ at each index $s_2,\dots,s_m$ (but not at index s_1). This is correct, i.e., $\mathsf{LCP}^{rev}[s_q] = \ell$ for $2 \leq q \leq m$, because $\omega^{rev}c_{q-1}$ is a prefix of the suffix at index e_{q-1} and $\omega^{rev}c_q$ is a prefix of the suffix at index $s_q = e_{q-1}+1$.

Thus, whenever Algorithm 2 fills an entry in the lcp-array LCP^{rev}, it assigns the correct value. However, the algorithm does not fill LCP^{rev} completely; in

Algorithm 3. Given a partial LCP-array that contains at least all irreducible LCP-values, this procedure computes the whole LCP-array.

```
for all $c \in \Sigma$ do        /* initialize the array $count$ */
    $count[c] \leftarrow C[c]$
$todo[1..n] \leftarrow [0, \dots, 0]$        /* initialize the bit array $todo$ */
for $i \leftarrow 1$ to $n$ do        /* replace undefined LCP-values with $LF$-values */
    $c \leftarrow \mathsf{BWT}[i]$        /* BWT is accessed sequentially */
    $count[c] \leftarrow count[c] + 1$
    if $\mathsf{LCP}[i] = \bot$ then
        $todo[i] \leftarrow 1$
        $\mathsf{LCP}[i] \leftarrow count[c]$        /* LCP[$i$] stores $LF(i)$ */
initialize an empty stack
for $i \leftarrow 1$ to $n$ do
    $k \leftarrow i$
    while $todo[k] = 1$ do
        $push(k)$
        $k \leftarrow \mathsf{LCP}[k]$        /* recall that LCP[$k$] contains $LF(k)$ */
    $\ell \leftarrow \mathsf{LCP}[k]$
    while stack is not empty do
        $\ell \leftarrow \ell - 1$
        $top \leftarrow pop()$
        $\mathsf{LCP}[top] \leftarrow \ell$
        $todo[top] \leftarrow 0$
```

Fig. 1 the computed entries are underlined. This is because whenever Algorithm 2 detects a $c\omega$-interval $[lb..rb]$ in the list returned by $getIntervals([i..j])$ with $lb \neq rb$ and $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[rb] + \ell + 1]$, then it does *not* determine the lexicographic ordering of the suffixes in the $\omega^{rev}c$-interval $[s..e]$. Instead, it fills $\mathsf{BWT}^{rev}[s..e]$ with a's because each occurrence of $c\omega$ in S is followed by the same character $a = S[\mathsf{SA}[lb] + \ell + 1]$. Consequently, if an entry $\mathsf{LCP}^{rev}[p]$ is not filled by Algorithm 2, then $\mathsf{BWT}^{rev}[p-1] = \mathsf{BWT}^{rev}[p]$. Hence $\mathsf{LCP}^{rev}[p]$ is reducible. To sum up, Algorithm 2 computes all irreducible lcp-values of LCP^{rev} (and possibly some reducible values). We next show how the remaining lcp-values can be computed in linear time.

Algorithm 3 shows pseudo-code for the computation of the whole LCP-array of a string,[1] provided that all irreducible LCP-values are already stored in the LCP-array. It uses an auxiliary array *count* of size σ, which is inizialized in the first for-loop by setting $count[c] = C[c]$ for all $c \in \Sigma$.[2] Furthermore, it uses a bit array *todo* of size n which initially contains a series of zeros. The key property utilized by Algorithm 3 is that a reducible value $\mathsf{LCP}[i]$ can be computed by the equation (see [13, Lemma 1] and [10, Lemma 4])

$$\mathsf{LCP}[i] = \mathsf{LCP}[LF[i]] - 1 \tag{1}$$

[1] In our context, the string under consideration is S^{rev}.

[2] If the array C is not needed later, then it can be used instead of *count*.

because $\mathsf{BWT}[i] = \mathsf{BWT}[i-1]$ is equivalent to $LF[i] = LF[i-1]+1$. In its second for-loop, Algorithm 3 computes the LF-mapping. Recall that $LF(i) = C[c] + Occ(c,i)$, where $c = \mathsf{BWT}[i]$ and $Occ(c,i)$ is the number of occurrences of the character c in $\mathsf{BWT}[1..i]$. The algorithm scans the BWT from left to right and counts how often each character appeared already. Every time character c appears during the scan of BWT, $count[c]$ is incremented by one. When the algorithm finds the j-th occurrence of character c at index i in BWT, then $LF(i) = C[c] + j = count[c]$. During the execution of the second for-loop, if an undefined entry $\mathsf{LCP}[i]$ is detected, then the value $LF(i)$ is stored in $\mathsf{LCP}[i]$ and $todo[i]$ is set to 1 (from now on $todo[i] = 1$ if and only if the value $\mathsf{LCP}[i]$ still needs to be computed). In the third for-loop, Algorithm 3 computes the missing LCP-values. For each index i, it follows LF-pointers until an index k is reached with $todo[k] = 0$, and it stores the sequence $i, LF(i), \ldots, LF^q(i)$ on a stack, where $k = LF^{q+1}(i)$. Since $todo[k] = 0$, $\mathsf{LCP}[k]$ contains the correct lcp-value ℓ. It is a consequence of Equation 1 that the LCP-value at index $LF^q(i)$—the topmost element of the stack—is $\mathsf{LCP}[LF^q(i)] = \ell - 1$. After $LF^q(i)$ has been popped from the stack, the subsequent values $\mathsf{LCP}[LF^{q-1}(i)], \ldots, \mathsf{LCP}[LF(i)], \mathsf{LCP}[i]$ are obtained similarly by a repeated application of Equation 1.

5 Parallelization and Experimental Results

Based on Lemma 1 and the observation that each entry $\mathsf{BWT}^{rev}\ [k]$ is accessed only once in Algorithm 2, the arrays BWT^{rev} and LCP^{rev} can be efficiently computed in parallel. The parallel algorithm for a shared-memory multi-core system can be obtained by modifying Algorithm 2 as follows: the algorithm uses a queue and instead of a recursive call $bwtrev(k, [lb..rb], \ell + 1)$ the tuple $(k, [lb..rb], \ell + 1)$ is added to the queue. Processors successively remove tuples from the queue and process them until the queue is empty. The time complexity of the algorithm is $O(p + n/p)$, where p is the number of processors.

We implemented Algorithm 2 (called bwtrev) using Simon Gog's library sdsl (http://www.uni-ulm.de/in/theo/research/sdsl.html). To compare it with methods that compute BWT^{rev} from SA^{rev}, we used Yuta Mori's library libdivsufsort library (http://code.google.com/p/libdivsufsort), which is the fastest sequential suffix array construction algorithm. Table 1 shows the running times of constructing BWT^{rev} with and without the LCP^{rev}-array (the $go\Phi$ algorithm [6] from the sdsl was used to construct LCP^{rev} from SA^{rev}). Chromosome 1 of the human genome can be found at http://www.ncbi.nlm.nih.gov/ and the other test files at http://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks. The results in Table 1 show that our sequential algorithm is competitive.

We then conducted experiments with the parallel implementation of Algorithm 2. For a fair comparison, we used two parallel suffix sorting algorithms: the mkESA package [9] (only for DNA/protein datasets) and our own implementation (called pbs) of an improved version of the algorithm of [5]. (To the best of our knowledge, no other implementation of a parallel suffix sorting algorithm is available.) Our experiments were conducted on a multi-core machine of 32 processors (8 Quadcore AMD Opteron processors with 2.3 GHz, L1 Cache=64K,

Table 1. Running times in seconds for different algorithms and datasets

Data	σ	size	libdivsufsort	libdivsufsort + lcp	bwtrev	bwtrev + lcp
human chr1	4	200MB	173.0	280.0	219.0	236.0
swissprot	20	22MB	16.2	27.5	24.5	26.0
dickens	100	10.2MB	6.0	10.0	11.1	11.9

Table 2. Running times in seconds for the datasets of Table 1 with different numbers of processors (e.g. p32 = 32 processors). The numbers $x(y)$ in each table entry should be interpreted as follows: x is the time for constructing BWT^{rev} and y is the time for constructing BWT^{rev} and LCP^{rev}. There are no entries for the mkESA package for dickens because it can solely handle biological data.

human chr1						
Tool	p1	p2	p4	p8	p16	p32
mkESA	235 (342)	230 (337)	222 (329)	222 (329)	222 (329)	222 (329)
pbs	397 (504)	274 (381)	187 (294)	156 (263)	128 (235)	127 (234)
bwtrev	219 (236)	151 (159)	72 (78)	41 (44)	22 (27)	18 (19)
swissprot						
Tool	p1	p2	p4	p8	p16	p32
mkESA	121.5 (132.8)	101.9 (113.2)	90.7 (102.0)	82.3 (93.6)	70.4 (81.7)	70.4 (81.7)
pbs	41.3 (52.6)	26.6 (37.9)	17.2 (28.5)	10.6 (21.9)	10.6 (21.9)	10.2 (21.5)
bwtrev	24.5 (26.0)	14.1 (14.9)	6.9 (7.4)	3.7 (4.1)	2.0 (2.4)	2.0 (2.4)
dickens						
Tool	p1	p2	p4	p8	p16	p32
pbs	17.6 (21.6)	10.8 (14.8)	7.8 (11.8)	6.5 (10.5)	5.7 (9.7)	5.4 (9.4)
bwtrev	11.7 (11.8)	6.0 (6.2)	2.9 (3.1)	1.7 (1.8)	1.1 (1.3)	1.1 (1.3)

and L2 Cache= 512K) and a total of 256 GB RAM. All programs were compiled with gcc/g++ using the -O3 optimization option. The results of Table 2 show that our algorithm outperforms the other tools in terms of absolute running time and scalability. It should be pointed out that with just two processors our algorithm outperforms libdivsufsort, the best sequential algorithm.

References

1. Beller, T., Gog, S., Ohlebusch, E., Schnattinger, T.: Computing the Longest Common Prefix Array Based on the Burrows-Wheeler Transform. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 197–208. Springer, Heidelberg (2011)
2. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center (1994)
3. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k Ranked Document Search in General Text Databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)

4. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
5. Futamura, N., Aluru, S., Kurtz, S.: Parallel suffix sorting. In: Proc. 9th International Conference on Advanced Computing and Communications, pp. 76–81. IEEE (2001)
6. Gog, S., Ohlebusch, E.: Lightweight LCP-array construction in linear time (2011), http://arxiv.org/pdf/1012.4263
7. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th Annual Symposium on Discrete Algorithms, pp. 841–850 (2003)
8. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
9. Homann, R., Fleer, D., Giegerich, R., Rehmsmeier, M.: mkESA: Enhanced suffix array construction tool. Bioinformatics 25(8), 1084–1085 (2009)
10. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted Longest-Common-Prefix Array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
11. Kasai, T., Lee, G.H., Arimura, H., Arikawa, S., Park, K.: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
12. Lam, T.-W., Li, R., Tam, A., Wong, S., Wu, E., Yiu, S.-M.: High throughput short read alignment via bi-directional BWT. In: Proc. International Conference on Bioinformatics and Biomedicine, pp. 31–36. IEEE Computer Society (2009)
13. Manzini, G.: Two Space Saving Tricks for Linear Time LCP Array Computation. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 372–383. Springer, Heidelberg (2004)
14. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
15. Schnattinger, T., Ohlebusch, E., Gog, S.: Bidirectional Search in a String with Wavelet Trees. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 40–50. Springer, Heidelberg (2010)
16. Simpson, J.T., Durbin, R.: Efficient construction of an assembly string graph using the FM-index. Bioinformatics 26(12), i367–i373 (2010)
17. Välimäki, N., Ladra, S., Mäkinen, V.: Approximate All-Pairs Suffix/Prefix Overlaps. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 76–87. Springer, Heidelberg (2010)

Efficient Algorithm for Circular Burrows-Wheeler Transform[⋆]

Wing-Kai Hon[1], Tsung-Han Ku[1], Chen-Hua Lu[2] Rahul Shah[3], and Sharma V. Thankachan[3]

[1] National Tsing Hua University, Taiwan
{wkhon,thku}@cs.nthu.edu.tw
[2] Academia Sinica, Taiwan
walchl@iis.sinica.edu.tw
[3] Louisiana State University, USA
{rahul,thanks}@csc.lsu.edu

Abstract. Given a set $\mathcal{P}$ of d patterns, the circular dictionary matching problem is to index $\mathcal{P}$ such that for any online query text T, we can quickly locate the occurrences of any cyclic shift of any pattern of $\mathcal{P}$ within T efficiently. This problem can be applied on practical problems that arise in bioinformatics and computational geometry. Recently, Hon et al. (2011) applied a variant of the well-known Burrows-Wheeler transform, called circular Burrows-Wheeler transform (circular BWT) [Mantaci, Restivo, Rosone, and Sciortino, *Theoretical Computer Science*, 2007], and showed that this can be used to solve the circular dictionary matching problem efficiently. In this paper, we give the first construction algorithm for the circular BWT, which takes $O(n \log n)$ time and requires $O(n \log \sigma)$ bits working space, where n denotes the total length of the patterns in $\mathcal{P}$ and σ is the alphabet size.

1 Introduction

Given a set $\mathcal{P}$ of d patterns, the dictionary matching problem is to index $\mathcal{P}$ such that for any online query text T, we can quickly locate the occurrences of any pattern of $\mathcal{P}$ within T efficiently. This problem has been well-studied in the literature [1,4,10], and an index taking optimal space and simultaneously supporting optimal-time query is achieved [2,8]. A central technique to derive optimal-spaced indexes is to apply the well-known Burrows-Wheeler transform (BWT) [3] to represent $\mathcal{P}$ in an alternative way, and augment this representation with auxiliary data structures so as to support the desired pattern matching queries efficiently.

Cyclic shifts of the indexed patterns can also be valid patterns in some practical applications in bioinformatics and computational geometry [12]. In bioinformatics, the genomes of many viruses, such as herpes simplex virus (HSV-1),

[⋆] This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W. Hon) and US NSF Grant CCF–1017623 (R. Shah).

exist as circular strings [17]. Also, due to the launch of next-generation sequencing technology, the biologists in the metagenomics area are now collecting and sequencing microbial samples from an environment directly, without isolating and culturing the samples. Those samples have circular chromosomes and plasmid DNA [6,16]. In computational geometry, a polygon may be stored by listing the co-ordinates of its vertices in clockwise order. As applications in these two fields usually involve huge data sets, it is very important to find a compact indexing scheme for the circular patterns supporting online pattern matching query efficiently. Hon et al. [7] recently applied a variant of the BWT, called circular Burrows-Wheeler transform (circular BWT) that was originally proposed by Mantaci et al. [14], and showed that this can be used to solve the above circular dictionary matching problem efficiently.

Although there are several efficient construction algorithms for the BWT in the literature, such as [9] and [11], none of them can be applied directly to construct circular BWT due to the different nature of the two definitions. Technically speaking, most of the existing construction algorithms of BWT assume each input pattern ends with a special end-marking character, and utilize this character to distinguish the lexicographical ordering among the patterns and their suffixes. In contrast, the circular BWT is defined on patterns of infinite length (see Section 2 for its definition) so that adding such an end-marker will intuitively destroy the *circular* nature of the patterns. In this paper, we give the first construction algorithm for the circular BWT, which takes $O(n \log n)$ time and requires $O(n \log \sigma)$ bits working space, where n denotes the total length of the patterns in $\mathcal{P}$ and σ is the alphabet size. Our result stems from some observations about the circular patterns, which in turn allows us to adapt the algorithm in [9] for our construction.

2 Preliminaries

Let $P = P[1..|P|]$ be a pattern of length $|P|$. A cyclic shift of P is called a *circular suffix* of P. For example, the circular suffixes of `abc` are `abc`, `bca`, and `cab`. Let P^∞ denote the string formed by repeating P infinite number of times. Then we have the following definition.

Definition 1. *Let P and Q be two patterns. We say P is* circularly *larger than Q, or simply P is larger than Q, if and only if P^∞ is lexicographically larger than Q^∞. The notions of "smaller than", or "equal to", are defined analogously.*

For example, suppose that $P = \mathtt{ba}$, $Q = \mathtt{bbba}$, and $R = \mathtt{b}$. Then, we have P is smaller than Q, and both patterns are smaller than R.

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_d\}$ be a set of d patterns of total length n, with characters drawn from an alphabet of size σ. Let $S = P_1 P_2 \cdots P_d$ be the concatenation of the patterns in $\mathcal{P}$. Any cyclic shift of a pattern P_i in $\mathcal{P}$ will be called a circular suffix of S. In addition, the circular suffix which starts at position j in S, denoted by S_j, will be referred to as the jth circular suffix of S. For example, suppose that $\mathcal{P} = \{\mathtt{ba}, \mathtt{bbba}, \mathtt{b}\}$. Then, we have $S = \mathtt{babbbab}$, and the circular suffixes of S are $S_1 = \texttt{ba}$, $S_2 = \texttt{ab}$, $S_3 = \texttt{bbba}$, $S_4 = \texttt{bbab}$, $S_5 = \texttt{babb}$, $S_6 = \texttt{abbb}$, and $S_7 = \texttt{b}$.

We define the *circular suffix array* $SA_\circ[1..n]$ of S, such that $SA_\circ[i] = j$ if S_j is the ith smallest circular suffix of S. For example, $\mathcal{P} = \{\texttt{ba}, \texttt{bbba}, \texttt{b}\}$, and $SA_\circ[1..7] = [2, 6, 1, 5, 4, 3, 7]$. Note that if two circular suffixes are equal, we arrange them arbitrarily in $SA_\circ$.[1] We also define $SA_\circ^{-1}[1..n]$ of S, such that $SA_\circ^{-1}[j] = i$ if and only if $SA_\circ[i] = j$.

Let L be a binary vector which represents the length of each pattern in S, such that a length of value k is encoded in unary by $\texttt{10}^{k-1}$. We append an extra $\texttt{1}$ at the end of L as a sentinel. For instance, $L = \texttt{10100011}$ in our running example. Let $rank_b(L, i)$ denote the number of b's in $L[1..i]$, and $select_b(L, j)$ denote the position of the jth b in L. Based on L and its operations, we define the circular $\Psi_\circ[1..n]$ function of S, which is analogous to the Ψ function in [9], as follows.

$$\Psi_\circ[i] = \begin{cases} SA_\circ^{-1}[j+1] & \text{if } L[j+1] \neq 1 \\ SA_\circ^{-1}[select_1[rank_1[j+1]-1] & \text{otherwise} \end{cases}$$

where $j = SA_\circ[i]$.[2] Then we have the following lemma.

Lemma 1. *Suppose that $SA_\circ[1..n]$ and L of S are given. We can construct the $\Psi_\circ[1..n]$ of S in $O(n)$ time using $O(n \log n)$ bits working space.*

Proof. Based on $SA_\circ$, we can easily construct $SA_\circ^{-1}[1..n]$ in $O(n)$ time in $O(n \log n)$ bits space. Next, we construct an $o(n)$-bit auxiliary index of L in $O(n)$ time so that the *rank* and *select* queries on L can be answered in $O(1)$ time [15]. The lemma thus follows. □

Finally, the *circular Burrows-Wheeler transform (cbwt)* on S, which is a variant of the well-known Burrows-Wheeler transform [3], is a character string $cbwt[1..n]$ of length n such that the ith character $cbwt[i]$ is the last character of the ith smallest circular suffix of S. The *cbwt* for our running example is $cbwt[1..7] = \texttt{bbabbab}$.

Lemma 2. *Suppose that S and $\Psi_\circ[1..n]$ are given. Then we can construct cbwt in $O(n)$ time using $O(n \log \sigma)$ bits working space.*

[1] For ease of discussion, we assume that each pattern P_i cannot be written as P^k for some period string P and some integer $k > 1$. Also, we further assume that no pattern is a cyclic shift of another one. Under these assumptions, the lexicographical ordering of the circular suffixes will be distinct. Handling the general case involves minor adaptation to the indexing scheme, where the main idea is to represent a periodic string by its shortest period and its length. We defer the details in the full paper.

[2] In other words, suppose that S_j denotes the ith smallest circular suffix of S, and S_j comes from the pattern P_k in $\mathcal{P}$. Then $\Psi_\circ[i]$ is the rank of the circular suffix S_{j+1} in $SA_\circ$ if S_{j+1} also comes from the same pattern P_k, and otherwise it is the rank of P_k in $SA_\circ$.

Proof. We sort the characters in S in $O(n)$ time and $O(n \log \sigma)$ bits space by counting sort, so that $S_{\texttt{sorted}}[i]$ stores the first character of the ith smallest circular suffix. Then by computing $j = \Psi_\circ[i]$ iteratively for each i, we can fill in $cbwt[j] = S_{\texttt{sorted}}[i]$ accordingly. The total time is $O(n)$. □

3 Construction of Circular Burrows-Wheeler Transform

To construct the circular BWT, we can first obtain the $\Psi_\circ$, and then convert it into the circular BWT in linear time based on Lemma 2. Thus, in the remainder of this paper, we focus on the efficient construction of $\Psi_\circ$ instead. Basically, such a construction requires us to implicitly sort the circular suffixes of S. We begin with the following definition and observation.

Definition 2 (Period [5]). *Let x be a nonempty string. An integer p such that $0 < p \le |x|$ is called a* period *of x if $x[i] = x[i + p]$ for $i = 1, 2, \ldots, |x| - p$. For example, if $x =$ `abababab`, then the possible periods of x are 2, 4, 6, and 8.*

Lemma 3. *Let S_i and S_j be two circular suffixes of S, and $|S_i| < |S_j|$. If S_i is circularly equal to S_j, $|S_i|$ will be a period of S_j.*

Proof. Suppose S_i is circularly equal to S_j. That means $S_j^\infty = S_i^\infty$. Let p be the length of S_i. Then, we will have $S_j[k] = S_i^\infty[k]$ and $S_i^\infty[k + p] = S_j[k + p]$ for $1 \le k \le |S_j| - p$. Also, $S_i^\infty[k] = S_i^\infty[k + p]$ for any k. Thus, by Definition 2, p is a period of S_j. □

Note that the converse of the above lemma is not true. That is, if $|S_i|$ is a period of S_j, S_i may not be equal to S_j. For example, consider $S_i =$ `aab` and $S_j =$ `aabaa`. Although $|S_i| = 3$ is a period of S_j, but S_i is not circularly equal to S_j. So, if we want to know whether S_i is equal to S_j or not, we cannot just verify whether $|S_i|$ is a period of S_j and $S_i = S_j[1..|S_i|]$. To do so, we will need the following lemmas.

Lemma 4 (Weak Periodicity Lemma [5]). *Let p and q be two periods of a string x. If $p + q \le |x|$, then $\gcd(p, q)$ is also a period of x. Here, $\gcd(p, q)$ denotes the greatest common divisor of p and q.*

Lemma 5. *Let S_i and S_j be two circular suffixes of S, and $|S_i| < |S_j|$. $|S_i|$ is a period of $S_j S_j$ and $S_i = S_j[1..|S_i|]$ if and only if S_i is circularly equal to S_j.*

Proof. (Only if part:) Let $|S_i| = p$ and $|S_j| = q$ such that $p < q$. Suppose p is a period of $S_j S_j$ and $S_i = S_j[1..p]$. Since p and q are periods of $S_j S_j$ and $p + q < 2q$, by Lemma 4, $\gcd(p, q)$ is also a period of $S_j S_j$. As $\gcd(p, q)$ divides q, this implies that the prefix of S_j of length $\gcd(p, q)$ can be repeated integral number of times to form $S_j S_j$ (and thus S_j as well). Similarly, as S_i is equal to the prefix of S_j in the first p characters, and $\gcd(p, q)$ divides p, the prefix of S_j of length $\gcd(p, q)$ can be repeated integral number of times to form S_i. This immediately implies that S_i is circularly equal to S_j. (If part:) This part is obviously true. □

Lemma 5 implies that to compare two circular suffixes S_i and S_j, we only need to compare S_i^∞ and S_j^∞ directly up to length $2|S_j|$ (assuming $|S_i| < |S_j|$).[3] Thus, to construct circular BWT, one way is to apply the above idea to sort all the circular suffixes, and then obtain the last character of the corresponding suffix to fill in the cbwt array. However, this would need $\Omega(n^2 \log n)$ time in worst case.

In the following, we adapt the algorithm in [9] to improve the efficiency. The basic idea is to obtain the $\Psi_\circ$ function when we only consider patterns in $\mathcal{P}$ that are *short*, and then repeatedly update the $\Psi_\circ$ function to include the circular suffixes of each long pattern in $\mathcal{P}$.

3.1 Constructing $\Psi_\circ$ for All Short Patterns

We first sort all the d patterns by their lengths. Without losing of generality, we suppose that the result is $|P_1| \le |P_2| \le \cdots \le |P_d|$, and $S = P_1 P_2 \cdots P_d$. Then, we separate patterns into short pattern group and long pattern group. Short patterns are those patterns whose lengths are shorter than $0.5n/\log n$; otherwise, they are long patterns.

Suppose $P_1, P_2, \ldots, P_s$ are the short patterns. First, we find the first i such that the total lengths of $P_1, P_2, \ldots, P_i$ is just greater than or equal to $0.5n/\log n$.[4] As each pattern is short, the total length is bounded by $n/\log n$. Next, we sort the circular suffixes of the patterns in $\{P_1, P_2, \ldots, P_i\}$ based on Larsson and Sadakane's suffix sorting algorithm [13]. After sorting, we would have the $\Psi_\circ$ function of $P_1..P_i$.

We repeat the procedure for $P_{i+1}, \ldots, P_j$, whose total length is just greater than or equal to $0.5n/\log n$. Then, we apply the suffix sorting algorithm, and would have another $\Psi_\circ$ function of $P_{i+1}..P_j$. After that, we combine the $\Psi_\circ$ functions of $P_1 \cdots P_i$ and of $P_{i+1} \cdots P_j$, based on Hon et al.'s algorithm [9], and obtain the $\Psi_\circ$ of $P_1 \cdots P_j$.

The above procedures are repeated until we obtain the $\Psi_\circ$ of $P_1..P_s$. Algorithm 1 shows how we perform the sorting of the circular suffixes, and Algorithm 2 shows how we can combine the $\Psi_\circ$ functions into one.

Example: Suppose $\mathcal{P} = \{\texttt{ba}, \texttt{bbba}, \texttt{b}\}$. Since the longest pattern is `bbba` whose length is 4, we will execute the for-loop of Algorithm 1 for three times.

- Step 1: We sort `b`, `a`, `b`, `b`, `b`, `a`, `b`. After sorting, we rename each character. In this case, `a` $\leftrightarrow 1$ and `b` $\leftrightarrow 2$. Thus, `ba` $\rightarrow 21$, `bbba` $\rightarrow 2221$, and `b` $\rightarrow 2$.
- Step 2: Then, we sort `ba`, `ab`, `bb`, `bb`, `ba`, `ab`, `bb` by sorting the corresponding tuples, which are $(2,1), (1,2), (2,2), (2,2), (2,1), (1,2), (2,2)$, respectively. After sorting, we rename each tuple. That is $(1,2) \rightarrow 1$, $(2,1) \rightarrow 2$, $(2,2) \rightarrow 3$. So, `ba` $\rightarrow 21$, `bbba` $\rightarrow 3321$, and `b` $\rightarrow 3$.
- Step 3: Furthermore, we sort `baba`, `abab`, `bbba`, `bbab`, `babb`, `abbb`, `bbbb` by sorting the corresponding tuples, which are (2,2), (1,1), (3,2), (3,1), (2,3), (1,2), (3,3). After sorting, we rename each tuples. That means $(1,1) \rightarrow 1$,

[3] In fact, Mantaci et al. [14] have shown a better bound of $|S_i| + |S_j| - gcd(|S_i|, |S_j|)$, based on the Fine and Wilf theorem.

[4] If no such i is found, we set i to be s.

Algorithm 1. Sorting circular suffixes by Larsson-Sadakane's algorithm [13]

```
Input:  Patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_i\}$, with $|P_1| < |P_2| < \cdots < |P_i|$
Output: Ordering of all circular suffixes of $\mathcal{P}$
1: Sort all characters in $\mathcal{P}$, and rename each character by its rank
2: for $r \leftarrow 1$ to $\lceil \log 2|P_i| \rceil$ do
3:     For each consecutive $2^r$ characters starting at each position of each pattern, we label it by a tuple $(u, v)$, where $u$ and $v$ are the corresponding names of the first $2^{r-1}$ consecutive characters and the last $2^{r-1}$ consecutive characters, respectively. We assume the patterns are circular so that consecutive characters are defined for each position.
4:     Sort all the tuples and rename each tuple according to its rank.
       /* After this step, the lexicographical ordering to each circular substring of length $2^r$ is determined */
5: end for
6: The ordering of each circular suffix among all the circular suffixes is denoted by the final name corresponding to it
```

$(1,2) \to 2$, $(2,2) \to 3$, $(2,3) \to 4$, $(3,1) \to 5$, $(3,2) \to 6$, $(3,3) \to 7$. So, `ba` $\to 31$, `bbba` $\to 6542$, and `b` $\to 7$.

- Step 4: Finally, we sort `babababa`, `abababab`, `bbbabbba`, `bbabbbab`, `babbbabb`, `abbbabbb`, `bbbbbbbb` by sorting the corresponding tuples, which are (3,3), (1,1), (6,6), (5,5), (4,4), (2,2), (7,7). After sorting, we rename each tuples. That means $(1,1) \to 1$, $(2,2) \to 2$, $(3,3) \to 3$, $(4,4) \to 4$, $(5,5) \to 5$, $(6,6) \to 6$, $(7,7) \to 5$. So, `ba` $\to 31$, `bbba` $\to 6542$, and `b` $\to 7$.
- Finally, we conclude that

$$(\texttt{abab})^\infty < (\texttt{abbb})^\infty < (\texttt{baba})^\infty < (\texttt{babb})^\infty < (\texttt{bbab})^\infty < (\texttt{bbba})^\infty < \texttt{b}^\infty.$$

Theorem 1. *Algorithm 1 correctly computes the ordering between the circular suffixes in $\mathcal{P}$.*

Proof. Since we will compare all the substring of length 2^r of all the circular patterns, for $0 \le r \le \lceil \log 2|P_i| \rceil$, we will compare the prefixes of each pair of circular suffixes for at least twice of their lengths. By Lemma 5, we know that we will get the correct ordering of each pair of circular suffixes, so that the final output arrangement is correct. □

In Algorithm 2, we construct an auxiliary bit vector V in Step 2 to aid the subsequent steps. Suppose that it is obtained correctly. Then, it is straightforward to obtain the following lemmas.

Lemma 6. *For any circular suffix x of $P_1 P_2 \cdots P_i$, let r be the rank of x among all circular suffixes in $P_1 P_2 \cdots P_j$ and r' be the rank of x among all circular suffixes in $P_1 P_2 \cdot P_i$. Then, $r = select_0(V, r' + 1)$ and $r' = rank_0(V, r)$.*

Lemma 7. *For any circular suffix x of $P_1P_2\cdots P_i$, let r be the rank of x among all circular suffixes in $P_1P_2\cdots P_j$. Suppose that the next circular suffix of x is y. Then,*

- *the rank of y among all circular suffixes in $P_1P_2\cdots P_i$ is $\Psi_1[rank_0(V,r)] = r'$;*
- *the rank of y among all circular suffixes in $P_1P_2\cdots P_j$ is $select_0(V, r'+1)$.*

Lemma 8. *For any circular suffix x of $P_{i+1}P_{i+2}\cdots P_j$, let r be the rank of x in $P_1P_2\cdots P_j$. Suppose that the next circular suffix of x is y. Then,*

- *the rank of y in $P_{i+1}P_{i+2}\cdots P_j$ is $\Psi_2[rank_1(V,r)] = r'$.*
- *the rank of y in $P_1P_2\cdots P_j$ is $r' + rank_0(V,h)$, where $h = select_1(V,r')$.*

Algorithm 2. Merging two $\Psi_\circ$ functions by Hon et al.'s algorithm [9]

Input: $\Psi_1 = \Psi_\circ$ for $P_1P_2\cdots P_i$ and $\Psi_2 = \Psi_\circ$ for $P_{i+1}P_{i+2}\cdots P_j$
Output: $\Psi_\circ$ for $P_1P_2\cdots P_j$

```
1: For each circular suffix in $P_{i+1}P_{i+2}\cdots P_j$, compute its relative rank among
   all circular suffixes in $P_1P_2\cdots P_i$. The relative ranks are stored in an array
   $R$
2: Let $m$ denote the length of $P_1P_2\cdots P_j$. Use $R$ to construct a bit vector
   $V[1..m]$ (with the auxiliary data structures of [15] to support $O(1)$-time
   rank/select operations), such that $V[i] = 0$ if the circular suffix of $P_1P_2\cdots P_j$
   with rank $i$ is a circular suffix of $P_1P_2\ldots P_i$, and $V[i] = 1$ otherwise
3: /* the values of $\Psi_\circ$ are obtained iteratively */
4: for $r \leftarrow 1$ to $m-1$ do
5:   if $V[r] = 0$ then
6:     $p \leftarrow rank_0(V,r)$;   $r' \leftarrow \Psi_1[p]$;   $\Psi_\circ[r] \leftarrow select_0(V, r'+1)$;
7:   else
8:     $p \leftarrow rank_1(V,r)$;   $r' \leftarrow \Psi_2[p]$;   $h \leftarrow select_1(V,r')$;   $\Psi_\circ[r] \leftarrow$
       $(r' + rank_0(V,h))$;
9:   end if
10: end for
```

Based on the above lemmas, we have the following theorem.

Theorem 2. *Algorithm 2 correctly computes the $\Psi_\circ$ for $P_1P_2\cdots P_j$.*

To finish the discussion of Algorithm 2, it remains to show how to compute the relative ranks in Step 1 and how to construct the bit vector V in Step 2, efficiently. Firstly, let \$ be a symbol whose alphabetical ordering is smaller than any character in the alphabet of $\mathcal{P}$. We observe that for a circular suffix π of P_h, where $h \in [i+1, j]$, the following equalities hold:

$$\begin{aligned} &\text{rank of } \pi^\infty \text{ among all circular suffixes in } P_1P_2\cdots P_i \\ = &\text{ rank of } \pi\pi\hat{\pi} \text{ among all circular suffixes in } P_1P_2\cdots P_i \\ = &\text{ rank of } \pi\pi\hat{\pi}\$ \text{ among all circular suffixes in } P_1P_2\cdots P_i \end{aligned}$$

where $\hat{\pi}$ denotes a prefix of π. In the above, the first equality follows from Lemma 5, and the second equality follows from the alphabetical order of \$. Thus, the desired relative rank of any circular suffix π of P_h can be obtained if we can obtain the relative rank of each suffix of $P_h P_h P_h\$$ in the circular suffixes of $P_1 P_2 \cdots P_i$. This can be done by applying the backward search algorithm similar to the one in [9] as follows: Let $\#(a)$ be the number of circular suffixes of $P_1 P_2 \cdots P_i$ starting with character a, and let $\alpha(a)$ be the number of circular suffixes of $P_1 P_2 \cdots P_i$ which is lexicographically smaller than the character a. Firstly, we find the relative rank of \$ among all circular suffixes in $P_1 P_2 \cdots P_i$. It is 0 because \$ is alphabetically the smallest character. Next, suppose that the relative rank $r(Q)$ of a string Q is known. We can compute the relative rank r of the string cQ, for any character c, by finding (i) $r_1 = \alpha(c)$, and (ii) the rank r_2 of cQ among all circular suffixes of $P_1 P_2 \cdots P_i$ that begins with the character c. The desired r is equal to $r_1 + r_2$. The value of r_2 can be obtained by binary searching $r(Q)$ in the range $\Psi_1[\alpha(c)+1, \alpha(c+1)]$ (the values in the range is known to be increasing [7]). Thus, we can iteratively repeat the above step, and obtain the rank of each suffix of $P_h P_h P_h\$$ as desired. Finally, we store these relative ranks in an array R. After sorting R (which may contain equal values), we can obtain the desired bit vector V of Step 2 in linear time by scanning the sorted R.

Based on the above algorithms, we can compute the $\Psi_\circ$ function of $P_1 P_2 \cdots P_s$ as follows. First, we partition the small patterns into subgroups so that each subgroup (except the last one) has total length $\Theta(n/\log n)$. Then, we sort the circular suffixes in the first and the second subgroups (by Algorithm 1), obtain $\Psi_\circ$ functions of these two subgroups (by Lemmas 1 and 2), and apply Algorithm 2 to merge the $\Psi_\circ$ functions. Next, we sort the third subgroup and obtain its $\Psi_\circ$ function, and apply Algorithm 2 to merge this with the $\Psi_\circ$ function for the first two subgroups. We repeat these steps to merge each of the remaining subgroups. This gives the following theorem.

Theorem 3. *The $\Psi_\circ$ function for $P_1 P_2 \cdots P_s$ can be obtained in $O(n \log n)$ time using $O(n \log \sigma)$ bits working space.*

Proof (sketch). Each time when we apply Algorithm 1 to sort the circular suffixes of the patterns whose total length is $O(n/\log n)$, it can be done in $O(n)$ time. Each time when we apply Algorithm 2, we need $O(n)$ time to perform $O(n/\log n)$ binary searches, each taking $O(\log n)$ time, and $O(n)$ time to construct the bit vector V. After that, the computation of $\Psi_\circ$ can be done in $O(n)$ time. As the number of subgroups of short patterns is $O(\log n)$, the total time is bounded by $\mathrm{O}(n \log n)$. For the space requirement, we need to construct the $\Psi_\circ$ function. As its values are obtained iteratively, the $\Psi_\circ$ can be stored and encoded on the fly by the same manner as in [9], which takes $O(n \log \sigma)$ bits. The other arrays, such as L, $\#$, α, V, and R can be represented in $O(n \log \sigma)$ bits as well. The theorem thus follows.

3.2 Updating $\Psi_\circ$ for Long Patterns

Next, we describe how we deal with the long patterns. One may simply apply the method in the previous section to include the circular suffixes of each long pattern P_k. However, when only $O(n \log \sigma)$ bits of working space is allowed, we have two problems: (i) We cannot apply Algorithm 1 as a subroutine to compute the $\Psi_\circ$ of P_k, and (ii) we cannot use the array R in Algorithm 2 (Step 1) to store the relative ranks of each circular suffix of P_k with respect to those included in the current $\Psi_\circ$ function. To circumvent the first problem, we use the ordinary Ψ function of $P_k P_k P_k\$$ as a replacement of the circular $\Psi_\circ$ function of P_k, where the former can be constructed in $O(|P_k| \log n)$ time using $O(n \log \sigma)$ bits of working space [9]. Based on this Ψ function, we can maintain the relative ranks between the circular suffixes of P_k among themselves. To circumvent the second problem, we partition P_k into ℓ segments of length $\Theta(n/\log n)$, say $P_k = \pi_1\pi_2 \cdots \pi_\ell$, and apply a modified version of Algorithm 2 for ℓ times so that the circular suffixes of P_k corresponding to the same segment will be included as a batch. Details are as follows.

We use a running example in this section to ease our discussion. Let $\mathcal{P} = \{\texttt{abba}, \texttt{cdca}, \texttt{bcaacabb}\}$, which contains two short patterns and one long pattern. Suppose that the $\Psi_\circ$ function Ψ_1 for the short patterns are already obtained:

$$\Psi_1 = [2, 5, 7, 1, 4, 3, 8, 6].$$

Next, we update the current Ψ_1 to include the circular suffixes of a long pattern, say P_k. We construct the ordinary Ψ function of $P = P_k P_k P_k\$$. This function, together with its $o(|P_k|)$-bit auxiliary data structure that is constructed together [9], allows us to obtain the rank $\rho(i)$ of each suffix $P[i..]$ among themselves in $O(\log n)$ time. In the running example, we include our long pattern $P_3 = \texttt{bcaacabb}$, and the corresponding ρ function will be as follows (which is not stored explicitly):

$$\rho = [19, 22, 4, 10, 25, 7, 14, 16, 18, 21, 3, 9, 24, 6, 13, 15, 17, 20, 2, 8, 23, 5, 12, 11, 1].$$

Next, we apply Algorithm 2 to include the circular suffixes corresponding to each segment π_j of $P_k = \pi_1\pi_2 \cdots \pi_\ell$, one segment at a time. More precisely, the inclusion is performed in a backward manner, so that we include those circular suffixes corresponding to π_ℓ first, and then those corresponding to $\pi_{\ell-1}$, and so on. To process each segment π_j, we will construct a new bit vector V (as in Step 2 of Algorithm 2) to indicate whether each position in the updated Ψ_1 is due to an existing circular suffix from the current Ψ_1, or a new circular suffix corresponding to π_j. To do so, at the very beginning, we use backward search of $P_k P_k\$$ in Ψ_1 obtain the rank of $P_k P_k\$$ with respect to those circular suffixes in Ψ_1. Then to process each segment π_j, we will already have the rank of $\pi_{j+1}\pi_{j+2} \cdots \pi_\ell P_k P_k\$$ with respect to those circular suffixes in the current Ψ_1, so that a backward search with the portion π_j in the current Ψ_1 is sufficient to locate all the positions of 1 in the bit vector V.

In our running example, at the beginning, we use backward search of $P_3 P_3\$$ and obtain its rank (which is 5) among all circular suffixes in Ψ_1. Next, when we process `cabb` of P_3, a backward search with `cabb` allows us to obtain the relative ranks of the new circular suffixes with respect to those in the current Ψ_1, which are $5, 5, 2, 5$ (in backward order). This in turn allows us to obtain the desired V as follows:

$$V = [0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0].$$

By the time when we process `bcaa` of P_3, we will have updated the current Ψ_1 to include those circular suffixes from `cabb` of P_3. Also, we will have the rank of `cabb`$P_3P_3\$$ among all circular suffixes in the current Ψ_1, which is 9. A backward search with `bcaa` now allows us to obtain the relative ranks of the new circular suffixes with respect to those in the current Ψ_1, which are 3, 1, 8, 8 (in backward order). This in turn allows us to obtain the desired V in this step as follows:

$$V = [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0].$$

Once the bit vector V is ready, we can compute the entries of the new $\Psi_\circ$ as follows. First, for those entries of $\Psi_\circ$ that are due to circular suffixes in the current Ψ_1, the corresponding V entries will be 0, and the new $\Psi_\circ$ will be updated in the same way as before (see Lines 5 and 6 in Algorithm 2). Second, the remaining entries of $\Psi_\circ$ are all due to the new circular suffixes of the current segment π_j. More precisely, the pth 1 in V corresponds to the pth lexicographically smallest suffix in the segment π_j, whose $\Psi_\circ$ entry should be filled with the rank of its next suffix among all circular suffixes in the new $\Psi_\circ$. To obtain such a value, we use ρ to compute the relative ranks of the new circular suffixes from π_j. Based on that, for each new circular suffix x, its next suffix y among the new circular suffixes can be determined, which consequently allows us to compute the desired $\Psi_\circ$ corresponding to x.

In the running example, when we first process `cabb` of P_3, we extract the ranks of the corresponding suffixes from ρ, which are $25, 7, 14, 16$. After sorting, we obtain the relative ranks to be $4, 1, 2, 3$. In this case, the $\Psi_\circ$ entry corresponding to the lexicographically 4th smallest suffix (i.e., the $\mathit{select}_1(V, 4)$th entry) should be set to $\mathit{select}_1(V, 1)$, the entry corresponding to the lexicographically 1st smallest suffix should be set to $\mathit{select}_1(V, 2)$. In general, if we use H to store the relative ranks, then for $1 \leq z \leq |p_j| - 1$, we should set

$$\Psi_\circ[\mathit{select}_1(V, H[z])] \leftarrow \mathit{select}_1(V, H[z+1]).$$

For the boundary case when $z = |p_j|$, the corresponding next suffix will be $(\pi_{j+1} \cdots \pi_\ell \pi_1 \cdots \pi_j)^\infty$, so that we should set $\Psi_\circ[\mathit{select}_1(V, H[z])]$ to be the rank of the corresponding next suffix among the circular suffixes in $\Psi_\circ$. Thus, we would have computed the following values:

$$\Psi_\circ[3] \leftarrow 7, \quad \Psi_\circ[7] \leftarrow 8, \quad \Psi_\circ[8] \leftarrow 8^{5}, \quad \Psi_\circ[9] \leftarrow 3.$$

By merging the values of $\Psi_\circ$ computed in the two cases, we can obtain the complete $\Psi_\circ$, in ascending order of entries, as follows:

$$\Psi_\circ = [2, 6, 7, 11, 1, 5, 8, 8^\star, 3, 4, 12, 10].$$

[5] This value corresponds to the rank of $P_3P_3\$$ among all the circular suffixes in the current $\Psi_\circ$.

Note that there is a repeated rank (with value 8) in the $\Psi_\circ$, since one of them corresponds to the rank of $P_k P_k\$$, where such a string does not correspond to any circular suffix considered in $\Psi_\circ$.

Next, we update Ψ_1 to be the $\Psi_\circ$ we have just obtained, and process `bcaa` of P_3. Following the above procedures, we get $H = [3, 4, 1, 2]$, and we would have computed the following $\Psi_\circ$ values corresponding to the entries of the new circular suffixes:

$$\Psi_\circ[2] \leftarrow 5, \quad \Psi_\circ[5] \leftarrow 13[6], \quad \Psi_\circ[11] \leftarrow 12, \quad \Psi_\circ[12] \leftarrow 2.$$

By merging the values of $\Psi_\circ$ computed in the two cases, we can obtain the complete $\Psi_\circ$, in ascending order of entries, as follows:

$$\Psi_\circ = [3, 5, 8, 9, 13, 15, 1, 7, 10, 10^\star, 12, 2, 4, 6, 16, 14].$$

Again, there is a repeated rank (with value 10) in the $\Psi_\circ$, since one of them corresponds to the rank of $P_k P_k\$$, where such a string does not correspond to any circular suffix in $\Psi_\circ$.

Finally, when all the segments are processed, the entry with the rank of $P_k P_k\$$ in the $\Psi_\circ$ can now be updated as the rank of $P_k P_k P_k\$$, as both strings represent the same string P_k^∞, and the latter one truly exists in the circular suffixes considered in $\Psi_\circ$. Essentially, the updating will always increase the rank by exactly 1, so that the final $\Psi_\circ$ becomes:

$$\Psi_\circ = [3, 5, 8, 9, 13, 15, 1, 7, 10, 11, 12, 2, 4, 6, 16, 14].$$

This gives the following theorem.

Theorem 4. *Suppose that the $\Psi_\circ$ function for $P_1 \cdots P_{k-1}$ is given. Then the $\Psi_\circ$ function for $P_1 \cdots P_k$ can be obtained in $O(|P_k| \log n)$ time using $O(n \log \sigma)$ bits working space.*

Proof (sketch). In the above steps, the total time for all backward searches is $\sum_{j=1}^{\ell} O(|\pi_j| \log n) = O(|P_k| \log n)$. Overall, the bit vector V is constructed ℓ times, which takes $O(n\ell)$ time in total. Next, for the H array, it is constructed ℓ times. Each time we need to extract $|\pi_j| = \Theta(n/\log n)$ ρ values, and perform sorting on these values, both requiring $O(n)$ time. Thus, the total time to construct all the H arrays is bounded by $O(n\ell)$. Finally, the $\Psi_\circ$ array is constructed ℓ times, each time taking $O(n)$ time, so that the total time is again bounded by $O(n\ell)$. As $n\ell = O(|P_k| \log n)$, the construction time bound in the theorem follows. For the space requirement, the $\Psi_\circ$ can be stored and encoded on the fly using $O(n \log \sigma)$ bits, while the other arrays, such as H and V, require only $O(n)$ bits. The working space bound in the theorem follows. □

Using the above algorithm, we can repeatedly add the circular suffixes of the next long pattern into the current $\Psi_\circ$. By combining the above result with Theorem 3, we obtain the following theorem.

[6] This value corresponds to the rank of `cabb`$P_3 P_3\$$ among all the circular suffixes in the current $\Psi_\circ$.

Theorem 5. *The* $\Psi_\circ$ *function for* $P_1 P_2 \cdots P_d$ *can be obtained in* $O(n \log n)$ *time using* $O(n \log \sigma)$ *bits working space.*

References

1. Aho, A., Corasick, M.: Efficient String Matching: An Aid to Bibligoraphic Search. Communications of the ACM 18(6), 333–340 (1975)
2. Belazzougui, D.: Succinct Dictionary Matching with No Slowdown. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 88–100. Springer, Heidelberg (2010)
3. Burrows, M., Wheeler, D.J.: A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA (1994)
4. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed Indexes for Dynamic Text Collections. ACM Transactions on Algorithms 3(2) (2007)
5. Crochemore, M., Rytter, W.: Text Algorithms. Oxford University Press, New York (1994)
6. Eisen, J.A.: Environmental Shotgun Sequencing: Its Potential and Challenges for Studying the Hidden World of Microbes. PLoS Biology 5(3), e82 (2007)
7. Hon, W.-K., Lu, C.-H., Shah, R., Thankachan, S.V.: Succinct Indexes for Circular Patterns. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 673–682. Springer, Heidelberg (2011)
8. Hon, W.-K., Ku, T.-H., Shah, R., Thankachan, S.V., Vitter, J.S.: Faster Compressed Dictionary Matching. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 191–200. Springer, Heidelberg (2010)
9. Hon, W.K., Lam, T.W., Sadakane, K., Sung, W.K., Yiu, S.M.: A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays. Algorithmica 48(1), 28–36 (2007)
10. Hon, W.K., Lam, T.W., Shah, R., Tam, S.L., Vitter, J.S.: Compressed Index for Dictionary Matching. In: DCC, pp. 23–32 (2008)
11. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. SIAM J. Computing 38(6), 2162–2178 (2009)
12. Iliopoulos, C.S., Rahman, M.S.: Indexing Circular Patterns. In: Nakano, S.-I., Rahman, M.S. (eds.) WALCOM 2008. LNCS, vol. 4921, pp. 46–57. Springer, Heidelberg (2008)
13. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Theoretical Computer Science 387(3), 258–272 (2007)
14. Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: An Extension of the Burrows Wheeler Transform. Theoretical Computer Science 387(3), 298–312 (2007)
15. Raman, R., Raman, V., Rao, S.S.: Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. In: SODA, pp. 233–242 (2002)
16. Simon, C., Daniel, R.: Metagenomic Analyses: Past and Future Trends. Applied and Environmental Microbiology 77(4), 1153–1161 (2011)
17. Strang, B.L., Stow, N.D.: Circularization of the Herpes Simplex Virus Type 1 Genome upon Lytic Infection. Journal of Virology 79(19), 12487–12494 (2005)

Least Random Suffix/Prefix Matches in Output-Sensitive Time

Niko Välimäki[*]

Helsinki Institute for Information Technology, Department of Computer Science, University of Helsinki, Finland
nvalimak@cs.helsinki.fi

Abstract. We study the problem of finding suffix/prefix matches (*overlaps*) when given a set of r strings of total length n. Gusfield et al. (1992) gave an algorithm to find the longest *exact* overlaps between all string-pairs in the optimal $O(n + t_{\mathsf{output}})$ time, where $t_{\mathsf{output}} \leq r^2$ is the number of non-zero length overlaps found. So far the best worst-case time for finding *approximate* overlaps within edit distance k has been $O(knr)$ (Landau et al. 1998), which gives $\Omega(r^2)$ time regardless of the output size. We propose the first output-sensitive algorithm to find either the longest or the *least random* approximate overlaps. Given the maximum edit distance k allowed in an overlap, the approximate overlaps can be found in linear space and in $O((n + t_{\mathsf{output}})\,\text{polylog}(n))$ time for any constant k. If all input strings are shorter than $\log n/(k^{\frac{1}{k}}\sigma)$, we achieve the time complexity $O(n \log^k n + t_{\mathsf{output}})$ for any k. For strings longer than $\epsilon \log^k r$, we improve the previous best worst-case time from $O(knr)$ to $O(\frac{c^k}{k!}nr)$ for moderate k and constants $c > 1$ and $\epsilon > 0$.

1 Introduction

In the *suffix/prefix matching* problem, we are given a (multi-)set of strings, $\mathcal{T}$, having r strings of total length n. The string $A[1\,..\,a]$ is said to have a suffix/prefix match (*overlap*) of length ℓ with string $B[1\,..\,b]$ if the suffix $A[a - \ell + 1\,..\,a]$ matches the prefix $B[1\,..\,\ell]$. The goal is to output the longest overlap for each ordered-pair of strings in $\mathcal{T}$. More specifically, we output only those overlaps that have non-zero lengths, or lengths greater than a given minimum threshold. Gusfield et al. [5] gave an optimal-time algorithm to find the longest exact overlaps in $O(n + t_{\mathsf{output}})$ time, where $t_{\mathsf{output}} \leq r^2$ denotes the output size.

A natural extension to the above problem is to find *approximate* overlaps, that is, allow some number of insertions, deletions and mismatches between the suffix and prefix. Insertions and deletions can, however, make it non-trivial to choose the "best" approximate overlap. Now a more appropriate goal is to find the *least random overlaps* [8]. All the previous algorithms [8,9,15,21] for approximate overlaps have $\Omega(r^2)$ worst-case time regardless of the output size. Comparing

[*] Funded by the Academy of Finland grant 118653 (ALGODAN), and Helsinki Doctoral Programme in Computer Science (HECSE).

a quadratic number of sequence pairs becomes infeasible e.g. for the current iteration of DNA sequencing machines, some of which produce huge volumes of constant length sequences [12]. We propose the first algorithms that achieve time proportional to the output size; for example, if the number of allowed errors k is constant, we can find the least random overlaps in $O((n + t_{\mathsf{output}}) \operatorname{polylog}(n))$ time, which is within $\operatorname{polylog}(n)$ factor from the optimal time. For sequences shorter than $\log n/(k^{\frac{1}{k}}\sigma)$, we can achieve $O(n \log^k n + t_{\mathsf{output}})$ time for any k. All of our results have a linear working-space and assume an alphabet size of $\sigma = O(\operatorname{polylog}(n))$.

If the number of strings is relatively low, output sensitivity does not make a difference since r^2 becomes $O(n)$. Even so, we still improve the previous best worst-case bound from $O(knr)$ [9] to $O(\frac{c^k}{k!}nr)$ when all the strings are longer than $\epsilon \log^k r$ for any $\epsilon > 0$ and $k < \log r/ \log\log r$.

The next two sections give a short review of the earlier work on exact and approximate overlaps, respectively. The latter sections describe our algorithms, including an extension to find the least random overlaps. The last section sketches how to decrease the working-space to $O(n \log \sigma)$ bits.

2 Preliminaries

A *string* $S[1..n] = S[1]S[2]\cdots S[n]$ is a *sequence* of *symbols*. Each symbol is an element of an ordered *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$, where we assume $\sigma = O(\operatorname{polylog}(n))$. A *substring* of S is written $S[i..j] = S[i]S[i+1]\cdots S[j]$. A *prefix* of S is a substring of the form $S[1..j]$, and a *suffix* is a substring of the form $S[i..n]$. The lexicographical order "<" among strings is defined in the obvious way. The *unit-cost edit distance* $ed(T, T')$ is defined as the minimum number of insertions, deletions and replacements of symbols to transform string T into T' [10]. *Hamming distance* $h(T, T')$ is the number of mismatching symbols between strings T and T'. Hamming distance requires that $|T| = |T'|$ while the edit distance can be computed for arbitrary length strings.

2.1 Finding Exact Overlaps

The exact all-pairs suffix/prefix matching problem is to find, for each ordered pair $T_i, T_j \in \mathcal{T}$, their longest non-zero length overlap. The problem is interesting here because there exists an elegant output-sensitive solution. The problem can be solved in optimal time by building a *generalized suffix tree* for the input strings:

Theorem 1 ([5,4]). *Given a set of r strings of total length n, the longest* exact *overlaps can be found in linear space and in $O(n+t_{\mathsf{output}})$ time, where $t_{\mathsf{output}} \le r^2$ is the number of non-zero length overlaps found.*

The algorithm computes the longest suffix/prefix matches as follows. First, the strings $T_1, T_2, \ldots, T_r$ are concatenated into one string $T = T_1\$_1T_2\$_2\cdots T_r\$_r$, separating each string by a unique terminator symbol $\$_i$. The generalized suffix tree of T contains in total n leaves and at most $n-1$ internal nodes. We require two special definitions: an internal node v is said to have a *terminating edge* i if there exists a leaf node branching from v with the symbol $\$_i$; and a leaf node l is called a *whole-suffix* j if the labels from the root node to leaf l spell out the whole string T_j. Notice that there are at most n terminating edges and exactly r whole-suffixes.

To output the longest overlaps, we traverse the suffix tree once in the depth-first order and keep track of terminating edges using r stacks. For each internal node v encountered during the traversal, we check if it has one or more terminating edges, and for each terminating edge i of v, we push the string depth of v into the i-th stack. Now, whenever we encounter a leaf that corresponds to some whole-suffix j, we can output the longest overlaps by outputting the topmost values in non-empty stacks: if the topmost value of i-th stack is ℓ, then the prefix of $T_j[1\mathinner{..}\ell]$ matches the suffix $T_i[|T_i|-\ell+1\mathinner{..}|T_i|]$, and it must be the longest prefix of T_j that matches a suffix of T_i. When we visit v the second time (postorder), we pop the stack values inserted at v. Finally, after traversing the whole suffix tree, we have covered all different combinations of overlapping string-pairs. To avoid using $O(r^2)$ time to check all r stacks for all r whole-suffixes, we can keep track of non-empty stacks with a doubly-linked list [5]. See [14] for a space-efficient variant of the above algorithm.

3 Related Work on Approximate String Matching

There are several results for finding overlaps in the approximate case. We omit here all the algorithms that might work well in *practice* — their worst-case time is typically $\Omega(n^2)$. This includes e.g. the q-gram filter from Rasmussen et al. [15], and the suffix filter from Välimäki et al. [21], to name a few. We will next introduce the concept of *least random overlaps* and revisit some basic results on approximate pattern and dictionary matching.

Least Random Overlaps. Let $A[1\mathinner{..}a]$ and $B[1\mathinner{..}b]$ denote two random strings from the Bernoulli trial over symbols in Σ. Kececioglu and Myers [8] proposed a precomputed table $\Pr_\sigma(l,d)$ that gives an upper-bound for the probability that A and B match with d errors and l matching symbols. They considered only insertions and deletions to estimate $\Pr_\sigma(l,d)$, thus, the number of matching symbols is simply $l=(a+b-d)/2$. The goal is to find the overlap that minimizes $\Pr_\sigma(l,d)$ or maximizes the *likelihood* of the alignment, say $L_\sigma(l,d)=-\log_\sigma \Pr_\sigma(l,d)$. The likelihood values are increasing in l and decreasing in d [8]. Let $l_{\max}=\max\{|T_i| : T_i \in \mathcal{T}\}$ denote the largest string length in $\mathcal{T}$. If we assume that the likelihood values are precomputed for $0 \le l \le l_{\max}$ and $0 \le d \le k$, the least random overlaps can be found in $O(\epsilon n^2)$ time, where $\epsilon > 0$ is an error rate parameter and $k=\lceil \epsilon l_{\max} \rceil$ [8].

Landau et al. [9] improved the above worst-case bound to $O(k\,|T_j|)$ for a string-pair T_i and T_j. Then the total time over all string-pairs is $O(knr)$. They also generalized the likelihood computation for the edit distance model: since now there is no direct way to compute l, they assumed a precomputed table $L_\sigma(m, n, d)$ for $0 \leq m, n \leq l_{\max}$ and $0 \leq d \leq k$. Then the likelihood of A matching B with edit distance d is given by $L_\sigma(a, b, d)$.

Approximate Pattern Matching. We will next give a summary of basic results on approximate string matching under the *unit-cost edit distance* model [10]. Let $U_k(P)$ denote the set of strings within k edit distance from string $P[1..m]$. The set $U_k(P)$ is called the *neighborhood* of P. A classical result shows that $U_k(P) = O(\sigma^k m^k)$ [20]. A simple depth-first search [7, Remark 2] on the suffix tree of T can be used to search the neighborhood of P. We start the traversal from the root and traverse down until the current path does not appear as a prefix of any of the strings in $U_k(P)$ — see e.g. [13] for a more detailed explanation. In the worst case, string T contains all the possible strings in $U_k(P)$ and the total number of columns evaluated in the dynamic programming becomes $O(m \min\{n, \sigma^k m^k\})$. Since each column can be computed in $O(k)$ time and space [19], the total time to traverse the neighborhood becomes $O(\sigma^k m^{k+1} k)$. The result is a subset of suffix tree nodes whose upward path label (or a prefix of it) belongs to the neighborhood of P.

The above traversal can be done in backward manner by using an *FM-index* [3] to cover all suffixes of P and their neighborhoods in one traversal. The FM-index can be constructed in $O(n \log n \log \sigma)$ time and in $nH_k(T) + o(n \log \sigma)$ bits of space, where $H_k(T) \leq \log \sigma$ denotes the k-th order entropy of T [11]. Since we assume $\sigma = O(\text{polylog}(n))$, pattern matching is supported in constant time per symbol [3], thus, the time complexity of the above traversal is retained. Each step of the backward search gives a suffix array range $[s, e]$ that can be mapped to the corresponding suffix tree node v with a constant-time *lowest common ancestor* (LCA) query [17], that is, $v = \mathsf{lca}(l_s, l_e)$, where l_s and l_e are the s-th and e-th leaf in lexicographical order. Finally, notice that a single backward traversal is enough to cover all suffixes of P and their neighborhoods $U_k(P[i..m])$ for $i \in [1, m]$ because strings $U_k(P[i..m])$ occur as suffixes of the strings in $U_k(P)$.

Approximate Dictionary Queries. We review a result by Cole et al. [2] for approximate dictionary queries. The result assumes a *weakly nonuniform* RAM model, that is, a fixed number of precomputed constants that depend on the word length [6]. It will be utilized later in Sect. 4 as a part of our algorithm.

Theorem 2 ([2]). *Given a set of r strings of total length n, a* dictionary query *asks which strings match to a given pattern of length m. After preprocessing the dictionary in a deterministic*

$$O\left(n + r\frac{(c_1 \log r)^k}{k!} + \min\{n, \sigma\} \log n\right) \textit{ time and } O\left(n + r\frac{(c_1 \log r)^k}{k!}\right) \textit{ space,}$$

approximate matches within Hamming distance $k \leq \text{polylog}(r)$ can be found in[1]

[1] The $(\log \log r)$-factor in the query time is explained at the end of Sect. 5 in Cole et al. [2]. It requires that $k \leq \text{polylog}(r)$. For larger k, the factor grows to $(\log \log n)$.

$$O\left(m + \frac{(c_2 \log r)^k}{k!} \log\log r + t_{\mathsf{output}}\right) \textit{ time,}$$

where t_{output} *is the number of matching strings and* $c_1, c_2 > 1$ *are constants. The same complexities hold also for edit distance apart from the output size which grows to* $3^k \cdot t_{\mathsf{output}}$.

The data structure is called the *k-error tree* and has several interesting properties: (P1) the k-error tree of r strings contains in total $O(r(c_1 \log r)^k/k!)$ nodes and leaves, (P2) each leaf of the k-error tree corresponds to one of the r strings, and (P3) one string corresponds to at most $O((c_1 \log r)^k/k!)$ leaves. Notice that Cole et al. [2] provide both deterministic and randomized preprocessing time complexities; the additional time required by the deterministic preprocessing is $O(\min\{n, \sigma\} \log n)$ which simplifies to $O(n)$ here because $\sigma = O(\text{polylog}(n))$.

The k-error tree is used for approximate pattern matching as follows. First, the given pattern is preprocessed so that subsequent LCP queries between any suffix of the pattern and the k-error tree can be computed in $O(\log\log r)$ time. This preprocessing is done only once and requires $O(m)$ time. Now the actual pattern matching can be completed solely by LCP queries, requiring in total $O((c_2 \log r)^k/k!)$ LCP queries. The total time complexity is then $O(m + (\log\log r)(c_2 \log r)^k/k!)$. As a result, the pattern matching query returns a set of $O((c_2 \log r)^k/k!)$ nodes from the k-error tree. Occurrences of the pattern, i.e. strings whose prefix matches the pattern with $\leq k$ errors, can be enumerated by outputting the string identifiers from the leaves that belong to the result nodes' subtrees.

4 Output-Sensitive Algorithms

In this section, we show how to find the *longest approximate overlaps*. Since the suffix and prefix can have different lengths in the edit distance model, we define the *length of the overlap* as the length of the suffix in the alignment, and report the longest suffix length that matches. The next two subsections describe an algorithm for short and long strings, respectively. The last subsection combines these algorithms for sets of variable sized strings.

Remark 1. In all cases, we can easily report only those overlaps that are longer than a given minimum length threshold $\ell_{\min}$. The threshold should be $\ell_{\min} > k$ since suffixes of length $\ell \leq k$ match any prefix of length ℓ with edit distance k.

4.1 Method for Short Strings

We propose the following algorithm to deal with short strings, that is, we assume that all strings in $\mathcal{T}$ are of length at most β.

Lemma 1. *We are given a set* $\mathcal{T}$ *of* r *strings, where each string is of length at most* β *and the total length is* n. *After preprocessing the input strings in* $O(n \log n \log \sigma)$ *time, the longest approximate overlaps within edit distance* k

can be found in $O(n\,\beta^k\,\sigma^k\,k + t_{\mathsf{output}})$ time and $O(n)$ space, where $t_{\mathsf{output}} \le r^2$ is the total number of overlaps outputted.

Proof. Let us first show how to achieve the time complexity with a three-step algorithm. (i) We start by building the suffix tree of string $T = T_1\$_1T_2\$_2\cdots T_r\$_r$ and preprocess it for constant-time LCA queries [17]. We also build the FM-index[2] of Ferragina et al. [3] for T in $O(n \log n \log \sigma)$ time [11]. (ii) Then, we do approximate search on each string T_i separately. More specifically, the goal is to *mark* nodes of the suffix tree that have approximate match with one or more suffixes of T_i. The marks are represented as 3-tuples $\langle i, \ell, d\rangle$, where i denotes the string identifier of T_i, ℓ denotes the length of the suffix of T_i that was aligned and $d \le k$ denotes the edit distance. The marks are attached to the nodes as e.g. linked lists. The approximate pattern matching of T_i is done backwards using the FM-index (see Sect. 3): at each step of the backward search, we map the current suffix array range $[s, e]$ to the corresponding suffix tree node $v = \mathsf{lca}(l_s, l_e)$, and check if v already has a mark from T_i (such a mark must be the last one in v's linked list). Then, we record from the dynamic programming matrix the longest suffix length ℓ having edit distance $d \le k$, and insert a new mark $\langle i, \ell, d\rangle$ if v does not yet have a mark from T_i. If v already has a mark $\langle i, \ell', d'\rangle$ and $\ell > \ell'$, we update the mark to new values $\langle i, \ell, d\rangle$. The search covers all suffixes of T_i in $O(\sigma^k|T_i|^{k+1}k)$ time (see Sect. 3). Since we add at most one mark per node, there are in total $O(\min\{|T_i|^{k+1}\sigma^k, n\})$ marks inserted for T_i. The total time over all r strings is $O(\sum \sigma^k|T_i|^{k+1}k) = O(\beta^k\sigma^k k \sum |T_i|)$, where $\sum |T_i| = n$. (iii) Finally, we do a depth-first traversal through the suffix tree similar to Gusfield et al's algorithm in Sect. 2.1: now the r stacks are used to keep track of the inserted marks, so that, each mark $\langle i, \ell, d\rangle$ we encounter gets pushed to stack i if ℓ is larger than the current topmost suffix length in stack i. When we encounter the whole-suffix of T_j, we output the topmost mark $\langle i, \ell, d\rangle$ from each stack, which represents the longest suffix of T_i that matches T_j with an edit distance of at most k.

Finally, let us analyze the space complexity. Searching all r strings at once requires more than linear space since then the total number of markers inserted to the tree would be $O(n\beta^k\sigma^k)$. To achieve linear space we partition the set $\mathcal{T}$ into disjoint sets $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_\beta$ so that the set $\mathcal{T}_i$ contains strings of length i. Some of the sets can be empty. Let r_i denote the number of strings in $\mathcal{T}_i$ so that now $\sum r_i i = n$, that is, the total length of the original set $\mathcal{T}$. We process each set $\mathcal{T}_i$ separately: we search $\lceil n/i^{k+1}\sigma^k\rceil$ strings of $\mathcal{T}_i$ at a time. Then the number of inserted marks is $O(\lceil n/i^{k+1}\sigma^k\rceil \cdot \min\{i^{k+1}\sigma^k, n\}) = O(n)$. However, it requires us to repeat the suffix tree traversal $O\left(\frac{r_i}{n}\, i^{k+1}\sigma^k\right)$ times for each $\mathcal{T}_i$. This does not affect the total time complexity since

$$O\left(n \cdot \sum_{i=1}^{\beta} \frac{r_i}{n}\, i^{k+1}\sigma^k\right) = O\left(\beta^k\sigma^k \sum_{i=1}^{\beta} r_i\, i\right) = O\left(n\beta^k\sigma^k\right).$$

[2] The construction in [11] does not retain the lexicographical order $\$_i < \$_{i+1}$. This is not a problem since we never search over a $\$_i$. Also, the order $\$_i < c \in \Sigma$ is retained similar to the generalized suffix tree.

In other words, if we process at most $\lceil n/i^{k+1}\sigma^k \rceil$ strings at a time at step (ii), we never exceed $O(n)$ space for marks. The step (iii) needs to be repeated multiple times, but the total time needed to do all these traversals is $O(n\beta^k\sigma^k)$. The total number of push/pop operations is not affected since step (iii) is repeated over disjoint sets of marks. □

If all the strings in $\mathcal{T}$ are shorter than $\log n/(k^{\frac{1}{k}}\sigma)$ the above lemma directly gives an $O(n \log^k n + t_{\mathsf{output}})$ time algorithm to find the longest approximate overlaps between all pairs of strings for any $k \geq 2$.

4.2 Method for Long Strings

We propose the following algorithm to deal with long strings. Here we assume that all strings are of length at least β, thus, the number of strings is bounded to $r \leq \lfloor n/\beta \rfloor$. Let us first consider this one-against-all problem:

Lemma 2. *We are given a pattern P of length m and a set $\mathcal{T}$ of r strings, where each string is of length $\geq \beta$ and their total length is n. After preprocessing the set $\mathcal{T}$ in*

$$O\left(n + r\frac{(c_1 \log r)^k}{k!}\right) \textit{ time and } O\left(n + r\frac{(c_1 \log r)^k}{k!}\right) \textit{ space,}$$

the longest approximate overlaps within edit distance $k \leq \operatorname{polylog}(r)$ and between the pattern and all the strings in $\mathcal{T}$ can be found in

$$O\left(m + m\,\frac{(c_2 \log r)^k}{k!}\,\log\log r + t_{\mathsf{output}}\,\frac{(c_1 \log r)^k}{k!}\right) \textit{ time,}$$

where $t_{\mathsf{output}} \leq r \leq \lfloor n/\beta \rfloor$ is the total number of overlaps outputted and $c_1, c_2 > 1$ are constants. For larger k, the term $(\log\log r)$ grows to $(\log\log n)$.

Proof. First, build the index in Theorem 2 for the set of r strings, which gives the above preprocessing time and space. Then, the pattern is preprocessed in $O(m)$ time (see Sect. 3). After the preprocessing, we can search each suffix of P in just $O(\log\log r\,(c_2 \log r)^k/k!)$ time per suffix. More specifically, we search each suffix of the pattern in order from the longest to the shortest suffix. This ensures that the longest overlaps are found first since the length of the overlap was defined to be the length of the suffix in the alignment. As a result we find in total $O(m(c_2 \log r)^k/k!)$ nodes from the k-error tree. Each result node corresponds to one interval in the array of leaves, however, these intervals can overlap: the interval $[i\,..\,j]$ can either be inside other interval $[i'\,..\,j']$ that has been already outputted, or it can cover some number of intervals $[i_1\,..\,j_1], \ldots, [i_p\,..\,j_p]$ that have been already outputted. We can detect these two cases (plus the case when the interval does not overlap) using a *y-fast trie* [22] as follows. When we are outputting results from an interval $[i\,..\,j]$, we first check if the interval is already processed using a $\langle i', j' \rangle \leftarrow \operatorname{predecessor}(i)$ query. If $j' \geq j$ the interval

$[i\,..\,j]$ appears inside $[i'\,..\,j']$ and has been already outputted. If not, we need to check if there are some intervals $[i_1\,..\,j_1], \ldots, [i_p\,..\,j_p]$ inside $[i\,..\,j]$. We can avoid re-outputting them by first computing $\langle i_1, j_1\rangle \leftarrow \text{successor}(i)$ and then using subsequent $\langle i_{i'}, j_{i'}\rangle \leftarrow \text{successor}(i_{i'-1}+1)$ queries to cover each $i' \in [2, p]$. Finally, we mark the interval as processed by adding a new key-value pair $\langle i, j\rangle$ into the y-fast trie, and remove the intervals $[i_1\,..\,j_1], \ldots [i_p\,..\,j_p]$ to avoid later re-computation. In total, we do at most one insertion, deletion and predecessor operation per result node. Also, we do at most two successor operations per result node. Thus, the total number of required operations on the y-fast trie is $O(m(c_2 \log r)^k/k!)$, each of them costing $O(\log\log r)$ (amortized) worst-case time [22]. The total space is not affected since the number of inserted intervals is limited by the size of the k-error tree.

We can ensure that the actual output size is $t_{\mathsf{output}} \leq r$ by using a bit-vector of length r to mark strings whose overlap against P is already reported. Furthermore, we can reset the bit-vector in $O(t_{\mathsf{output}})$ time by simply recording the output into a linked list. The total number of queries to this bit-vector is still $O(t_{\mathsf{output}}(c_1 \log r)^k/k!)$ because each string identifier is repeated $O((c_1 \log r)^k/k!)$ times in the leaves. □

Constructing the index only once and iterating the above algorithm over all strings in $\mathcal{T}$ solves the *all-against-all* problem for longest approximate overlaps when all strings have length $\geq \beta$:

Corollary 1. *Given a set $\mathcal{T}$ of r strings, where each string is of length $\geq \beta$ and their total length is n, the longest approximate overlaps within edit distance $k \leq \text{polylog}(r)$ can be found in*

$$O\left(n + n\,\frac{(c_2 \log r)^k}{k!}\,\log\log r + t_{\mathsf{output}}\,\frac{(c_1 \log r)^k}{k!}\right)\ \textit{time}$$

and $O(n + r\,(c_1 \log r)^k/k!)$ space, where $t_{\mathsf{output}} \leq r^2 \leq r\lfloor n/\beta\rfloor$ is the total number of overlaps outputted and $c_1, c_2 > 1$ are constants. For larger k, the term $(\log\log r)$ grows to $(\log\log n)$.

If all the strings in $\mathcal{T}$ are longer than $\beta = \epsilon \log^k r$, for any $\epsilon > 0$, and $k < \log r/\log\log r$, the above corollary simplifies to $O(\frac{c^k}{k!}nr)$ time and linear space, where $c = \max\{c_1, c_2\}$ is a constant. More precisely, the space is $O(n + c^k n/k!) = O(n)$ because $r \leq \lfloor n/\beta\rfloor$, and the terms in the above time complexity are

$$n\,\frac{(c_2 \log r)^k}{k!}\,\log\log r = O\left(\frac{c^k}{k!}nr\right)$$

when $k < \log r/\log\log r$, and

$$t_{\mathsf{output}}\,\frac{(c_1 \log r)^k}{k!} \leq r\left\lfloor\frac{n}{\beta}\right\rfloor\,\frac{(c_1 \log r)^k}{k!} = O\left(\frac{c^k}{k!}nr\right).$$

This is an improvement over the previous best worst-case time of $O(knr)$ [9] because $\frac{c^k}{k!}nr = o(knr)$ when k is not a constant.

4.3 Sets of Mixed Length Strings

Lemma 1 and 2 can be combined to support input sets with mixed length strings:

Theorem 3. *Given a set of r strings of total length n and a maximum number of errors $k = O(1)$, the longest approximate overlap for each string-pair can be found in $O((n + t_{\mathsf{output}})\mathrm{polylog}(n))$ time and $O(n)$ space, where $t_{\mathsf{output}} \leq 2r^2$ is the total number of overlaps outputted and $c_1, c_2 > 1$ are constants.*

Proof. We choose $\beta = (c_1 \log n)^k/k!$ and call strings of length $\leq \beta$ and $> \beta$ short and long, respectively. Long strings can overlap short strings, and vice-versa; the preprocessing of Lemma 2 is done for all strings in $\mathcal{T}$ of length $\geq \beta - k$. This requires $O(n\,\mathrm{polylog}(n))$ time and $O(n + r'(c_1 \log r)^k/k!) = O(n)$ space, where $r' \leq \lfloor n/(\beta - k)\rfloor = O(n/\beta)$ is the number of strings to preprocess. Then, Corollary 1 gives us $O((n+t'_{\mathsf{output}})\mathrm{polylog}(n))$ time to find overlaps for all string-pairs that have length $\geq \beta - k$. The preprocessing of Lemma 1 is done for the whole set $\mathcal{T}$. We search all short strings, including suffixes of length β from long strings, using the technique in Lemma 1. This gives the time complexity $O(n\beta^k\sigma^k k + t''_{\mathsf{output}}) = O(n\,\mathrm{polylog}(n) + t''_{\mathsf{output}})$ to output the longest approximate overlaps up to length β for all pairs of short strings and long strings. The final output, $t'_{\mathsf{output}} + t''_{\mathsf{output}}$, can contain two overlaps for some string-pairs having length from $\beta - k$ to β. □

5 Least Random Overlaps

We can modify Lemma 1 to find the *least random overlaps* (see Sect. 3) instead of the longest overlaps. We assume that the likelihood values $L_\sigma(m, n, d)$ are precomputed. First, we need to modify step (ii) so that we do not insert the mark for the longest suffix, but instead, for the one with maximum likelihood. The maximum value can be computed simultaneously with the dynamic programming: at each step of the backward search, we know the length of the prefix in the alignment ℓ', and from the dynamic programming matrix, we can choose the suffix length ℓ that maximizes $L_\sigma(\ell, \ell', d)$: there are $O(k)$ possible suffix lengths within k edit distance, and we add the mark for the best combination of d and ℓ. Also, we need to modify the process of collecting the inserted marks into the r stacks during the step (iii) of Lemma 1. Now, we insert the mark $\langle i, \ell, d\rangle$ to stack i only if its likelihood value is larger than the topmost value in the stack i. When we arrive at a whole-suffix, the topmost values in each stack correspond to overlaps having the highest likelihoods.

Lemma 2 can also be modified to find the least random overlaps. For each query, we receive $O(m(c_2 \log r)^k/k!)$ result nodes from the k-error tree. For each result node, we know the suffix and prefix length, say ℓ and ℓ', and the number of errors $d \leq k$. The goal is to sort the result nodes based on $L_\sigma(\ell, \ell', d)$ values. We cannot afford to store the whole result set, however, we can keep track of the highest likelihood of each result node found so far; then the result size is bounded also by the size of the k-error tree. Thus, we traverse the

$O(\min\{m(c_2 \log r)^k/k!, r(c_1 \log r)^k/k!\})$ nodes in depth-first order[3], and use a stack to record the highest likelihood ancestor — we output only those nodes that have higher likelihood than their ancestor. Similar to Lemma 2, we need to take care not to output the same interval multiple times. Outputting the final results in this order means that the overlaps having the highest likelihoods are outputted first.

Corollary 2. *We can find the* least random approximate overlaps*, which maximize the precomputed likelihood values $L_\sigma(m, n, d)$, in the same time and space as the longest approximate overlaps in the above subsections.*

6 Discussion

All the algorithms that we proposed require a linear working-space, that is, $O(n \log n)$ bits of memory. Let $H_k(T) \leq \log \sigma$ denote the k-th order entropy of T [18]. It seems possible to achieve $nH_k(T) + O(n) = O(n \log \sigma)$ bits of working-space with a negligible slowdown. Lemma 1 can be made space-efficient by substituting the suffix tree with a *compressed suffix tree* (CST) that admits constant time navigation in $|CST| = nH_k(T) + \Theta(n)$ bits of space [16]. The space required by the markers can be reduced to $O(n)$ bits by increasing the number of suffix tree traversals in step (iii) of Lemma 1 by a factor of $\log n$. Furthermore, the suffix tree used inside the k-error tree can be substituted with a CST (similar to what Chan et al. [1] did), and a larger β can be chosen to force the data structure of Lemma 2 into $|CST| + O(n)$ bits. A space-efficient variant of e.g. Theorem 3 retains the time complexity given. The exact details are out of the scope of this paper.

Acknowledgments. The author would like to thank Veli Mäkinen, Kimmo Fredriksson and Enno Ohlebusch for their constructive feedback.

References

1. Chan, H.-L., Lam, T.-W., Sung, W.-K., Tam, S.-L., Wong, S.-S.: A Linear Size Index for Approximate Pattern Matching. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 49–59. Springer, Heidelberg (2006)
2. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proc. STOC 2004, pp. 91–100. ACM (2004)
3. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Algorithms 3(2), 20–44 (2007)
4. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)

[3] We can do the traversal in $O(m(\log\log r)(c_2 \log r)^k/k!)$ time and $O(r(c_1 \log r)^k/k!)$ space by using an additional y-fast trie to track the pre- and post-order numbers of the result nodes.

5. Gusfield, D., Landau, G.M., Schieber, B.: An efficient algorithm for the all pairs suffix-prefix problem. Inf. Process. Lett. 41(4), 181–185 (1992)
6. Hagerup, T., Miltersen, P.B., Pagh, R.: Deterministic dictionaries. J. Algorithms 41(1), 69–85 (2001)
7. Jokinen, P., Ukkonen, E.: Two Algorithms for Approximate String Matching in Static Texts. In: Tarlecki, A. (ed.) MFCS 1991. LNCS, vol. 520, pp. 240–248. Springer, Heidelberg (1991)
8. Kececioglu, J.D., Myers, E.W.: Combinatiorial algorithms for dna sequence assembly. Algorithmica 13(1/2), 7–51 (1995)
9. Landau, G.M., Myers, E.W., Schmidt, J.P.: Incremental string comparison. SIAM J. Comput. 27(2), 557–582 (1998)
10. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
11. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Trans. Algorithms 4, 32:1–32:38 (2008)
12. Metzker, M.L.: Sequencing technologies - the next generation. Nature Reviews Genetics 11(1), 31–46 (2010)
13. Navarro, G., Baeza-Yates, R.A., Sutinen, E., Tarhio, J.: Indexing methods for approximate string matching. IEEE Data Engineering Bulletin 24(4), 19–27 (2001)
14. Ohlebusch, E., Gog, S.: Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem. Inf. Process. Lett. 110(3), 123–128 (2010)
15. Rasmussen, K.R., Stoye, J., Myers, E.W.: Efficient q-gram filters for finding all e-matches over a given length. J. of Computational Biology 13(2), 296–308 (2006)
16. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41(4), 589–607 (2007)
17. Schieber, B., Vishkin, U.: On finding lowest common ancestors: simplification and parallelization. SIAM Journal on Computing 17(6), 1253–1262 (1988)
18. Shannon, C.E.: A mathematical theory of communication. Bell System Technical Journal 27(1), 379–423 (1948)
19. Ukkonen, E.: Algorithms for approximate string matching. Information and Control 64(1-3), 100–118 (1985)
20. Ukkonen, E.: Finding approximate patterns in strings. J. Algorithms 6(1), 132–137 (1985)
21. Välimäki, N., Ladra, S., Mäkinen, V.: Approximate all-pairs suffix/prefix overlaps. Information and Computation 213, 49–58 (2012); CPM 2010 Special Issue
22. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space Theta(N). Inf. Process. Lett. 17(2), 81–84 (1983)

Compressed String Dictionary Look-Up with Edit Distance One[⋆]

Djamal Belazzougui[1] and Rossano Venturini[2]

[1] LIAFA, Univ. Paris Diderot - Paris 7
`dbelaz@liafa.jussieu.fr`
[2] Dept. of Computer Science, University of Pisa
`rossano@di.unipi.it`

Abstract. In this paper we present different solutions for the problem of indexing a dictionary of strings in compressed space. Given a pattern P, the index has to report all the strings in the dictionary having *edit distance* at most one with P. Our first solution is able to solve queries in (almost optimal) $O(|P| + occ)$ time where occ is the number of strings in the dictionary having edit distance at most one with P. The space complexity of this solution is bounded in terms of the k-th order entropy of the indexed dictionary. Our second solution further improves this space complexity at the cost of increasing the query time.

1 Introduction

Modern web search, information retrieval, data base and data mining applications often require solving string processing and searching tasks. Most of such tasks boil down to some basic algorithmic primitives which involve a large dictionary of strings with variable length. The interest in approximate searches over dictionaries of strings is increasing since they appear frequently in many practical scenarios. In Web search, for example, users query the engine with possibly misspelled terms that can be corrected by choosing among the closest terms stored in a trustable dictionary. In data mining and data base applications, instead, an automatically built dictionary may contain noise in the form of misspelled strings. Thus, we may need to resort to approximate searches in order to identify the closest dictionary strings with respect to a (correct) input string.

The *Edit distance* (also known as Levenstein distance) is the most commonly used distance to deal with misspelled strings. The edit distance between two strings is defined as the minimal number of edit operations required to transform the first string into the second string. There are three possible edit operations:

[⋆] This work has been partially supported by the MADALGO Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation, the French ANR-2010-COSI-004 MAPPI project, the S-CUBE EU-FP7- 215483 project, the ASSETS CIP-ICT-PSP-250527 project, MIUR PRIN MadWeb 2008, and FIRB Linguistica 2006.

deletion of a symbol, insertion of a symbol and substitution of a symbol with another.

The problem *String Dictionary Look-up with Edit Distance One* is defined as follows. Let $\mathcal{D} = \{S_1, S_2, \dots, S_d\}$ be a set of d strings of total length n drawn from an alphabet Σ of size σ. We want to build a (compressed) index that, given any string $P[1, p]$, reports all the strings in $\mathcal{D}$ having edit distance at most 1 with P. In the following we assume that the strings in $\mathcal{D}$ are all distinct and sorted lexicographically (namely, for any $1 \leq i < d$, $S_i < S_{i+1}$).

In this paper we provide two efficient and compressed solutions for the problem above. The first solution guarantees (almost) optimal query time while requiring compressed space. Namely, we show how to obtain an index of $2nH_k + n \cdot o(\log \sigma) + 2d \log d$ bits, that is able to report all the *occ* strings having edit distance at most 1 with P in time $O(p + occ)$. Here H_k denotes the k-th order entropy of the strings in the dictionary. Interestingly, the time complexity of this solution is independent of alphabet size. This is quite an uncommon result for compressed data structures dealing with texts. The second solution provides possible space/time tradeoffs by using a completely different approach. Its space occupancy, indeed, decreases to $nH_k + n \cdot o(\log \sigma)$ bits. This better space bound is obtained at the cost of increasing the query time to $O(p \log \log \sigma)$.

Interestingly, our first solution can be extended to support an additional operation which has interesting practical applications. We assume that each string S_i in $\mathcal{D}$ has been assigned a score $c(S_i)$. For example, the score could establish the relative importance of any string with respect to the others. It is possible to extend our solution in order to support the extra operation $\mathsf{Top}(P[1, p], k)$ that reports the k highest scored strings in $\mathcal{D}$ having edit distance at most 1 with P. This operation is solved in $O(p + k \log k)$ time.

2 Related Work

The literature presents several solutions to the problem of indexing string dictionaries to efficiently search strings with error distance one. In the following we restrict our attention only on results that currently have the best time/space complexities.

The work in [3] proposes two solutions to solve the string dictionary lookup with Hamming distance one[1]. The first solution has $O(p + occ)$ query time and uses $O(\sigma \cdot n \log n)$ bits of space. The main data structure is a trie that indexes strings in $\mathcal{D}$ plus extra strings. An extra string is a string that does not belong to $\mathcal{D}$ but has Hamming distance one with at least a string in the set. Clearly, each root to a leaf path in the trie represents either a string in $\mathcal{D}$ or an extra string. In every leaf representing a string S there is stored the list of indices of strings in $\mathcal{D}$ that have Hamming distance one with S. The query for P is solved by navigating the trie. If a leaf is reached, it reports all the indices stored in the leaf. The major drawback of this solution is represented by its space occupancy for non-constant

[1] However, they can be easily extended to deal with the more general Edit distance.

size alphabet and by its construction time. Indeed, it is unknown how to build this data structure in $O(n\sigma)$ time.

The second solution in [3] is slower than the previous one by an additive term $O(\log n)$ (namely, query time is $O(p+\log n+occ)$). The advantage is represented by its space occupancy which is $O(n \log n)$ and, thus, it is better for non-constant size alphabets. The solution resorts to two tries and a balanced search tree. The first trie contains the set of strings $\mathcal{D}$ while the second trie indexes the strings in $\mathcal{D}$ reversed. The query algorithm exploits the following property: if there exists a string S in $\mathcal{D}$ having distance one with $P[1,p]$, it can be factorized as $S = P[1,i] \cdot c \cdot P[i+2,p]$, for some index i and symbol $c \in \Sigma$. This is a key property that has been exploited by almost all the subsequent solutions, including ours. These solutions differ from each other in data structures and algorithms they use to discover all these factorizations. For each string $S[1,s]$ in $\mathcal{D}$, we consider all triplets $(\mathsf{np}_i(S),\ S[i+1],\ \mathsf{ns}_{i+2}(S))$ where $\mathsf{np}_i(S)$ is the identifier of the node corresponding to prefix $S[1,i]$ in the first trie and $\mathsf{ns}_{i+2}(S)$ is the identifier of the node corresponding to $S[i+2,s]$ reversed in the second trie. These triples are inserted in a search tree that is able to report, given a pair of node identifiers u and v, all the triples with u in the first component and v in the third component. The query algorithm works as follows. For any index i, it identifies the nodes $\mathsf{np}_i(P)$ and $\mathsf{ns}_{i+2}(P)$ and uses the search tree by querying for these two nodes. If the triple $(\mathsf{np}_i(P),\ c,\ \mathsf{ns}_{i+2}(P))$ is returned, then the string $S = P[1,i] \cdot c \cdot P[i+2,p]$ is in $\mathcal{D}$ and has distance one from P. Remarkably, this solution can be easily made dynamic: inserting or deleting a string $P[1,p]$ from $\mathcal{D}$ costs $O(p + \log n)$.

The current best solution is the one presented in [1]. This solution follows a similar approach but obtains significantly better time and space complexities. Indeed, this solution achieves $O(p + occ)$ query time by requiring optimal $O(n \log \sigma)$ bits of space. This is obtained by carefully combining compact tries, (minimal) perfect hash functions and Rabin-Karp fingerprinting.

The solution presented in [4] solves the problem for Hamming distance and it deals with binary strings having a fixed length L. The strong limitation on the length of strings allows to achieve optimal $O(L/w)$ query time, where w is the size of a memory word. However, the space usage grows to $O(n \cdot L \lg L)$ bits. Moreover, this solution can only report a single matching string from the dictionary.

Finally, we observe that currently known solutions to solve the more general problem of approximate full-text indexing are not competitive with solutions presented in this paper. Indeed, all known solutions for approximate full-text indexing for edit distance one incur at least a factor $\Omega(\log n)$ in space usage and/or an additive $\Omega(\log n)$ term in query time.

3 Background

In this section we collect a set of algorithmic tools that will be used by our solutions. We report each result together with a brief description. More details can be obtained by consulting the corresponding references.

Compressed Strings with Fast Random Access. We will require the availability of a storage scheme for a text T which uses compressed space and is able to decode in $O(1)$ time any symbol of T. To this aim, we use the following result in [9].

Lemma 1. *Given a text $T[1, n]$ drawn from an alphabet of size σ, there exists a compressed data structure that supports the access in constant time of any substring of T of length $O(\log n)$ bits requiring $nH_k(T) + \mathsf{extra}$ bits, where $H_k(T)$ denotes the kth empirical entropy of T and $k = o(\log_\sigma n)$. The extra space depends on the alphabet size σ: $\mathsf{extra} = o(n)$ if $\log \sigma = o(\log n / \log\log n)$, $\mathsf{extra} = n \cdot o(\log \sigma)$ otherwise.*

The scheme can be also used in cases in which T is the concatenation of a set of strings (namely, $T = S_1 \cdot S_2 \cdot \ldots \cdot S_d$). The starting positions of strings in T are stored by resorting to Elias-Fano's representation [7,8] within $d \log(\frac{n}{d}) + O(d)$ bits. This additional structure allows us to access an arbitrary portion of any string in optimal time.

Rabin-Karp Signature. Given a string $S[1, s]$, the Rabin-Karp signature [13] $\mathsf{rk}(S)$ is equal to $\sum_{i=1}^{s} S[i] \cdot t^i \pmod M$, where M is a prime number and t is a randomly chosen integer in $[1, M-1]$. Given a set of strings $\mathcal{D}$ of d strings of total length n, it can be obtained an instance $\mathsf{rk}()$ of the Rabin-Karp signature that maps strings in $\mathcal{D}$ to the first $O(M)$ integers without collisions, with M chosen among the first $O(n \cdot d^2)$ integers. It is known that a value of t that guarantees no collisions can be found in expected $O(1)$ attempts (e.g., see the analysis in [6]). The representation of the suitable function requires $O(\log n)$ bits of space.

Interestingly, Rabin-Karp signature guarantees that, after a preprocessing phase over a string S, signatures of strings close enough to S can be computed in constant time. This property is formally stated by the following lemma.

Lemma 2. *Given a string $S[1, s]$, for every prefix P of S, $\mathsf{rk}(P)$ can be computed in constant time. Moreover, for every string Q at distance 1 from S, $\mathsf{rk}(Q)$ can be computed in constant time. It is required a preprocessing phase that takes time $O(s)$.*

Minimal Perfect Hash Function. Result in [12] shows how to build a space/time optimal minimal perfect hash function. This result is summarized in the following lemma.

Lemma 3. *Given a subset of $S \subseteq U = 2^w$ of size n, there exists a minimal perfect hash function for S that can be evaluated in $O(1)$ time and requires $n \log e + o(n)$ bits of space.*

Compressed Static Function. Often we have to represent satellite data associated with the keys in S. Repetitions in these associated values can be exploited in order to reduce space requirements. The following result can be proven by using standard techniques.

Theorem 1. *A function F that assigns values from $[\sigma]$ (with $\sigma = \omega(1)$) to keys in $S = \{x_1, x_2, \ldots, x_n\} \subset U \subseteq 2^w$ can be represented in $nH_0 + n \cdot o(\log \sigma)$ bits such that the evaluation of F requires constant time, where H_0 denotes the empirical entropy of the assigned values $\{F(x_1), F(x_2), \cdots, F(x_n)\}$.*

Proof. We use a minimal perfect hash function $\mathsf{m}()$ to map keys to the first n integers by paying $\log e + o(1)$ bits per key (Lemma 3). We construct a sequence A that has the value associated with key x_j in position $m(x_j)$. Sequence A is represented in compressed form by using schema of Lemma 1.

4 A Compressed and Fast Solution

Our first solution can be seen as a compressed variant of the solution presented in [1]. However, we need to apply significant and non-trivial changes to that solution in order to achieve compressed space and to retain exactly the same (almost optimal) query time. More formally, in this section we prove the following theorem.

Theorem 2. *Given a set of strings $\mathcal{D} = \{S_1, S_2, \ldots, S_d\}$ of d strings of total length n drawn from an alphabet Σ of size σ, there exists an index that, given any pattern $P[1, p]$, reports in $O(p + occ)$ time all the occ strings in $\mathcal{D}$ having edit distance at most one with P. It requires:*

1. *$nH_k + o(n) + 2d \log d$ bits of space for any $k = o(\log n)$, if $\sigma = O(1)$;*
2. *$2nH_k + n \cdot o(\log \sigma) + 2d \log d$ bits of space for a fixed $k = o(\log_\sigma n)$, otherwise.*

At a high level our solution works as follows. Firstly, it identifies a set of $O(p+occ)$ *candidate strings* being a superset of the strings that have edit distance at most one with P. Then, it discards all candidate strings that actually do not belong to $\mathcal{D}$. For the moment, let us assume that establishing whether or not a candidate string belongs to $\mathcal{D}$ costs constant time. Later, we will discuss how to efficiently perform this non-trivial task[2].

Our solution requires identifying strings in $\mathcal{D}$ that share prefixes and suffixes with the query string P. For this aim we resort to two patricia tries $\mathcal{PT}$ and $\mathcal{PT}_r$ that index respectively the strings in $\mathcal{D}$ and the strings in $\mathcal{D}$ written in reversed order. As common, a node in each patricia trie is uniquely identified by the time of its visit in the preorder visit of the tree. The tree structure of each patricia trie is represented in $O(d)$ bits with standard succinct solutions [14]. In order to perform searches on patricia tries, we add data structures to compute the length of longest common prefix (lcp) and longest common suffix (lcp_r) for any pair of strings in $\mathcal{D}$. A standard constant time solution requiring $O(d(1+\log \frac{n}{d}))$ bits of space is obtained by writing lcps between lexicographically consecutive strings (resp. reverse strings) using Elias-Fano's representation [7,8] and by resorting to Range Minimum Queries (rmq) (see e.g., [11]) on these

[2] Notice that just accessing each symbol of these candidate strings would cost $O(p + p \cdot occ)$ time which is much higher than our claimed complexity.

arrays. Fast percolation of the tries is obtained by augmenting the branching nodes with monotone minimal perfect hash functions as described in [2]. In this way choosing the correct edge to follow from the current node can be done in constant time regardless of the alphabet size. The extra cost in term of space is bounded by $O(d \log\log\sigma)$ bits. The correctness of the steps performed during the search is established by comparing the searched string and labels on the followed edges. This is done by accessing directly to the appropriate portion of strings in $\mathcal{D}$ from their compressed representations. For this aim $\mathcal{D}$ is represented by resorting to the compressed scheme of Lemma 1 that allows constant time access to any symbol of any string in $\mathcal{D}$. The space required by this is bounded by kth order entropy accordingly to Lemma 1. Since the strings do not keep their original order in the trie $\mathcal{PT}_r$, we store a permutation π of $\{1, 2, \ldots, d\}$ that keeps track of the original order in $\mathcal{D}$ of each leaf of $\mathcal{PT}_r$. Namely, $\pi(i)$ is the index in $\mathcal{D}$ of the ith lexicographically smaller string in $\mathcal{PT}_r$. Clearly, storing π requires $d \log d + O(d)$ bits.

Candidate Strings Obtained by Deleting a Symbol. The identification of candidate strings for deletion is an easy task. Indeed, we observe that there are just p possible candidate strings obtainable from $P[1, p]$ by deleting one of its symbol. Thus, we simply consider any string $P[1, i] \cdot P[i + 2, p]$ as a candidate string. However, any of these strings is reported only after having checked that it actually belongs to $\mathcal{D}$. As said above, for the moment we assume that this non-trivial task can be done in $O(1)$ (amortized) time.

Candidate Strings Obtained by Inserting or Substituting a Symbol. Identifying candidate strings for insertion or substitution of a symbol is an easy task whenever the alphabet has constant size. In this case there are, indeed, $O(\sigma \cdot p) = O(p)$ candidate strings obtained by inserting or substituting any possible symbol of Σ in any position of P. This implies that data structures above suffice for Point 1 in Theorem 2[3]. Identifying insertions and substitutions with a larger alphabet is a much harder task, which requires an additional data structure. Our additional data structure follows the idea presented in [1] which allows us to reduce the number of candidate strings from $O(\sigma \cdot p)$ to $O(p + occ)$. However, our solution is forced to use more sophisticated arguments in order to achieve space bounded in term of kth order entropy. In the following we consider only insertions since substitutions are solved similarly.

Given the set of strings $\mathcal{D}$ and the two patricia tries $\mathcal{PT}$ and $\mathcal{PT}_r$, our first step consists in building a set $\mathcal{T}$ of tuples. For each string S in $\mathcal{D}$ of length s, we consider each of its factorizations of the form $S = S[1, i] \cdot c \cdot S[i + 2, s]$. For each of them, we add to $\mathcal{T}$ the tuple $\langle \mathsf{np}, i, c = S[i + 1], s - (i + 2), \mathsf{ns} \rangle$ where np (resp. ns) is the index of the highest node in $\mathcal{PT}$ (resp. $\mathcal{PT}_r$) prefixed by $S[1, i]$ (resp. $S[i + 2, s]$ reversed). Observe that the cardinality of $\mathcal{T}$ is at most n, since we add at most s tuples for a string S of length s.

[3] Recall that we are still assuming that we can check in $O(1)$ whether a candidate string belongs to $\mathcal{D}$.

The set $\mathcal{T}$ contains enough information to allow the identification of all the candidate strings. For insertion we consider all the factorizations of P having the form $P = P[1,i] \cdot P[i+1,p]$. For each of them, we identify the (highest) nodes np_i and ns_{i+1} in $\mathcal{PT}$ and $\mathcal{PT}_r$ that are prefixed respectively by $P[1,i]$ and $P[i+1,p]$ reversed. Clearly, identifying all these nodes for all the factorizations of P requires $O(p)$ time by resorting to the patricia tries.

The key observation to identify candidate strings is the following: If there exists a tuple $\langle \mathsf{np}_i, i, c, p-i-1, \mathsf{ns}_{i+1}\rangle$ in $\mathcal{T}$, then the string $S = P[1,i]\cdot c\cdot P[i+1,p]$ belongs to $\mathcal{D}$ and, obviously, has distance one from P[4].

Our data structure is built on top of $\mathcal{T}$ and allows us to easily identify the required tuples. We notice that there may exist several tuples of the form $\langle \mathsf{np}, i, \star, \mathsf{ns}, i'\rangle$. These groups of tuples share the same four components np, i, ns and i', and differ just for the symbol c. In order to distinguish them, we arbitrarily rank tuples in the same group and we assign to each of them its position in the ranking. We build a data structure that, given the indexes np and ns of two nodes, two lengths i and i' and rank r, returns the symbol c of the rth tuple of the form $\langle \mathsf{np}, i, \star, \mathsf{ns}, i'\rangle$ in $\mathcal{T}$. The data structure is allowed to return an arbitrary symbol whenever such a tuple does not exist. The use of such a data structure to solve our problem is simple. For each factorization $P[1,i]\cdot P[i+1,p]$ of P, we query the data structure above several times by passing the parameters np_i, i, $p-i-1$, ns_{i+1} and r. The value of r is initially set to 0 and increased by 1 for the subsequent query. After every query, we check if the string $S = P[1,i]\cdot c\cdot P[i+1,p]$ belongs to $\mathcal{D}$, where c is the symbol returned by the data structure. We pass to the next factorization as soon as we discover that either the string S does not belong to $\mathcal{D}$ or symbol c has been already seen for the same factorization. Both these conditions provide the evidence that no tuple $\langle \mathsf{np}_i, i, \star, p-i-1, \mathsf{ns}_{i+1}\rangle$ with rank r or larger can belong to $\mathcal{T}$. It is easy to see that the overall number of queries is $O(p + occ)$.

We are now ready to present a data structure to index $\mathcal{T}$ as described above that requires $O(1)$ time per query and uses entropy bounded space.[5] The first possible compressed solution consists in appropriately defining a function $F()$ which is then represented by using solution in Theorem 1. For any tuple $\langle \mathsf{np}, i, c, \mathsf{ns}, i'\rangle$ having rank r in $\mathcal{T}$, we set $F(\mathsf{np}, i, \mathsf{ns}, i', r)$ equal to c. Queries above are solved by appropriately evaluating function $F()$. Accordingly to Theorem 1, each query is solved in constant time. As far as space occupancy is concerned, we observe that $F()$ is defined for at most n values and that any symbol of any string in $\mathcal{D}$ is assigned at most once. Thus, by combining these considerations with Theorem 1, it follows that the representation of $F()$ requires at most $nH_0 + n \cdot o(\log\sigma)$ bits. A boost of this space complexity to nH_k is obtained by defining several functions F, one for each possible context of length k. Here $k = o(\log_\sigma n)$ is an arbitrary but fixed parameter. The function $F_{\mathsf{cntxt}}()$ is defined only for tuples $\langle \mathsf{np}, i, c, \mathsf{ns}, i'\rangle$ where the symbol c is preceded by the

[4] Observe that similar considerations hold also for substitutions with the difference that we skip ith symbol in factorizations of the form $P = P[1,i-1]\cdot P[i]\cdot P[i+1,p]$.

[5] We remark that the set of tuples $\mathcal{T}$ is just conceptual and not explicitly stored.

context cntxt in the string that induced the tuple. By summing up the cost of storing the representations of these functions, we have that the space occupancy is bounded by $nH_k + n \cdot o(\log \sigma)$ bits for the fixed $k = o(\log_\sigma n)$. Notice that splitting F in several functions is not an issue for our aim. In the algorithm above, indeed, we can query the correct function since we are always aware of the correct context.

Checking Candidate Strings. It is left to explain how to establish, in constant time, whether a candidate string belongs to $\mathcal{D}$. Observe that any candidate string has the form $S = P[1,i] \cdot P[i+2,p]$ in case of deletion, $S = P[1,i] \cdot c \cdot P[i+1,p]$ in case of insertion, or $S = P[1,i] \cdot c \cdot P[i+2,p]$ in case of substitution, for some symbol c and index i. The main issue behind this task is given by the fact that strings may not fit in a constant number of memory words. Thus, we cannot manage them directly in constant time. For this aim Rabin-Karp function $\mathsf{rk}()$ is used to create small sketches of the strings in $\mathcal{D}$ that fit in $O(1)$ memory words and that uniquely identify each string. Observe that the signatures assigned by function $\mathsf{rk}()$ are values smaller than M and, thus, each of them fits in $O(1)$ words of memory.

Once we have these perfect signatures, we use a minimal perfect hash function to connect each signature to the corresponding string in $\mathcal{D}$. Let $\mathcal{D}_{\mathsf{rk}}$ be the set of signatures assigned by $\mathsf{rk}()$ to strings in $\mathcal{D}$ (i.e., $\mathcal{D}_{\mathsf{rk}} = \{\mathsf{rk}(S) \mid S \in \mathcal{D}\}$). We construct a minimal perfect hash function mph that maps signatures in $\mathcal{D}_{\mathsf{rk}}$ to the first n integers. Lemma 3 guarantees $O(1)$ evaluation time by requiring $O(d)$ bits of space. As satellite data, the entry for the string S stores in $\log d + O(1)$ bits the index of the leaf in $\mathcal{PT}_r$ that corresponds to S reversed. Clearly, if S belongs to $\mathcal{D}$, $\mathsf{mph}(\mathsf{rk}(S))$ gives us in constant time the index of S reversed in $\mathcal{PT}_r$ while $\pi(\mathsf{mph}(\mathsf{rk}(S)))$ reports the index of S in $\mathcal{PT}$. It is worth noticing that the result of these operations are meaningless whenever S does not belong to $\mathcal{D}$.

The check of candidate strings requires a preprocessing phase shared among all the candidate strings. Firstly, we compute in $O(p)$ the Rabin-Karp signatures of all prefixes and suffixes of P. In this way, the signature of any candidate string S can be computed in constant time by appropriately combining two of those signatures (Lemma 2). Then, we identify a leaf pleaf in $\mathcal{PT}$ that shares the longest common prefix with P. Similarly, we identify a leaf sleaf in $\mathcal{PT}_r$ having the longest common prefix with P reversed. Given the properties of patricia tries and our succinct representation, identifying these two leaves cost $O(p)$ time.

The check for the single candidate string $S = P[1,i] \cdot c \cdot P[i+1,p]$ obtained by inserting symbol c in $(i+1)$th position is done as follows [6]. We compute in constant time the values $k = \pi(\mathsf{mph}(\mathsf{rk}(S)))$ and $k' = \mathsf{mph}(\mathsf{rk}(S))$. Then, we have to check that the candidate string S is equal to the string S_k in $\mathcal{D}$. Instead of comparing S and S_k symbol by symbol, we exploit the fact that S and S_k coincide if and only if the following three conditions are satisfied:

[6] Checks for other types of errors are done in a similar way.

- $\mathsf{lcp}(k, \mathsf{pleaf})$ is at least i;
- $\mathsf{lcp}_r(k', \mathsf{sleaf})$ is at least $p - i$;
- $(i + 1)$th symbol of S_k is equal to c.

Clearly, these three conditions are checkable in constant time. The $O(p)$ preprocessing time is amortized over the $O(p + occ)$ candidate strings.

Finding Top-k Strings. As we mentioned in the Introduction, our solution could be extended to support an additional operation which has interesting practical applications. Assume that each string S_i in $\mathcal{D}$ has assigned a score $c(S_i)$. For example, the score could establish the relative importance of any string with respect to the others. It is possible to extend our solution in order to support the extra operation $\mathsf{Top}(P[1, p], k)$ that reports the k highest scored strings in $\mathcal{D}$ having edit distance at most 1 with P. This operation is solved in $O(p + k \log k)$ time. We assume that values $c()$ are available for free. Notice that we can easily avoid this assumption by storing in $d \log d + O(d)$ bits the ranking of strings in $\mathcal{D}$ induced by $c()$.

We first present a simpler $O((p + k) \log k)$ time algorithm which is, then, modified in order to achieve the claimed time complexity. We said above that an arbitrary rank is assigned to tuples in $\mathcal{T}$ belonging the same group (namely, tuples of the form $\langle \mathsf{np}, i, \star, \mathsf{ns}, i' \rangle$ that differ just for the symbol $\star$). Instead, this algorithm requires that the assigned ranks respecting the order induced by $c()$. Namely, lower ranks are assigned to tuples corresponding to strings with higher values of $c()$. The searching algorithm is similar to the previous one. The main difference is in the order in which the factorizations of $P[1, p]$ are processed. The algorithm works in steps and keeps a heap. The role of the heap is that of keeping track of the top-k candidate strings seen so far. Each factorization is initially considered *active* and becomes *inactive* later in the execution. Once a factorization becomes inactive, it is no longer taken into consideration. Each factorization has also associated a score which is initially set to $+\infty$. At each step, we process the active factorization with the largest score. We query function $F()$ with the correct value of r for the current factorization. Let S be the candidate string identified by resorting to $F()$. If S does not belong to $\mathcal{D}$, the current factorization becomes inactive and we continue with the next factorization. Otherwise, we insert S into the heap with its score $c(S)$ and we decrease the score associated to the current factorization to $c(S)$. At each step we also check the number of string into the heap. If they are $k + 1$, we remove the string with the lowest score and we declare inactive the factorization that introduced that string.

Notice that, apart from the first k steps, in each step a factorization becomes inactive. Since there are $O(p)$ factorizations, our algorithm performs at most $O(p + k)$ insertions into a heap containing at most k strings. Thus, the claimed time complexity easily follows. The improvement is obtained by observing that most of the time (i.e., $O(p \log k)$) is spent in inserting the first string of each factorization into the heap. This is no longer necessary if we use the following strategy. We first collect the first string of each factorization together with its score and we apply the classical linear time selection algorithm to identify the k-th smallest score. This step costs $O(p)$ time. We immediately declare inactive the $p-k$ factorizations whose strings have a smaller score. We insert the remaining k strings into the heap and we use the previous algorithm to complete the task. The latter step costs now $O(k \log k)$ time, since we have just k active factorizations.

5 A More Compressed Solution

The factor 2 multiplying the H_k term in space bound of Theorem 2 may be annoying in some scenario. In this section we provide a solution which is able to overcome this limitation at the cost of (slightly) increasing the query time. Formally, we prove the following theorem.

Theorem 3. *Given a set of strings $\mathcal{D} = \{S_1, S_2, \ldots, S_d\}$ of d strings of total length n drawn from an alphabet Σ of size σ, there exists an index requiring $nH_k + n \cdot o(\log \sigma)$ bits of space for a fixed $k = o(\log_\sigma n)$ that, given any pattern $P[1,p]$ reports all the occ strings in $\mathcal{D}$ having edit distance at most one with P in:*

1. *$O(p(min(p, \log_\sigma n \log\log n)) + occ)$ worst-case time when $\sigma = \log^c n$ for some constant c.*
2. *$O(p \log\log \sigma + occ)$ worst-case time when $\sigma = \omega(\log^c n)$ for any constant c.*

This solution uses a completely different approach with respect to our previous one. Indeed, it resorts to a collection of compressed permuterm indexes [10] built on the dictionary $\mathcal{D}$. More precisely, we divide strings in $\mathcal{D}$ based on their length. Let $\mathcal{D}_\ell$ denote the set of strings in $\mathcal{D}$ of length ℓ. A compressed permuterm index R_ℓ is built for each set $\mathcal{D}_\ell$.[7]

In [10] it is shown how design a Burrows-Wheeler Transform [5] (BWT) based index for a dictionary of strings. Among the other types of queries, the index solves efficiently the so-called PrefixSuffix query that, given a prefix P and a suffix S, identifies all the strings in the dictionary having P as prefix and S as suffix. In our solution we are interested in this type of query which is solved by using a slightly different variant of the compressed permuterm index. The main difference is the sorting strategy used to obtain the underlying Burrows-Wheeler Transform (BWT) [5]. In [10] a text is obtained by concatenating the strings of the dictionary by using, as separator, a special symbol # not appearing in Σ. Then, all the suffixes of this text are sorted lexicographically to obtain the rows of the Burrows-Wheeler matrix. In our variant we first append the symbol # at the end of each string, then we construct the BWT matrix by sorting lexicographically all the cyclic rotations of the strings in the set. This different way to proceed guarantees us that symbols in any row belong to the same string.

[7] We notice that the number of distinct lengths and, thus, compressed permuterm indexes is $O(\sqrt{n})$.

This fact turns out to be useful below when we will define parent and depth operations. The searching algorithm presented in [10] does not change[8].

Given a pattern $P[1,p]$, we query only three compressed permuterm indexes: R_{p-1} for deletions, R_p for substitutions and R_{p+1} for insertions. In the following we will only describe the solution for insertion. Deletion and substitution are solved in a similar way. The basic idea behind our searching algorithm is the following. For each cyclic rotation $P_i = P[i,p]\#P[1,i-1]$ of P, we use the compressed permuterm index R_{p+1} in order to identify the range of rows of Burrows-Wheeler Transform [5] which are prefixed by P_i, if any (see [10] and references therein for more details). We observe that having that range $[r,l]$ suffices for identifying the strings in $\mathcal{D}$ obtained by inserting a symbol in ith position on P. These symbols are, indeed, the ones contained in $\mathsf{BWT}_{p+1}[l,r]$, where BWT_{p+1} is the Burrows-Wheeler Transform of set $\mathcal{D}_{p+1}$. However, we cannot compute all these ranges in a naïve way (i.e., searching each P_i separately), since it would cost at least p^2 time.

A faster solution requires to augment the compressed permuterm index with a data structure that supports the two operations: parent and depth. Consider the conceptual compact trie build on top of rows of BWT_{p+1}. Given a range $[l,r]$, let u be the node of the above trie corresponding to range $[l,r]$. The two operations are defined as follows:

1. $\mathsf{parent}(u)$ returns the range $[l',r']$ corresponding to the parent of the node u;
2. $\mathsf{depth}(u)$ returns the length of the locus of node u.

Using the solution presented in [15], we are able to support both these operations in $O(\log_\sigma \hat{n} \log\log \hat{n})$ time by requiring $O(\hat{n}\frac{\log \sigma}{\log\log \hat{n}})$ bits of additional space, where $\hat{n}$ is the total size of the indexed dictionary.

Our solution works in two phases. In the first phase, it identifies the range of rows of BWT_{p+1} sharing the longest common prefix with $P_0 = \#P[1,p]$. This is done by using the following strategy. We search P_0 backwards. At any step i, we keep the following invariant: $[l_i, r_i]$ is the largest range of rows of BWT_{p+1} which are prefixed by the longest prefix of $P_0[p-i,p+1]$. We also keep a counter ℓ that tells us the length of this prefix. Notice that it may happen that a backwards step from $[l_i,r_i]$ with the next symbol $P[p-i-1]$ returns an empty range. In this case, we repeatedly enlarge the range $[l_i,r_i]$ via parent operations until the subsequent backwards step is successful. The value of ℓ is kept updated by increasing it by one after every successful backwards step or by setting it equal to the value returned by depth after every call to parent.

Similarly, the second phase matches suffixes of P backwards. The main difference is given by the fact that the starting range $[l_1,r_1]$ and value of ℓ are the ones computed in the previous phase. In each step, we claim that we have identified the range of rows prefixed by some $P_i = P[i,p]\#P[1,i-1]$, for the appropriate i, as soon as the value of ℓ reaches $p+1$. The overall time complexity of these two phases is $O(p \log_\sigma \hat{n} \log\log \hat{n})$, since we have at most $2p$ calls to parent and depth.

[8] Actually, the different construction of the Burrows-Wheeler Transform defined here was already implicitly in use in [10] and simulated at query time by means of function jump2end (see [10] for more details).

The discussion above provides a proof of Point 1 of Theorem 3. Observe that the time complexity of the above solution is dominated by the time spent in performing parent and depth operations.

Point 2 of Theorem 3 is obtained by showing that, for sufficiently large alphabet (i.e., $\sigma = \omega(\log^c n)$, for any constant c), faster implementations of these operations (i.e., $O(\log\log\sigma)$ time) are possible. We can, indeed, improve the time complexity of the solution above if we are allowed to use more space. More precisely, using more space, we can improve the time of parent (for all cases) and depth (for the case of large depths):

1. The operation parent can be supported in constant time using $O(n)$ additional bits of space. This is feasible by using the Sadakane's compressed suffix tree [16].
2. The operation depth can be supported in constant time using $O(n \log t)$ bits of space when the string depth is at least $p - t$ for some parameter t. For this aim, we can just store a table Δ which stores $\log(t + 1)$ bits per node. These bits will store a special value whenever the depth is less than $p - t$, otherwise, we store the difference between the depth and p.

Now that we have a constant time parent operation, the depth operation remains as the only bottleneck for achieving faster query time. Assume that the compressed suffix tree supports the depth operation in time t. We first notice that a given range obtained after $t' < t$ backwards steps can correspond to a P_i if and only if the depth of the node obtained after the last parent operation was precisely $p - t'$. This condition can be checked directly by probing the table Δ. If this is not the case, we adopt a lazy strategy. Instead of computing a depth after each parent, we safely wait until we performed t backwards steps after the last parent operation. The $O(t)$ time required by depth is amortized on the cost of these (at least) t backwards steps. Point 2 of Theorem 3 is proven by setting $t = O(\log_\sigma n \log\log n)$ and by observing that the backwards steps become the dominant cost (i.e., $O(p \log\log\sigma)$).

6 Conclusion

In this paper we described two different compressed solutions for the look-up with Edit distance one in a dictionary of strings. The first solution requires $2H_k(S) + n \cdot o(\log\sigma) + 2d \log d$ bits of space for a fixed $k = o(\log_\sigma n)$. It is able to solve queries in (almost optimal) $O(|P| + occ)$ time where occ is the number of strings in the dictionary having edit distance at most one with the query pattern P. The second solution further improves this space complexity which is reduced to $nH_k + n \cdot o(\log\sigma)$ bits. However, the time complexity grows to $O(|P| \log_\sigma n \log\log n + occ)$ or $O(|P| \log\log\sigma + occ)$ depending on the alphabet size. Interestingly enough, the two solutions solve the problem at hand by resorting to two different approaches: the former is based on (perfect) hashing while the latter is based on the compressed permuterm index.

An interesting open problem asks to design an index that obtains simultaneously the time complexity of the former solution and the space complexity of the second one. Furthermore, it is still open the question regarding the possibility of designing a solution that solves the problem in $O(|P| \cdot \log \sigma / w + occ)$ time, where w is the size of a word machine. At the moment, there does not exist any solution achieving such a time complexity, even non compressed one.

Finally, building efficient dictionaries for edit distance d larger than 1 is still an open problem. However, the approaches we used in our two solutions are not easily extendible to efficiently solve query for higher edit distance. Indeed, we could just solve a query in $O(\sigma^{d-1}|P|^d + occ)$ time for edit distance d by resorting to the standard dynamic programming approach.

References

1. Belazzougui, D.: Faster and Space-Optimal Edit Distance "1" Dictionary. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009 Lille. LNCS, vol. 5577, pp. 154–167. Springer, Heidelberg (2009)
2. Belazzougui, D., Navarro, G.: Alphabet-Independent Compressed Text Indexing. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 748–759. Springer, Heidelberg (2011)
3. Brodal, G.S., Gąsieniec, L.: Approximate Dictionary Queries. In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 65–74. Springer, Heidelberg (1996)
4. Brodal, G.S., Srinivasan, V.: Improved bounds for dictionary look-up with one error. Information Processing Letters 75(1-2), 57–59 (2000)
5. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
6. Dietzfelbinger, M., Gil, J., Matias, Y., Pippenger, N.: Polynomial Hash Functions are Reliable (Extended abstract). In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 235–246. Springer, Heidelberg (1992)
7. Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM 21, 246–260 (1974)
8. Fano, R.M.: On the number of bits required to implement an associative memory. Memorandum 61, Computer Structures Group, Project MAC (1971)
9. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theor. Comput. Sci. 372(1), 115–121 (2007)
10. Ferragina, P., Venturini, R.: The compressed permuterm index. ACM Transactions on Algorithms 7(1), 10 (2010)
11. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. 40(2), 465–492 (2011)
12. Hagerup, T., Tholey, T.: Efficient Minimal Perfect Hashing in Nearly Minimal Space. In: Ferreira, A., Reichel, H. (eds.) STACS 2001. LNCS, vol. 2010, pp. 317–326. Springer, Heidelberg (2001)
13. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2), 249–260 (1987)
14. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001)
15. Russo, L.M.S., Navarro, G., Oliveira, A.L.: Fully compressed suffix trees. ACM Transactions on Algorithms 7(4), 53 (2011)
16. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. 41(4), 589–607 (2007)

Time-Space Trade-Offs for Longest Common Extensions*

Philip Bille[1], Inge Li Gørtz[1], Benjamin Sach[2], and Hjalte Wedel Vildhøj[1]

[1] Technical University of Denmark, DTU Informatics
{phbi,ilg,hwvi}@imm.dtu.dk

[2] University of Warwick, Department of Computer Science
sach@dcs.warwick.ac.uk

Abstract. We revisit the longest common extension (LCE) problem, that is, preprocess a string T into a compact data structure that supports fast LCE queries. An LCE query takes a pair (i, j) of indices in T and returns the length of the longest common prefix of the suffixes of T starting at positions i and j. We study the time-space trade-offs for the problem, that is, the space used for the data structure vs. the worst-case time for answering an LCE query. Let n be the length of T. Given a parameter τ, $1 \leq \tau \leq n$, we show how to achieve either $O(n/\sqrt{\tau})$ space and $O(\tau)$ query time, or $O(n/\tau)$ space and $O(\tau \log(|\text{LCE}(i, j)|/\tau))$ query time, where $|\text{LCE}(i, j)|$ denotes the length of the LCE returned by the query. These bounds provide the first smooth trade-offs for the LCE problem and almost match the previously known bounds at the extremes when $\tau = 1$ or $\tau = n$. We apply the result to obtain improved bounds for several applications where the LCE problem is the computational bottleneck, including approximate string matching and computing palindromes. Finally, we also present an efficient technique to reduce LCE queries on two strings to one string.

1 Introduction

Given a string T, the *longest common extension* of suffix i and j, denoted $\text{LCE}(i, j)$, is the length of the longest common prefix of the suffixes of T starting at position i and j. The *longest common extension problem* (LCE problem) is to preprocess T into a compact data structure supporting fast longest common extension queries.

The LCE problem is a basic primitive that appears as a subproblem in a wide range of string matching problems such as approximate string matching and its variations [3, 6, 18, 20, 25], computing exact or approximate tandem repeats [10, 19, 22], and computing palindromes. In many of the applications, the LCE problem is the computational bottleneck.

In this paper we study the time-space trade-offs for the LCE problem, that is, the space used by the preprocessed data structure vs. the worst-case time used by

* This work was partly supported by EPSRC.

LCE queries. We assume that the input string is given in read-only memory and is not counted in the space complexity. There are essentially only two time-space trade-offs known: At one extreme we can store a suffix tree combined with an efficient nearest common ancestor (NCA) data structure [12] (other combinations of $O(n)$ space data structures for the string can also be used to achieve this bound). This solution uses $O(n)$ space and supports LCE queries in $O(1)$ time. At the other extreme we do not store any data structure and instead answer queries simply by comparing characters from left-to-right in T. This solution uses $O(1)$ space and answers an $\mathrm{LCE}(i,j)$ query in $O(|\mathrm{LCE}(i,j)|) = O(n)$ time. Using succinct data structures the space for the first solution can be slightly improved to $O(n)$ *bits* (see e.g. [8]). The second solution was recently shown to be very practical [13].

1.1 Our Results

We show the following main result.

Theorem 1. *For a string T of length n, and any parameter τ, $1 \le \tau \le n$, we can solve the longest common extension problem*

(i) in $O\left(\frac{n}{\sqrt{\tau}}\right)$ space, $O(\tau)$ query time, and $O(\frac{n^2}{\sqrt{\tau}})$ preprocessing time, or in

(ii) in $O\left(\frac{n}{\tau}\right)$ space, $O\left(\tau \log\left(\frac{|\mathrm{LCE}(i,j)|}{\tau}\right)\right)$ query time, and $O(n \log n)$ preprocessing time. The preprocessing time bound is achieved with high probability, that is, with probability at least $1 - 1/n^c$ for any constant c. All other bounds are worst case.

Hence, we provide a smooth time-space trade-off that allows several new and non-trivial bounds. For instance, with $\tau = \sqrt{n}$ Theorem 1(i), gives a solution using $O(n^{3/4})$ space and $O(\sqrt{n})$ time. If we allow randomisation, we can use Theorem 1(ii) to further reduce the space to $O(\sqrt{n})$ while using query time $O(\sqrt{n}\log(|\mathrm{LCE}(i,j)|/\sqrt{n})) = O(\sqrt{n}\log n)$. Note that at both extremes of the trade-off ($\tau = 1$ or $\tau = n$) we almost match the previously known bounds.

Furthermore, we also consider LCE queries between two strings, i.e. the pair of indices to an LCE query is from different strings. We present a general result that reduces the query on two strings to a single one of them. When one of the strings is significantly smaller than the other, we can combine this reduction with Theorem 1 to obtain even better time-space trade-offs.

1.2 Techniques

The high-level idea in Theorem 1 is to combine and balance out the two extreme solutions for the LCE problem. For Theorem 1(i) we use *difference covers* to sample a set of suffixes of T of size $O(n/\sqrt{\tau})$. We store a compact trie combined with an NCA data structure for this sample using $O(n/\sqrt{\tau})$ space. To answer an LCE query we compare characters from T until we get a mismatch or reach a pair of sampled suffixes, which we then immediately compute the answer for. By the properties of difference covers we compare at most $O(\tau)$ characters before reaching a pair of sampled suffixes. Similar ideas have previously been used to achieve trade-offs for suffix array and LCP array construction [15,26].

For Theorem 1(ii) we show how to use Rabin-Karp fingerprinting [16] instead of difference covers to reduce the space further. We show how to store a sample of $O(n/\tau)$ fingerprints, and how to use it to answer LCE queries using doubling search combined with directly comparing characters. This leads to the output-sensitive $O(\tau \log(|\mathrm{LCE}(i,j)|/\tau))$ query time. We reduce space compared to Theorem 1 by computing fingerprints on-the-fly as we need them. Initially, we give a Monte-Carlo style randomised data structure that may answer queries incorrectly (see Theorem 5). However, this solution uses only $O(n)$ preprocessing time and is therefore of independent interest in applications that can tolerate errors. To get the error-free Las-Vegas style bound of Theorem 1(ii) we need to verify the fingerprints we compute are collision free; i.e. two fingerprints are equal iff the corresponding substrings of T are equal. The main challenge is to do this in only $O(n \log n)$ time. We achieve this by showing how to efficiently verify fingerprints of composed samples which we have already verified, and by developing a search strategy that reduces the fingerprints we need to consider.

Finally, the reduction for LCE on two strings to a single string is based on a simple and compact encoding of the larger string using the smaller string. The encoding could be of independent interest in related problems, where we want to take advantage of different length input strings.

1.3 Applications

With Theorem 1 we immediately obtain new results for problems based on LCE queries. We review some the most important below.

Approximate String Matching. Given strings P and T and an error threshold k, the *approximate string matching problem* is to report all ending positions of substrings of T whose *edit distance* to P is at most k. The edit distance between two strings is the minimum number of insertions, deletions, and substitutions needed to convert one string to the other. Let m and n be the lengths of P and T. The Landau-Vishkin algorithm [20] solves approximate string matching using $O(nk)$ LCE queries on P and substrings of T of length $O(m)$. Using the standard linear space and constant time LCE data structure, this leads to a solution using $O(nk)$ time and $O(m)$ space (the $O(m)$ space bound follows by the standard trick of splitting T into overlapping pieces of size $O(m)$). If we plug in the results from Theorem 1 we immediately obtain the following result.

Theorem 2. *Given strings P and T of lengths m and n, respectively, and a parameter τ, $1 \le \tau \le m$, we can solve approximate string matching*

(i) in $O\left(\frac{m}{\sqrt{\tau}}\right)$ space and $O(nk \cdot \tau + \frac{nm}{\sqrt{\tau}})$ time, or
(ii) in $O\left(\frac{m}{\tau}\right)$ space and $O(nk \cdot \tau \log m)$ time with high probability.

To the best of our knowledge these are the first non-trivial bounds for approximate string matching using $o(m)$ space.

Palindromes. Given a string T the *palindrome problem* is to report the set of all *maximal palindromes* in T. A substring $T[i \dots j]$ is a maximal palindrome iff $T[i \dots j] = T[i \dots j]^R$ and $T[i-1 \dots j+1] \neq T[i-1 \dots j+1]^R$. Here $T[i \dots j]^R$ denotes the reverse of $T[i \dots j]$. Any palindrome in T occurs in the middle of a maximal palindrome, and thus the set of maximal palindromes compactly represents all palindromes in T. The palindrome problem appears in a diverse range of applications, see e.g. [2,4,9,14,17,21,24].

We can trivially solve the problem in $O(n^2)$ time and $O(1)$ space by a linear search at each position in T to find the maximal palindrome. With LCE queries we can immediately speed up this search. Using the standard $O(n)$ space and constant time solution to the LCE problem this immediately implies an algorithm for the palindrome problem that uses $O(n)$ time and space (this bound can also be achieved without LCE queries [23]). Using Theorem 1 we immediately obtain the following result.

Theorem 3. *Given a string of length n and a parameter τ, $1 \leq \tau \leq n$, we can solve the palindrome problem*

(i) in $O\left(\frac{n}{\sqrt{\tau}}\right)$ space and $O\left(\frac{n^2}{\sqrt{\tau}} + n\tau\right)$ time.
(ii) in $O\left(\frac{n}{\tau}\right)$ space and $O(n \cdot \tau \log n)$ time with high probability.

For $\tau = \omega(1)$, these are the first sublinear space bounds using $o(n^2)$ time. Similarly, we can substitute our LCP data structures in the LCP-based variants of palindrome problems such as *complemented palindromes*, *approximate palindromes*, or *gapped palindromes*, see e.g. [17].

Tandem Repeats. Given a string T, the *tandem repeats problem* is to report all squares, i.e. consecutive repeated substrings in T. Main and Lorentz [22] gave a simple solution for this problem based on LCP queries that achieves $O(n)$ space and $O(n \log n + \text{occ})$ time, where occ is the number of tandem repeats in T. Using different techniques Gąsieniecs et al. [11] gave a solution using $O(1)$ space and $O(n \log n + \text{occ})$ time. Using Theorem 1 we immediately obtain the following result.

Theorem 4. *Given a string of length n and parameter τ, $1 \leq \tau \leq n$, we can solve the tandem repeats problem*

(i) in $O\left(\frac{n}{\sqrt{\tau}}\right)$ space and $O\left(\frac{n^2}{\sqrt{\tau}} + n\tau \cdot \log n + occ\right)$ time.
(ii) in $O\left(\frac{n}{\tau}\right)$ space and $O\left(n\tau \cdot \log^2 n + \text{occ}\right)$ time with high probability.

While this does not improve the result by Gąsieniecs et al. it provides a simple LCP-based solution. Furthermore, our result generalizes to the approximate versions of the tandem repeats problem, which also have solutions based on LCP queries [19].

2 The Deterministic Data Structure

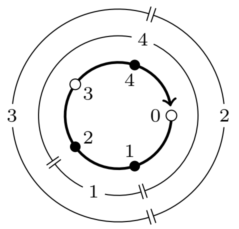
We now show Theorem 1(i). Our deterministic time-space trade-off is based on sampling suffixes using *difference covers*.

2.1 Difference Covers

A *difference cover modulo* τ is a set of integers $D \subseteq \{0, 1, \ldots, \tau - 1\}$ such that for any distance $d \in \{0, 1, \ldots, \tau - 1\}$, D contains two elements separated by distance d modulo τ (see example 1).

Example 1. The set $D = \{1, 2, 4\}$ is a difference cover modulo 5.

d	0	1	2	3	4
i, j	1, 1	2, 1	1, 4	4, 1	1, 2



A difference cover D can cover at most $|D|^2$ differences, and hence D must have size at least $\sqrt{\tau}$. We can also efficiently compute a difference cover within a constant factor of this bound.

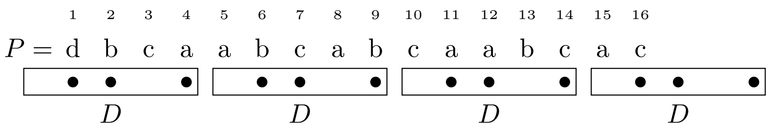
Lemma 1 (Colbourn and Ling [5]). *For any τ, a difference cover modulo τ of size at most $\sqrt{1.5\tau} + 6$ can be computed in $O(\sqrt{\tau})$ time.*

2.2 The Data Structure

Let T be a string of length n and let τ, $1 \leq \tau \leq n$, be a parameter. Our data structure consists of the compact trie of a sampled set of suffixes from T combined with a NCA data structure. The sampled set of suffixes $\mathcal{S}$ is the set of suffixes obtained by overlaying a difference cover on T with period τ, that is,

$$\mathcal{S} = \{i \mid 1 \leq i \leq n \ \wedge \ i \bmod \tau \in D\} \ .$$

Example 2. Consider the string T = dbcaabcabcaabcac. As shown below, using the difference cover from Example 1, we obtain the suffix sample $\mathcal{S} = \{1, 2, 4, 6, 7, 9, 11, 12, 14, 16\}$.



By Lemma 1 the size of $\mathcal{S}$ is $O(n/\sqrt{\tau})$. Hence the compact trie and the NCA data structures use $O(n/\sqrt{\tau})$ space. We construct the data structure in $O(n^2/\sqrt{\tau})$ time by inserting each of the $O(n/\sqrt{\tau})$ sampled suffixes in $O(n)$ time.

To answer an LCE(i,j) query we explicitly compare characters starting from i and j until we either get a mismatch or we encounter a pair of sampled suffixes. If we get a mismatch we simply report the length of the LCE. Otherwise, we do a NCA query on the sampled suffixes to compute the LCE. Since the distance to a pair of sampled suffixes is at most τ the total time to answer a query is $O(\tau)$. This concludes the proof of Theorem 1(i).

3 The Monte-Carlo Data Structure

We now show Theorem 5 below which is an intermediate step towards proving Theorem 1(ii) but is also of independent interest, providing a Monte-Carlo time-space trade-off. The technique is based on sampling suffixes using *Rabin-Karp fingerprints*. These fingerprints will be used to speed up queries with large LCE$_P$ values while queries with small LCE$_P$ values will be handled naively.

Theorem 5. *For a string T of length n, and any parameter τ, $1 \leq \tau \leq n$, we can solve the* LCE *problem in $O\left(\frac{n}{\tau}\right)$ space, $O\left(\tau \log\left(\frac{|\mathrm{LCE}(i,j)|}{\tau}\right)\right)$ query time, and $O(n)$ preprocessing. The solution is randomised (Monte-Carlo); with high probability, all queries are answered correctly.*

3.1 Rabin-Karp fingerprints

Rabin-Karp fingerprints are defined as follows. Let $2n^{c+4} < p \leq 4n^{c+4}$ be some prime and choose $b \in \mathbb{Z}_p$ uniformly at random. Let S be any substring of T, the fingerprint $\phi(S)$ is given by,

$$\phi(S) = \sum_{k=1}^{|S|} S[k]b^k \bmod p\,.$$

Lemma 2 gives a crucial property of these fingerprints (see e.g. [16] for a proof). That is with high probability we can determine whether any two substrings of T match in constant time by comparing their fingerprints.

Lemma 2. *Let ϕ be a fingerprinting function picked uniformly at random (as described above). With high probability,*

$$\begin{aligned} \phi(T[i \ldots i+\alpha-1]) &= \phi(T[j \ldots j+\alpha-1]) \\ \textit{iff}\ T[i \ldots i+\alpha-1] &= \quad T[j \ldots j+\alpha-1] \quad \textit{for all } i,j,\alpha. \end{aligned} \tag{1}$$

3.2 The Data Structure

The data structure consists of the fingerprint, ϕ_k, of each suffix of the form $T[k\tau \ldots n]$ for $0 < k < n/\tau$, i.e. $\phi_k = \phi(T[k\tau \ldots n])$. Note that there are $O(n/\tau)$

such suffixes and the starting points of two consecutive suffix are τ characters apart. Since each fingerprint uses constant space the space usage of the data structure is $O(n/\tau)$. The n/τ fingerprints can be computed in left-to-right order by a single scan of T in $O(n)$ time.

Queries. The key property we use to answer a query is given by Lemma 3.

Lemma 3. *The fingerprint $\phi(T[i \dots i+\alpha-1])$ of any substring $T[i \dots i+\alpha-1]$ can be constructed in $O(\tau)$ time. If i, α are divisible by τ, the time becomes $O(1)$.*

Proof. Let $k_1 = \lceil i/\tau \rceil$ and $k_2 = \lceil (i+\alpha)/\tau \rceil$ and observe that we have ϕ_{k_1} and ϕ_{k_2} stored. By the definition of ϕ, we can compute $\phi(T[k_1\tau \dots k_2\tau - 1]) = \phi_{k_1} - \phi_{k_2} \cdot b^{(k_2-k_1)\tau} \bmod p$ in $O(1)$ time. If $i, \alpha = 0 \bmod \tau$ then $k_1\tau = i$ and $k_2\tau = i+\alpha$ and we are done. Otherwise, similarly we can then convert $\phi(T[k_1\tau \dots k_2\tau-1])$ into $\phi(T[k_1\tau-1 \dots k_2\tau-2])$ in $O(1)$ time by inspecting $T[k_1\tau-1]$ and $T[k_2\tau-1]$. By repeating this final step we obtain $T[i \dots i+\alpha-1]$ in $O(\tau)$ time.

We now describe how to perform a query by using fingerprints to compare substrings. We define $\phi_k^\ell = \phi(T[k\tau \dots (k+2^\ell)\tau - 1])$ which we can compute in $O(1)$ time for any k, ℓ by Lemma 3.

First consider the problem of answering a query of the form $\mathrm{LCE}(i\tau, j\tau)$. Find the largest ℓ such that $\phi_i^\ell = \phi_j^\ell$. When the correct ℓ is found convert the query into a new query $\mathrm{LCE}((i+2^\ell)\tau, (j+2^\ell)\tau)$ and repeat. If no such ℓ exists, explicitly compare $T[i\tau \dots (i+1)\tau - 1]$ and $T[j\tau \dots (j+1)\tau - 1]$ one character at a time until a mismatch is found. Since no false negatives can occur when comparing fingerprints, such a mismatch exists. Let ℓ_0 be the value of ℓ obtained for the initial query, $\mathrm{LCE}(i\tau, j\tau)$, and ℓ_q the value obtained during the q-th recursion. For the initial query, we search for ℓ_0 in increasing order, starting with $\ell_0 = 0$. After recursing, we search for ℓ_q in descending order, starting with ℓ_{q-1}. By the maximality of ℓ_{q-1}, we find the correct ℓ_q. Summing over all recursions we have $O(\ell_0)$ total searching time and $O(\tau)$ time scanning T. The desired query time follows from observing that by the maximality of ℓ_0, we have that $O(\tau + \ell_0) = O(\tau + \log(|\mathrm{LCE}|/\tau))$.

Now consider the problem of answering a query of the form $\mathrm{LCE}(i\tau, j\tau + \gamma)$ where $0 < \gamma < \tau$. By Lemma 3 we can obtain the fingerprint of any substring in $O(\tau)$ time. This allows us to use a similar approach to the first case. We find the largest ℓ such that $\phi(T[j\tau + \gamma \dots (j+2^\ell)\tau + \gamma - 1]) = \phi_i^\ell$ and convert the current query into a new query, $\mathrm{LCE}((i+2^\ell)\tau, (j+2^\ell)\tau + \gamma)$. As we have to apply Lemma 3 before every comparison, we obtain a total complexity of $O(\tau \log(|\mathrm{LCE}|/\tau))$.

The general case can be reduced to one of the first two cases by scanning $O(\tau)$ characters in T. By Lemma 2, all fingerprint comparisons are correct with high probability and the result follows.

4 The Las-Vegas Data Structure

We now show Theorem 1(ii). The important observation is that when we compare the fingerprints of two strings during a query in Section 3, one of them is of the form $T[j\tau \dots j\tau + \tau \cdot 2^\ell - 1]$ for some ℓ, j. Consequently, to ensure all queries are correctly computed, it suffices that ϕ is τ-*good*:

Definition 1. *A fingerprinting function, ϕ is τ-*good *on T iff*

$$\begin{aligned} &\phi(T[j\tau \dots j\tau + \tau \cdot 2^\ell - 1]) = \phi(T[i \dots i + \tau \cdot 2^\ell - 1]) \\ &\textit{iff}\ T[j\tau \dots j\tau + \tau \cdot 2^\ell - 1] \ = \ T[i \dots i + \tau \cdot 2^\ell - 1] \quad \textit{for all } (i, j, \ell). \end{aligned} \tag{2}$$

In this section we give an algorithm which decides whether a given ϕ is τ-good on string T. The algorithm uses $O(n/\tau)$ space and takes $O(n \log n)$ time with high probability. By Lemma 2, a uniformly chosen ϕ is τ-good with high probability and therefore (by repetition) we can generate such a ϕ in the same time/space bounds. For brevity we assume that n and τ are powers-of-two.

4.1 The Algorithm

We begin by giving a brief overview of Algorithm 1. For each value of ℓ in ascending order (the outermost loop), Algorithm 1 checks (2) for all i, j. For some outermost loop iteration ℓ, the algorithm inserts the fingerprint of each block-aligned substring into a dynamic perfect dictionary, D_ℓ (lines 3-9). A substring is block-aligned if it is of the form, $T[j\tau \dots (j + 2^\ell)\tau - 1]$ for some j (and block-unaligned otherwise). If more than one block-aligned substring has the same fingerprint, we insert only the left-most as a representative. For the first iteration, $\ell = 0$ we also build an Aho-Corasick automaton [1], denoted AC, with a pattern dictionary containing every block-aligned substring.

The second stage (lines 12-21) uses a sliding window technique, checking each time we slide whether the fingerprint of the current $(2^\ell\tau)$-length substring occurs in the dynamic dictionary, D_ℓ. If so we check whether the corresponding substrings match (if not a collision has been revealed and we abort). For $\ell > 0$, we use the fact that (2) holds for all i, j with $\ell - 1$ (otherwise, Algorithm 1 would have already aborted) to perform the check in constant time (line 18). I.e. if there is a collision it will be revealed by comparing the fingerprints of the left-half ($L'_i \neq L_k$) or right-half ($R'_i \neq R_k$) of the underlying strings. For $\ell = 0$, the check is performed using the AC automaton (lines 20-21). We achieve this by feeding T one character at a time into the AC. By inspecting the state of the AC we can decide whether the current τ-length substring of T matches any block-aligned substring.

Correctness. We first consider all points at which Algorithm 1 may abort. First observe that if line 21 causes an abort then (2) is violated for $(i, k, 0)$. Second, if line 18 causes an abort either $L'_i \neq L_k$ or $R'_i \neq R_k$. By the definition of ϕ, in either case, this implies that $T[i \dots i + \tau \cdot 2^\ell - 1] \neq T[k\tau \dots k\tau + 2^\ell\tau - 1]$. By line 16, we have that $f'_i = f_k$ and therefore (2) is violated for (i, k, ℓ). Thus, Algorithm 1 does not abort if ϕ is τ-good.

Algorithm 1. Verifying a fingerprinting function, ϕ on string T

```
1: // $AC$ is an Aho-Corasick automaton and each $D_\ell$ is a dynamic dictionary
2: for $\ell = 0 \dots \log_2(n/\tau)$ do
3:    // Insert all distinct block-aligned substring fingerprints into $D_\ell$
4:    for $j = 1 \dots n/\tau - 2^\ell$ do
5:       $f_j \leftarrow \phi(T[j\tau \dots (j+2^\ell)\tau - 1])$
6:       $L_j \leftarrow \phi(T[j\tau \dots (j+2^{\ell-1})\tau - 1])$, $R_j \leftarrow \phi(T[(j+2^{\ell-1})\tau \dots (j+2^\ell)\tau - 1])$
7:       if $\nexists (f_k, L_k, R_k, k) \in D_\ell$ such that $f_j = f_k$ then
8:          Insert $(f_j, L_j, R_j, j)$ into $D_\ell$ indexed by $f_j$
9:          if $\ell = 0$ then Insert $T[j\tau \dots (j+1)\tau - 1]$ into $AC$ dictionary
10:   // Check for collisions between any block-aligned and unaligned substrings
11:   if $\ell = 0$ then Feed $T[1 \dots \tau - 1]$ into $AC$
12:   for $i = 1 \dots n - \tau \cdot 2^\ell + 1$ do
13:      $f'_i \leftarrow \phi(T[i \dots i + \tau \cdot 2^\ell - 1])$
14:      $L'_i \leftarrow \phi(T[i \dots i + \tau \cdot 2^{\ell-1} - 1])$, $R'_i \leftarrow \phi(T[(i+2^{\ell-1})\tau \dots i + \tau \cdot 2^\ell - 1])$
15:      if $\ell = 0$ then Feed $T[i+\tau-1]$ into $AC$ // $AC$ now points at $T[i \dots i+\tau-1]$
16:      if $\exists (f_k, L_k, R_k, k) \in D_\ell$ such that $f'_i = f_k$ then
17:         if $\ell > 0$ then
18:            if $(L'_i \neq L_k$ or $R'_i \neq R_k)$ then abort
19:         else
20:            Compare $T[i \dots i+\tau-1]$ to $T[k\tau \dots (k+1)\tau - 1]$ by inspecting $AC$
21:            if $T[i \dots i+\tau-1] \neq T[k\tau \dots (k+1)\tau - 1]$ then abort
```

It remains to show that Algorithm 1 always aborts if ϕ is not τ-good. Consider the total ordering on triples (i, j, ℓ) obtained by stably sorting (non-decreasing) by ℓ then j then i. E.g. $(1,3,1) < (3,2,3) < (2,5,3) < (4,5,3)$. Let (i^*, j^*, ℓ^*) be the (unique) smallest triple under this order which violates (2). We first argue that $(f_{j^*}, L_{j^*}, R_{j^*}, j^*)$ will be inserted into D_{ℓ^*} (and AC if $\ell^* = 0$). For a contradiction assume that when Algorithm 1 checks for f_{j^*} in D_{ℓ^*} (line 7, with $j = j^*, \ell = \ell^*$) we find that some $f_k = f_{j^*}$ already exists in D_{ℓ^*}, implying that $k < j^*$. If $T[j^*\tau \dots j^*\tau + \tau 2^\ell - 1] \neq T[k\tau \dots k\tau + \tau 2^\ell - 1]$ then $(j^*\tau, k, \ell^*)$ violates (2). Otherwise, (i^*, k, ℓ^*) violates (2). In either case this contradicts the minimality of (i^*, j^*, ℓ^*) under the given order.

We now consider iteration $i = i^*$ of the second inner loop (when $\ell = \ell^*$). We have shown that $(f_{j^*}, L_{j^*}, R_{j^*}, j^*) \in D_{\ell^*}$ and we have that $f'_{i^*} = f_{j^*}$ (so $k = j^*$) but the underlying strings are not equal. If $\ell = 0$ then we also have that $T[j^*\tau \dots (j^*+1)\tau - 1]$ is in the AC dictionary. Therefore inspecting the current AC state, will cause an abort (lines 20-21). If $\ell > 0$ then as (i^*, j^*, ℓ^*) is minimal, either $L'_{i^*} \neq L_{j^*}$ or $R'_{i^*} \neq R_{j^*}$ which again causes an abort (line 18), concluding the correctness.

Time-Space Complexity. We begin by upper bounding the space used and the time taken to performs all dictionary operations on D_ℓ for any ℓ. First observe that there are at most $O(n/\tau)$ insertions (line 8) and at most $O(n)$ look-up operations (lines 7,16). We choose the dictionary data structure employed based on the relationship between n and τ. If $\tau > \sqrt{n}$ then we use the deterministic dynamic dictionary of Ružić [27]. Using the correct choice of constants, this dictionary supports look-ups and insert operations in $O(1)$ and $O(\sqrt{n})$ time respectively (and linear space). As there are only $O(n/\tau) = O(\sqrt{n})$ inserts, the total time taken is $O(n)$ and the space used is $O(n/\tau)$. If $\tau \leq \sqrt{n}$ we use the Las-Vegas dynamic dictionary of Dietzfelbinger and Meyer auf der Heide [7]. If $\Theta(\sqrt{n}) = O(n/\tau)$ space is used for D_ℓ, as we perform $O(n)$ operations, every operation takes $O(1)$ time with high probability. In either case, over all ℓ we take $O(n \log n)$ total time processing dictionary operations.

The operations performed on AC fall conceptually into three categories, each totalling $O(n \log n)$ time. First we insert $O(n/\tau)$ τ-length substrings into the AC dictionary (line 9). Second, we feed T into the automaton (line 11,15) and third, we inspect the AC state at most n times (line 20). We store AC in the standard compacted form with edge labels stored as pointers into T. This means that the space used is $O(n/\tau)$, the number of strings in the AC dictionary.

Finally we bound the time spent constructing fingerprints. We first consider computing f'_i (line 13) for $i > 1$. We can compute f'_i in $O(1)$ time from f'_{i-1}, $T[i-1]$ and $T[i + \tau \cdot 2^\ell]$. This follows immediately from the definition of ϕ. We can compute L'_i and R'_i analogously. Over all i, ℓ, this gives $O(n \log n)$ time. Similarly we can compute f_j from f_{j-1}, $T[(j-1)\tau \ldots j\tau - 1]$ and $T[(j-1+2^\ell)\tau \ldots (j+2^\ell)-1]$ in $O(\tau)$ time. Again this is analogous for L'_i and R'_i. Summing over all j, ℓ this gives $O(n \log n)$ time again. Finally observe that the algorithm only needs to store the current and previous values for each fingerprint so this does not dominate the space usage.

5 Longest Common Extensions on Two Strings

We now show how to efficiently reduce LCE queries between two strings to LCE queries on a single string. We generalize our notation as follows. Let P and T be strings of lengths n and m, respectively. Define $\mathrm{LCE}_{P,T}(i,j)$ to be the length of the longest common prefix of the substrings of P and T starting at i and j, respectively. For a single string P, we define $\mathrm{LCE}_P(i,j)$ as usual. We can always trivially solve the LCE problem for P and T by solving it for the string obtained by concatenating P and T. We show the following improved result.

Theorem 6. *Let P and T be strings of lengths m and n, respectively. Given a solution to the LCE problem on P using $s(m)$ space and $q(m)$ time and a parameter τ, $1 \leq \tau \leq n$, we can solve the LCE problem on P and T using $O(\frac{n}{\tau} + s(m))$ space and $O(\tau + q(m))$ time.*

For instance, plugging in Theorem 1(i) in Theorem 6 we obtain a solution using $O(\frac{n}{\tau} + \frac{m}{\sqrt{\tau}})$ space and $O(\tau)$ time. Compared with the direct solution on the concatenated string that uses $O(\frac{n+m}{\sqrt{\tau}})$ we save substantial space when $m \ll n$.

5.1 The Data Structure

The basic idea for our data structure is inspired by a trick for reducing constant factors in the space for the LCE data structures [9, Ch. 9.1.2]. We show how to extend it to obtain asymptotic improvements. Let P and T be strings of lengths m and n, respectively. Our data structure for LCE queries on P and T consists of the following information.

– A data structure that supports LCE queries for P using $s(m)$ space and $q(m)$ query time.
– An array A of length $\lfloor \frac{n}{\tau} \rfloor$ such that $A[i]$ is the maximum LCE value between any suffix of P and the suffix of T starting at position $i \cdot \tau$, that is, $A[i] = \max_{j=1\ldots m} \mathrm{LCE}_{P,T}(j, i\tau)$.
– An array B of length $\lfloor \frac{n}{\tau} \rfloor$ such that $B[i]$ is the index in P of a suffix that maximizes the LCE value, that is, $B[i] = \arg\max_{j=1\ldots m} \mathrm{LCE}_{P,T}(j, i\tau)$.

Arrays A and B use $O(n/\tau)$ space and hence the total space is $O(n/\tau + s(m))$. We answer an $\mathrm{LCE}_{P,T}$ query as follows. Suppose that $\mathrm{LCE}_{P,T}(i,j) < \tau$. In that case we can determine the value of $\mathrm{LCE}_{P,T}(i,j)$ in $O(\tau)$ time by explicitly comparing the characters from position i in P and j in T until we encounter the mismatch. If $\mathrm{LCE}_{P,T}(i,j) \geq \tau$, we explicitly compare $k < \tau$ characters until $j + k \equiv 0 \pmod{\tau}$. When this occurs we can lookup a suffix of P, which the suffix $j + k$ of T follows at least as long as it follows the suffix $i + k$ of P. This allows us to reduce the remaining part of the $\mathrm{LCE}_{P,T}$ query to an LCE_P query between these two suffixes of P as follows.

$$\mathrm{LCE}_{P,T}(i,j) = k + \min\left(A\left[\frac{j+k}{\tau}\right], \mathrm{LCE}_P\left(i+k,\, B\left[\frac{j+k}{\tau}\right]\right)\right) .$$

We need to take the minimum of the two values, since, as shown by Example 3, it can happen that the LCE value between the two suffixes of P is greater than that between suffix $i+k$ of P and suffix $j+k$ of T. In total, we use $O(\tau + q(m))$ time to answer a query. This concludes the proof of Theorem 6.

Example 3. Consider the query $\mathrm{LCE}_{P,T}(2, 13)$ on the string P from Example 2 and

$$\begin{array}{cccccccccccccccccccccccc} & 1 & 2 & 3 & 4 & \underline{5} & 6 & 7 & 8 & 9 & \underline{10} & 11 & 12 & 13 & 14 & \underline{15} & 16 & 17 & 18 & 19 & \underline{20} & 21 & 22 \\ T = & \mathrm{c} & \mathrm{a} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{a} & \mathrm{c} & \mathrm{a} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{a} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{c} & \mathrm{a} & \mathrm{e} \end{array}$$

The underlined positions in T indicate the positions divisible by 5. As shown below, we can use the array $A = [0, 6, 4, 2]$ and $B = [16, 3, 11, 10]$.

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		
i	iv	$P =$	d	b	c	a	a	b	c	a	b	c	a	a	b	c	a	c	$A[i]$	$B[i]$
1	5																	$\cancel{e}$	0	16
2	10				c	a	a	b	c	a	$\cancel{a}$								6	3
3	15												a	a	b	c	$\cancel{d}$		4	11
4	20											c	a	$\cancel{e}$					2	10

To answer the query $\mathrm{LCE}_{P,T}(2,13)$ we make $k = 2$ character comparisons and find that

$$\begin{aligned}\mathrm{LCE}_{P,T}(2,13) &= 2 + \min\left(A\left[\frac{13+2}{5}\right], \mathrm{LCE}_P\left(2+2, B\left[\frac{13+2}{5}\right]\right)\right) \\ &= 2 + \min(4,5) = 6\ .\end{aligned}$$

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM 18 (1975)
2. Allouche, J., Baake, M., Cassaigne, J., Damanik, D.: Palindrome complexity. Theoret. Comput. Sci. 292(1), 9–31 (2003)
3. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. J. Algorithms 50(2), 257–275 (2004)
4. Breslauer, D., Galil, Z.: Finding all periods and initial palindromes of a string in parallel. Algorithmica 14(4), 355–366 (1995)
5. Colbourn, C.J., Ling, A.C.: Quorums from difference covers. Inf. Process. Lett. 75(1-2), 9–12 (2000)
6. Cole, R., Hariharan, R.: Approximate String Matching: A Simpler Faster Algorithm. SIAM J. Comput. 31(6), 1761–1782 (2002)
7. Dietzfelbinger, M., Meyer auf der Heide, F.: A New Universal Class of Hash Functions and Dynamic Hashing in Real Time. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 6–19. Springer, Heidelberg (1990)
8. Fischer, J., Heun, V.: Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 36–48. Springer, Heidelberg (2006)
9. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology, Cambridge (1997)
10. Gusfield, D., Stoye, J.: Linear time algorithms for finding and representing all the tandem repeats in a string. J. Comput. Syst. Sci. 69, 525–546 (2004)
11. Gąsieniec, L., Kolpakov, R., Potapov, I.: Space efficient search for maximal repetitions. Theoret. Comput. Sci. 339(1), 35–48 (2005)
12. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
13. Ilie, L., Navarro, G., Tinta, L.: The longest common extension problem revisited and applications to approximate string searching. J. of Discrete Algorithms 8, 418–428 (2010)
14. Jeuring, J.: The Derivation of On-Line Algorithms, with an Application to Finding Palindromes. Algorithmica 11(2), 146–184 (1994)
15. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)
16. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. 31(2), 249–260 (1987)
17. Kolpakov, R., Kucherov, G.: Searching for Gapped Palindromes. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 18–30. Springer, Heidelberg (2008)
18. Landau, G.M., Myers, E.W., Schmidt, J.P.: Incremental string comparison. SIAM J. Comput. 27(2), 557–582 (1998)

19. Landau, G.M., Schmidt, J.P.: An Algorithm for Approximate Tandem Repeats. J. Comput. Biol. 8(1), 1–18 (2001)
20. Landau, G.M., Vishkin, U.: Fast Parallel and Serial Approximate String Matching. J. Algorithms 10, 157–169 (1989)
21. Lu, L., Jia, H., Dröge, P., Li, J.: The human genome-wide distribution of DNA palindromes. Funct. Integr. Genomics 7(3), 221–227 (2007)
22. Main, M.G., Lorentz, R.J.: An O (n log n) algorithm for finding all repetitions in a string. J. Algorithms 5(3), 422–432 (1984)
23. Manacher, G.: A New Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String. J. ACM 22(3), 346–351 (1975)
24. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Efficient algorithms to compute compressed longest common substrings and compressed palindromes. Theoret. Comput. Sci. 410(8-10), 900–913 (2009)
25. Myers, E.W.: An $O(ND)$ difference algorithm and its variations. Algorithmica 1(2), 251–266 (1986)
26. Puglisi, S.J., Turpin, A.: Space-Time Tradeoffs for Longest-Common-Prefix Array Computation. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 124–135. Springer, Heidelberg (2008)
27. Ružić, M.: Uniform deterministic dictionaries. ACM Trans. Algorithms 4, 1–23 (2008)

Local Exact Pattern Matching for Non-fixed RNA Structures

Mika Amit[1,⋆], Rolf Backofen[3,4,⋆⋆], Steffen Heyne[3,⋆⋆], Gad M. Landau[1,2,⋆⋆⋆], Mathias Möhl[3,⋆⋆], Christina Schmiedl[3,⋆⋆], and Sebastian Will[3,5,⋆⋆]

[1] Department of Computer Science, University of Haifa, Mount Carmel, Haifa, Israel
[2] Department of Computer Science and Engineering, NYU-Poly, Brooklyn NY, USA
[3] Bioinformatics, Institute of Computer Science, Albert-Ludwigs-Universität, Freiburg, Germany
[4] Center for Biological Signaling Studies (BIOSS), Albert-Ludwigs-Universität, Freiburg, Germany
[5] CSAIL and Mathematics Department, MIT, Cambridge MA, USA

Abstract. Detecting local common sequence-structure regions of RNAs is a biologically meaningful problem. By detecting such regions, biologists are able to identify functional similarity between the inspected molecules. We developed dynamic programming algorithms for finding common structure-sequence patterns between two RNAs. The RNAs are given by their sequence and a set of potential base pairs with associated probabilities. In contrast to prior work which matches fixed structures, we support the *arc breaking* edit operation; this allows to match only a subset of the given base pairs. We present an $O(n^3)$ algorithm for local exact pattern matching between two nested RNAs, and an $O(n^3 \log n)$ algorithm for one nested RNA and one bounded-unlimited RNA.

1 Introduction

Ribonucleic acid (RNA) is a chain of nucleotides present in the cells of all living organisms. Most RNAs are single-stranded. RNA strands have a backbone made from groups of phosphates and ribose sugar, to which one of four bases can attach (Adenine, Cytosine, Guanine, and Uracil). The bases are linked together by their phosphodiester bonds (usually referred to as *backbone connection*), and interact with each other using hydrogen bonds (usually referred to as *bond connections*), forming the RNA structure. We further denote two bases that are connected by bond connection as *base pairs* and a base that has only backbone connections as a *single base*. RNA performs important functions for living organisms, ranging from the regulation of gene expression to assistance with copying genes. The important role that small RNA take in operating the cell's control has been discovered recently and it was referred to as the breakthrough of the year 2002 in Science magazine [4].

⋆ Partially supported by the Israel Science Foundation grant 347/09 and DFG.
⋆⋆ Partially supported by the German Research Foundation (BA 2168/3-1and MO 2402/1-1).
⋆⋆⋆ Partially supported by the National Science Foundation Award 0904246, Israel Science Foundation grant 347/09, Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG.

Finding similarity between sequences and structures of RNAs is an important and well studied task. The reason is that the activity and functionality of RNA is determined by its sequence and mainly by its secondary and tertiary structure [15]. Furthermore, the structure of a molecule is usually much more preserved during evolution than its sequence alone. Thus, analyzing and comparing the secondary (and tertiary) structures of given RNAs plays a very important role in the RNA research.

The complexity of RNA secondary structure is defined by the amount and order of the *base pairs* that it contains. It is commonly categorized as follows:

- *Plain*: no arcs at all (this is the primary structure of the RNA)
- *Nested*: each base can be connected to at most one other base, and there are no crossing arcs
- *Crossing*: each base can be maximally connected to one other base
- *Bounded-Unlimited*: each base can be maximally connected to a constant number of other bases
- *Unlimited*: no restrictions on the arcs

Figure 1 demonstrates three ways of visualizing RNA nested structure. Throughout this work we use the arc-annotated sequence, that represents both the sequence and the structure of the RNA by adding an arc between each two bases that have a bond connection. This representation can describe both nested and bounded-unlimited RNA structures (see figure 1).

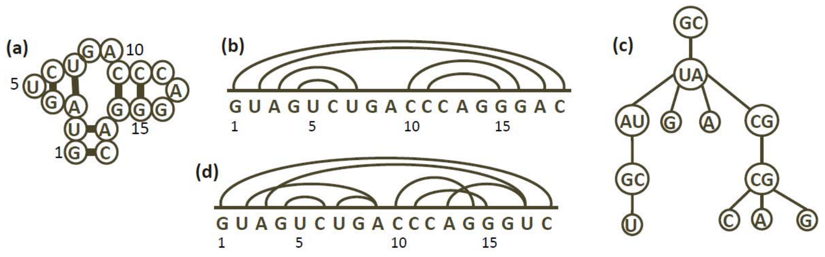

(a)
(b)
(c)
(d)
G U A G U C U G A C C C A G G G A C
G U A G U C U G A C C C A G G G U C
GC
UA
AU
CG
GC
U
1
5
10
15

Fig. 1. RNA secondary structure representations: figures (a-c) represent the same RNA sample of length 18 with depth 5. (a) schematic two dimensional description of RNA folding (b) arc annotated sequence (c) an ordered tree: a single base is represented as a leaf and a base pair is represented as either a leaf (if the base pair's size is 2) or as an internal node with child nodes (of the base pairs and single bases that it contains). Figure (d) represents a bounded-unlimited RNA structure with an arc-annotated sequence.

There are several approaches to compute the similarity between two given RNAs, among them are tree similarity algorithms such as edit distance ([19], [20], [11], [5], [2], [6]), alignment ([10], [16], [1], [14]), and LAPCS ([7], [13], [9]). An edit distance between two ordered trees, T_1 and T_2, of sizes n and m ($n > m$), is a set of edit operations applied on T_1 in order to turn it into T_2. The optimal edit distance between two trees is such set of edit operations with minimum cost. Tree alignment restricts the edit operations such that insertions are made for both T_1 and T_2 to make them isomorphic, and then relabeling of the nodes is done (see [3] for a thorough survey). Zhang and Shasha [20] present an edit distance algorithm that works in $O(nm \times min\{D_1, L_1\} \times min\{D_2, L_2\})$ where D_i is the depth of tree i and L_i is the number of leaves in tree i. Klein [11] presents an $O(m^2 n \log n)$ algorithm, which in some cases performs better than the previous algorithm. Recently an optimal $O(n^3)$ decomposition algorithm for tree edit distance was given by Demaine et al. [5]. Ma et al. [21] compute the edit distance between two RNAs where at least one is of nested structure. This algorithm runs in $O(n^2 D_1 D_2)$, and an explanation of how to modify it to run in $O(n^3 \log n)$ is given.

Jansson and Peng [8] describe $O(n^4)$ algorithms for finding a subforest F of T_1 such that F has a minimal edit distance from T_2. The structure of F is restricted to being a *simple*, *sibling* or *closed* subforest, where a *simple* subforest is a subtree, a *sibling* subforest is a set of simple subforests whose roots are siblings in T_1, and *closed* is a complete subtree of T_1.

Another approach for similarity checking is finding common motifs between two RNAs. In this problem, local maximal exact sequence-structure patterns are computed. Backofen and Siebert [17] solve this problem in $O(n^2)$ time.

1.1 Our Results

In this work, we are looking for local exact pattern matching between two RNA molecules. We use the definitions from [17], and add an additional edit operation: *arc breaking*, which breaks a base pair into two single bases. Adding the *arc breaking* operation means that the bonds are not necessarily preserved in the common substructure. This enhancement to the pattern matching algorithm allows greater flexibility in both the input and the output. Instead of representing a fixed structure, the input can be interpreted as a set of weighted secondary structures. This is encoded by base pairs with probabilities. For this purpose we score the match of two base pairs according to their probabilities. The *arc breaking* operation is not supported in [17],[21], or any other algorithms based on tree edit distance, and it is one of our major achievements in this paper. Figure 2 demonstrates the *arc breaking* edit operation. The formal definitions of the problems are given in section 2.

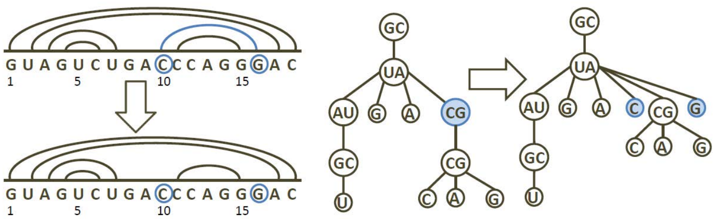

G U A G U C U G A C C C A G G G A C
1
5
10
15
G U A G U C U G A C C C A G G G A C
1
5
10
15
GC
UA
AU
G
A
CG
GC
U
CG
C
A
G
GC
UA
AU
G
A
C
CG
G
GC
U
C
A
G

Fig. 2. Arc breaking operation: both representations show the result of the arc breaking operation for base pair CG in positions (10,16)

We present a simple $O(n^4)$ algorithm for computing the local exact pattern matching between two nested RNAs (section 3). In section 4, we continue with an $O(n^3 \log n)$

algorithm, and in section 6 we show how to modify the algorithm to support one nested and one bounded-unlimited input structure ($(Nested, Bounded-Unlimited)$, in short). In section 5 we show how to improve the algorithm for $(Nested, Nested)$ RNAs to $O(n^3)$.

Due to space limitations we will describe the following algorithms in the extended version of this paper:

– An $O(n^3k^2)$ algorithm for computing the local approximate matching between two nested RNAs with at most k mismatches. This algorithm can be also modified to work in $O(n^3k^2 \log n)$ for $(Nested, Bounded-Unlimited)$ RNAs.
– An $O(n^3)$ algorithm for computing the most similar sibling substructure between two $(Nested, Nested)$ RNAs, as defined in [8].

2 Notations and Definitions

RNA sequence is an ordered pair $R = (S, B)$, where $S = s_1, \ldots, s_{|S|}$, and s_i is defined over the alphabet $\Sigma = \{A, C, G, U\}$ and represents the RNA primary structure. B, the optional secondary structure, is a set of tuples $\{(a, b, p)|1 \leq a < b \leq |S|,\ 0 < p \leq 1\}$, such that a tuple $bp = (a, b, p) \in B$ represents a hydrogen bond (a base pair) between bases a and b that exists with probability p in R. We denote a and b as the *left* and *right* endpoints of *bp*, respectively. A base that is neither left nor right endpoint is denoted as a *single base*. We further distinguish between two connection types of bases in R: the connection between a base i and its subsequent base $i + 1$ is denoted as a *backbone connection*, and a base pair connection is denoted as a *bond connection*. The *size* of a base pair $bp = (a, b, p) \in B$ is the number of bases that it contains. i.e., $|bp| = (b - a + 1)$. We assume that the number of base pairs in R is $O(n)$, which holds for nested and bounded-unlimited structures by definition.

Definition 1 (*Parent-child* relation between bases). *A parent of base pair bp* $= (a, b, p) \in B$ *(resp. single base i) is the smallest size base pair pbp* $= (c, d, q) \in B$ *that contains bp (resp. i) in it. That is, c, d are the closest endpoints of a base pair such that $c < a < b < d$ (resp. $c < i < d$). We denote bp (resp. i) as the child of pbp.*

We proceed with definitions of substructures of R (see figure 3 for examples):

Definition 2 (Path). *A path in RNA R is a sequence of positions $(i_1, \ldots, i_y)$ such that $\forall 1 \leq k < y$, i_k is connected to i_{k+1} with either a backbone or a bond connection. If i_k is connected to i_{k+1} with a bond connection we say that the base pair bp* $= (i_k, i_{k+1}, p)$ *is contained in the path.*

Definition 3 (Pattern). *A pattern in RNA R is a set of positions $P = \{i_1, \ldots, i_y\}$ such that $\forall k, l \in P$ there exists a path in P that connects i_k and i_l.*

Definition 4 (Exact Pattern Matching). *Given two RNAs $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$, with sizes n and m respectively, an exact pattern matching (in short, matching) M, over R_1 and R_2 is a set of pairs $M = \{(i_1, j_1), \ldots, (i_k, j_k)|\ \forall 1 \leq \ell \leq k, 1 \leq i_\ell \leq n, 1 \leq j_\ell \leq m\}$ that satisfies the following conditions:*

1. $S_1(i_\ell) = S_2(j_\ell)\ \forall 1 \leq \ell \leq k$.
2. $P_1 = \{i_1, \ldots, i_k\}$ *is a pattern in* R_1.
3. $P_2 = \{j_1, \ldots, j_k\}$ *is a pattern in* R_2.
4. For each $1 \leq x, y \leq k$, *a base pair* $bp_1 = (i_x, i_y, p)$ *is contained in* P_1 *if and only if a base pair* $bp_2 = (j_x, j_y, q)$ *is contained in* P_2.
5. M *is maximally extended.*

The first condition applies to the sequence equivalence requirement, whereas the rest of the conditions apply to the structural equivalence requirement. The last condition refers to the maximality of the matching, meaning that it cannot be extended sequence- or structure- wise. For two base pairs in the matching, $bp_1 = (a, b, p) \in B_1$ and $bp_2 = (c, d, q) \in B_2$, we say that $(bp_1, bp_2) \in M$.

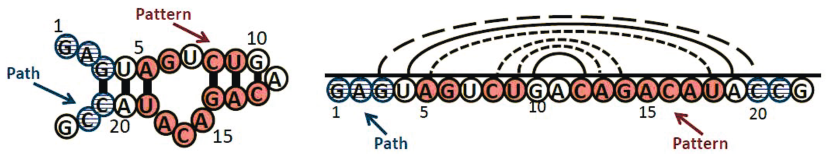


Fig. 3. Path and pattern examples in two different representations of the same RNA sample. A path is marked with horizontal lines and contains the bases {1,2,3,20,21}, a pattern is shadowed and contains the bases {5,6,8,9,12,13,14,15,16,17,18}. Note that the pattern contains the base pairs (5,18), (8,14) and (9,13), whereas the base pairs (3,20) and (10,12) are not included.

Each matching M has an associated *score* that can be described as:

$$score(M) = \sum_{(i,j)\in M} \alpha(i,j) + \sum_{(bp_1,bp_2)\in M} \beta(bp_1, bp_2),$$

where $\alpha : [1, |\Sigma|] \times [1, |\Sigma|] \rightarrow \mathbb{R}$ returns the score of matching two single bases:

$$\alpha(i,j) = 1 \text{ if } S_1(i) = S_2(j) \text{ or } -\infty \text{ otherwise,}$$

and $\beta : ([1, |B_1|]) \times ([1, |B_2|]) \rightarrow \mathbb{R}$ returns the score of matching two base pairs $bp_1 = (a, b, p), bp_2 = (c, d, q)$:

$$\beta(bp_1, bp_2) = ((1+p) \times (1+q)) \text{ if } S_1(a) = S_2(c) \text{ and } S_1(b) = S_2(d), \text{ or } -\infty, \text{ otherwise.}$$

The definition of the scoring functions enables finding biologically meaningful structures via the scoring. In the general case the scoring functions can be defined to return scores other than 1 or $(1+p) \times (1+q)$ when the bases match. The optimal sequence-structure matching depends on both the matching of single bases and base pairs. This enables us to sometimes prefer a matching of a base pair with a high probability over matching a single base, or prefer matching large sequence of single bases over low probability base pair (see figure 4).

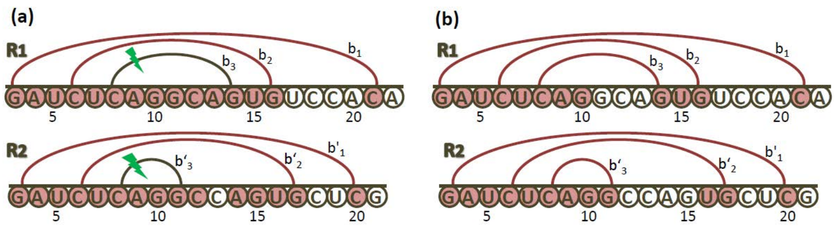


Fig. 4. Two matchings example. The figure presents two matching examples that can be defined between R_1 and R_2. In both cases the matchings are maximally extended. Note that matching (a) contains the base pairs (b_1, b'_1) and (b_2, b'_2), and matching (b) contains (b_1, b'_1), (b_2, b'_2) and (b_3, b'_3). The matching scores depend on the definition of α and β functions. Given $b_1 = (3, 21, 0.9)$, $b_2 = (6, 16, 0.6)$, $b_3 = (8, 14, 0.1)$, $b'_1 = (3, 20, 0.8)$, $b'_2 = (6, 17, 0.5)$, and $b'_3 = (8, 11, 0.3)$ and using our function definitions, $score(a) = 15 + (1.9 \cdot 1.8) + (1.6 \cdot 1.5) = 20.82$ and $score(b) = 12 + (1.9 \cdot 1.8) + (1.6 \cdot 1.5) + (1.1 \cdot 1.3) = 19.25$, thus the matching with the maximal score is (a).

2.1 Local Exact Pattern Matching Problem Definition

Given two RNAs, $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$ with sizes n and m, resp. $(n \geq m)$, scoring functions $\alpha()$ and $\beta()$, and a number c, we want to find the set $\mathcal{M}$ containing all matchings with a score greater than c. i.e,

$$\mathcal{M} = \{M | M \textit{ is a matching and } score(M) \ \geq c)\}$$

Note that the definition of the problem does not restrict the structure of the given RNA molecules. We will explore two different settings of RNAs: $(Nested, Nested)$ and $(Nested, Bounded - Unlimited)$.

3 A Simple $O(n^4)$ Algorithm for Local Exact Pattern Matching

In this section we solve the local exact pattern matching problem following its definition in section 2.1. We use similar ideas to those in Zhang and Shasha's tree edit distance algorithm [20]. The algorithm distinguishes between two cases of matchings: those that don't contain any base pair matching and those that contain at least one. In the first case, no base pair from B_1 is matched with a base pair from B_2. The problem is, therefore, finding common substrings using suffix trees in time and space $O(n + m)$ ([12]). The second case is the more interesting one, and we will explore its implementation in the following sections. The key idea is that we find the matchings between each combination of a *base pair* from B_1 and a *base pair* from B_2. For convenience reasons, we refer to arc-annotated substrings as *substrings*.

3.1 Finding the Maximal Matching between Two Base Pairs

The algorithm divides the process of finding the matching into two stages: finding the maximal matching in between the two endpoints of both base pairs (discussed in sections 3.2), and extending the match "outside" of the base pairs (discussed in section 3.3). On each of these stages, the maximal score is saved in table M, of size $O(|B_1||B_2|)$, in which an entry M_{bp_1,bp_2} contains the scores of comparing the two base pairs $bp_1 \in B_1$ and $bp_2 \in B_2$: inside the base pairs, their maximal extensions and the total score. We denote these scores as $M^{in}_{bp_1,bp_2}$, $M^{out}_{bp_1,bp_2}$, and $M^{total}_{bp_1,bp_2}$ respectively.

3.2 Finding the Maximal Score Matching Inside the Base Pairs

The input of the algorithm is two RNAs $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$ and the output is M^{in} table, in which an entry $M^{in}_{bp_1,bp_2}$ contains the maximal matching score between the base pairs $bp_1 \in B_1$ and $bp_2 \in B_2$ and their inner parts. The values of M^{in} table are computed in increasing order of the base pairs' sizes in order to enable reuse of calculations: if two base pairs are contained in two other base pairs, then the calculation of the smaller base pairs' maximal matching is already calculated and there is no need to recalculate it (see figure 5 case (c) for an example).

The main procedure of the algorithm computes for every combination of a base pair $bp_1 = (a, b, p) \in B_1$ and a base pair $bp_2 = (c, d, q) \in B_2$, their maximal matching score by comparing the two substrings $s = (s_a, \ldots, s_b)$ and $t = (t_c, \ldots, t_d)$ that are defined over bp_1 and bp_2, respectively. It is a dynamic programming algorithm that computes matchings between prefixes of the substrings s and t, in increasing order of their sizes.

We next describe the $patternMatch()$ function that computes the maximal matching score between two substring s and t.

The Pattern Matching Function
For every two substrings $s = (s_a, \ldots, s_i)$ and $t = (t_c, \ldots, t_j)$ the function computes four different matchings:

- $Lmatch$: The maximal left-to-right matching that starts at positions (a, c) and continues going from left to right using a backbone or bond connections until either a mismatch occurs or the rightmost bases of s or t are reached.
- $Rmatch$: The maximal right-to-left matching that starts at (i, j) and continues going from right to left until either a mismatch occurs or the leftmost bases of s or t are reached.
- $Full$: The maximal matching that contains both (a, c) and (i, j) indices, if such matching exists.
- $Score$: The maximal left to right and right to left matchings between the two substrings, such that they do not overlap and are maximally extended.

Note that the maximal matching score does not necessarily include both $Rmatch$ and $Lmatch$, since the bases they contain may overlap. Another observation is that the score of a $Full$ matching is not always greater than $Score$ (see figure 5 for examples).

We use $Score(a \ldots i, c \ldots j)$ to refer to $Score$ between substrings $s = (s_a, \ldots, s_i)$ and $t = (t_c, \ldots, t_j)$. We refer to $Lmatch$, $Rmatch$ and $Full$ properties in a similar way. The values are computed according to the following equations (in the same order):

$$Full(a \ldots i, c \ldots j) = \max \begin{cases} Full(a \ldots i-1, c \ldots j-1) + \alpha(i, j) \\ Full(a \ldots e-1, c \ldots f-1) + M^{in}_{b_1,b_2} \end{cases} \quad (1)$$

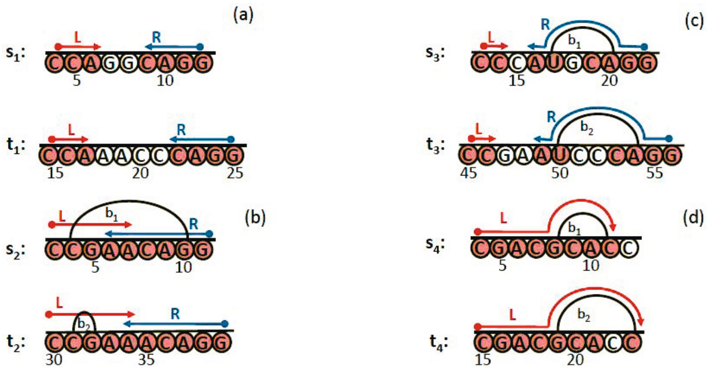


Fig. 5. $Lmatch$,$Rmatch$,$Full$ and $Score$ matchings between substrings s_i and t_i: $Lmatch$ is marked with 'L' right arrows, and $Rmatch$ is marked with 'R' left arrows. The probabilities of b_1 and b_2 to exist in s_i and t_i are 0.2 and 0.2, resp.
(a) Non-overlapping: $Lmatch(s_1, t_1)$ contains 'CCA' and $Rmatch(s_1, t_1)$ contains 'GGAC', $Full = -\infty$ and $Score(s_1, t_1) = 7$ (both 'CCA' and 'GGAC') .
(b) Overlapping: $Lmatch(s_2, t_2)$ contains positions (6,33) and (7,34), $Rmatch(s_2, t_2)$ contains positions (7,35) and (6,34). Note that using both (7,34) and (7,35) (or both (6,33) and (6,34)) would have created an overlapping matching. Therefore, $Score(s_2, t_2) = 9$ (all single bases of s_2 and t_2 excluding base 35), and $Full(s_2, t_2) = -\infty$ since there is no matching that contains both (3,30) and (11,39). Note that in this case, the maximal score is the one that uses arc breaking operation: $Score(s_2, t_2)$ does not include "jumping over" b_1 and b_2.
(c) "Jumping over" base pairs: $Lmatch(s_3, t_3)$ contains 'CC', $Rmatch(s_3, t_3)$ contains 'GG', (b_1, b_2), and 'A'. Note that since b_1 and b_2 are contained in $Rmatch(s_3, t_3)$, the matching bases inside of them ('AC' and 'U') are also contained in $Rmatch(s_3, t_3)$. Also note that the matching between b_1 and b_2 is a $Full$ matching: the matching bases $\{(17, 50), (19, 53), (20, 54)\}$ contains both endpoints of b_1 and b_2. $Full(s_3, t_3) = -\infty$ and $Score(s_3, t_3) = 9.43$ (contains 'CC' from left and 'GG', (b_1, b_2), and 'A' from right).
(d) $Full < Lmatch$: $Full(s_4, t_4) = 9$ by matching all bases of both s_4 and t_4 and arc-breaking b_1 and b_2. $Lmatch(s_4, t_4) = 9.43$ by "jumping over" the base pairs b_1 and b_2, $Rmatch(s_4, t_4) = 9$. Hence, $Score(s_4, t_4) = Lmatch(s_4, t_4)$.

$$Lmatch(a \dots i, c \dots j) = \max \begin{cases} Lmatch(a \dots i-1, c \dots j) \\ Lmatch(a \dots i, c \dots j-1) \\ Full(a \dots i, c \dots j) \end{cases} \tag{2}$$

$$Rmatch(a \dots i, c \dots j) = \max \begin{cases} Rmatch(a \dots i-1, c \dots j-1) + \alpha(i,j) \\ Rmatch(a \dots e-1, c \dots f-1) + M^{in}_{b_1,b_2} \\ 0 \end{cases} \tag{3}$$

$$Score(a \dots i, c \dots j) = \max \begin{cases} Lmatch(a \dots i, c \dots j) \\ Score(a \dots i-1, c \dots j-1) + \alpha(i,j) \\ Score(a \dots e-1, c \dots f-1) + M^{in}_{b_1,b_2} \end{cases} \tag{4}$$

where $b_1 = (e, i, r) \in B_1$ and $b_2 = (f, j, w) \in B_2$ (if such base pairs do not exist the value of $M^{in}_{b_1,b_2}$ is $-\infty$).

Finally, the score of $M^{in}_{bp_1,bp_2}$ is set as follows:
$M^{in}_{bp_1,bp_2} = \beta(bp_1, bp_2) + Score(a \ldots b, c \ldots d)$.

The computation of $Full$ values is straight-forward: either the matching is extended to include the rightmost bases, or it is extended to include the rightmost base pairs and their inner parts. If the matching cannot be extended, the value is set to $-\infty$. $Lmatch$ value is the maximum between previously computed $Lmatch$ scores and the current computed $Full$ value. $Rmatch$ contains the maximal score that includes i, j, therefore, if the bases mismatch, it is set to 0. Otherwise, it is the maximum between extending the matching with the rightmost bases or base pairs. The value of $Score$ is the maximum between extending the maximal score with either single base or base pairs matching, or the maximal left to right matching, $Lmatch$, that was computed between the substrings. The reason for that is that each one of the allowed operations can set $Rmatch$ score to 0. $Lmatch$, on the other hand, cannot be decreased and it can only be increased to contain the $Full$ matching score (if it is bigger).

Note that in any of the computations the structure of the rightmost bases is not checked, which can lead to *arc-breaking* - the case when a base pair is treated as two single bases with no bond connection between them.

The value of $Rmatch$ is not used for the total score in this algorithm, but in the improved algorithm it will be used and for clarity we define it here.

3.3 Extending the Match Outside the Base Pairs

This section describes the algorithm for computing the maximal extension of the matching outside the endpoints of base pairs. The input of the algorithm is two RNAs, $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$, and the table M^{in}. The output is M^{out} table. Each base pairs comparison can be extended to both left and right, in this section we describe the algorithm for the extension to the right; the extension to the left is similar.

The algorithm computes the maximal extensions scores for every position $i \in R_1$ and $j \in R_2$, in decreasing order of i and j. The values are kept in $Rextend$ table (of size $O(n^2)$), in which an entry $Rextend(i, j)$ contains the maximal extension starting at positions i, j going right. If a mismatch occurs between s_i and t_j, the value is set to 0. Otherwise, the value is the maximum between matching single bases and matching base pairs, as follows:

$$Rextend(i,j) = \max \begin{cases} Rextend(i+1, j+1) + \alpha(i,j) \\ Rextend(b+1, d+1) + M^{in}_{b_1,b_2} \\ 0 \end{cases} \tag{5}$$

where $b_1 = (i, b, r) \in B_1$ and $b_2 = (j, d, w) \in B_2$.

Eventually, for every two base pairs, $bp_1 = (a, b, p) \in B_1$ and $bp_2 = (c, d, q) \in B_2$, the values in $M^{out}_{bp_1,bp_2}$ table are set as follows: $M^{out}_{bp_1,bp_2} = Rextend(b + 1, d + 1) + Lextend(a - 1, c - 1)$.

3.4 Complete $O(n^4)$ Algorithm

The algorithm for computing the local exact pattern matching between two given RNA molecules is as follows:

(a) Compute the pattern matching inside all base pairs into M^{in}.
(b) Compute the extension tables $Rextend$ and $Lextend$ and the table M^{out} accordingly.
(c) For each base pair $bp_1 \in B_1$ and each base pair $bp_2 \in B_2$: $M^{total}_{bp_1,bp_2} = M^{in}_{bp_1,bp_2} + M^{out}_{bp_1,bp_2}$.

Time Complexity: the time complexity of step (a) is equal to the total number of prefixes compared (since each of the allowed operations computation is done in constant time), which can be bounded by $O(n^4)$. In step (b), the computation of each entry in $Rextend$ and $Lextend$ tables is again done in constant time. Therefore, its time complexity is the number of entries in the tables, which is $O(n^2)$. In addition, in step (b) the algorithm computes M^{out} table is $O(n^2)$ time.

The last step runs in $O(n^2)$ time: for each combination of a base pair from B_1 and a base pair from B_2, the computation is done in constant time.

Therefore, the time complexity of the complete algorithm is $O(n^4 + n^2 + n^2) = O(n^4)$.

From this time complexity analysis we immediately observe that the bottleneck of the algorithm is computing the maximal matching score inside the base pairs. In the next sections (4 and 5) we show how to improve this time complexity.

4 An $O(n^3 \log n)$ Algorithm for Local Exact Pattern Matching

In this algorithm we take advantage of the fact that not all substrings that are compared as part of the $O(n^4)$ algorithm need to be compared. We use similar ideas of Klein's tree edit distance algorithm [11]. We first explain the heavy path decomposition concept in regarding RNAs and continue with the modifications to the $O(n^4)$ algorithm.

Definition 5 (*heavy-light* base pairs). *For a given RNA $R = (S, B)$, we define each base pair in B as heavy or light by the following recursive definition: the base pair $bp_1 = (1, |R|, p)$ is defined light (if such base pair does not exist, we add it as a fictive base pair). For each base pair $bp \in B$, we pick a child base pair of bp with maximal size among the children of bp and mark it as heavy, the rest of the children are marked as light. We say that $heavy(bp) = hp$ if hp is the heavy child base pair of bp.*

The sequence of $bp_1, heavy(bp_1), heavy(heavy(bp_1)), \ldots$ defines a descending path called the *heavy path*, let $P(bp_1)$ denote this path. We recursively decompose R into heavy paths: we start with $P(bp_1)$ and add the heavy path of each light child base pair of bp_1 (see figure 6). We denote each light base pair as the *root* of the heavy path that it contains.

The following Lemma of Sleator and Tarjan [18] bounds the number of light base pairs that contain a base in R:

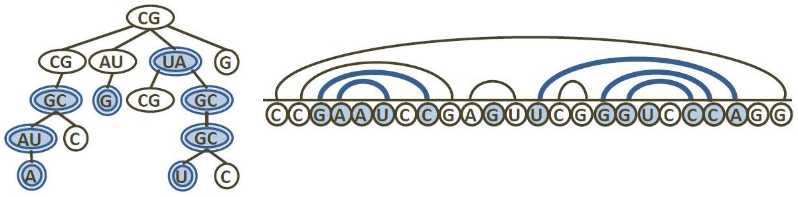


Fig. 6. Heavy path decomposition: in this RNA structure, we have three heavy path routes. They are presented in both tree and arc-annotated structures.

Lemma 1 (Sleator and Tarjan [18]). *Each base in RNA* $R = (S, B)$*, of size* n*, is contained in at most* $O(\log n)$ *light base pairs.*

Definition 6 (*Special Substrings*). *The set of special substrings of a substring* $s = (s_a, \ldots, s_b)$*, that is defined over a base pair* $bp = (a, b, p) \in B_1$ *with* $heavy(bp) = (x, y, r) \in B_1$*, consists of the suffixes of* $(s_a, \ldots, s_y)$ *starting at positions* $a, \ldots, x$*, and the prefixes of* $(s_a, \ldots, s_b)$ *ending at positions* $y, \ldots, b$ *(see figure 7).*

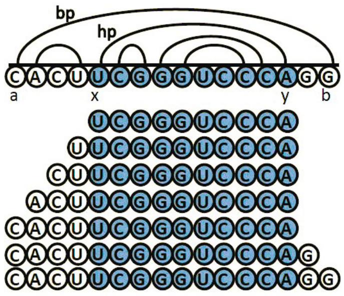


Fig. 7. Special Substrings Example: the special substrings of a base pair, $bp = (a, b, p)$, with a heavy child base pair, $hp = (x, y, r)$

Let s be a substring. We denote *last(s)* as either the rightmost or the leftmost base of s. We define $last(s)$ of a suffix special substring, s, to be the leftmost base in s, and $last(s)$ of a prefix special substring s to be its rightmost base. Each base i in $(a, \ldots, b)$ that is not contained in the heavy child base pair of bp, hp, defines exactly one special substring that contains i as its $last$ base. Thus, the number of special substrings defined over the base pair is: $size(bp) - size(hp)$.

Let $bp_1 = (a, b, p) \in B_1$ with $heavy(bp) = hp = (x, y, r) \in B_1$, $bp_2 = (c, d, q) \in B_2$, and let $s = (s_a, \ldots, s_b)$, $h = (s_x, \ldots, s_y)$ and $t = (t_c, \ldots, t_d)$ be the substrings defined over bp_1, hp and bp_2, respectively.

The algorithm is based on two changes to the $O(n^4)$ algorithm: the first modification is in the compared substrings: we compare *all* substrings of t and only the special substrings of s as part of the $patternMatch()$ function. The special substrings are compared in increasing order of their sizes: we start with the heavy child base pair's substring, h, and increase the substring from left, until the left endpoint of bp is reached (the suffixes special substrings). Then, we continue with the prefixes of bp, starting from $s_a, \ldots, s_y$, and continue going from left to right until the right endpoint of bp is reached.

The second modification is in the main procedure of $patternMatch()$: in the previous algorithm, $last(s)$ was always the rightmost base, in this version it is sometimes the leftmost base. Thus, the function should support ignoring or matching of both $last(s)$ positions. The function is therefore the combination of two $patternMatch()$ versions: for the prefixes comparisons the computation is exactly as described in section 3.2. For the suffixes comparisons the values are set according to the following equations:

$$Full(i \ldots b, j \ldots d) = \max \begin{cases} Full(i+1 \ldots b, j+1 \ldots d) + \alpha(i,j) \\ Full(e+1 \ldots b, f+1 \ldots d) + M^{in}_{b_1,b_2} \end{cases} \tag{6}$$

$$Lmatch(i \ldots b, j \ldots d) = \max \begin{cases} Lmatch(i+1 \ldots b, j+1 \ldots d) + \alpha(i,j) \\ Lmatch(e+1 \ldots b, f+1 \ldots d) + M^{in}_{b_1,b_2} \\ 0 \end{cases} \tag{7}$$

$$Rmatch(i \ldots b, j \ldots d) = \max \begin{cases} Rmatch(i+1 \ldots b, j \ldots d) \\ Rmatch(i \ldots b, j+1 \ldots d) \\ Full(i \ldots b, j \ldots d) \end{cases} \tag{8}$$

$$Score(i \ldots b, j \ldots d) = \max \begin{cases} Rmatch(i \ldots b, j \ldots d) \\ Score(i+1 \ldots b, j+1 \ldots d) + \alpha(i,j) \\ Score(e+1 \ldots b, f+1 \ldots d) + M^{in}_{b_1,b_2} \end{cases} \tag{9}$$

where $b_1 = (i, e, r) \in B_1$ and $b_2 = (j, f, w) \in B_2$.
Eventually, the value in M^{in} table is set to:
$M^{in}_{bp_1,bp_2} = \beta(bp_1, bp_2) + Score(a \ldots b, c \ldots d)$.

Time Complexity: the same reasons that the $O(n^4)$ algorithm gave constant time for each $patternMatch(s,t)$ function call apply here, too. We therefore count the number of compared substrings: following Lemma 1, each base is defined as $last(s)$ of at most $O(\log n)$ special substrings, which gives a total of $O(n \log n)$ special substrings. The set of substrings t, are all $O(n^2)$ substrings of R_2. The number of compared substrings is therefore $O(n \log n \times n^2) = O(n^3 \log n)$.

Thus, the time complexity of the above algorithm for computing the matching inside each combination of a base pair from B_1 and a base pair from B_2 is $O(n^3 \log n)$ time.

5 An $O(n^3)$ Algorithm for Local Exact Pattern Matching

In the previous algorithm (section 4) we select the larger RNA structure as the *dominant* structure. w.l.o.g. we defined R_1 to be the dominant structure, and for each $bp_1 \in B_1$, bp_1 was the dominant base pair, by which special substrings were defined.

An improvement for this algorithm can be done using the optimal decomposition algorithm described in [5]. The key observation is that the dominant structure can be decided for each combination of base pairs comparison rather than once for the entire algorithm. The complete description and proof of the algorithm are given in [5]. In this section we give the highlights of the algorithm and "translate" it into the arc-annotated representation of RNA molecules.

As an initialization step of the algorithm, *both* R_1 and R_2 are recursively decomposed into heavy paths (see figure 8). The algorithm computes the matching between each combination of a base pair $bp_1 \in B_1$ and a base pair $bp_2 \in B_2$. The difference is that on each such comparison, the algorithm selects the dominant base pair to be the one with the larger root (i.e. $|root(bp_1)|$ and $|root(bp_2)|$). The rest of the algorithm is exactly the same as the previous $O(n^3 \log n)$ algorithm, meaning that the special substrings of the dominant base pair are compared with all substrings of the other base pair (see figure 8 for an example).

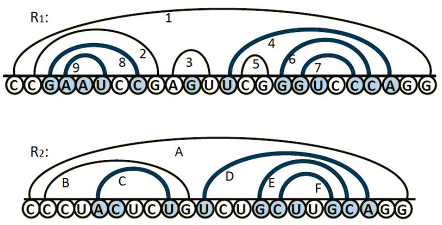


Fig. 8. Heavy Path Decomposition of Both RNA Molecules: R_1 contains the heavy path $(1, 4, 6, 7, U)$. In addition R_1 contains the heavy paths $(2, 8, 9, A)$, $(3, G)$, and 5. In the comparison between $6 \in B_1$ and $E \in B_2$ the dominant base pair is 6, whereas in the comparison between $8 \in B_1$ (or $2 \in B_1$) and $B \in B_2$ the dominant base pair is B.

This enhancement to the algorithm improves the time complexity to $O(n^3)$. The intuition to this improvement is that on each comparison between two base pairs, we compare all substrings of the *relatively smaller* base pair with the special substrings of the *relatively larger* base pair (see complete proof in [5]).

6 Local Exact Pattern Matching for (Nested, Bounded-Unlimited) Inputs

The input of this algorithm consists of two RNA structures $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$, where R_1 is a nested structure and R_2 is a bounded-unlimited structure. The output is the maximal local exact matching set M defined over R_1 and R_2.

The algorithm is similar to the $O(n^3 \log n)$ algorithm described in section 4. The difference is that the bounded-unlimited structure of R_2 needs to be handled: as opposed to the previous algorithm, where each base can be connected by a bond connection to at most one other base, in the bounded-unlimited structure it can be connected to $O(1)$ other bases. Let i be $last(s)$ of substring s, and let the $last(s)$ be the rightmost base in s, w.l.o.g.. If i is a right endpoint of a base pair $bp_1 = (e, i, p) \in R_1$, there can be several base pairs in R_2 with j being their right endpoint (e.g. $bp_k = (f_k, j, q_k) \in R_2$). All of these base pairs should be considered in the matching between s and t.

Note that even though R_2 has a bounded-unlimited structure, the output matching structure is always nested. Hence the only modification that is necessary is to iterate over all base pairs with right endpoint j and pick the one that gives the maximal total score.

In an analogous way, the algorithm for extending the matching outside of the base pairs, as described in section 3.3, is also modified to support the bounded-unlimited structure of R_2. Again, on each base pairs comparison the algorithm compares at most $O(1)$ options of base pairs matching.

Time Complexity: the only modification to $patternMatch()$ function is that we compare $O(1)$ base pairs of substring t with the base pair that starts at $last(s)$, if such exist. This, of course, does not add to the overall time complexity analysis. In a similar way, the modification to the algorithm for computing the maximal extensions does not change its time complexity.

The total time complexity of the entire algorithm is therefore $O(n^3 \log n)$.

References

1. Backofen, R., Chen, S., Hermelin, D., Landau, G.M., Roytberg, M.A., Weimann, O., Zhang, K.: Locality and gaps in RNA comparison. Journal of Computational Biology 14, 1074–1087 (2007)
2. Backofen, R., Landau, G.M., Möhl, M., Tsur, D., Weimann, O.: Fast RNA Structure Alignment for Crossing Input Structures. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 236–248. Springer, Heidelberg (2009)
3. Bille, P.: A survey on tree edit distance and related problems. Theoretical Computer Science 337, 217–239 (2005)
4. Couzin, J.: Small RNAS make big splash. Science 298(5602), 2296–2297 (2002)
5. Demaine, E.D., Mozes, S., Rossman, B., Weimann, O.: An Optimal Decomposition Algorithm for Tree Edit Distance. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 146–157. Springer, Heidelberg (2007)
6. Dulucq, S., Touzet, H.: Decomposition algorithms for the tree edit distance problem. J. Discrete Algorithms 3(2-4), 448–471 (2005)
7. Evans, P.A.: Algorithms and Complexity for Annotated Sequence Analysis. PhD thesis, University of Alberta (1999)
8. Jansson, J., Peng, Z.: Algorithms for finding a most similar subforest. Theory Comput. Syst 48(4), 865–887 (2011)
9. Jiang, T., Lin, G., Ma, B., Zhang, K.: The longest common subsequence problem for arc-annotated sequences. J. Discrete Algorithms 2(2), 257–270 (2004)
10. Jiang, T., Wang, L., Zhang, K.: Alignment of trees – an alternative to tree edit. TCS: Theoretical Computer Science 143 (1995)
11. Klein, P.N.: Computing the Edit-Distance between Unrooted Ordered Trees. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 91–102. Springer, Heidelberg (1998)
12. Landau, G.M., Vishkin, U.: Fast parallel and serial approximate string matching. Journal of Algorithms 10, 157–169 (1989)

13. Lin, G.H., Chen, Z.Z., Jiang, T., Wen, J.: The longest common subsequence problem for sequences with nested arc annotations. JCSS: Journal of Computer and System Sciences 65(3), 465–480 (2002)
14. Möhl, M., Will, S., Backofen, R.: Lifting Prediction to Alignment of RNA Pseudoknots. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 285–301. Springer, Heidelberg (2009)
15. Moore, P.B.: Structural motifs in RNA. Annual Review of Biochemistry 68, 287–300 (1999)
16. Schirmer, S., Giegerich, R.: Forest Alignment with Affine Gaps and Anchors. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 104–117. Springer, Heidelberg (2011)
17. Siebert, S., Backofen, R.: A dynamic programming approach for finding common patterns in RNAS. Journal of Computational Biology 14(1), 33–44 (2007)
18. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. Journal of Computer and System Sciences 26(3), 362–391 (1983)
19. Tai, K.C.: The tree-to-tree correction problem. JACM: Journal of the ACM 26(3), 422–433 (1979)
20. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM Journal on Computing 18(6), 1245–1262 (1989)
21. Zhang, K., Wang, L., Ma, B.: Computing Similarity between RNA Structures. In: Crochemore, M., Paterson, M. (eds.) CPM 1999. LNCS, vol. 1645, pp. 281–293. Springer, Heidelberg (1999)

Impact of the Energy Model on the Complexity of RNA Folding with Pseudoknots

Saad Sheikh[1,4], Rolf Backofen[2], and Yann Ponty[3,4,⋆]

[1] University of Florida, Gainesville, USA
[2] Albert Ludwigs University, Freiburg, Germany
[3] Ecole Polytechnique, CNRS UMR 7161, Palaiseau, France
[4] AMIB Team-Project, INRIA, Saclay, France

Abstract. Predicting the folding of an RNA sequence, while allowing general pseudoknots (PK), consists in finding a minimal free-energy matching of its n positions. Assuming independently contributing base-pairs, the problem can be solved in $\Theta(n^3)$-time using a variant of the maximal weighted matching. By contrast, the problem was previously proven NP-Hard in the more realistic nearest-neighbor energy model.

In this work, we consider an intermediate model, called the stacking-pairs energy model. We extend a result by Lyngsø, showing that RNA folding with PK is NP-Hard within a large class of parametrization for the model. We also show the approximability of the problem, by giving a practical $\Theta(n^3)$ algorithm that achieves at least a 5-approximation for any parametrization of the stacking model. This contrasts nicely with the nearest-neighbor version of the problem, which we prove cannot be approximated within any positive ratio, unless $P = NP$.

Keywords: RNA folding, General pseudoknots, Hardness, Inapproximability.

1 Introduction

Ribonucleic Acid (RNA) is one of the key pieces to the puzzle of molecular biology. It plays a very large number of roles, not only by coding for proteins, but also through catalytic and regulatory functions. To play such roles, RNA folds into an intricate structure which is stabilized by the pairing, mediated by hydrogen bonds, of some of its positions. The conformations that arise from this folding process are instrumental to the function of an RNA. Consequently, the process of RNA folding has been extensively studied by molecular biology and biochemistry, and its *in silico* prediction has given rise to a wealth of computational approaches. Early work on the subject have focused on the secondary structure, a restriction of all admissible base-pairs that forbids crossing-interactions. Under the assumption of reasonable, additive, energy models, such a restriction implies an optimal-substructure property for computing the most stable conformation, i.e. the one having minimal free-energy. Polynomial-time

⋆ To whom correspondance should be addressed.

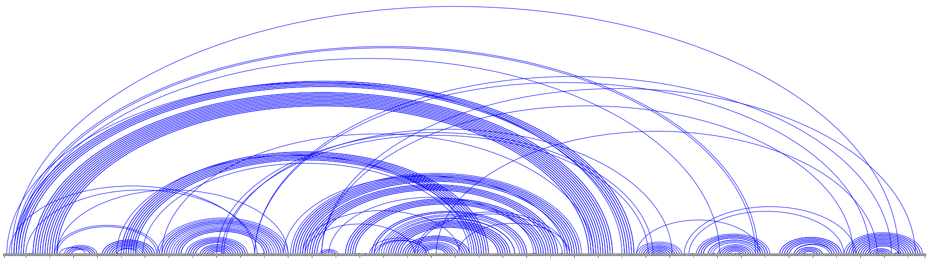


Fig. 1. Canonical base-pairs of the Oceanobacillus iheyensis group II intron, derived from a 3D model (PDB id: 3IGI) using `RNAView` [19]

algorithms, based on dynamic-programming (DP), have consequently been proposed for predicting the minimal free-energy *secondary structure* conformation of an RNA from its sequence. Unfortunately, the assumption of a non-crossing conformation may impede the quality of the prediction, since functionally essential crossing-interactions are found in many families of non-protein-coding RNAs (ncRNAs), as illustrated by Figure 1. For instance, **pseudoknots** (PK), can be found within the RFAM consensus [8] of at least 70 functional families of ncRNAs and are conserved throughout the evolution.

Taking general pseudoknots into account is known to turns RNA minimal-free energy prediction into a rather challenging problem. A pioneering work by Cary, Tabaska *et al* [6,17] considered a simple additive model, associating independent energy contributions to each putative base-pair, and used a $\mathcal{O}(n^3)$ maximal-weighted matching algorithm to extract a minimal free-energy folding. Unfortunately, this energy model is regarded as unrealistic because of its incapacity to capture the interaction of consecutive – stacking – base-pairs, which constitute a primary stabilizing force in RNA folding. Such energy contributions are captured by the **nearest neighbor energy model**, in which the contribution of each base-pair depends on the base-pairing status and partners of its consecutive positions. The hardness of RNA folding assuming a nearest-neighbor energy model was independently established by Lyngsø and Pedersen [12], and Akutsu [1]. Subsequent efforts have therefore focused on providing either parameterized complexity algorithms [11,20], heuristics [4,18] or exact DP schemes for tractable subsets of pseudoknots [16,15].

It is frequent that the complexity of solving any problem in computational biology optimally is tied to the chosen model (e.g. [3]). However, despite a significant amount of research focusing on predicting pseudoknots, the impact of a specific instantiation of the energy model on the computational complexity of RNA folding with pseudoknots has only been partly unexplored. In this extended abstract, we further study the influence of the energy model on the complexity and approximability of RNA folding with unconstrained pseudoknots. In addition to the base-pair and nearest-neighbor models, we consider the **stacking base-pairs energy model**, which captures the dependency between consecutive base-pairs. The computational complexity of RNA folding under this energy model was first studied by Ieong *et al* [9]. They were able to show the NP-hardness of maximizing the number of stacking-pair among the set of planar secondary structures, a restriction of general pseudoknots. This restriction was lifted by Lyngsø [13], who established the hardness of maximizing the number of base-pairs, allowing general types of pseudoknots. Approximation algorithms we also sought, leading to the current best 8/3-approximation $O(n^{10})$-time algorithm reported by Jiang [10]. However all of these works consider a purely combinatorial model, maximizing the number of base-stacking, while the contribution of stacking pairs to the free-energy may vary significantly. It is therefore a natural question to ask to what extent the hardness of folding with pseudoknots is affected by perturbations of the energy model. More generally, understanding what makes the problem hard, and just how hard, could be instrumental to the development of future algorithms, achieving better tradeoffs between sensibility and complexity.

This extended abstract is organized as follows. First we formally define in Section 2 our main problem, along with the different energy models considered. We discuss the NP-hardness of the stacking base-pairs in Section 3, and present an approximation in Section 4. In Section 5, we show the inapproximability of RNA folding with pseudoknots under the nearest-neighbor energy model. Finally Section 6 summarizes the contributions and describes futures lines of research.

2 Problem Statement and Free-Energy Models

Let $\omega \in \{\mathsf{A},\mathsf{C},\mathsf{G},\mathsf{U}\}^*$ be an RNA sequence, and m be a partial matching of the positions in ω, i.e. a set of non overlapping pairs of positions in ω. An **energy model** is a real-valued function E_w that associates a **free-energy** to ω by summing over the contributions of local motifs in the matching. The precise definition of local motifs will depend on the exact energy model.

A low free-energy indicates a stable folding. Furthermore, any free-energy contribution is usually determined up to an additive constant. Therefore one can assume that the contribution to the free-energy of any local motif is negative, with the exception of extremely unfrequent motifs which will be forbidden and assigned $+\infty$ contributions. Let us then rephrase the problem of as an optimization (minimization) problem.

RNA-PK-Fold(E) **problem**
Input: An RNA sequence w.
Ouput: A partial matching m over w, i.e. a set of pairwise disjoint pairs of positions in $[1,|w|]$, which minimizes $E_w(m)$.

The three reference energy models are usually considered:

– **Base-pairs model** $\mathcal{B}$ **[14,6,17]:** Here, local motifs are simply individual base pairs, independently contributing to the free-energy:

$$\mathcal{B}_w(m) = \sum_{(i,j)\in m} \Delta_{\mathcal{B}}(w_i, w_j)$$

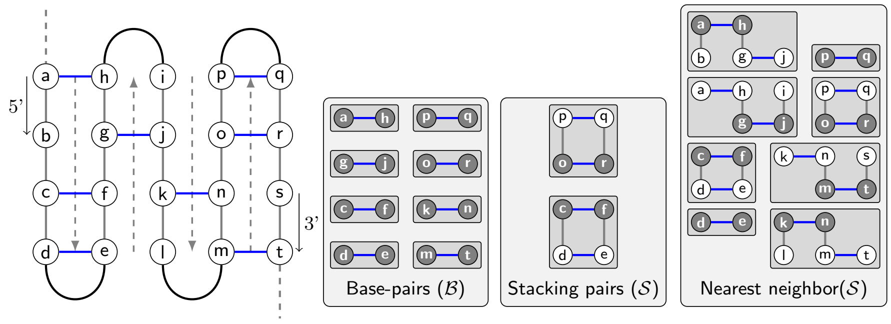

5'
3'
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
Base-pairs ($\mathcal{B}$)
Stacking pairs ($\mathcal{S}$)
Nearest neighbor($\mathcal{S}$)

Fig. 2. Typical picture of a standard pseudoknot/matching (Left) and individual contributions of local motifs to the three energy models considered (Right). Dark nodes indicate the supporting base pair for each motif (i.e. (i,j) pairs in our definition below).

where $\Delta_\mathcal{B} : \{\mathsf{A},\mathsf{C},\mathsf{G},\mathsf{U}\}^2 \to \mathbb{R}^- \cup \{+\infty\}$.

– **Stacking pairs model $\mathcal{S}$ [13,5]:** This model only considers *consecutively nested* pairs as motifs, and disregards isolated pairs:

$$\mathcal{S}_w(m) = \sum_{(i,j),(i+1,j-1)\in m} \Delta_\mathcal{S}(w_i, w_j, w_{i+1}, w_{j-1})$$

where $\Delta_\mathcal{S} : \{\mathsf{A},\mathsf{C},\mathsf{G},\mathsf{U}\}^4 \to \mathbb{R}^- \cup \{+\infty\}$.

– **Nearest-neighbors model $\mathcal{N}$ [12,16]:** This motif definition is even more expressive, allowing different contributions for each base-pair, depending on its bases, the base-pairing status of its consecutive neighbors and own their own partners:

$$\mathcal{N}_w(m) = \sum_{\substack{(i,j)\in m\\ i<j}} \Delta_\mathcal{N}(w_i, w_j, w_{i+1}, w_{j-1}, w_{m_{i+1}}, w_{m_{j-1}})$$

where $\Delta_\mathcal{N}$ is any function $\{\mathsf{A},\mathsf{C},\mathsf{G},\mathsf{U}\}^4 \times \{\mathsf{A},\mathsf{C},\mathsf{G},\mathsf{U},\varnothing\}^2 \to \mathbb{R}^- \cup \{+\infty\}$, m_i denotes the partner of a position i in m (or $\varnothing$ if i is unpaired, while $w_\varnothing \equiv \varnothing$ by convention).

These three models induce different decompositions into motifs for any given structure, as illustrated by Figure 2.

3 NP-Hardness of RNA-PK-Fold($\mathcal{S}$) in Any Non-degenerate Stacking Energy Model

Consider the set of **canonical base-pairs** (A,U), (G,C) and (G,U). A **combinatorial stacking model** $\mathcal{S}^*$ specializes the stacking pairs model by assigning a -1.0 kcal.mol^{-1} contribution to any canonical stacking pair, and $+\infty$ to others. It was showed by Lyngsø [13] that the RNA-PK-Fold($\mathcal{S}^*$) problem is NP-complete, using a reduction from the BIN-PACKING problem.

Here we complement this result by showing its robustness, i.e. the NP-hardness of the problem under a wide class of stacking energy model.

Theorem 1. *Let $\mathcal{S}$ be a stacking energy model that allows (G, C) pairs, and forbids (A, C) and (C, G) pairs. Then RNA-PK-Fold($\mathcal{S}$) is NP-hard.*

Proof. In order to prove the hardness of RNA-PK-Fold($\mathcal{S}$), let us remind the statement of the 3-PARTITION problem:

3-PARTITION problem
Input: A multiset of integral values $X = \{x_i\}_{i=1}^n$ of cardinality $n = 3m$, such that $\sum_{i=1}^n x_i = m \cdot K$ for some $K \in \mathbb{N}$, and $\lfloor K/4 \rfloor < x < \lfloor K/2 \rfloor, \forall x \in X$.
Ouput: `True` if there exists a partition of X into m triplets $((x_{a_j}, x_{b_j}, x_{c_j}))_{j=1}^m$ such that

$$x_{a_j} + x_{b_j} + x_{c_j} = K, \forall j \in [1, m],$$

and `False` otherwise.

From Garey and Johnson [7], it is known that 3-PARTITION is **strongly NP-complete**, i.e. not only is the problem NP-hard, but it remains hard even when the elements of X are upper-bounded by some polynomial function $P(n)$.

Lemma 2. *Let X be a 3-PARTITION instance whose values are bounded by $P(n)$, and w_X be an RNA sequence such that:*

$$w_X = \mathsf{C}^{x_1}\mathsf{A}\mathsf{C}^{x_2}\mathsf{A}\mathsf{C}^{x_3}\mathsf{A}\cdots\mathsf{A}\mathsf{C}^{x_n}\underbrace{\mathsf{A}\mathsf{G}^K\mathsf{A}\mathsf{G}^K\mathsf{A}\cdots\mathsf{A}\mathsf{G}^K}_{m\ times}.$$

There exists a 3-partition of X into equally summing triplets if and only if there exists a solution to RNA-PK-Fold($\mathcal{S}$) over w_X having energy $k = \delta \cdot (K-3) \cdot m$ kcal.mol^{-1} with $\delta := \Delta_\mathcal{S}(\mathsf{C}, \mathsf{G}, \mathsf{C}, \mathsf{G})$.

Let us summarize the argument:

– A matching has minimal free-energy iff any block C^x is entirely paired to some contiguous substring of a single G^K block.
– A matching has minimal free-energy iff every position in every G^K block is connected.
– Any optimal matching thus gives us a mapping between the C^x and G^K blocks, which can be transformed in polynomial-time into a solution to the **3-Partition** problem.

Proof. X **is 3-partitionable** $\Rightarrow \exists m^*$ **such that** $\mathcal{S}_{w_X}(m^*) = \delta \cdot (K-3) \cdot m$**:**

If X is 3-partitionable, then there exists m disjoint triplets $((x_{a_j}, x_{b_j}, x_{c_j}))_{j=1}^m$ whose sum is identically K. If follows that the C^x blocks in w_X can be partitioned into triplets $(\mathsf{C}^{x_{a_j}}, \mathsf{C}^{x_{b_j}}, \mathsf{C}^{x_{c_j}})$ that can form a three-tier *perfect* helix with the

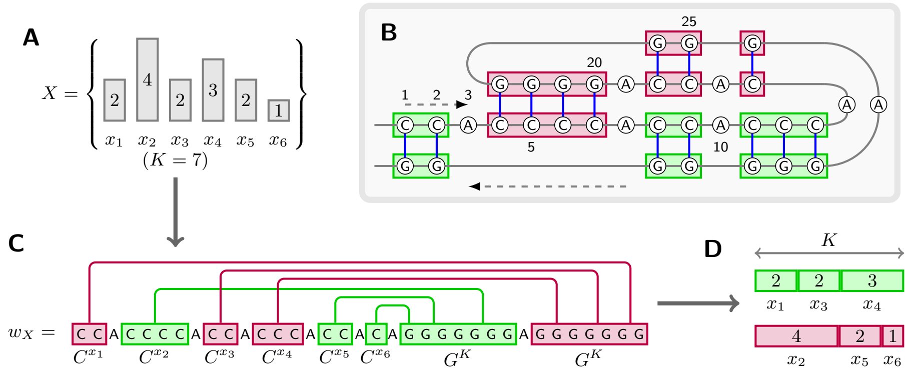


Fig. 3. Illustrating the reduction: Finding a 3-partition of a set of numbers X (**A**) is equivalent to finding a matching for w_X that produces a maximal number of stacking pairs (**C**), from which one easily deduces a set of equally summing triplets (**D**). Such a matching can be represented as a pseudoknotted secondary structure (**B**).

j-th block $\mathsf{G}^K \equiv \mathsf{G}^{x_{a_j}}.\mathsf{G}^{x_{b_j}}.\mathsf{G}^{x_{c_j}}$. By creating x_{a_j} (resp. x_{b_j} and x_{c_j}) nested base-pairs between the block $\mathsf{C}^{x_{a_j}}$ ($\mathsf{C}^{x_{b_j}}$ and $\mathsf{C}^{x_{c_j}}$) and the beginning (resp. middle and ending) of the j-th block G^K, one obtains exactly $(x_{a_j})-1+(x_{b_j})-1+(x_{c_j}-1) = K-3$ stacking pairs. Repeating the operation for each triplet j yields a valid conformation with $(K-3)\cdot m$ stacking $(\mathsf{G},\mathsf{C})/(\mathsf{G},\mathsf{C})$ pairs, and the implication follows.

$\exists m^*$ **such that** $\mathcal{S}_{w_X}(m^*) = \delta\cdot(K-3)\cdot m \Rightarrow X$ **is 3-partitionable:**
Let us remark that the absence of U implies that the only admissible base-pairs are C/G or G/C, arising from interactions between C^x and G^K blocks respectively.

First, let us show that each G^K block contributes to at most $K-3$ stacking pairs, and that this upper-bound cannot be reached unless G^K creates K base-pairs with exactly 3 distinct C^x blocks. Indeed, it is easily seen that any G^K block, connected to b blocks $(\mathsf{C}^{x_{d_1}},\ldots,\mathsf{C}^{x_{d_b}})$ by at least one base-pair, for a total number P of base-pairs, creates at most $P-b$ stacking pairs. This bound is reached when G^K is split into b portions, each forming a perfect helix with the corresponding $\mathsf{C}^{x_{d_i}}$ block. Noting that $x_i < \lfloor K/2\rfloor$ is equivalent to $x_i \leq \lfloor K/2\rfloor - 1$, one has that any connection of G^K with b blocks can therefore create at most $\min(K, b\cdot(\lfloor K/2\rfloor - 1))$ base-pairs. It follows that, for $b=1$ and $b=2$, the maximum number of stackings involving G^K is bounded by $\lfloor K/2\rfloor - 2$ and $2\lfloor K/2\rfloor - 4 \leq K-4$ respectively. For $b\geq 3$, the number of base-pairs is potentially no longer limited by the lack of occurrences of C, but by the K occurrences of G in G^K. It follows that, when $b\geq 3$, the maximum number of stacking pairs is $K-3$, and is reached for $b=3$ when every position in G^K is paired.

Then let us assume the existence of a matching with $(K-3)\cdot m$ stacking pairs. Since $K-3$ is the upper-bound on the number of stacking pairs supported by a given G^K block, then each of the m G^K blocks must achieve this upper-bound. It follows that each G^K block must create a total of exactly K base-pairs with a triplet of blocks $(\mathsf{C}^x_a, \mathsf{C}^x_b, \mathsf{C}^x_c)$. A direct corollary is that every G and C in w_X must be paired.

We have now established that, within any matching having $m \cdot (K-3)$ stacking pairs, each G^K block creates exactly K base-pairs with a triplet of blocks $(\mathsf{C}^x_{a_i}, \mathsf{C}^x_{b_i}, \mathsf{C}^x_{c_i})$. To conclude on the implication, we need to show that each C^x block interacts with a single G^K block, i.e. that the $(\mathsf{C}^x_{a_i}, \mathsf{C}^x_{b_i}, \mathsf{C}^x_{c_i})$ triplets are mutually disjoint. Indeed, if a block C^{x_i} is found in two distinct triplets, then there exists a block C^{x_j} that is not within any triplet (remember that there are $3K$ blocks C^x and K triplets). It follows that at most $m \cdot K - x_j$ base-pairs exist within this matching, which contradicts K base-pairs for every G^K block. Consequently, no C^{x_i} can be present in two distinct triplets, and the triplets are therefore disjoint.

Therefore, the interacting blocks found in a matching having energy $\delta \cdot (K-3) \cdot m$ induce a partition of the $\{\mathsf{C}^{x_i}\}_{i=1}^{3m}$ blocks into m triplets. Furthermore, each $(\mathsf{C}^x_a, \mathsf{C}^x_b, \mathsf{C}^x_c)$ triplet must give rise to K base-pairs, and therefore $x_a+x_b+x_c \geq K$. Since the triplets are disjoint and partition a set of a total $m \cdot K$ occurrences of C, then any excess of C within a triplet implies a lack of C within another, so one has $x_a + x_b + x_c = K$. We conclude that any matching of w_X having energy $\delta \cdot (K-3) \cdot m$ induces the existence of a partition $\{\mathsf{C}^{x_i}\}_{i=1}^{3m}$ blocks into disjoint triplets $(\mathsf{C}^{x_a}, \mathsf{C}^{x_b}, \mathsf{C}^{x_c})$ such that $x_a + x_b + x_c = K$ which, in turn, implies the existence of a 3-partition for X. □

It follows from Lemma 2 that any algorithm for RNA-PK-Fold($\mathcal{S}$) gives an algorithm for the 3-PARTITION problem. Furthermore the length of w_X exactly equals $\sum_{i=1}^n x_i + K \cdot m + 2m - 1 = 2K \cdot m + 2m - 1 \in \mathcal{O}(n^2 \cdot P(n))$ where $P(n)$ is the polynomial upper bound on the value of each x_i. Therefore any polynomial algorithm for RNA-PK-Fold($\mathcal{S}$) gives a polynomial algorithm for the 3-PARTITION problem. Since 3-PARTITION is NP-hard, then so is RNA-PK-Fold($\mathcal{S}$) and Theorem 1 follows. □

4 Approximability of RNA-PK-Fold($\mathcal{S}$) in the Stacking Model

Since objective functions are usually derived experimentally or statistically, it is a natural question to ask whether hard problems can be efficiently approximated. Previous works on the subjetc only considered a combinatorial version of the problem, and the current best algorithm [10] produces a matching whose number of stacking pairs is guaranteed to be at least $3/8 \cdot OPT$, where OPT is the maximal number of stacking pairs in any matching. Unfortunately, this result does not hold for arbitrary-valued stacking energy models, as the free-energy of valid stacking pairs may greatly vary. For instance, the latest version of the Turner model reports a factor ~ 3.6 discrepancy between stacking canonical pairs, bringing the guaranteed approximation ratio down to 1/10. By contrast, we show that RNA-PK-Fold($\mathcal{S}$) can be approximated in polynomial time up to a factor at least 1/5, for any stacking model $\mathcal{S}$.

```
Input   : An RNA sequence $w$
Output: A matching $m$ of non-overlapping pairs of positions
$G = (V, E) \leftarrow ([1, |w| - 1], \varnothing)$;
$M \leftarrow \varnothing$;
foreach $u, v \in V$ do
    if $w_u$ base pairs with $w_{v+1}$ and $w_{u+1}$ base pairs with $w_v$ then
        // Label each edge with its weight/energy
        $E \leftarrow E \cup (u, v, -\Delta_{\mathcal{S}}(w_u, w_{v+1}, w_{u+1}, w_v))$;
    end
end
$m' \leftarrow$ MaxWeightedMatching$(G)$;
foreach $(u, v) \in m'$ sorted by increasing value $\Delta_{\mathcal{S}}(w_u, w_{v+1}, w_{u+1}, w_v)$ do
    if $\forall (u', v') \in m, \{u', v'\} \cap \{u, v+1, u+1, v\} = \varnothing$ or
    $(u', v') \in \{(u, v+1), (u+1, v)\}$ then
        $m \leftarrow m \cup \{(u, v+1), (u+1, v)\}$;
    end
end
return $m$
```

Algorithm 1: A 5-approximation for any stacking energy model.

Theorem 3. *In any stacking energy model, RNA-PK-Fold($\mathcal{S}$) $\in APX$, and can be approximated in polynomial time within a factor at least* $1/5$.

Proof. To prove the approximability of RNA-PK-Fold($\mathcal{S}$), let us consider Algorithm 1. This algorithm contracts consecutive positions in the RNA sequence as vertices, and adds an edge, weighted according to the energy function, between any pair of compatible positions. Computing a maximal weighted matching on this graph gives a set of stacking-pair which is not necessarily a valid matching, since distinct pairs of stacking pairs may induce more than a single partners for a given position. Therefore the algorithm considers the returned stacking pairs in decreasing order, and only retains the stacking pairs that do not conflict with the current selection of stacking-pairs.

Now let m^* be the optimal matching for the given RNA string w, m' be the maximal matching over G, and m be the matching finally returned by the algorithm. Let us remark that m' induces a set of matched pairs over w that does not strictly constitutes a matching, as some position may be matched twice. Nevertheless let us write $\mathcal{S}_w(m')$ as a shorthand for the total energy of m', obtained by summing over the stacking pairs induced by m'. Any matching can be decomposed as a set of stacking pairs (leaving a set of isolated, non-contributive, base-pairs), hence one has $\mathcal{S}_w(m') \leq \mathcal{S}_w(m^*) \leq 0$. Any edge (i, j) in m' may conflict with at most 4 other, adjacent, stacking-pairs. Furthermore, the algorithm considers the edges in m' by decreasing contribution, so the stacking pairs induced by any edge (i, j) in m' may only conflict with four stacking pairs having (negative) contribution of smaller absolute value. Discarding these competitors guarantees that at least $\frac{1}{5}$ of the total energy of m' is retained in m, i.e. $\mathcal{S}_w(m) \leq \frac{1}{5}\mathcal{S}_w(m') \leq 0$, and one therefore concludes that $\frac{\mathcal{S}_w(m)}{\mathcal{S}_w(m^*)} \geq \frac{1}{5}$. □

Remark that the actual approximation ratio achieved by Algorithm 1 might be better than 1/5, even in the worst-case scenario. However, this crude upper-bound already establishes the approximability of the problem, nicely contrasting with our upcoming inaproximability result for the nearest-neighbor version of the problem.

5 Inapproximability of RNA-PK-Fold($\mathcal{N}$) in the Nearest-Neighbor Energy Model

The stacking model, considered in the above sections, makes the prediction of RNA structure NP-Hard, yet remains approximable in general. By contrast, let us show that RNA-PK-Fold($\mathcal{N}$), the nearest-neighbor version of the problem, is non-approximable. More precisely, let us show the stronger property that, unless $P = NP$, there is no polynomial-time algorithm that guarantees to find a matching whose free-energy approximates that of the optimal matching up to a strictly positive factor $r(n)$.

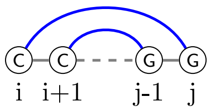
Theorem 4. *There exists instances of the nearest-neighbor model such that RNA-PK-Fold($\mathcal{N}$) $\notin$ APX.*

Proof. Let us briefly outline our proof strategy. We encode any set of numbers X as a string w, having length polynomial on the sum of values in X, and whose matchings are either forbidden ($+\infty$ free-energy), empty (0 free-energy), or have negative energy. Focusing on the latter category, we show that any negative energy matching can be turned, in polynomial-time, into a solution to the 3-PARTITION problem. It follows that any polynomial-time algorithm that guarantees a positive-ratio approximation, thereby producing a matching having negative free-energy anytime such a matching exists, immediately yields a polynomial-time algorithm for the 3-PARTITION problem. The NP-hardness of this problem allows us to conclude on the hardness of approximating RNA-PK-Fold($\mathcal{N}^*$), within any positive ratio, in the nearest-neighbor energy problem.

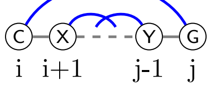
Let us consider the 3-PARTITION problem, fully defined in Section 3. For any instance $X = \{x_i\}_{i=1}^{3m}$ of the problem, let us consider the following RNA sequence:

$$w = \mathsf{C}^{x_1}\mathsf{A}\mathsf{C}^{x_2}\mathsf{A}\cdots\mathsf{A}\mathsf{C}^{x_{3m}}\mathsf{A}\underbrace{\mathsf{G}^{K}\mathsf{U}\mathsf{G}^{K}\mathsf{U}\cdots\mathsf{G}^{K}\mathsf{U}}_{m\text{ times}}\mathsf{U}^{2m}$$

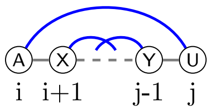
Moreover let us consider a nearest-neighbor energy model $\mathcal{N}^*$, defined by a function $\Delta^*_{\mathcal{N}}$ such that:

$\Delta^*_{\mathcal{N}}:$ (A) 

$\longrightarrow -1,\quad \forall i < j,$

(B) 

$\longrightarrow -1,\quad \forall i < j, \forall X \neq C, \forall Y,$
($i+1$ and $j-1$ **must both** base-pair somewhere, possibly together)

(C) 

$\longrightarrow -1,\quad \forall i < j, \forall (X,Y),$
($i+1$ and $j-1$ **must both** base-pair somewhere, possibly together)

(D) Any other motif $\longrightarrow +\infty,\ \forall i < j.$

Lemma 5. *Let X be a 3-PARTITION instance whose values are bounded by $P(n)$. Then the following statements are equivalent:*

– *There exists a 3-partition of X into m triplets of equal sum.*
– *There exists a matching of strictly negative energy over w under $\mathcal{N}^*$.*

Proof. **X is 3-PARTITIONABLE $\Rightarrow \exists m^*$ such that $\mathcal{N}^*_w(m^*) < 0$:** Since X is 3-PARTITIONABLE, then there exists a partition of X into m disjoint triplets $((x_{a_j}, x_{b_j}, x_{c_j}))_{j=1}^m$ whose sum are identically K. Consider the matching that pairs each G^K block with one of the triplet of blocks $\mathsf{C}^{x_{a_j}}$, $\mathsf{C}^{x_{b_j}}$ and $\mathsf{C}^{x_{c_j}}$, creating nested sequences of base-pairs, and completed with $3 \cdot m$ (A, U) unconstrained base-pairs over the remaining positions. Clearly, all the positions are involved in a base-pair, and consecutive $\mathsf{CC} \cdots \mathsf{GG}$ are nested as required by energy rule (A). Therefore, any base-pair falls within the scope of energy rules (A), (B) or (C), and the final energy of the matching is $\mathcal{N}^*_w(m^*) = -m \cdot (K+3) < 0$.

$\exists m^*$ such that $\mathcal{N}^*_w(m^*) < 0 \Rightarrow X$ is 3-PARTITIONABLE: Let us start by proving that, within an energy model $\mathcal{N}^*$, any matching of w having negative energy is a perfect matching, i.e. every position in the matching is paired. Since $\mathcal{N}^*$ only allows (C, G) and (A, U) pairs, therefore any valid (finite, negative contribution) base-pair (i, j) must involve a position in the left half of w (C or A) and a position in its right half (G or U), i.e. such that $i \leq m \cdot (K+3) < j$. In order to be valid, (i, j) must also be in a context where $i+1$ (resp. $j-1$) is paired to $j' \geq m \cdot (K+3)$ (resp. $i' < m \cdot (K+3)$). The same argument applies to $(i+1, j')$ and $(i', j-1)$, and one easily shows by induction that any matching featuring a base-pair (i, j) has infinite energy unless every position in $[i, j]$ is paired. It follows that any matching having negative energy is perfect on some interval $[a, b]$, $a \leq m \cdot (K+3) < b$, and leaves the remaining positions unpaired.

Now let us consider which bounds for the interval $[a, b]$ are compatible with a negative energy. Let us denote by $w_{[a,b]}$ the $[a, b]$ factor in a sequence w, and by $|w|_t$ the number of occurrences of some letter t in w, then one has $|w_{[a,b]}|_A = |w_{[a,b]}|_U$ and $|w_{[a,b]}|_\mathsf{C} = |w_{[a,b]}|_\mathsf{G}$. Observe that, since $x_i < K/2$, one has $\frac{|w_{[a,b]}|_A}{|w_{[a,b]}|_C} \leq \frac{1}{1+K/2}$. Furthermore, if b falls before the final run U^{2m}, then $b < m \cdot (2K+4)$ and one has $\frac{1}{1+K} \leq \frac{|w_{[a,b]}|_U}{|w_{[a,b]}|_G}$. It follows that $\frac{|w_{[a,b]}|_\mathsf{A}}{|w_{[a,b]}|_\mathsf{C}} < \frac{|w_{[a,b]}|_\mathsf{U}}{|w_{[a,b]}|_\mathsf{G}}$, i.e. the matching cannot be perfect on $[a,b]$, and its energy cannot be negative. We are then left to consider the case where $m \cdot (2K+4) \leq b \leq |w_X|$. In such a case, one has $|w_{[a,b]}|_\mathsf{G} = m \cdot K$ and one has $a = 1$. Indeed any greater value a would lead to less than $\sum x_i = m \cdot K$ copies of C, and some G would be left alone. Remark that $|w_{[1,b]}|_\mathsf{A} = 3m$, so one must have $b = |w|$, from which we conclude that any matching having negative energy is perfect, i.e. base-pairs every position.

Let us finally show that a 3-PARTITION of X can be retrieved from a matching having negative energy. Remind that energy rule (A) forces two consecutive occurrences of C to pair with consecutive occurrences of G. This property extends transitively, and any C^{x_i} block in w must therefore be entirely connected to a single G^K block. Since a matching of negative energy is perfect, then all the positions in a G^K block must be base-paired. Two C^{x_i} blocks are not sufficient ($x_i < K/2$) to saturate a G^K block, and four blocks would be too large ($x_i > K/4$), violating the constraint that each block must be entirely paired to a single G^K block. Therefore a triplet $(\mathsf{C}^{x_{a_i}}, \mathsf{C}^{x_{b_i}}, \mathsf{C}^{x_{c_i}})$ blocks is in total interaction with each G^K block, and the corresponding values (a_i, b_i, c_i) constitute a 3-PARTITION of X. □

From Lemma 5, one knows that the existence of a 3-PARTITION for X can be derived from the existence of a matching of w having negative energy under $\mathcal{N}^*$. Now assume there exists a polynomial-time algorithm $\mathcal{A}$ that guarantees an $r(n) > 0$ approximation ratio. Then $\mathcal{A}$ would produce a matching m such that $\mathcal{N}^*_w(m) = \mathcal{N}^*_w(m^*)/r(n)$, for m^* the optimal matching. In particular, $\mathcal{A}$ would produce a matching having negative energy anytime such a matching exists. One could then decide, in polynomial time, the 3-partitionability of any set X. Since the decision version of 3-PARTITION is NP-Hard, then there is no such algorithm unless $P = NP$. □

6 Conclusion/Perspectives

We considered the influence of the energy model on the computational complexity of RNA folding with general pseudoknots. In the simplest base-pair model, the problem is exactly equivalent to finding a maximal weighted matching in the graph of compatible positions, and can be solved in $\Theta(n^3)$ [17]. By contrast, it was previously established that the more expressive nearest-neighbor model made the problem NP-Hard [1,12]. We completed this result by showing that this problem is actually inapproximable within any ratio. Turning to a less expressive – yet realistic – stacking energy model, we have showed that, although NP-hard, the problem could be approximated in polynomial time, at least up to a $\frac{1}{5}$ approximation ratio.

Quite nicely, a similar approach could be used to refine the computational complexity of RNA-RNA interaction prediction. Already proven NP-complete by Alkan *et al* [2], it can be verified that our approximation algorithm achieves the same ratio for RNA-RNA interactions. Furthermore, our NP-hardness and inapproximability results consider bi-partite strings, for which an algorithm for the RNA-RNA interaction problem, suitably parameterized, would yield the same matching as an algorithm for RNA folding with general pseudoknots.

These results show a difference in essence between the nearest-neighbor and the stacking models, which could serve as a starting point for a design of practical (approximation) algorithms for the stacking version of the problem. To that purpose, we plan to complement this study by investigating the existence of a polynomial-time approximation scheme for the problem. Another direction for complementing this study would consider the impact of the energy model on the parameterized-complexity of the problem.

Acknowledgement. This work was supported by the French ANR MAGNUM `ANR 2010 BLAN 0204` grant (YP) and by an INRIA Postdoc Program (SS).

References

1. Akutsu, T.: Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. Discrete Appl. Math. 104(1-3), 45–62 (2000)
2. Alkan, C., Karakoç, E., Nadeau, J.H., Şahinalp, S.C., Zhang, K.: RNA-RNA Interaction Prediction and Antisense RNA Target Search. In: Miyano, S., Mesirov, J., Kasif, S., Istrail, S., Pevzner, P.A., Waterman, M. (eds.) RECOMB 2005. LNCS (LNBI), vol. 3500, pp. 152–171. Springer, Heidelberg (2005)
3. Ashley, M.V., Berger-Wolf, T.Y., Chaovalitwongse, W., Dasgupta, B., Khokhar, A., Sheikh, S.: On Approximating an Implicit Cover Problem in Biology. In: Goldberg, A.V., Zhou, Y. (eds.) AAIM 2009. LNCS, vol. 5564, pp. 43–54. Springer, Heidelberg (2009)
4. Bindewald, E., Kluth, T., Shapiro, B.A.: Cylofold: secondary structure prediction including pseudoknots. Nucleic Acids Research 38(suppl. 2), W368–W372 (2010)
5. Bon, M.: Prédiction de structures secondaires d'ARN avec pseudo-noeuds. Ph.D. thesis, Ecole Polytechnique (September 2009)
6. Cary, R.B., Stormo, G.D.: Graph-theoretic approach to RNA modeling using comparative data. In: Proceedings International Conference on Intelligent Systems for Molecular Biology, vol. 3, pp. 75–80 (1995)
7. Garey, M.R., Johnson, D.S.: Complexity results for multiprocessor scheduling under resource constraints. SIAM Journal on Computing 4(4), 397–411 (1975)
8. Griffiths-Jones, S., Bateman, A., Marshall, M., Khanna, A., Eddy, S.R.: Rfam: an RNA family database. Nucleic Acids Research 31(1), 439–441 (2003)
9. Ieong, S., Kao, M.-Y., Lam, T.W., Sung, W.-K., Yiu, S.-M.: Predicting RNA secondary structures with arbitrary pseudoknots by maximizing the number of stacking pairs. Journal of Computational Biology 10(6), 981–995 (2003)
10. Jiang, M.: Approximation algorithms for predicting RNA secondary structures with arbitrary pseudoknots. IEEE/ACM Trans. Comput. Biology Bioinform. 7(2), 323–332 (2010)

11. Liu, C., Song, Y., Shapiro, L.: RNA Folding Including Pseudoknots: A New Parameterized Algorithm and Improved Upper Bound. In: Giancarlo, R., Hannenhalli, S. (eds.) WABI 2007. LNCS (LNBI), vol. 4645, pp. 310–322. Springer, Heidelberg (2007)
12. Lyngsø, R.B., Pedersen, C.N.: RNA pseudoknot prediction in energy-based models. J. Comput. Biol. 7(3-4), 409–427 (2000)
13. Lyngsø, R.B.: Complexity of Pseudoknot Prediction in Simple Models. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 919–931. Springer, Heidelberg (2004)
14. Nussinov, R., Jacobson, A.: Fast algorithm for predicting the secondary structure of single-stranded RNA. Proc. Natl. Acad. Sci. U S A 77, 6903–6913 (1980)
15. Reidys, C.M., Huang, F.W.D., Andersen, J.E., Penner, R.C., Stadler, P.F., Nebel, M.E.: Topology and prediction of RNA pseudoknots. Bioinformatics 27(8), 1076–1085 (2011)
16. Rivas, E., Eddy, S.: A dynamic programming algorithm for RNA structure prediction including pseudoknots. J. Mol. Biol. 285, 2053–2068 (1999)
17. Tabaska, J.E., Cary, R.B., Gabow, H.N., Stormo, G.D.: An RNA folding method capable of identifying pseudoknots and base triples. Bioinformatics 14(8), 691–699 (1998)
18. Theis, C., Janssen, S., Giegerich, R.: Prediction of RNA Secondary Structure Including Kissing Hairpin Motifs. In: Moulton, V., Singh, M. (eds.) WABI 2010. LNCS, vol. 6293, pp. 52–64. Springer, Heidelberg (2010)
19. Yang, H., Jossinet, F., Leontis, N., Chen, L., Westbrook, J., Berman, H., Westhof, E.: Tools for the automatic identification and classification of rna base pairs. Nucleic Acids Research 31(13), 4250–4263 (2003)
20. Zhao, J., Malmberg, R., Cai, L.: Rapid ab initio prediction of RNA pseudoknots via graph tree decomposition. Journal of Mathematical Biology 56, 145–159 (2008)

Finding Longest Common Segments in Protein Structures in Nearly Linear Time

Yen Kaow Ng[1], Hirotaka Ono[2], Ling Ge[3], and Shuai Cheng Li[4,⋆]

[1] Department of Computer Science, Faculty of Information and Communication Technology, Universiti Tunku Abdul Rahman, Malaysia
ykng@utar.edu.my

[2] Department of Economic Engineering, Faculty of Economics, Kyushu University, Japan
hirotaka@en.kyushu-u.ac.jp

[3] College of Business, University of Massachusetts Dartmouth, North Dartmouth, USA
lge@umassd.edu

[4] Department of Computer Science, City University of Hong Kong, Hong Kong
shuaicli@cityu.edu.hk

Abstract. The Local/Global Alignment (Zemla, 2003), or LGA, is a popular method for the comparison of protein structures. One of the two components of LGA requires us to compute the longest common contiguous segments between two protein structures. That is, given two structures $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$ where a_k, $b_k \in \mathbb{R}^3$, we are to find, among all the segments $f = (a_i, \ldots, a_j)$ and $g = (b_i, \ldots, b_j)$ that fulfill a certain criterion regarding their similarity, those of the maximum length. We consider the following criteria: (1) the root mean square deviation (RMSD) between f and g is to be within a given $t \in \mathbb{R}$; (2) f and g can be superposed such that for each k, $i \leq k \leq j$, $\|a_k - b_k\| \leq t$ for a given $t \in \mathbb{R}$. We give an algorithm of $O(n \log n + n\boldsymbol{l})$ time complexity when the first requirement applies, where $\boldsymbol{l}$ is the maximum length of the segments fulfilling the criterion. We show an FPTAS which, for any $\epsilon \in \mathbb{R}$, finds a segment of length at least $\boldsymbol{l}$, but of RMSD up to $(1+\epsilon)t$, in $O(n \log n + n/\epsilon)$ time. We propose an FPTAS which for any given $\epsilon \in \mathbb{R}$, finds all the segments f and g of the maximum length which can be superposed such that for each k, $i \leq k \leq j$, $\|a_k - b_k\| \leq (1+\epsilon)t$, thus fulfilling the second requirement approximately. The algorithm has a time complexity of $O(n \log^2 n/\epsilon^5)$ when consecutive points in A are separated by the same distance (which is the case with protein structures).

1 Introduction

Scoring functions for the purpose of comparing the similarity between two given 3-dimensional structures are commonly used in protein science. They serve at least two important functions in protein structure prediction: (1) for the evaluation of constructed models against the native structures, and (2) for the selection

⋆ Corresponding author.

of consensus structures out of a collection of similar structures [13,15]. A traditional candidate for such scoring functions is the root mean square deviation (RMSD) [8,9]. However, the use of the RMSD for the first purpose suffers from a few shortcomings [2,7,3]. For example, the RMSD measure tends to overestimate the difference between two structures when only a small part of the structure differs, but differs very significantly. The significance of RMSD values also differs on structures of different lengths, since it is in general less likely, and therefore more significant, for longer structures to have a low RMSD.

These shortcomings have resulted in the proposal of other similarity measures [6,10,12]. In this paper we are interested in the Local/Global Alignment (LGA) [16], a method for protein model comparison which has become very popular in the protein structure prediction community. The LGA method consists of two computations: the Global Distance Test (GDT) and the Longest Continuous Segments (LCS). The GDT can be computed in $O(n^7)$ time using methods from algorithmic geometry [4], and it also allows a practical PTAS [11]. Less is known of the computational complexity of the LCS. To our best knowledge, no algorithm with theoretical bounds (such as runtime complexity and approximation ratio) has been published on the LCS.

Given two protein structures $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ where the elements a_k, $b_k \in \mathbb{R}^3$, the LCS uses the longest common contiguous residues of A and B with an RMSD less than or equal to a cutoff threshold t, in evaluating how similar the structures are. Hence the LCS requires us to find all the segments $f = (a_i, a_{i+1}, \ldots, a_j)$ and $g = (b_i, b_{i+1}, \ldots, b_j)$, $1 \le i \le j \le n$, of the longest length where the RMSD between f and g is within t. A trivial algorithm is to enumerate all $O(n^2)$ segments and analyze each in $O(n)$ time, hence requiring $O(n^3)$ time. We show in this paper that the problem can be solved in $O(n \log n + n\boldsymbol{l})$ time, where $\boldsymbol{l}$ is the length of the longest segments that fulfill the criterion. Furthermore, for any $\epsilon \in \mathbb{R}$ one can obtain, in $O(n \log n + n/\epsilon)$ time, a segment of length at least $\boldsymbol{l}$ but with an RMSD of up to $(1 + \epsilon)t$.

An alternative formulation of the LCS is to find $f = (a_i, a_{i+1}, \ldots, a_j)$ and $g = (b_i, b_{i+1}, \ldots, b_j)$, $1 \le i \le j \le n$, of the longest length that can be superposed such that for each k, $i \le k \le j$, $\|a_k - b_k\| \le t$ for a given $t \in \mathbb{R}$. The $O(n^7)$ algorithm for the GDT mentioned above can be adapted to solve this problem. However, the high runtime would render the algorithm impractical for routine use in protein structure prediction. We show that for protein structures, where consecutive residues are separated by the same distance, there is an FPTAS of $O(n \log^2 n/\epsilon^5)$ time complexity, which computes an approximate solution where for each k, $i \le k \le j$, $\|a_k - b_k\| \le (1 + \epsilon)t$, for any specified $\epsilon \in \mathbb{R}$. Theoretically, this algorithm demonstrates a non-trivial use of a natural generalization of the *radial pair* (or *radial axis*) introduced in [11].

2 Preliminary

In this paper $\mathcal{R}$ denotes the set of all rotation matrices, and $\mathcal{T}$ denotes the set of all translation vectors. A protein structure A is modeled as a sequence of three dimensional points, denoted as $(a_1, a_2, \ldots, a_n)$, where $a_i \in \mathbb{R}^3$ for $1 \leq i \leq n$. For integers i, j, $1 \leq i \leq j \leq n$, $A[i, j]$ denotes the segment $(a_i, a_{i+1}, \ldots, a_j)$. For a segment $f = (a_i, a_{i+1}, \ldots, a_j)$, $S(f)$ denotes the set $\{a_i, a_{i+1}, \ldots, a_j\}$. We write $f \subseteq A$ iff $f = A[i, j]$ for some i, j, $1 \leq i \leq j \leq n$.

Given two structures $A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_n)$, the RMSD of A and B is defined to be

$$\mathrm{RMSD}(A, B) = \min_{R \in \mathcal{R}, T \in \mathcal{T}} \sqrt{\frac{\sum_{i=1}^{n} \|Ra_i - b_i - T\|^2}{n}}.$$

For $A = (a_1, \ldots, a_n)$, $B = (b_1, \ldots, b_n)$, and i, j, $1 \leq i \leq j \leq n$, we call $B[i, j]$ the *corresponding segment of* $A[i, j]$.

For a given $t \in \mathbb{R}$, we define a *longest contiguous segment of A and B under the RMSD*, denoted $\mathrm{LCS}_{\mathrm{RMSD}}(A, B, t)$, to be a segment of the longest length among all $A[i, j]$, $1 \leq i \leq j \leq n$, which fulfills $\mathrm{RMSD}(A[i, j], B[i, j]) \leq t$. $\mathrm{ALCS}_{\mathrm{RMSD}}(A, B, t)$ denotes the set of all $\mathrm{LCS}_{\mathrm{RMSD}}(A, B, t)$.

In this paper we also define a *longest contiguous segment of A and B under the bottleneck distance*, $\mathrm{LCS}_{\mathrm{dist}}(A, B, t)$, to be a segment of the longest length among all $f \subseteq A$ where under some superposition, each residue a_k in f fulfills $\|a_k - b_k\| \leq t$. $\mathrm{ALCS}_{\mathrm{dist}}(A, B, t)$ denotes the set of all $\mathrm{LCS}_{\mathrm{dist}}(A, B, t)$.

Solving $\mathrm{ALCS}_{\mathrm{RMSD}}$ and solving $\mathrm{ALCS}_{\mathrm{dist}}$ require different strategies for a few reasons. In solving $\mathrm{ALCS}_{\mathrm{RMSD}}$, the RMSD can be very efficiently computed due to the existence of efficient closed-form solutions [8,1,14]. However, for $\mathrm{ALCS}_{\mathrm{dist}}$, approximations have to be considered since we know of no practically efficient way to exactly determine if there is a superposition that matches corresponding points in two segments to within a threshold distance. As far as we know, the algorithm closest to solving this problem exactly has a high runtime of $O(n^7)$ [4].

On the other hand, $\mathrm{ALCS}_{\mathrm{dist}}$ is easier to deal with in the following way. If the residues in two corresponding segments cannot be matched to within distance t, all the segments which extend these segments will not be matchable to within distance t as well. These extended segments need not be evaluated when solving $\mathrm{ALCS}_{\mathrm{dist}}$. A similar strategy cannot be used in the case of $\mathrm{ALCS}_{\mathrm{RMSD}}$, since an extension of segments f and g where $\mathrm{RMSD}(f, g) > t$ may have an RMSD within t, due to the $1/n$ factor in the RMSD measure.

3 Algorithms for $\mathrm{ALCS}_{\mathrm{RMSD}}$

As mentioned, closed form solutions exist for computing the RMSD. We use the well-known Kabsch algorithm, which has an $O(n)$ time complexity [8]. Using the algorithm, a simple way to compute $\mathrm{ALCS}_{\mathrm{RMSD}}(A, B, t)$ is possible: Compute the RMSDs of all the contiguous segments, of all possible lengths, and from the segments with RMSDs not exceeding t, obtain those of the longest length. Since there are only $O(n^2)$ segments in total, this completes in $O(n^3)$ time.

For our algorithms, we first claim the following.

Lemma 1. *There exists an algorithm which, given $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, completes a pre-computation step in $O(n)$ time and thereafter, for any i, j, $1 \leq i \leq j \leq n$, computes RMSD$((a_i, \ldots, a_j), (b_i, \ldots, b_j))$ in $O(1)$ time.*

Proof. We explain how each of the steps in the Kabsch algorithm can be computed in $O(1)$ time through the use of pre-computed values. Given two segments $P = (a_1, a_2, \ldots, a_n)$ and $Q = (b_1, b_2, \ldots, b_n)$, the Kabsch algorithm for computing RMSD(P, Q) consists of the steps (S1)-(S5) below. Here, each coordinate is treated as a 3-dimensional vector, while P and Q are treated as $n \times 3$ matrices.

(S1) Compute the centroids of P and Q respectively.
(S2) Translate P or Q so that their centroids coincide.
(S3) Compute $P^T Q$. This gives us a 3×3 matrix C.
(S4) Compute the singular value decomposition, VSW^T say, of C.
(S5) RMSD(P, Q) is then computed in $O(1)$ time from V, S, and W.

For i, $1 \leq i \leq n$, we pre-compute the values

– $A_i = \sum_{k=1}^{i} a_k$.
– $B_i = \sum_{k=1}^{i} b_k$.
– $s_i^{u,v} = \sum_{k=1}^{i} a_k^u b_k^v$ for each u, v, $1 \leq v, u \leq 3$, where a_k^u denotes the u-th component of a_k, and b_k^v denotes the v-th component of b_k.

It is clear that this pre-computation can be completed in $O(n)$ time. Now, given the segments $P' = P[i, j]$ and $Q' = Q[i, j]$ where $1 \leq i \leq j \leq n$,

– The centroid of P' is $\sum_{k=i}^{j} a_k/(j - i + 1)$. This can be computed as $(A_j - A_{i-1})/(j - i + 1)$. The centroid of Q' can be computed similarly. Hence (S1) can be computed in $O(1)$ time.

– The (u, v)-th component of $P'^T Q'$ is $\sum_{k=i}^{j} a_k^u b_k^v$. This can be computed as $s_j^{u,v} - s_{i-1}^{u,v}$. Hence (S3) can be computed in $O(1)$ time.

– To compute step (S2), it suffices that we reflect the translation's effect in the matrix C in step (S3). The (u, v)-th element of $C' = P'^T Q'$ is $\sum_{k=i}^{j} a_k^u b_k^v$. Suppose P' is to be translated from its original coordinates of P by $(\sigma^1, \sigma^2, \sigma^3)$. The (u, v)-th element of C' is then

$$\sum_{k=i}^{j} (a_k^u - \sigma^u) b_k^v = \sum_{k=i}^{j} a_k^u b_k^v - \sigma^u \sum_{k=i}^{j} b_k^v = s_j^{u,v} - s_{i-1}^{u,v} - \sigma^u (B_j^v - B_{i-1}^v).$$

where B_j^v denotes the v-th component of B_j.

Finally, (S4) is traditionally solved in $O(1)$ time using an analytical approach, that is, by solving the resultant cubic equation from $\det(C - \lambda I) = 0$. ∎

Table 1 shows an algorithm for computing ALCS$_{\text{RMSD}}(A, B, t)$. By Lemma 1, steps (2.2)-(2.2.2) can be computed in $O(n)$ time. Since they are repeated exactly n times, the algorithm completes in $O(n^2)$ time.

We now show an algorithm to estimate $\boldsymbol{l}$, the length of the segments in ALCS$_{\text{RMSD}}(A, B, t)$. Our estimation ℓ of $\boldsymbol{l}$ will fulfill $\ell \leq \boldsymbol{l} \leq 6\ell$. This will allow us to restrict the values of l to examine in step (2) in the algorithm in Table 1 to only ℓ, $\ell + 1$, …, 6ℓ, thus bringing that algorithm's runtime to $O(n\boldsymbol{l})$. Our algorithm for finding ℓ is presented in Table 2. To show its correctness, we first state the following lemma.

Lemma 2. *For any two structures P and Q of length l, if RMSD$(P, Q) \leq t$ for $t \in \mathbb{R}$, then for any x, $1 \leq x < l$, either RMSD$(P[1, x], Q[1, x]) \leq t$ or RMSD$(P[x + 1, l], Q[x + 1, l]) \leq t$.*

Table 1. Algorithm for computing $\mathrm{ALCS}_{\mathrm{RMSD}}(A, B, t)$

```
Input:   Structures $A = (a_1, \ldots, a_n)$, $B = (b_1, \ldots, b_n)$, and RMSDthreshold $t \in \mathbb{R}$.
Output: $\mathrm{ALCS}_{\mathrm{RMSD}}(A, B, t)$.
(1)      Set $\mathrm{ALCS}_l \leftarrow \emptyset$ for $l$ from 1 to $n$.
(2)      For $l \leftarrow 1$ to $n$ (l specifies the length of the segments to examine),
(2.2)        For $i \leftarrow 1$ to $n - l + 1$, (i specifies the position to obtain the segment)
(2.2.1)          If $\mathrm{RMSD}(A[i, i+l-1], B[i, i+l-1]) \le t$,
(2.2.2)              Add $A[i, i+l-1]$ to $\mathrm{ALCS}_l$.
(3)      Find maximum $\ell$ where $\mathrm{ALCS}_\ell \neq \emptyset$ and return $\mathrm{ALCS}_\ell$.
```

Proof. Let R and T be such that

$$\mathrm{RMSD}(P, Q) = \sqrt{\frac{\sum_{i=1}^{l} \|Ra_i - b_i - T\|^2}{l}}.$$

By definition,

$$x\mathrm{RMSD}(P[1, x], Q[1, x])^2 \le \sum_{i=1}^{x} \|Ra_i - b_i - T\|^2, \tag{1}$$

and

$$(l - x)\mathrm{RMSD}(P[x+1, l], Q[x+1, l])^2 \le \sum_{i=x+1}^{l} \|Ra_i - b_i - T\|^2. \tag{2}$$

Summing (1) and (2) gives

$$x\mathrm{RMSD}(P[1, x], Q[1, x])^2 + (l-x)\mathrm{RMSD}(P[x+1, l], Q[x+1, l])^2 \le \sum_{i=1}^{l} \|Ra_i - b_i - T\|^2.$$

Dividing both sides by l, we have

$$\frac{1}{l}\left(x\mathrm{RMSD}(P[1, x], Q[1, x])^2 + (l-x)\mathrm{RMSD}(P[x+1, l], Q[x+1, l])^2\right)$$

$$\le \frac{\sum_{i=1}^{l} \|Ra_i - b_i - T\|^2}{l} = \mathrm{RMSD}(P, Q)^2 \le t^2. \tag{3}$$

Now suppose both $\mathrm{RMSD}(P[1, x], Q[1, x]) > t$ and $\mathrm{RMSD}(P[x+1, l], Q[x+1, l]) > t$. Then

$$\frac{1}{l}\left(x\mathrm{RMSD}(P[1, x], Q[1, x])^2 + (l-x)\mathrm{RMSD}(P[x+1, l], Q[x+1, l])^2\right)$$

$$> \frac{1}{l}\left(xt^2 + (l-x)t^2\right) = t^2,$$

which contradicts (3). ∎

Table 2. Algorithm for estimating the length of the elements in $\mathrm{ALCS}_{\mathrm{RMSD}}(A, B, t)$

```
Input:   Structures $A = (a_1, \ldots, a_n)$, $B = (b_1, \ldots, b_n)$, and threshold $t \in \mathbb{R}$.
Output:  An estimation $\ell$ of the length $\boldsymbol{l}$ of the elements in $\mathrm{ALCS}_{\mathrm{RMSD}}(A, B, t)$,
         where $\ell \leq \boldsymbol{l} \leq 6\ell$.
(1)    Set $\ell \leftarrow \lfloor n/2 \rfloor$.
(2)    Partition $A$ into $\lfloor n/\ell \rfloor + 1$ contiguous segments of length $\ell$, $P_1^\ell, P_2^\ell, \ldots$ say.
(3)        (The last segment may not have the same length as the other segments.)
(4)    For each $i$, $1 \leq i \leq \lfloor n/\ell \rfloor - 2$, find if there exists either
(4.1)      (i) $P' \subseteq P_i^\ell P_{i+1}^\ell$ ending in $P_{i+1}^\ell$ (i.e. $P_{i+1}^\ell$ is the suffix of $P'$), or
(4.2)      (ii) $P' \subseteq P_{i+2}^\ell P_{i+3}^\ell$ beginning in $P_{i+2}^\ell$ (i.e. $P_{i+2}^\ell$ is the prefix of $P'$),
(4.3)    with $\mathrm{RMSD}(P', Q') \leq t$, where $Q' \subseteq B$ is the corresponding segment of $P'$.
(5)    If either (i) or (ii) is found, output $\ell$ and terminate program.
(6)    Else, let $\ell \leftarrow \lfloor \ell/2 \rfloor$. If $\ell \geq 2$, repeat from (2). Otherwise, output $\ell$.
```

If a segment $P \subseteq A$ of length l has $\mathrm{RMSD}(P, Q) \leq t$ (where $Q \subseteq B$ is its corresponding segment), then P' in step (4) must exist when $\ell \leq (l+1)/3$, because then, P necessarily spans two consecutive segments, P_{i+1}^ℓ and P_{i+2}^ℓ say, entirely. Then, by Lemma 2, either the segment P' described by (4.1), or that described by (4.2) must fulfill the RMSD requirement.

If (4) is unfulfilled prior to ℓ, then no segment of length $l \geq 3(2\ell) - 1 = 6\ell - 1$ which fulfills the RMSD requirement exists. Hence, $\boldsymbol{l} < 6\ell - 1 \leq 6\ell$. On the other hand, if (4) is fulfilled at ℓ, then there exists a segment of length at least ℓ which fulfills the RMSD requirement. The correctness of the algorithm follows.

The runtime of the algorithm is as follows. By Lemma 1, steps (4)-(4.3) can be computed in $O(n)$ time. Since ℓ is halved in each iteration, these steps are repeated $O(\log n)$ times, and the algorithm completes in $O(n \log n)$ time. Hence,

Theorem 1. *There is an algorithm which, given structures $A = (a_1, \ldots, a_n)$, $B = (b_1, \ldots, b_n)$, and $t \in \mathbb{R}$, computes $ALCS_{RMSD}(A, B, t)$ in $O(n \log n + n\boldsymbol{l})$ time, where $\boldsymbol{l}$ is the length of the segments in $ALCS_{RMSD}(A, B, t)$.*

With the approximation ℓ, we can very efficiently obtain a segment of at least length $\boldsymbol{l}$, but compromises on the threshold t. First, for structures $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, we define

$$\mathrm{RSD}(A, B) = \min_{R \in \mathcal{R}, T \in \mathcal{T}} \sqrt{\sum_{i=1}^{n} \|Ra_i - b_i - T\|^2}.$$

It is clear that for $P \subseteq A$ and $Q \subseteq B$, $\mathrm{RSD}(P, Q) \leq \mathrm{RSD}(A, B)$. Since $\mathrm{RSD}(A, B)$ can be computed from $\mathrm{RMSD}(A, B)$, a similar lemma as Lemma 1 can be obtained for RSD. These properties give us the following result.

Lemma 3. *There is an algorithm of $O(n)$ time complexity which, given structures A, B of length n, and threshold $x \in R$, computes a segment of the longest length among all $P \subseteq A$ with $RSD(P, Q) \leq x$, where $Q \subseteq B$ is the corresponding segment of P.*

Proof. Table 3 shows an algorithm which, given a threshold $x \in \mathbb{R}$, computes a longest segment $P \subseteq A$ fulfilling $\mathrm{RSD}(P, Q) \leq x$, where $Q \subseteq B$ is the corresponding segment of P.

Table 3. Algorithm for computing a longest segment of RSD within x

```
Input:  Structures $A = (a_1, \ldots, a_n)$, $B = (b_1, \ldots, b_n)$, threshold $x \in \mathbb{R}$.
Output: A segment $P \subseteq A$ of the longest length fulfilling $\mathrm{RSD}(P, Q) \leq x$, where
        $Q \subseteq B$ is the corresponding segment of $P$.

(1)    Pre-compute values for computing RSD in $O(1)$ time.
(2)    Let $pos \leftarrow 1$, $l \leftarrow 1$, and maxlen $\leftarrow 1$.
(3)    Let $P \leftarrow A[pos, pos + l]$ and $Q \leftarrow B[pos, pos + l]$.
(4)    If $\mathrm{RSD}(P, Q) \leq x$ and $l \geq$ maxlen *(A longer segment is found.)*
(4.1)      Let $\mathrm{LCS}_{\mathrm{RSD}} \leftarrow P$ and maxlen $\leftarrow l + 1$.
(4.2)      Let $l \leftarrow l + 1$. **(Extend)**
(4.3)      Repeat from (3).
(5)    Otherwise, *(RSD$(P, Q) > x$)*
(5.1)      Let $pos \leftarrow pos + 1$ and $l \leftarrow$ maxlen $- 1$. **(Move)**
(5.2)      Repeat from (3).
(At any point if $pos + l > n$, output $\mathrm{LCS}_{\mathrm{RSD}}$ and terminate program.)
```

The algorithm makes use of the fact that if a segment P results in a larger RSD than x, then all the segments which extend P will result in RSDs larger than x. As a result, at any point in the algorithm, if $\mathrm{LCS}_{\mathrm{RSD}}$ is set as $A[i, j]$, then for all $i' < i$, every $A[i', j']$ where $j' > j$ will result in an RSD larger than x (hence need not be evaluated). The correctness of the algorithm is clear from this analysis.

Since there can be at most $O(n)$ **Move** and **Extend** steps, step (4) is performed at most $O(n)$ times. Each computation of the RSD requires $O(1)$ time, given the pre-computation in step (1), which requires $O(n)$ time. Hence the time complexity of the algorithm is $O(n)$. ∎

We use the algorithm in Lemma 3 to find, for each of k different values of d from ℓ to $6\ell - \frac{5\ell}{k}$ (that is, $d = \ell, \ell + \frac{5\ell}{k}, \ell + \frac{10\ell}{k}, \ldots, 6\ell - \frac{5\ell}{k}$), a segment $P \subseteq A$ of $\mathrm{RSD}(P, Q) \leq t\sqrt{d + \frac{5\ell}{k}}$ and of length at least d. We select the longest of these segments.

When $d \leq \boldsymbol{l} \leq d + \frac{5\ell}{k}$, a segment of length at least $\boldsymbol{l}$ will be found, since for any segment $P \in \mathrm{ALCS}_{\mathrm{RMSD}}(A, B, t)$,

$$\mathrm{RMSD}(P, Q) \leq t \Rightarrow \mathrm{RSD}(P, Q) \leq t\sqrt{\boldsymbol{l}} \leq t\sqrt{d + \frac{5\ell}{k}}.$$

Since we select the longest segment P which fulfills the condition, our segment has length $\boldsymbol{d} \geq \boldsymbol{l}$. Furthermore,

$$\mathrm{RMSD}(P,Q) \leq \frac{\mathrm{RSD}(P,Q)}{\sqrt{\boldsymbol{d}}} \leq \frac{t\sqrt{\boldsymbol{d}+\frac{5\ell}{k}}}{\sqrt{\boldsymbol{d}}} \leq \frac{t\sqrt{\boldsymbol{d}+\frac{5\boldsymbol{d}}{k}}}{\sqrt{\boldsymbol{d}}} \leq t\sqrt{1+\frac{5}{k}} \leq t(1+\frac{5}{2k})$$

Let $\epsilon = 5/2k$, then $\mathrm{RMSD}(P,Q) \leq (1+\epsilon)t$.

Theorem 2. *There is an algorithm of $O(n \log n + \frac{n}{\epsilon})$ time complexity which, given structures A, B of length n, threshold $t \in \mathbb{R}$, and precision $\epsilon \in \mathbb{R}$, computes a segment $P \subseteq A$ of length at least $\boldsymbol{l}$ (i.e. the length of the elements in $ALCS_{RMSD}(A,B,t)$) with $RMSD(P,Q) \leq (1+\epsilon)t$, where $Q \subseteq B$ is the corresponding segment of P.*

4 FPTAS for ALCS$_{\text{dist}}$

We first show an FPTAS of $O(n^2 \log n/\epsilon)$ time complexity for $\mathrm{ALCS}_{\text{dist}}$ $(A, B, t+\epsilon t)$ on general structures. This will help in explaining the workings of the nearly linear time FPTAS. As mentioned, the computation of $\mathrm{ALCS}_{\text{dist}}(A,B,t)$ is complicated by the lack of an efficient algorithm to compute a superposition that matches corresponding points in a segment to within a given bottleneck threshold t. We use ideas from an algorithm which computes such a superposition efficiently, but compromises on the threshold requirement of t [11]. Instead of $\mathrm{ALCS}_{\text{dist}}(A,B,t)$, the algorithm computes $\mathrm{ALCS}_{\text{dist}}(A,B,t+\epsilon t)$ for a variable $\epsilon \in \mathbb{R}$ which affects its runtime.

To decide if two segments $f \subseteq A$ and $g \subseteq B$ are matchable, one can choose a_i and a_j in f, and first transform f under a transformation T which brings a_i and a_j to within distance t from b_i and b_j respectively. Then, f is rotated along the axis formed by Ta_i and Ta_j, to see if there exists an angle which brings all corresponding points in f and g to within distance t. Some care is needed in the choice of T — an incorrect T may result in fewer matchable points. However, short of an analytically method to determine the correct T, we exhaustively try out every T that is formed through possible positions of Ta_i and Ta_j in a discretized space. The discretization may result in the loss of the optimal T. To address this issue, we carefully select a_i and a_j to fulfill the property of a and a' below.

Definition 1. *Given a finite set of points S, two points a, $a' \in S$, and positive integer k, we write $\langle [a] \xrightarrow{k} a' \rangle_S$ iff $\max\{\|a - a''\| \mid a'' \in S\} \leq k\|a - a'\|$. We call a and a' a k-*radial pair*, and refer to a as its* pivot*. (Note that $\langle [a] \xrightarrow{k} a' \rangle_S$ does not imply $\langle [a'] \xrightarrow{k} a \rangle_S$.) We also write $\langle [a] \xrightarrow{1} a' \rangle_S$ as $\langle [a]a' \rangle_S$, and refer to such a, a' as simply a* radial pair.

The k-radial pair naturally generalizes the radial pair in [11]. The following result applies to k-radial pairs.

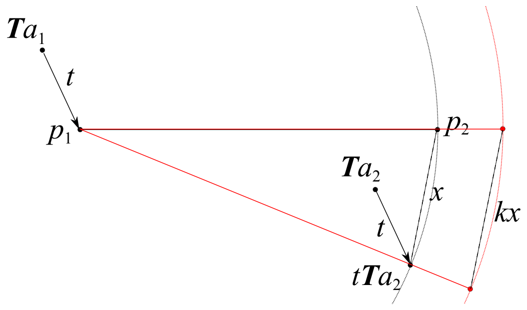
Lemma 4. *Given a set of points S, rigid transformations $\boldsymbol{T}$, T, and a_i, $a_j \in S$ where $\langle [a_i] \xrightarrow{k} a_j \rangle_S$, if $\|\boldsymbol{T}a_i - Ta_i\| \leq \delta$ and $\|\boldsymbol{T}a_j - Ta_j\| \leq \delta$, then there exists a rotation R about the axis through the points Ta_i and Ta_j, such that $\forall x \in S$, $\|RTx - \boldsymbol{T}x\| \leq (2k+1)\delta$.*

Proof. Let p_1, p_2 be two arbitrary points where $\|p_1 - \boldsymbol{T}a_1\| \leq \delta$, $\|p_2 - \boldsymbol{T}a_2\| \leq \delta$ and $\|p_1 - p_2\| = \|a_1 - a_2\|$. To prove the hypothesis it suffices that we show that there exists a transformation T' where $T'\boldsymbol{T}a_1 = p_1$, $T'\boldsymbol{T}a_2 = p_2$, and $\forall a \in S$, $\|T'\boldsymbol{T}a - \boldsymbol{T}a\| \leq (2k+1)\delta$. (That is, to show the hypothesis with T and R as stated, one only needs to further note that any transformation $T'\boldsymbol{T}$ can be decomposed into RT.)

We consider T' as the composition of two transformations, t and r, as follows:

– t is the translation where $t\boldsymbol{T}a_1 = p_1$.

– r is the rotation about the axis through a_1, orthogonal to the plane defined by p_1, $t\boldsymbol{T}a_2$ and p_2, with rotation angle $\alpha = \angle(t\boldsymbol{T}a_2\ p_1\ p_2)$.



Clearly $rt\boldsymbol{T}a_1 = p_1$ and $rt\boldsymbol{T}a_2 = p_2$.

Now $\forall a \in S$, $\|\boldsymbol{T}a - t\boldsymbol{T}a\| = \|t\| = \|\boldsymbol{T}a_1 - p_1\| \leq \delta$. Since $\|\boldsymbol{T}a_2 - p_2\| \leq \delta$, we have $\|t\boldsymbol{T}a_2 - p_2\| \leq 2\delta$. Since $\langle [a_1] \xrightarrow{k} a_2 \rangle_S$, $\forall a \in S$,

(1) $\|a_1 - a\| \leq k\|a_1 - a_2\|$,

(2) $\angle t\boldsymbol{T}a\ a_1\ rt\boldsymbol{T}a = \alpha$.

By (1) and (2), we have $\|t\boldsymbol{T}a - rt\boldsymbol{T}a\| \leq k\|t\boldsymbol{T}a_2 - p_2\| \leq 2k\delta$. Then, since $\|t\boldsymbol{T}a - \boldsymbol{T}a\| \leq \delta$, by triangle inequality we have $\|rt\boldsymbol{T}a - \boldsymbol{T}a\| \leq (2k+1)\delta$. Let $T' = rt$ and we are done. ∎

By Lemma 4, if T introduces an error of at most δ to Ta_i and Ta_j with respect to an optimal transformation $\boldsymbol{T}$, there will be a rotation such that the error introduced by T in each of the other $x \in S$ is no larger than $(2k+1)\delta$. By using a discretization of unit size $\delta = \epsilon t/(2k+1)$, the possible error introduced in each point is at most ϵt, as allowed for our approximation algorithm. Using this discretization, there are $O(k^5/\epsilon^5)$ possible combinations for the positions of Ta_i and Ta_j, each giving rise to a rotation axis. To see this, note the following. Since we require $\|a_i - b_i\| \leq t$ and $\|a_j - b_j\| \leq t$, it suffices that we examine coordinates of a_i and a_j which fulfill these conditions. By Lemma 4, we require a discretization of resolution at least $\epsilon t/(2k+1)$. We first discretize a sphere of radius $(1 + \frac{\epsilon}{2k+1})t$ centered at q_i using cubes of side length $\epsilon t/(2k+1)$. Each cube corresponds to a grid point in the discretized space. This gives us a total of $O(k^3/\epsilon^3)$ grid points to examine for a_i. Once a_i is fixed at a grid point, all the possible positions for a_j must be on a sphere cap centered at a_i with radius $\|a_i - a_j\|$, and are to be contained in the sphere of radius $(1 + \frac{\epsilon}{2k+1})t$ centered at b_j. If the two regions do not overlap, then clearly f and g are not matchable, and we are done. If the two regions overlap, then the overlap has an area of $O((1 + \frac{\epsilon}{2k+1})^2 t^2)$. We discretize this area with grids of resolution $\epsilon t/(2k + 1)$, resulting in $O(k^2/\epsilon^2)$ grid points to evaluate for a_j. Hence there are $O(k^5/\epsilon^5)$ possible combinations for the positions of a_i and a_j, each giving rise to a rotation axis.

For each rotation axis, the task is to examine if there exists a rotation R such that for each a_i in f, $\|Ra_i - b_i\| \leq t$. This can be done, as follows, in $O(l \log l)$ time, where l is the length of f. We first find, for each a_k in f, the pair of rotation angles which brings a_k into and out of distance t from b_k. Each of these pair of angles $[p, q]$ gives us an interval where two corresponding points are matched. (If $p > q$, the interval is broken into two parts $[p, 2\pi)$ and $[0, q]$.) We construct a tree out of these intervals according to a suggestion in [5] (see Problem 14.1). One construction of the data structure is as follows. A red-black tree T is contructed, with each interval forming two of its nodes — one node formed out of the interval's smallest value and one out of its largest value. The label $l(v)$ of each node v is the value from which the node is formed. All the nodes in the left subtree of a node v has a label smaller than $l(v)$, while all the nodes in the right subtree of a node v has a label larger than $l(v)$. That is, each node represents a point in $[0, 2\pi)$ when the number of interval overlaps may change — an in-order traversal of T would enumerate these points in increasing order. We keep a value $inorout(v)$ for each node v, where

$$inorout(v) = \begin{cases} 1 & \text{if } v \text{ is formed out of an interval's smallest value,} \\ -1 & \text{if } v \text{ is formed out of an interval's largest value.} \end{cases}$$

Let $v_1, v_2, \ldots$ be an in-order traversal of T, then $overlap(v_j) = \sum_{i=1}^{j} inorout(v_i)$ gives us the number of overlaps at the point represented by v_j.

To be able to compute the maximum of $overlap(v)$ for each node $v \in T$ efficiently, each v is further augmented with the following values: (1) $sum(v) = \sum_{u \in T(v)} inorout(u)$, where $T(v)$ is the subtree rooted at v; (2) $maxsum(v)$, which is defined as follows. Suppose an in-order traversal of $T(v)$ is $v_1, v_2, \ldots, v_k$. Then, $maxsum(v) = \max\{\sum_{i=1}^{j} inorout(v_i) \mid 1 \leq j \leq k\}$.

For a node $v \in T$, let $left(v)$ denote the left child of v and $right(v)$ denote the right child of v. Then, $sum(v)$ for each node v can be computed in $O(1)$ time with $sum(v) = sum(left(v)) + inorout(v) + sum(right(v))$, whereas $maxsum(v)$ can be computed in $O(1)$ time with $maxsum(v) = \max\{maxsum(left(v)), sum(left(v)) + inorout(v), sum(left(v)) + inorout(v) + maxsum(v)\}$. To find the maximum number of overlaps in the intervals, it suffices that we return $maxsum(root)$, which can be done in $O(1)$ time. If $maxsum(root) = l$, then there exists an interval where all l points are matched. Inserting/removing an interval into T requires the insertion/removal of two nodes, each which involves modification to $O(\log |T|)$ nodes to maintain the red-black tree property. Hence, the tree can be constructed in $O(l \log l)$ time. (In Section 4.1, we will further make use of the $O(\log l)$ update time of the tree.) The lemma below follows.

Lemma 5. *Given two segments $f = (a'_1, \ldots, a'_l)$, $g = (b'_1, \ldots, b'_l)$, and $a'_i, a'_j \in S(f)$ such that $\langle [a'_i] \xrightarrow{k} a'_j \rangle_{S(f)}$, whether there exists a transformation T where $(\forall m, 1 \le m \le l)[\|Ta'_m - b'_m\| \le (1+\epsilon)t]$ can be decided in $O((\frac{k}{\epsilon})^5 \, l \log l)$ time.*

Our approximation algorithm for $\mathrm{ALCS}_{\mathrm{dist}}(A, B, t)$ is in Table 4. The algorithm maintains a set ALCS of all the currently found longest contiguous segments, as well as their length, maxlen. The algorithm examines each pair $A[i, j]$, $B[i, j]$ in the following manner, starting with the pair $A[1, 2]$ and $B[1, 2]$.

- If a pair $A[i, j]$, $B[i, j]$ is matchable, $A[i, j]$ is recorded in ALCS, and the algorithm proceeds to examine $A[i, j+1]$ and $B[i, j+1]$ (**Extend** the segment).

- Otherwise, the algorithm will proceed to examine $A[i+1, i+\text{maxlen}]$, $B[i+1, i+\text{maxlen}]$ (**Move** the position of the segment to examine by a point).

Table 4. Algorithm for computing threshold approximation to $\mathrm{ALCS}_{\mathrm{dist}}(A, B, t)$

```
Input:  Structures $A = (a_1, \ldots, a_n)$, $B = (b_1, \ldots, b_n)$, threshold $t \in \mathbb{R}$, and
        precision $\epsilon \in \mathbb{R}$.
Output: $\mathrm{ALCS}_{\mathrm{dist}}(A, B, t + \epsilon t)$.
(1)    Let $pos \leftarrow 1$, $l \leftarrow 1$, maxlen $\leftarrow 2$, and ALCS $\leftarrow \emptyset$.
(2)    Let $f \leftarrow A[pos, pos + l]$ and $g \leftarrow B[pos, pos + l]$.
(3)    Find $a, a' \in S(f)$ s.t. $\langle [a]a' \rangle_{S(f)}$. (Find radial pair)
(4)    If $f$ and $g$ are matchabe by $t$ and $\epsilon$ (Computed via Lemma 5)
(4.1)      ((4.2)-(4.3) simply adds segment $A[pos, pos + l]$ to ALCS)
(4.2)      If $l + 1 =$ maxlen, add $A[pos, pos + l]$ into ALCS.
(4.3)      Otherwise, let ALCS $\leftarrow \{A[pos, pos + l]\}$ and maxlen $\leftarrow l + 1$.
(4.4)      Let $l \leftarrow l + 1$. (Extend)
(4.5)      Repeat from (2).
(5)    Otherwise, (f and g are not matchable)
(5.1)      Let $pos \leftarrow pos + 1$ and $l \leftarrow$ maxlen $- 1$. (Move)
(5.2)      Repeat from (2).
(At any point if $pos + l > n$, output ALCS and terminate program.)
```

The algorithm makes use of the fact that if a segment f cannot be matched under the threshold t and ϵ, then all the segments which extend f are not matchable under the same threshold. As a result, at any point in the algorithm, if $A[i, j]$ is added to ALCS, then for all $i' < i$, (1) all $A[i', j']$ where $j' > j$ are not matchable (hence need not be evaluated), while (2) all matchable $A[i', i' + (j - i)]$ are in ALCS. The correctness of the algorithm is clear from this analysis.

Since there can be at most $O(n)$ **Move** and **Extend** steps, step (4) is performed at most $O(n)$ times. Hence the total time required for the step is $O(n^2 \log n/\epsilon^5)$. To find a radial pair in $S(f)$ in step (3), we use an arbitrary point in $S(f)$ as a, and find from the remaining points in $S(f)$ the furthest point from a. This operation takes $O(l)$ time, and is carried out at most $O(n)$ times in the algorithm. Hence the $O(n^2 \log n/\epsilon^5)$ term dominates the runtime.

Theorem 3. *There is an algorithm of time complexity $O(n^2 \log n/\epsilon^5)$ which accepts two structures A, B, and a bottleneck distance $t \in \mathbb{R}$ as input, and outputs $ALCS_{dist}(A, B, t + \epsilon t)$.*

4.1 Nearly Linear-Time FPTAS for ALCS$_{\mathbf{dist}}$ on Protein Structures

The distances between consecutive points in a protein structure are the same (i.e. 3.8Å). This fact allows us to compute ALCS$_{\text{dist}}$ more efficiently. In this section we assume the distance between consecutive points in the structure $A = (a_1, \ldots, a_n)$ to be the same, i.e. $(\forall i, 1 < i < n)[\|a_{i-1} - a_i\| = \|a_i - a_{i+1}\|]$.

As in the discussion which leads to Lemma 5, given $f = A[i, i + l - 1]$ and $g = B[i, i + l - 1]$, we can use the tree of intervals to compute if f and g are matchable in $O((\frac{k}{\epsilon})^5\, l \log l)$ time. These trees can be reused, to compute if the subsequent segments $f' = A[i + 1, i + l]$ and $g' = B[i + 1, i + l]$ are matchable in only $O((\frac{k}{\epsilon})^5 \log l)$ time, provided that the *same k-radial pair can be used.* This latter runtime is for the removal of one interval from, as well as the addition of another interval to each tree constructed earlier, due to the removal of the point a_i and the addition of the point a_{i+l}. As mentioned, this modification to the tree structure requires $O(\log l)$ time. In order to exploit this lower runtime, our algorithm will attempt to reuse the same k-radial pair in examining subsequent segments.

Our algorithm first splits A into segments of maximal lengths, using the subroutine in Table 5. Its runtime is as follows. The subroutine uses 2-radial pairs in computing whether segments are matchable. We examine the number of times these 2-radial pairs are changed (i.e. when the condition in (4.2) is fulfilled) throughout the computation for a segment of length l in *Segments*. Consider how points are to be arranged in order that the fewest points are required between two subsequent changes in p. Since consecutive points are separated by an equal distance, these points are necessarily arranged along a path of the shortest distance between the boundaries of two concentric spheres, one of radius $\|p - pivot\|$ and the the other $2\|p - pivot\|$. That is, they are spread out along a radial line from *pivot*. Applying this rule recursively, for a fixed number of points, the condition in (4.2) is fulfilled most frequently when the points are arranged along a straight line. In which case, the condition is fulfilled $O(\log l)$ times. Each time it is fulfilled, $O(l \log l)$ time is required to recompute a tree of intervals. Hence, $O(l \log^2 l/\epsilon^5)$ time is required in total for these cases.

On the other hand, the condition in (4.2) fails for $O(l)$ times. Whenever that happens, $O(\log l)$ time is required to modify and $O(1)$ time to examine each tree of intervals, hence requiring a total runtime of $O(l \log l/\epsilon^5)$ for these cases. The earlier $O(l \log^2 l/\epsilon^5)$ runtime dominates. Since $\sum_{l \in \{|s| | s \in Segments\}} l = n$, the overall runtime complexity of the subroutine is $O(n \log^2 n/\epsilon^5)$.

It is clear that any longest contiguous segment cannot span more than two consecutive segments s_r and s_{r+1} in *Segments*. It follows that a longest contigu-

Table 5. Subroutine for partitioning A into maximally matchable segments

```
Input:  Structures $A = (a_1, \ldots, a_n)$, $B = (b_1, \ldots, b_n)$, threshold $t \in \mathbb{R}$, and
        precision $\epsilon \in \mathbb{R}$.
Output: A partitioning of $A$ into segments, where each partitioned segment $A[i, j]$
        is matchable to $B[i, j]$ but $A[i, j + 1]$ is not matchable to $B[i, j + 1]$.
(1)    Let $pos \leftarrow 1$, $l \leftarrow 1$, $Segments \leftarrow \emptyset$, and $r \leftarrow 0$.
(2)    Let $f \leftarrow A[pos, pos + l]$ and $g \leftarrow B[pos, pos + l]$.
(3)    Let $pivot \leftarrow a_{pos}$ and $p \leftarrow a_{pos+l}$.
(4)    If $f$ and $g$ are matchable by $t$ and $\epsilon$, (Then, extend $f$ as follows)
(4.1)      Let $l \leftarrow l + 1$, $f \leftarrow A[pos, pos + l]$, and $g \leftarrow B[pos, pos + l]$.
(4.2)      If the newly added point $p'$ in $f$ has $\|p' - pivot\| > 2\|p - pivot\|$
(4.3)          Let $p \leftarrow p'$ (Hence changing the 2-radial pair)
(4.4)          Repeat from (4).
(5)    Else (f and g are not matchable. Start a new segment f)
(5.1)      Let $r \leftarrow r + 1$, $s_r \leftarrow A[pos, pos + l - 1]$ and add $s_r$ into $Segments$.
(5.2)      Let $pos \leftarrow pos + l$ and $l \leftarrow 1$.
(5.3)      Repeat from (2).
(At any point if $pos + l > n$, output $Segments$ and terminate program.)
```

ous segment must either start within a segment s_r in *Segments* and end within s_{r+1}, or is exactly a segment in *Segments*. The algorithm in Table 6 accepts the concatenation of two consecutive segments in *Segments*, $A' = s_r s_{r+1}$, as well as its corresponding segment $B' \subseteq B$ as input, and computes ALCS$(A', B', t + \epsilon t)$. The strategy used by the algorithm is similar to that in Table 4, except for its use of 3-radial pairs instead of radial pairs. The correctness of the algorithm is argued similarly. The pivot of the 3-radial pairs is set to the point in the center of A'. This point is necessarily in every segment in ALCS$(A', B', t + \epsilon t)$, since any $f \in$ ALCS$(A', B', t+\epsilon t)$ must be as long as the longer of s_r and s_{r+1}. Hence, the same pivot is used throughout the computation.

The runtime of the main routine is as follows. Steps (1)-(4.6) can be analyzed in the same way as the subroutine in Table 5. These steps run in $O(l \log^2 l/\epsilon^5)$ time. Consider the number of times the condition in step (5.3) is fulfilled. If at any point of time, p is set to a point of an index larger than that of *pivot*, the condition will fail in all subsequent checks. Hence assume that p is always set to a point of a smaller index than *pivot*. Suppose there are ℓ consecutive points between *pivot* and p at some point of time. Since consecutive points are at a fixed distance of say, c apart, the distance between *pivot* and p is not more than $c(\ell + 1)$.

Consider the moment when the condition in step (5.3) is fulfilled, and a new p is to be chosen. By our condition that consecutive points are separated by c apart, there must exist a point at a distance at least $\frac{c(\ell+1)}{3}$, but no further than $\frac{c(\ell+1)}{3} + c$ from *pivot*. Hence, the point p' chosen as the new p is no further than $\frac{c(\ell+1)}{3} + c$ from *pivot*. Observe that the distance from the *pivot* to the new p is nearly one third of the distance from *pivot* to the original p.

Suppose after m number of times of such a reduction, the distance between *pivot* and p becomes no more than $3c$. We examine the smallest number of m

Table 6. Main routine for computing $\mathrm{ALCS}_{\mathrm{dist}}(A, B, t + \epsilon t)$ for protein structures

```
Input:  $A' = (a'_1, \ldots, a'_l)$, $B' = (b'_1, \ldots, b'_l)$, threshold $t \in \mathbb{R}$, and precision $\epsilon \in \mathbb{R}$.
Output: $\mathrm{ALCS}_{\mathrm{dist}}(A', B', t + \epsilon t)$.
(1)   Let $pos \leftarrow 1$, $l \leftarrow \lfloor \frac{l}{2} \rfloor - 1$, maxlen $\leftarrow \lfloor \frac{l}{2} \rfloor$, and ALCS $\leftarrow \emptyset$.
(2)   Let $f \leftarrow A'[pos, pos + l]$ and $g \leftarrow B'[pos, pos + l]$.
(3)   Let $pivot \leftarrow a'_{\mathrm{maxlen}}$ and find $p \in S(f)$ such that $\langle [pivot] p \rangle_{S(f)}$.
(4)   If $f$ and $g$ are matchable by $t$ and $\epsilon$, *(Then, extend $f$ as follows)*
(4.1)     If $l + 1 =$ maxlen, add $A[pos, pos + l]$ into ALCS.
(4.2)     Otherwise, let ALCS $\leftarrow \{A[pos, pos + l]\}$ and maxlen $\leftarrow l + 1$.
(4.3)     Let $l \leftarrow l + 1$, $f \leftarrow A'[pos, pos + l]$ and $g \leftarrow B'[pos, pos + l]$.
(4.4)     If the newly added point $p'$ in $f$ has $\|p' - pivot\| > 3\|p - pivot\|$
(4.5)         Let $p \leftarrow p'$. *(Hence changing the 3-radial pair)*
(4.6)     Repeat from (4).
(5)   Else *($f$ and $g$ are not matchable)*
(5.1)     Let $pos \leftarrow pos + 1$ and $l \leftarrow$ maxlen $- 1$. *(Move the segment)*
(5.2)     Let $f \leftarrow A'[pos, pos + l]$ and $g \leftarrow B'[pos, pos + l]$.
(5.3)     If $p \notin S(f)$, *($p$ was the point $a_i$ of the smallest $i$ in $f$ before the Move)*
(5.4)         Find in $f$ from the point of the largest to the smallest index,
(5.5)                 for a point $p'$ such that $\|p' - pivot\| > \|p - pivot\|/3$.
(5.6)         Let $p \leftarrow p'$. *(Hence changing the 3-radial pair)*
(5.7)     Repeat from (4).
(At any point if $pos + l > n$, output ALCS and terminate program.)
```

when this becomes true. It is easy to verify that the left hand side of the following inequality is the maximum distance between p and the *pivot* after the m-th time the condition is fulfilled.

$$\frac{c(\ell+1)}{3^m} + \left(\frac{c}{3^{m-1}} + \frac{c}{3^{m-2}} + \cdots + c\right) \leq 3c$$
$$\frac{\ell+1}{3^m} + \left(\frac{1}{2} - \frac{1}{2 \cdot 3^{m-1}}\right) \leq 2$$
$$\ell - \frac{1}{2} \leq \frac{3^{m+1}}{2}$$

The inequality shows that this happens when m is at least $\log_3(2\ell - 1) - 1$, or rather, of $O(\log \ell)$. When that happens, the point next to *pivot* with a larger index will be chosen as the new p, since it is at a distance of c from *pivot*. Henceforth, the condition will fail.

Since there are only l points in total, it follows that the condition is fulfilled at most $O(\log l)$ times. Each time the condition is fulfilled, $O(l \log l/\epsilon)$ time is required for the recomputation of the trees of intervals. This runtime dominates over those of the other computations, which are, $O(\log l/\epsilon^5)$ time for each of the $O(l)$ times when the condition fails, and $O(l)$ time for output of results (since $|\mathrm{ALCS}| \leq l$ and each segment can be described using two numbers). Hence, the runtime of the main routine is $O(l \log^2 l/\epsilon^5)$.

Since we perform the main routine for every two consecutive segments s_r and s_{r+1} in *Segments*, it has a total runtime of $O(\sum_{l \in \{|s_r|+|s_{r+1}| \mid s_r, s_{r+1} \in Segments\}} l \log^2 l/\epsilon^5) = O(n \log^2 n/\epsilon^5)$.

Theorem 4. *There is an algorithm of time complexity $O(n \log^2 n/\epsilon^5)$ which accepts (1) two structures A, B where the distance between subsequent points is fixed, and (2) a bottleneck distance $t \in \mathbb{R}$, as input, and outputs $ALCS_{dist}(A, B, t + \epsilon t)$.*

Acknowledgements. This research was supported by The Malaysian Ministry of Higher Education (MOHE) under Fundamental Research Grant Scheme, FRGS (FRGS/2/2010/SG/UTAR/03/2).

References

1. Arun, K.S., Huang, T.S., Blostein, S.D.: Least-squares fitting of two 3-d point sets. IEEE Trans. Pattern Anal. Mach. Intell. 9(5), 698–700 (1987)
2. Bowie, J.U., Luthy, R., Eisenberg, D.: A method to identify protein sequences that fold into a known 3-dimensional structure. Science 253(5016), 164–170 (1991)
3. Bryant, S.H., Altschul, S.F.: Statistics of sequence-structure threading. Current Opinion in Structural Biology 5(2), 236–244 (1995)
4. Choi, V., Goyal, N.: A Combinatorial Shape Matching Algorithm for Rigid Protein Docking. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 285–296. Springer, Heidelberg (2004)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press (2009)
6. Cristobal, S., Zemla, A., Fischer, D., Rychlewski, L., Elofsson, A.: A study of quality measures for protein threading models. BMC Bioinformatics 2(5) (2001)
7. Jones, D.T., Taylor, W.R., Thornton, J.M.: A new approach to protein fold recognition. Nature 358, 86–89 (1992)
8. Kabsch, W.: A solution for the best rotation to relate two sets of vectors. Acta Crystallographica Section A 32(5), 922–923 (1976)
9. Kabsch, W.: A discussion of the solution for the best rotation to relate two sets of vectors. Acta Crystallographica Section A 34(5), 827–828 (1978)
10. Leszek, R., Daniel, F., Arne, E.: Livebench-6: large-scale automated evaluation of protein structure prediction servers. Proteins 53(suppl. 6), 542–547 (2003)
11. Li, S.C., Bu, D., Xu, J., Li, M.: Finding nearly optimal GDT scores. J. Comput. Biol. 18(5), 693–704 (2011)
12. Siew, N., Elofsson, A., Rychlewski, L., Fischer, D.: Maxsub: an automated measure for the assessment of protein structure prediction quality. Bioinformatics 16(9), 776–785 (2000)
13. Simons, K.T., Kooperberg, C., Huang, E., Baker, D.: Assembly of protein tertiary structures from fragments with similar local sequences using simulated annealing and bayesian scoring functions. J. Mol. Biol. 268(1), 209–225 (1997)
14. Umeyama, S.: Least-squares estimation of transformation parameters between two point patterns. IEEE Trans. Pattern Anal. Mach. Intell. 13(4), 376–380 (1991)
15. Wu, S., Skolnick, J., Zhang, Y.: Ab initio modeling of small proteins by iterative tasser simulations. BMC Biology 5(17) (2007)
16. Zemla, A.: LGA: a method for finding 3D similarities in protein structures. Nucleic Acids Research 31(13), 3370–3374 (2003)

A Linear Kernel for the Complementary Maximal Strip Recovery Problem

Haitao Jiang[1,2] and Binhai Zhu[3]

[1] School of Computer Science and Technology, Shandong University, Jinan, Shandong 250100, China
htjiang@mail.sdu.edu.cn

[2] School of Mathematics and System Science, Shandong University, Jinan, Shandong 250100, China

[3] Department of Computer Science, Montana State University, Bozeman, MT 59717-3880, USA
bhz@cs.montana.edu

Abstract. In this paper, we compute the first linear kernel for the complementary problem of Maximal Strip Recovery (CMSR) — a well-known NP-complete problem in computational genomics. Let k be the parameter which represents the size of the solution. The core of the technique is to first obtain a tight $18k$ bound on the parameterized solution search space, which is done through a mixed global rules and local rules, and via an inverse amortized analysis. Then we apply additional data-reduction rules to obtain a tight $84k$ kernel for the problem. Combined with the known algorithm using bounded degree search, we obtain the best FPT algorithm for CMSR to this date, running in $O(2.36^k k^2 + n^2)$ time.

1 Introduction

The rapid development of the parameterized complexity theory greatly enhances our understanding beyond NP-completeness and the traditional computational complexity theory [6,22,13]. For many theoretically intractable applications, FPT (fixed-parameter tractable) algorithms can be very effective [7,11,21].

In the parameterized complexity theory, kernelization is a very useful tool [9,14]. Loosely, kernelization means the reduction of the problem instance size to a function of k (k is the parameter throughout this paper). In reality, small (especially small linear) kernel can make it feasible to use some traditional method like branch-and-bound or ILP, so it is always meaningful. On the other hand, there are various problems which do not admit small (or even polynomial) kernels unless the polynomial hierarchy collapses to its third level [1,8,10,12].

In the Complementary Maximal Strip Recovery (CMSR) problem, we need to delete at most k letters from the two input sequences (signed permutations) such that the remaining letters all form into strips (or maximal common substrings of length at least two, some could be in negated and reversed form). To this date, there are two bounded search tree algorithms running in $O^*(3^k)$ [17] and $O^*(2.36^k)$ [3] respectively for CMSR, but no (linear or even polynomial) kernel is known. Part of the reason that a (linear) kernel is elusive for the CMSR is that the only known local rule (see Lemma 1, i.e., 'long' maximal common substrings can be kept as strips) is not enough to establish any polynomial kernel.

In this paper, we obtain a linear $84k$ kernel for CMSR. The core of our idea is to first bound the *parameterized* solution search space (i.e., the set of letters, whose size is a function of k, from which an optimal solution can be obtained). By applying a set of global rules (together with the local rule induced by Lemma 1), we show that this space is of size at most $18k$. On top of this we can build successfully the linear kernel of size $84k$ for CMSR.

This paper is organized as follows. In Section 2, we define the MSR and CMSR problems and the corresponding concepts for FPT formally. In Section 3, we derive the $84k$ kernel bound for CMSR. In Section 4, we close the paper with several open problems.

2 Preliminaries

MSR and CMSR. Maximal Strip Recovery (MSR) was a problem originally proposed by the David Sankoff group to eliminate noise and ambiguities in genomic maps [5,24]. In comparative genomics, a genomic map (interchangeably, a sequence) is represented by a sequence of distinct gene markers (interchangeably, letters). A gene marker can appear in two different genomic maps, in either positive or negative form. A *strip* (syntenic block) is a sequence of distinct markers that appears as subsequences in two maps, either directly or in reversed and negated form. Given two genomic maps G_1 and G_2, the problem *Maximal Strip Recovery* (MSR) [5,24] is to find two subsequences of d strips (each of length at least two), denoted as $G_i^\star$, for $i = 1, 2$, and find two signed permutations π_i of $\langle 1, \ldots, d \rangle$, such that each sequence $G_i^\star = S_{\pi_i(1)} \ldots S_{\pi_i(d)}$ (here S_{-j} denotes the reversed and negated sequence of S_j) is a subsequence of G_i, and the total length of the strips S_j is maximized. Intuitively, those gene markers not included in $G_1^\star$ and $G_2^\star$ are noise and ambiguities. The complementary problem of deleting the minimum number of noise and ambiguous markers to have a feasible solution (i.e., every remaining marker must be in some strip) is exactly the *complement of MSR*, which will be abbreviated as CMSR.

We refer to Fig. 1 for an example. In this example, each integer represents a marker.

Not surprisingly, in [23], both MSR and CMSR were shown to be NP-complete. Most recently, MSR was shown to be APX-hard [2,15] and CMSR was also shown to be APX-hard [16]. For positive results, in [5,24], some heuristic approaches based on MIS and Max Clique were proposed. In [4], a factor-4 polynomial-time approximation algorithm was proposed for MSR. In [17], a factor-3 polynomial-time approximation algorithm was proposed for CMSR and an $O^*(3^k)$ FPT algorithm was proposed for CMSR (the latter improves and corrects an FPT bound in [23]). Recently, the approximation factor for CMSR was improved to 2.33 [20]. In this paper, we will focus only on the complement of MSR, or the CMSR problem.

$$
\begin{aligned}
G_1 &= \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 \rangle \\
G_2 &= \langle -9, -4, -7, -6, 8, 1, 3, 2, -12, -11, -10, -5 \rangle \\
S_1 &= \langle 1, 2 \rangle \\
S_2 &= \langle 6, 7, 9 \rangle \\
S_3 &= \langle 10, 11, 12 \rangle \\
\pi_1 &= \langle 1, 2, 3 \rangle \\
\pi_2 &= \langle -2, 1, -3 \rangle \\
G_1^\star &= \langle 1, 2, 6, 7, 9, 10, 11, 12 \rangle \\
G_2^\star &= \langle -9, -7, -6, 1, 2, -12, -11, -10 \rangle
\end{aligned}
$$

Fig. 1. An example for the problem MSR and CMSR. MSR has a solution size of eight (with $d = 3$ strips in $G_1^\star$ and $G_2^\star$; i.e., (1,2),(6,7,9) and (10,11,12)). CMSR has a solution size of four: the deleted markers are 3,4,5 and 8.

FPT and Kernel. We now present some definitions regarding FPT algorithms. Basically, a fixed-parameter tractable (FPT) algorithm for a *decision* problem Π with parameter k is an algorithm which solves the problem in $O(f(k)n^c) = O^*(f(k))$ time, where f is any function only on k, n is the input size and c is some fixed constant not related to k. FPT also stands for the set of problems which admit such an algorithm.

A useful technique in parameterized algorithmics is to provide polynomial time executable data-reduction rules that lead to a *problem kernel*. A data-reduction rule replaces (I,k) by an instance (I',k') in polynomial time such that: (1) $|I'| \leq |I|$, $k' \leq k$, (2) (I,k) is a Yes-instance if and only if (I',k') is a Yes-instance, and (3) $|I'| \leq g(k)$ for some function g. $|I'|$ is called the *size* of the kernel for the problem instance (I, k). A set of polynomial-time data-reduction rules for a problem are applied to an instance of the problem to achieve a *reduced* instance termed the *kernel*. A parameterized problem is FPT if and only if there is a polynomial time algorithm applying data-reduction rules that reduce any instance of the problem to a kernelized instance of size $g(k)$. More about parameterized complexity can be found in the monographs [7,11,21].

3 A Linear Kernel for CMSR

Our idea for constructing the linear $84k$ kernel for CMSR is based on first identifying the *parameterized solution search space* for CMSR. Formally, a *parameterized solution search space* for the CMSR problem is a subset S of the markers in G_1, G_2 such that we only need to delete k markers in S to obtain some optimal sequences $G_1^\star$ and $G_2^\star$; moreover, $|S| \leq g(k)$ for some function g. Once an S (of size $18k$) is obtained, it is relatively easy to obtain the linear kernel.

3.1 Bounding the Solution Search Space for CMSR

We first need to do some preprocessing. Before any marker is deleted, we can identify all maximal common substrings of length at least one (possibly in negated and reversed form, which will also be called maximal common substrings, or *block* for convenience) of G_1 and G_2. We also call a length-1 maximal common substring (which is a letter) an *isolated* letter or *isolate*. Two substrings are called *neighbors* if there is no other string in between them. The following lemma is proved in [17], and for completeness we include the proof here.

Lemma 1. *[17] Before any marker is deleted, if a length-4 maximal common substring $xyzw$ or $-w-z-y-x$ appears in both G_1 and G_2 (or, if $xyzw$ appears in G_1 and $-w-z-y-x$ appears in G_2, and vice versa), then there is an optimal solution for MSR which has $xyzw$ or $-w-z-y-x$ as a strip.*

Proof. Wlog, we only consider the case when $xyzw$ appears in G_1 and $-w-z-y-x$ appears in G_2. The cases when $xyzw$ $(-w-z-y-x)$ appears in both G_1 and G_2 are similar.

Let the length-6 substring in G_1 containing $xyzw$ be $p_1(x)xyzws_1(w)$ and let the length-6 substring in G_2 containing $-w-z-y-x$ be $p_2(w)-w-z-y-xs_2(x)$. Here $p_i(x), s_i(x)$ means the predecessor and successor of x in G_i. When deleting $xyzw$ from G_1 and $-w-z-y-x$ from G_2, at most two new strips can be obtained which could contain $\{p_1(x), s_1(w), p_2(w), s_2(x)\}$ (with a total size of 4). Clearly, retaining $xyzw$ and $-w-z-y-x$ as a strip can give us a solution at least as good as any optimal solution. Hence, the lemma is proven. □

An example for the above lemma is as follows: $G_1 = cdaxyzwbef$ and $G_2 = e-w-z-y-xfcd-b-a$. $xyzw$ appears in G_1, $-w-z-y-x$ appears in G_2. So we have one optimal solution $G_1^\star = cdxyzw$ and $G_2^\star = -w-z-y-xcd$. On the other hand, the optimal solution is not unique as we can select $G_1^+ = cdabef$ and $G_2^+ = efcd-b-a$.

The above lemma holds for maximal common substrings of length greater than 4. Now let us come back to our journey of obtaining a linear kernel for CMSR. Lemma 1 certainly provides a useful local rule to reduce the search space for solving CMSR. The difficulty now is how to handle length-2 and length-3 blocks. For example, let Q be a length-3 block and all P_i's have length 2, then in

$G_1 = xP_1QP_2y \cdot a_1b_1 \cdot a_2b_2 \cdot a_3P_3b_3 \cdot a_4P_4b_4 \cdot -w-z$

$G_2 = zP_3QP_4w \cdot a_4b_4 \cdot a_3b_3 \cdot a_2P_2b_2 \cdot a_1P_1b_1 \cdot -y-x$

the optimal solution in fact deletes Q, P_1, P_2, P_3, P_4. (Dot symbol is used for connection purpose.) Notice that Q has length-3 and has no isolated neighbor at all, yet it has to be deleted for an optimal solution! One could construct another counter-intuitive example where in a continuous (sequence of) length-2/3 blocks, only a part (i.e., not all) of them are deleted. So besides Lemma 1, it is in fact hard to apply any more local rules (with the ones we proposed early on, eventually counter-examples are found for each of them).

It turns out that we have to use a set of global rules together with a general graph method, which is described below in the algorithm.

Let Σ be the alphabet for the input maps G_1 and G_2. The kernelization procedure (for identifying S) is as follows.

1. Without deleting any gene marker in G_1 and G_2, identify a set of maximal common substrings (possibly in reversed and negated form) of length at least 4, of length-3, of length-2 and of length-1 (isolates). Then identify all maximal continuous blocks, each of length at least 2, in G_1 and G_2. We call the latter *super-blocks* henceforth, and denote them as $V_1 \in G_1$ and $V_2 \in G_2$.
2. (2.1) Firstly, for each block of length at least 4, change it to a new letter in Σ_1 (and delete the corresponding old letters in it from Σ whenever such a new letter in Σ_1 is created), with $\Sigma_1 \cap \Sigma = \emptyset$.
   (2.2) Secondly, for any pair of super-blocks $s_1 \in V_1, s_2 \in V_2$ which contain at least two pairs of common length-2 or length-3 blocks, identify the leftmost and rightmost such common blocks in s_1 (e.g., P_i, P_j) and in s_2 (e.g., P_l, P_r, with $P_i = P_l, P_j = P_r$ or $P_i = P_r, P_j = P_l$, some possibly in reversed and negated form). Change each block between and inclusive of P_i, P_j (resp. P_l, P_r) in s_1 (resp. s_2) into a new letter in Σ_1.
   (2.3) Thirdly, for any super-block (in V_1 or V_2) containing at least two length-3 blocks, identify the leftmost and rightmost length-3 blocks, say P_s, P_t. Change each block between and inclusive of P_s, P_t into a new letter in Σ_1.
   (2.4) Then, construct the simple bipartite graph $G = (V_1, V_2, E)$, where there is an edge $(v_1, v_2) \in E$ between two super-blocks $v_1 \in V_1, v_2 \in V_2$ iff they share a common length-2 or length-3 block not yet put in Σ_1. For any cycle in G, identify the length-2 or length-3 blocks involved in the cycle and change each of them into a new letter in Σ_1.
   (2.5) Finally, within any super-block, for all blocks between two letters in Σ_1, change each of them into a new letter in Σ_1.
3. Let the resulting sequences be G'_1, G'_2. Return $S \leftarrow \Sigma$ as a parameterized search space.

The correctness of Step 2 is as follows (Lemma 1 covers Rule (2.1)):

Lemma 2. *Rule (2.2) is correct.*

Proof. First, suppose that between P_i, P_j in V_1 there is a P' of length-2 or length-3 which is deleted in some optimal solution. As P' has no isolated neighbor in G_1, deleting it will create a new strip which includes at most two isolated neighbors of it in G_2. Therefore, we can keep P' as a strip and obtain another solution at least as good as the assumed optimal solution (which deletes P').

For P_i and P_j, as they are in V_1 and V_2, each of them has at most 2 isolated neighbors (one each in G_1 and G_2). If some optimal solution deletes one (or both) of them, by the same argument, we can keep one (or both) of them as strips to have a solution at least as good as the assumed optimal solution. □

Note that after Rule (2.2) is run, now a super-block could contain length-2 and length-3 blocks (no two common to another super-block), as well as letters in Σ_1.

Lemma 3. *Rule (2.3) is correct.*

Proof. First, suppose that between P_s, P_t in V_1 (resp. V_2) there is a P'' of length-2 or length-3 which is deleted in some optimal solution. As P'' has no isolated neighbor in G_1 (resp. G_2), deleting it will create a new strip which includes at most two isolated neighbors of it in G_2 (resp. G_1). Therefore, we can keep P'' as a strip and obtain another solution at least as good as the assumed optimal solution (which deletes P'').

For P_s and P_t, each of them has at most 3 isolated neighbors (1 in G_1 and 2 in G_2, or vice versa). If some optimal solution deletes one (or both) of them, by the fact that they are of length-3, we can keep one (or both) of them as strips to have a solution at least as good as the assumed optimal solution. □

After the run of Rule (2.3), a super-block could contain at most one length-3 block, as well as length-2 blocks and, of course, letters in Σ_1.

Lemma 4. *Rule (2.4) is correct.*

Proof. In the simple bipartite block graph G, if there is a cycle, with the involved length-2 or length-3 blocks being $P'_1, P'_2, ..., P'_u$, then $|P'_i| \geq 2$ for $1 \leq i \leq u$. If some optimal solution deletes some of these blocks, say $P'_{i1}, P'_{i2}, ..., P'_{ip}$, then in G we have deleted p edges, each associated with some P'_{ij}. P'_{ij} has at most two isolated neighbors (at most one each in G_1 and G_2). Consequently, we could keep $P'_{i1}, P'_{i2}, ..., P'_{ip}$ as strips to have a solution at least as good as the claimed optimal solution. □

Lemma 5. *Rule (2.5) is correct.*

Proof. In a super-block s_1 in G_1, any block $P' \in s_1$ between two letters in Σ_1 has at most 2 isolated neighbors in G_2. So if some optimal solution deletes P', we can put it back to have a solution at least as good as the assumed optimal solution. □

By now, it is easily seen that any given super-block s, after these run of five rules, has at most two continuous sequences of blocks which are not put in Σ_1. In other words, at this point, each superblock contains at most one letter in Σ_1.

Let Σ_1 be the set of all new letters used in the kernelization process, with $\Sigma_1 \cap \Sigma = \emptyset$. The three lemmas for obtaining the final results are:

Lemma 6. *There is an optimal CMSR solution of size k for G_1 and G_2 if and only if the solution can be obtained by deleting k markers in Σ from G'_1 and G'_2 respectively.*

Notice that after the kernelization step, we have no cycle and no vertex of degree zero in G. So if any connected component in G has q edges, then it must have exactly a set H of $q+1$ vertices. We have the following lemmas on H.

Lemma 7. *Let G contain m connected components $H_1, H_2, \cdots, H_m$, and let each H_i have q_i edges. Then, in between the vertices in G, there are at least $\sum_{i=1}^{m} q_i + m - 2$ sequences of neighboring isolates in G_1, G_2.*

Proof. The $q_i + 1$ vertices in H_i form a tree. In G_1 and G_2, these vertices correspond to continuous sequences of blocks (each of length at least 2, some of which could have been converted to letters in Σ_1), separated by sequences of neighboring isolates. Let H_i have a_i vertices in G_1 and b_i vertices in G_2. In G_1 the $\sum_{i=1}^m a_i$ vertices bound at least $\sum_{i=1}^m a_i - 1$ sequences of neighboring isolates. Similarly, in G_2 the $\sum_{i=1}^m b_i$ vertices bound at least $\sum_{i=1}^m b_i - 1$ sequences of neighboring isolates. In total the vertices in G have bounded at least

$$\left(\sum_{i=1}^m a_i\right) - 1 + \left(\sum_{i=1}^m b_i\right) - 1 = \sum_{i=1}^m q_i + m - 2$$

sequences of neighboring isolates, due to $a_i + b_i = q_i + 1$, for $i = 1..m$. □

Lemma 8. *Given any connected component H in G with q edges, the total length of all the blocks associated with the edges in H is at most $\lceil \frac{5q}{2} \rceil$.*

Proof. It is clear that $3q$ is a trivial upper bound, due to Rules (2.1-2.3). To have this tighter bound, first notice again that the $q + 1$ vertices in H form a tree. Then by the fact that no two incident edges can both correspond to length-3 blocks, we can conclude that the number of length-3 blocks allowed in H is exactly the size of maximum matching of H, which is obviously at most $\lceil q/2 \rceil$ (which occurs when H is in fact a path). Then the total length of all the blocks associated with the edges in H is at most $2q + \lceil q/2 \rceil = \lceil \frac{5q}{2} \rceil$. □

Finally, we have the following theorem.

Theorem 1. *In G_1' (resp. G_2'), there are at most $18k$ letters (markers) in Σ. In other words, CMSR has a parameterized solution search space of size $18k$.*

Proof. We use an inverse amortized analysis. Assume that we have some optimal MSR solution O^* (i.e., all letters in O^* are in some strips), we try to insert the deleted letters and length-2/3 blocks back into O^* to obtain G_1, G_2. There are four sets of letters/blocks: A — those letters/blocks we insert into G_1, G_2 (of a total length k); B — those isolated letters which were in some strips in O^*, but due to the insertion of type-A letters/blocks, they are broken into isolates; C — those blocks identified by our kernelization algorithm; and D — the remaining length-2/3 blocks associated with the edges in the block graph G. We need to show that

$$|A| + |B| + |D| \leq |A| + 17|A| = 18k.$$

Note that although A could contain sequences of blocks, they will be counted into $|A| = k$.

We charge a cost of 18 for each inserted type-A letter x (including x itself). Notice that x can break at most two strips in O^*, resulting in at most 4 type-B isolates.

The most general scenario is when we have a graph G each of its vertices corresponds to at most two sequences of type-D blocks, e.g., a vertex in G corresponds to $\mathcal{D} = P_1 P_2 \cdots P_i \cdot \Sigma_1$ letters $\cdot P_{i+1} P_{i+2} \cdots P_l$ (there could be no Σ_1

letters between P_i, P_{i+1}). For this scenario, first recall that now in G we have no cycle and no vertex of degree zero; moreover, in $\mathcal{D}$ we have at most one length-3 block. So if each connected component $H_i, 1 \le i \le m$ in G has q_i edges, then it must have exactly a set of $q_i + 1$ vertices. By Lemma 7, vertices in G bound at least $\sum_{i=1}^{m} q_i + m - 2$ sequences of isolated neighbors.

We now finish the final proof.

First, let us consider the (at least) $\sum_{i=1}^{m} q_i + m - 2$ sequences of isolated neighbors (also called *slots* for convenience) bounded by the vertices of G. These slots are introduced by the insertion of at least $\lceil (\sum_{i=1}^{m} q_i + m - 2)/4 \rceil$ type-A isolates. As what have just been discussed, each of these type-A isolates can introduce at most 4 type-B isolates. By Lemma 8, the total length of all the type-D blocks in G is at most $\sum_{i=1}^{m} \lceil \frac{5q_i}{2} \rceil$. Therefore, each type-A isolate can be charged a total cost of

$$\sum_{i=1}^{m} \lceil \frac{5q_i}{2} \rceil / \lceil \frac{\sum_{i=1}^{m} q_i + m - 2}{4} \rceil + 5,$$

which is at most 18 (when $m = 1$). To see why, let $t = \lceil (\sum_{i=1}^{m} q_i + m - 2)/4 \rceil$. Then

$$\sum_{i=1}^{m} q_i \le 4t - m + 2.$$

Therefore,

$$\sum_{i=1}^{m} \lceil \frac{5q_i}{2} \rceil \le \lfloor \frac{5\sum_{i=1}^{m} q_i}{2} \rfloor + m \le \lfloor \frac{5(4t - m + 2)}{2} \rfloor + m = 10t + 5 - \lceil \frac{3m}{2} \rceil,$$

which is at most $10t + 3 \le 13t$, with $m = 1$ and $t \ge 1$. Consequently, this means that our charge is safe.

Second, for each substring of r isolates not bounded (delimited) by vertices of G (we could have at most 4 such substrings of isolates, 2 each at the ends of G_1 and G_2), we first ignore the type-A isolates already contained in some slot and suppose that we have a remaining of r' isolates. These r' isolates can be either of type-A or type-B. It is easy to see that at least $\lceil r'/5 \rceil$ of these remaining r' isolates must be deleted. (The deleted ones are of type-A.) Clearly, $18\lceil r'/5 \rceil > r'$. So again our charge is safe.

As a simple example, assume that $G_1 = \boxed{\text{abc}}\, w_1 w_2 \boxed{\text{de}}\boxed{\text{fg}}\, x$ and $G_2 = \boxed{\text{abc}}\boxed{\text{de}} - w_2 x - w_1 \boxed{\text{fg}}$, they form G which contains a single connected component of 4 vertices and 3 edges. We have two slots: $w_1 w_2$ and $-w_2 x - w_1$. As x is charged for a total cost of 18 (including itself), while the length of G_1, G_2 is only 10, so the charge is safe.

Altogether, this gives us an upper bound of $18k$ for $|A| + |B| + |D|$. □

We can show that our kernelization algorithm for constructing the parameterized solution search space S is in fact tight, i.e., the size of S, returned by our algorithm, is at least $18k$ for $k = 1$. It can be done by modifying the $10k$ example at the end of the proof of Theorem 1 as follows.

$G_1 = \boxed{\text{abc}}\ \boxed{\text{de}}\ fxg\ \boxed{\text{hij}}\ \boxed{\text{kl}}\ mn\ \boxed{\text{opq}}$
$G_2 = \boxed{\text{abc}}\ fg\ \boxed{\text{de}}\ \boxed{\text{hij}}\ mxn\ \boxed{\text{kl}}\ \boxed{\text{opq}}$

The corresponding block graph G is a path. The optimal CMSR solution is to delete x (i.e., k=1). So the above parameterized search space bound is in fact tight.

3.2 Computing the Linear Kernel for CMSR

To obtain the linear kernel, we need to bound the number of letters in Σ_1 in addition to S. (Note that each letter in Σ_1 can be replaced by a unique block of length 4, although it could be of length at least 4 in the original input.) By the definition of superblocks, after Rule (2.2)-(2.5), each superblock contains at most one letter in Σ_1.

Now let us consider the number of superblocks, each containing at most one letter in Σ_1. Following the proof of Theorem 1, the number of superblocks is equal to the number of vertices of G, which is

$$\sum_{i=1}^{m}(q_i+1) = (\sum_{i=1}^{m} q_i) + m \leq (4t-m+2)+m = 4t+2,$$

which is bounded by $4k+2$. Therefore, the total number of letters in Σ_1 in G_1' and G_2' is at most

$$(4k+2)\times 2 = 8k+4.$$

Consequently, the total length of G_1' and G_2', expanding a letter in Σ_1 into a unique substring of length-4 in Σ^4, is bounded by

$$18k \times 2 + (8k+4)\times 4 = 68k+16 \leq 84k.$$

We thus have the main theorem of this paper.

Theorem 2. *CMSR has a linear kernel of size* $84k$.

Corollary 1. *Combined with the bounded search tree method, CMSR can be solved in* $O(2.36^k k^2 + n^2)$ *time.*

Proof. Without the linear kernel bound, using the bounded search tree method, there is an FPT algorithm which runs in $O(2.36^k n^2)$ time [3]. With the $84k$ linear kernel, the running time of the corresponding algorithm can be improved to $O(2.36^k k^2 + n^2)$ time. This is a standard procedure: just run the algorithm on the linear kernel. □

We comment that, by modifying the example at the end of Section 3.1, we can obtain a linear kernel of size $84k$ (for $k=1$). This shows that the $84k$ kernel bound for CMSR is tight (at least for $k=1$).

4 Concluding Remarks

We show a non-trivial $84k$ linear kernel for the Complementary Maximal Strip Recovery problem. Combined with a known bounded search tree algorithm, this results in the best known FPT algorithm for CMSR — in $O(2.36^k k^2 + n^2)$ time. An interesting question is whether these bounds can be further improved.

Using the recent concept of weak kernels [18], Theorem 2 in fact implies that CMSR has a (direct) weak kernel of size $18k$. However, as direct weak kernels can all be transformed into the traditional kernels (from the experience as in this paper), we think it is better to use weak kernels solely for the indirect ones. For problems admitting linear indirect weak kernels (e.g., Sorting by Reversals [18] and Sorting by Unsigned DCJ Operations [19]), no linear/polynomial kernels are known and no known bounded search tree algorithm can match up with the solutions provided by weak kernels.

Acknowledgments. This research is partially supported by NSF grant DMS-0918034, by NSF of China under grant 60928006, by the Open Fund of Top Key Discipline of Computer Software and Theory in Zhejiang Provincial Colleges at Zhejiang Normal University, and by the Shanghai Thousand Talents Program. We also thank Zhong Li, Guohui Lin and Lusheng Wang for several useful discussions, and the anonymous reviewers for several useful comments.

References

1. Bodlaender, H.L., Downey, R.G., Fellows, M.R., Hermelin, D.: On Problems without Polynomial Kernels (Extended Abstract). In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 563–574. Springer, Heidelberg (2008)
2. Bulteau, L., Fertin, G., Rusu, I.: Maximal Strip Recovery Problem with Gaps: Hardness and Approximation Algorithms. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 710–719. Springer, Heidelberg (2009)
3. Bulteau, L., Fertin, G., Jiang, M., Rusu, I.: Tractability and Approximability of Maximal Strip Recovery. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 336–349. Springer, Heidelberg (2011)
4. Chen, Z., Fu, B., Jiang, M., Zhu, B.: On recovering syntenic blocks from comparative maps. Journal of Combinatorial Optimization 18(3), 307–318 (2009)
5. Choi, V., Zheng, C., Zhu, Q., Sankoff, D.: Algorithms for the Extraction of Synteny Blocks from Comparative Maps. In: Giancarlo, R., Hannenhalli, S. (eds.) WABI 2007. LNCS (LNBI), vol. 4645, pp. 277–288. Springer, Heidelberg (2007)
6. Cook, S.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd ACM Symp. on Theory of Computing (STOC 1971), pp. 151–158 (1971)
7. Downey, R., Fellows, M.: Parameterized Complexity. Springer (1999)
8. Dell, H., van Melkebeek, D.: Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. In: Proc. 42nd ACM Symp. Theory of Computation (STOC 2010), Cambridge, MA, USA, pp. 251–260 (2010)
9. Fellows, M.: The Lost Continent of Polynomial Time: Preprocessing and Kernelization. In: Bodlaender, H.L., Langston, M.A. (eds.) IWPEC 2006. LNCS, vol. 4169, pp. 276–277. Springer, Heidelberg (2006)

10. Fernau, H., Fomin, F., Lokshtanov, D., Raible, D., Saurabh, S., Villanger, Y.: Kernel(s) for problems with no kernel: on out-trees with many leaves. In: Proc. 26th Intl. Symp. on Theoretical Aspects of Computer Science (STACS 2009), pp. 421–432 (2009)
11. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer (2006)
12. Fortnow, L., Santhanam, R.: Infeasibility of instance compression and succinct PCPs for NP. In: Proc. 40th ACM Symp. Theory of Computation (STOC 2008), Victoria, Canada, pp. 133–142 (2008)
13. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
14. Guo, J., Niedermeier, R.: Invitation to data reduction and problem kernelization. SIGACT News 38, 31–45 (2007)
15. Jiang, M.: Inapproximability of Maximal Strip Recovery. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 616–625. Springer, Heidelberg (2009)
16. Jiang, M.: Inapproximability of Maximal Strip Recovery: II. In: Lee, D.-T., Chen, D.Z., Ying, S. (eds.) FAW 2010. LNCS, vol. 6213, pp. 53–64. Springer, Heidelberg (2010)
17. Jiang, H., Li, Z., Lin, G., Wang, L., Zhu, B.: Exact and approximation algorithms for the complementary maximal strip recovery problem. J. of Combinatorial Optimization 23(4), 493–506 (2012)
18. Jiang, H., Zhang, C., Zhu, B.: Weak Kernels. ECCC Report, TR10-005 (October 2010)
19. Jiang, H., Zhu, B., Zhu, D.: Algorithms for sorting unsigned linear genomes by the DCJ operations. Bioinformatics 27, 311–316 (2011)
20. Li, Z., Goebel, R., Wang, L., Lin, G.: An Improved Approximation Algorithm for the Complementary Maximal Strip Recovery Problem. In: Atallah, M., Li, X.-Y., Zhu, B. (eds.) FAW-AAIM 2011. LNCS, vol. 6681, pp. 46–57. Springer, Heidelberg (2011)
21. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Univ. Press (2006)
22. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press, NY (1972)
23. Wang, L., Zhu, B.: On the tractability of maximal strip recovery. J. of Computational Biology 17(7), 907–914 (2010); Correction 18(1), 129 (2011)
24. Zheng, C., Zhu, Q., Sankoff, D.: Removing noise and ambiguities from comparative maps in rearrangement analysis. IEEE/ACM Transactions on Computational Biology and Bioinformatics 4, 515–522 (2007)

Efficient Exponential Time Algorithms for Edit Distance between Unordered Trees*

Tatsuya Akutsu[1], Takeyuki Tamura[1], Daiji Fukagawa[2], and Atsuhiro Takasu[3]

[1] Bioinformatics Center, Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto, 611-0011, Japan
{takutsu,tamura}@kuicr.kyoto-u.ac.jp
[2] Faculty of Culture and Information Science, Doshisha University, Kyoto 610-0394, Japan
dfukagaw@mail.doshisha.ac.jp
[3] National Institute of Informatics, Tokyo 101-8430, Japan
takasu@nii.ac.jp

Abstract. This paper presents efficient exponential time algorithms for the unordered tree edit distance problem, which is known to be NP-hard. For a general case, an $O(1.26^{n_1+n_2})$ time algorithm is presented, where n_1 and n_2 are the numbers of nodes in two input trees. This algorithm is obtained by a combination of dynamic programming, exhaustive search, and maximum weighted bipartite matching. For bounded degree trees over a fixed alphabet, it is shown that the problem can be solved in $O((1+\epsilon)^{n_1+n_2})$ time for any fixed $\epsilon > 0$. This result is achieved by avoiding duplicate calculations for identical subsets of small subtrees.

Keywords: tree edit distance, unordered trees, dynamic programming, maximum weight bipartite matching.

1 Introduction

Tree edit distance is one of the well-studied combinatorial pattern matching problems on tree structured data, which has applications in computational biology, XML databases, and image analysis [4,11].

Extensive studies have been done on tree edit distance for ordered trees since Tai developed an $O(n^6)$ time algorithm [12], where n is the maximum size of two input trees. Demaine et al. developed an $O(n^3)$ time algorithm and showed that this bound is optimal under some computation strategy [6].

However, the tree edit distance problem for unordered trees has been known to be NP-hard [14]. Furthermore, several MAX SNP-hardness results are known

* This work was partially supported by the Collaborative Research Programs of Institute for Chemical Research, Kyoto University and National Institute of Informatics. T.A. and T.T. were partially supported by JSPS, Japan: Grant-in-Aid 22650045 and Grant-in-Aid 23700017, respectively.

[8,9,15]. Fukagawa et al. developed an $O(h)$-approximation algorithm for unordered trees of height h [7]. For the closely related problem of finding a largest common subtree, Halldórsson and Tanaka developed a $2h$-approximation algorithm and an $O(\log^2 n)$-approximation algorithm [8], where the former was improved to $1.5h$-approximation [3]. Several exact algorithms have also been developed. Shasha et al. developed an $O(4^{l_1+l_2} \cdot poly(n_1, n_2))$ time algorithm [11], where l_i and n_i are respectively the numbers of leaves and nodes in an input tree T_i $(i = 1, 2)$. Halldórsson and Tanaka developed an $O(2^{b_1+b_2} \cdot poly(n_1, n_2))$ time algorithm [8], where b_i is the numbers of branching nodes of T_i[1]. Akutsu et al. developed $O(2.62^k \cdot poly(n))$ time algorithm under the unit cost model where k is the maximum bound of the edit distance [1]. Some polynomial time algorithms have also been developed for restricted editing operations [4].

In this paper, we present an $O(1.26^{n_1+n_2})$ time algorithm for edit distance between unordered trees. This clearly improves the result by Shasha et al. [11]. This also improves the result by Halldósson and Tanaka [8] when the time complexity is measured in terms of the numbers of nodes. Although all of these algorithms are based on combination of enumeration and dynamic programming, the way of the combination in our algorithm is different from those in [8,11]. Furthermore, our result includes a detailed combinatorial analysis of the number of relevant subsets of branching nodes. We further improve the algorithm for bounded degree trees over a fixed alphabet. We show that for such trees, the unordered tree edit distance problem can be solved in $O((1+\epsilon)^{n_1+n_2})$ time for any fixed $\epsilon > 0$. Due to the space limitation, some proofs are omitted in this version.

2 Preliminaries

For a rooted unordered tree $T = T(V, E)$, $V(T)$ denote the set of nodes, $E(T)$ denote the set of edges, and $r(T)$ denotes the root of T. For a node $v \in V(T)$, $p(v)$ denotes the parent of v where $v \neq r(T)$, $chd(v)$ denotes the set of children of v, $deg(v)$ denotes the *outdegree* of v (i.e., $deg(v) = |chd(v)|$), $\ell(v)$ denotes the label of v where a label is given from an alphabet Σ, $h(v)$ denotes the *height* of v where $h(v) = 0$ for leaves v, $des(v)$ denotes the set of descendants of v where $v \notin des(v)$, and $T(v)$ denotes the subtree induced by v and its descendants. For a tree T, $h(T)$ denotes the height of T (i.e., $h(T) = h(r(T))$) and $V^{\geq K}(T)$ denotes the set of nodes in T whose heights are no less than K.

We write $T_1 \approx T_2$ if a tree T_1 is isomorphic to a tree T_2 (including label information on nodes). We also write $F_1 \approx F_2$ if a forest F_1 is isomorphic to a forest F_2, where a forest is a multi-set of rooted trees.

An *edit operation on a tree T* is either a deletion, an insertion, or a substitution, where each operation is defined as follows (see also Fig. 1):

[1] This bound is given recently by Halldórsson via improved analysis.

Deletion: Delete a non-root node v in T with parent u, making the children of v become children of u. The children are inserted in the place of v into the set of the children of u.

Insertion: Inverse of delete. Insert a node v as a child of u in T, making v the parent of some of the children of u.

Substitution: Change the label of a node v in T.

We assign a *cost* for each editing operation: $\gamma(a, b)$ denotes the cost of substituting a node with label a to label b, $\gamma(a, \epsilon)$ denotes the cost of deleting a node labeled with a, and $\gamma(\epsilon, a)$ denotes the cost of inserting a node labeled with a.

The *edit distance* between two unordered trees $T_1 = T_1(V_1, E_1)$ and $T_2 = T_2(V_2, E_2)$ is defined as the cost of the minimum cost sequence of editing operations that transforms T_1 to T_2. The edit distance between T_1 and T_2 is denoted by $dist(T_1, T_2)$. In this paper, we assume that the cost function satisfies the conditions on a distance metric [4,14]: $\gamma(a, b) \geq 0$ for any $(a, b) \in \Sigma' \times \Sigma'$, $\gamma(a, a) = 0$ for any $a \in \Sigma'$, $\gamma(a, b) = \gamma(b, a)$ for any $(a, b) \in \Sigma' \times \Sigma'$, $\gamma(a, c) \leq \gamma(a, b) + \gamma(b, c)$ for any $a, b, c \in \Sigma' \times \Sigma' \times \Sigma'$, where $\Sigma' = \Sigma \cup \{\epsilon\}$. The cost function is called the *unit cost model* if $\gamma(x, y) = 1$ holds for all $x \neq y$. We call T_2 a *subtree* of T_1 if T_2 is obtained from T_1 only by deletion operations[2].

There exists a close relationship between the edit distance and the *edit distance mapping* [4,14]. $M \subseteq V(T_1) \times V(T_2)$ is called a *mapping* if the following conditions are satisfied for any two pairs (u_1, v_1), $(u_2, v_2) \in M$: $u_1 = u_2$ if and only if $v_1 = v_2$, u_1 is an ancestor of u_2 if and only if v_1 is an ancestor of v_2.

We define a *score function* $f(u, v)$ by $f(u, v) = \gamma(\ell(u), \epsilon) + \gamma(\epsilon, \ell(v)) - \gamma(\ell(u), \ell(v))$ where $u \in V_1$ and $v \in V_2$. We can see that $f(u, v) \geq 0$ holds for all (u, v). Then, the score of a mapping M is defined by $score(M) = \sum_{(u,v) \in M} f(u, v)$. Let M_{OPT} be a mapping with the maximum score. Then, it is well-known [2,4,8] that the following equality holds, where we assume without loss of generality (w.l.o.g.) that the roots of T_1 and T_2 correspond to each other in M_{OPT}:

$$dist(T_1, T_2) = \sum_{u \in V(T_1)} \gamma(\ell(u), \epsilon) + \sum_{v \in V(T_2)} \gamma(\epsilon, \ell(v)) - score(M_{OPT}).$$

In this paper, a subtree of T_1 (or T_2) consisting of the nodes of T_1 (resp. T_2) appearing in M is called a *common subtree* even if M contains some pairs of non-identical labels. The *largest common subtree* (LCST) is defined as the common subtree with the maximum score. Since the edit distance can be computed from LCST due to the above relationship, we hereafter focus on computation of LCST. The score of LCST between T_1 and T_2 (i.e., $score(M_{OPT})$) is denoted by $S(T_1, T_2)$ where $r(T_1)$ must correspond to $r(T_2)$ (because roots are not deleted). In the following, we let $n_1 = |V_1|$, $n_2 = |V_2|$, and $n = \max(n_1, n_2)$ unless otherwise stated.

[2] We also use the *subtree* for denoting a subgraph of a tree. However, the meaning of the subtree is clear from the context.

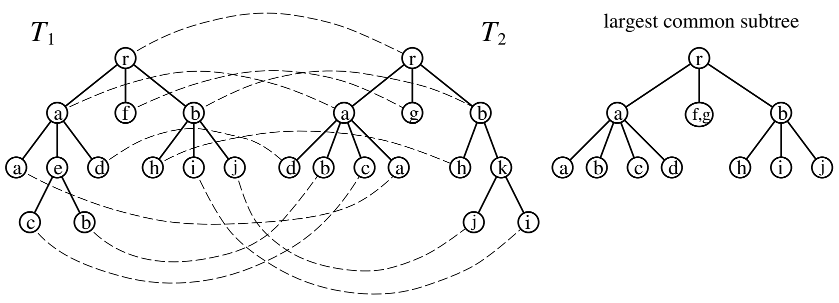


Fig. 1. Example of tree edit operation, mapping, and largest common subtree. T_2 is obtained from T_1 by deletion of node (labeled with) e, insertion of node k and substitution of node f. The corresponding mapping M is shown by broken curves. The largest common subtree is shown in the right-hand side.

3 Algorithm for a General Case

3.1 Algorithm

Akutsu et al. developed a heuristic exact algorithm for unordered tree edit distance by combining dynamic programming and maximum vertex weighted clique [1]. In this subsection, in order to obtain explicit time complexity results, we replace maximum clique with a combination of exhaustive search and maximum weight bipartite matching.

A set of nodes R is said to be *relevant* if R does not contain the root and

$$(\forall x, y \in R)(x \notin des(y) \text{ and } y \notin des(x))$$

holds, where $R = \emptyset$ is allowed. The key property of R is that there is no pair (x, y) in R that has an ancestor-descendant relationship.

For each pair of relevant sets R_u for $T_1(u)$ and R_v for $T_2(v)$, we construct a weighted bipartite graph $G_b(R_u, R_v; E_b)$ where $E_b = R_u \times R_v$ and the weight of an edge $(p, q) \in R_u \times R_v$ is given by $S[u, v]$, which should give the score of LCST between $T_1(u)$ and $T_2(v)$. Let $BPscore(R_u, R_v)$ denote the weight of the maximum weight bipartite matching of G_b. Procedure $UnordBasic(T_1, T_2)$ in the next page computes $S[u, v]$, which is equal to $S(T_1(u), T_2(v))$, for all pairs $(u, v) \in V(T_1) \times V(T_2)$ in a bottom up manner. It is to be noted that if u or v is a leaf, R_u or R_v is empty and thus $s_{max} = 0$ holds, where s_{max} finally gives the score of LCST between $T_1(u)$ and $T_2(v)$ excluding the score between u and v. This procedure essentially solves the tree-constrained bipartite matching problem [5] for all node pairs (u, v).

The following Proposition is clear from the description of the algorithm.

Proposition 1. *$UnordBasic(T_1, T_2)$ correctly computes $S(T_1(u), T_2(v))$ for all $(u, v) \in V(T_1) \times V(T_2)$.*

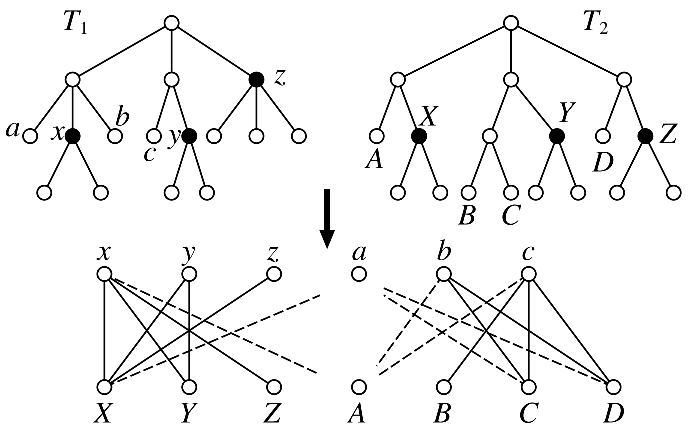


Fig. 2. Construction of a bipartite graph from relevant branching sets (shown by black circles) and remaining leaves (i.e., $L_u = \{a, b, c\}$ and $L_v = \{A, B, C, D\}$)

```
Procedure $UnordBasic(T_1, T_2)$
  for all pairs $(u, v) \in V(T_1) \times V(T_2)$ do (in a bottom-up way)
    $s_{max} \leftarrow 0$;
    for all relevant sets $R_u$ for $T_1(u)$ do
      for all relevant sets $R_v$ for $T_2(v)$ do
        $s \leftarrow BPscore(R_u, R_v)$;
        if $s > s_{max}$ then $s_{max} \leftarrow s$;
    $S[u, v] \leftarrow f(u, v) + s_{max}$.
```

Procedure $UnordBasic(T_1, T_2)$ examines all possible pairs of relevant sets for each pair (u, v). However, we can ignore leaves in relevant sets: we can regard all leaves that are not descendants of the selected internal nodes as candidates of leaves in LCST. Therefore, it is enough to examine all pairs of relevant sets of internal nodes. For a while, we assume that every internal node in T_1 and T_2 has at least two children, which means that every internal node is a branching node. R is called a *relevant branching* set if R is relevant and R does not contain any leaf or any node with outdegree 1. By modifying $UnordBasic(T_1, T_2)$ based on the above idea, we obtain procedure $UnordGeneral(T_1, T_2)$ (see the next page and Fig. 2), where $s_{max} = 0$ holds if u or v is a leaf.

Enumeration of relevant branching sets can be done in $O(|\mathcal{R}| \cdot poly(n))$ time where $\mathcal{R}$ is the set of all relevant branching sets, using $EnumRelevant(v, R)$ in the next page invoked with $v = r(T_i)$ and $R = \{\}$. $NextNode(v)$ returns u that is the next non-leaf node to v in DFS (depth first search) ordering of T_i, and $NextNonDes(v)$ returns u that is the next non-leaf node except descendants of u in DFS ordering of T_i, where we can use an arbitrarily fixed ordering of siblings in T_is. It is to be noted that this procedure can be combined with the main procedure so that it need not keep all relevant subsets.

```
Procedure $UnordGeneral(T_1, T_2)$
  for all pairs $(u, v) \in V(T_1) \times V(T_2)$ do (in a bottom-up way)
  $s_{max} \leftarrow 0$;
    for all relevant branching sets $R_u$ for $T_1(u)$ do
      for all relevant branching sets $R_v$ for $T_2(v)$ do
        Let $L_u$ be a set of leaves in $T_1(u)$ such that no $w \in L_u$ is
          a descendant of a node in $R_u$;
        Let $L_v$ be a set of leaves in $T_2(v)$ such that no $w \in L_v$ is
          a descendant of a node in $R_v$;
        $s \leftarrow BPscore(R_u \cup L_u, R_v \cup L_v)$;
        if $s > s_{max}$ then $s_{max} \leftarrow s$;
    $S[u, v] \leftarrow f(u, v) + s_{max}$.

Procedure $EnumRelevant(v, R)$
  if $v$ is emtpy then Output $R$ and return;
  $u \leftarrow NextNonDes(v)$;        /* $v$ is relevant */
  $EnumRelevant(u, R \cup \{v\})$;
  $u \leftarrow NextNode(v)$;          /* $v$ is not relevant */
  $EnumRelevant(u, R)$.
```

Since we are assuming that each internal node has at least two children, the number of internal nodes is at most $(n_1 - 1)/2 + (n_2 - 1)/2$. Therefore, the number of pairs of relevant branching sets per (u, v) is bounded by $2^{(n_1+n_2)/2}$. Since maximum weight bipartite matching can be done in polynomial time, $UnordGeneral(T_1, T_2)$ works in $O(2^{(n_1+n_2)/2} \cdot poly(n)) \leq O(1.415^{n_1+n_2})$ time.

In the above, we assumed that each internal node has at least two children. However, this assumption can be removed in the following way.

For node u in tree T, we define $\hat{u}$ by: $\hat{u} = u$ if u is a branching node, otherwise $\hat{u}$ is the highest branching node among the descendants of u, where $\hat{u}$ is not defined if $T(u)$ is a path. We compute $S'[u, v]$, which gives the score of LCST between $T_1(u)$ and $T_2(v)$, for all pairs (including non-branching node pairs) in $V(T_1) \times V(T_2)$ by

$$S'[u, v] \leftarrow \max \begin{cases} f(u, v), \\ f(u, v) + \max_{u' \in des(u), v' \in des(v)} S'[u', v'], \\ f(u, v) + \max_{R_{\hat{u}}, R_{\hat{v}}} BPscore(R_{\hat{u}} \cup L_u, R_{\hat{v}} \cup L_v), \end{cases}$$

where $R_{\hat{u}}$ (resp. $R_{\hat{v}}$) is taken over all relevant branching sets for $T_1(\hat{u})$ (resp. $T_2(\hat{v})$) and L_u (resp. L_v) is the set of leaves in $T_1(u)$ (resp. $T_2(v)$) such that no $w \in L_u$ (resp. no $w \in L_v$) is a descendant of a node in $R_{\hat{u}}$ (resp. $R_{\hat{v}}$).

The second and/or third lines are not executed if there do not exist the corresponding u', v', $\hat{u}$, and/or $\hat{v}$. $BPscore(R_{\hat{u}} \cup L_u, R_{\hat{v}} \cup L_v)$ is computed in the same way as in $UnordGeneral(T_1, T_2)$ using $S[u, v]$, which is computed by

$$S[u, v] \leftarrow \max_{u', v' | \hat{u'} = u, \hat{v'} = v} S'[u', v'].$$

If $T_i(u)$ is a path and u is a child of some branching node, u is treated as if it were a leaf where $S'[u,v]$ is used instead of $f(u,v)$ in the computation of $BPscore(R_{\hat{u}} \cup L_u, R_{\hat{v}} \cup L_v)$.

It is straight-forward to see that the exponential factor of the above procedure is the same as that of $UnordGeneral(T_1, T_2)$.

Lemma 1. *The edit distance between two unordered trees can be computed in* $O(1.415^{n_1+n_2})$ *time.*

3.2 Improved Analysis

In the above, we used a very rough estimate on the number of relevant branching sets. However, this number is much smaller than $2^{n/2}$ as shown below.

Lemma 2. *The number of relevant branching sets for a tree of size n is at most* $2^{\lceil (n-1)/3 \rceil}$.

This lemma is obtained by showing that the worst case number is essentially given when each children of the root has two leave children. To this end, we transform a given tree T into a tree having such structure step-by-step without decreasing the number of relevant sets, where the details are omitted in this version. Then, this lemma is obvious because such a tree has at most $\lceil (n-1)/3 \rceil$ branching nodes except the root.

Theorem 1. *The edit distance between two unordered trees can be computed in* $O(1.26^{n_1+n_2})$ *time.*

We can also see that the time complexity is above bounded by $O(2^{b_1+b_2} \cdot poly(n))$ time if we measure the complexity in terms of the numbers (b_1 and b_2) of branching nodes, where the number of relevant branching sets is much smaller than 2^{b_i} in most trees. It should be noted that this theorem holds also for the tree-constrained bipartite matching problem [5].

4 Algorithm for a Case of Bounded Degree and Fixed Alphabet

4.1 Algorithm

In Section 3, we considered all possible pairs of relevant (branching) sets. However, we need not consider all possible relevant sets. In the example of Fig. 3, bipartite graphs B_2 and B_3 are essentially the same, which comes from the fact that $T_1(u_4) \approx T_1(u_6)$. This suggests that we only need to mind the numbers of isomorphic subtrees once we have selected relevant nodes (shown by double circles in Fig. 3) with height greater than some threshold. In this section, we only consider trees of maximum outdegree D over a fixed alphabet Σ, where it is known that this restricted problem remains NP-hard [14].

Let K be a constant. We divide each relevant set R into R^H and R^L by $R^H = \{v \in R \mid h(v) \geq K\}$ and $R^L = \{v \in R \mid h(v) < K\}$, where we first determine R^H and then determine R and R^L in the algorithm[3].

[3] In this section, we do not consider relevant branching sets but consider relevant sets.

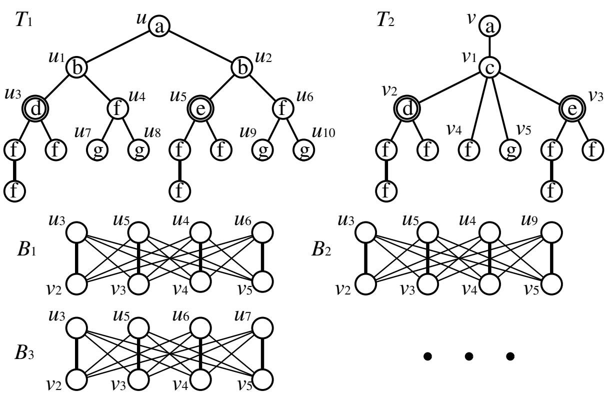


Fig. 3. Construction of bipartite graphs. B_2 and B_3 are essentially the same.

For each relevant set R for a tree T, $F(R)$ denotes the forest determined by $F(R) = \{T(v)|v \in R\}$. We say that two relevant sets R_1 and R_2 (resp. R_1^L and R_2^L) are *identical* if $R_1^H = R_2^H$ and $F(R_1^L) \approx F(R_2^L)$, which is denoted by $R_1 \simeq R_2$ (resp. $R_1^L \simeq R_2^L$).

For each relevant set R^H in a tree T, we define R^0 by $R^0 = \{v \in T \mid h(v) < K, v \notin des(u) \text{ for any } u \in R^H\}$.

Here, we consider the case of $K = 2$ in Fig. 3 as an example. Suppose $R_1 = \{u_3, u_4, u_5, u_6\}$, $R_2 = \{u_3, u_4, u_5, u_9\}$, $R_3 = \{u_3, u_5, u_6, u_7\}$. Then, $R_1^H = \{u_3, u_5\}$, $R_1^L = \{u_4, u_6\}$, $R_2^H = \{u_3, u_5\}$, $R_2^L = \{u_4, u_9\}$, $R_3^H = \{u_3, u_5\}$, $R_3^L = \{u_6, u_7\}$. $R_2 \simeq R_3$ and $R_2^L \simeq R_3^L$ hold, whereas $R_1 \simeq R_2$ does not hold. For any of R_1, R_2 and R_3, R^0 is determined as $R^0 = \{u_4, u_6, u_7, u_8, u_9, u_{10}\}$.

The following is a pseudocode of our proposed algorithm.

```
Procedure $UnordBounded(T_1, T_2)$
  **for all** pairs $(u, v) \in V(T_1) \times V(T_2)$ **do** (in a bottom-up way)
    $s_{max} \leftarrow 0$;
    **for all** relevant sets $R_u^H \subseteq (des(u) \cap V^{\geq K}(T_1))$ **do**
      **for all** relevant sets $R_v^H \subseteq (des(v) \cap V^{\geq K}(T_2))$ **do**
        Compute $R_u^0$ and $R_v^0$ from $R_u^H$ and $R_v^H$, respectively;
        **for all** non-identical relevant sets $R_u^L \subseteq R_u^0$ **do**
          **for all** non-identical relevant sets $R_v^L \subseteq R_v^0$ **do**
            $R_u \leftarrow R_u^H \cup R_u^L$;  $R_v \leftarrow R_v^H \cup R_v^L$;
            $s \leftarrow BPscore(R_u, R_v)$;
            **if** $s > s_{max}$ **then** $s_{max} \leftarrow s$;
    $S[u, v] \leftarrow f(u, v) + s_{max}$.
```

4.2 Analysis

In order to analyze the algorithm, we need to bound the number of R^Hs and the number of non-identical R^Ls.

Proposition 2. *The number of non-identical R^Ls is $O(n^{c_1})$ for some constant c_1 if K, D and $|\Sigma|$ are constants.*

Proof. Since K, D and $|\Sigma|$ are constants, the number of non-isomorphic trees of height less than K is bounded by some constant c_1. Therefore, the number of non-identical R^Ls is $O(n^{c_1})$. □

Enumeration of non-identical R^Ls can be done in the following way. Let $\{t_1, t_2, \ldots, t_{c_1}\}$ be the set of all possible non-isomorphic trees of height less than K. Let $b = (m_1, m_2, \ldots, m_{c_1})$ denote a forest that consists of m_1 copies of t_1, m_2 copies of t_2, $\cdots$, m_{c_1} copies of t_{c_1}. We use a 0-1 table $B[b]$ of size $O(n^{c_1})$[4]. First we initialize $B[b]$ by letting $B[b] \leftarrow 0$ for all b. For each t_i, let d_i be the number of non-isomorphic subforests of t_i. Let k_i denote the number of maximal $T(v)$s (not including a node from R^H) such that $T(v) \approx t_i$, where we let $k_i = 1$ even if there does not exist such $T(v)$[5]. For example, in T_1 of Fig. 3, $k_i = 2$ for t_i isomorphic to $T(u_4) \approx T(u_6)$ and $k_i = 1$ for the other t_is. Then, we can examine at most $k_1^{d_1} \times k_2^{d_2} \times \cdots \times k_{c_1}^{d_{c_1}}$ ways of making subforests, some of which may be isomorphic. If a subforest corresponding to b appears, we let $B[b] \leftarrow 1$. Accordingly, we can avoid the use of identical R^L. Since c_1 and d_is are constants (depending on K, D, and Σ) and $k_i \leq n$ holds, the above number is a polynomial of n as well as the size of $B[b]$. Since isomorphism of trees and forests can be tested in polynomial time, enumeration of non-identical R^Ls can be done in polynomial time.

Although we have not yet obtained a tight bound on c_1, we can estimate an upper bound of c_1 as follows. The size of a complete D-ary tree of height $K-1$ is

$$1 + D + D^2 + \cdots + D^{K-1} = \frac{D^K - 1}{D - 1}.$$

For each node v, we can consider $|\Sigma| + 1$ ways of label assignment where '+1' corresponds to deletion of v and its descendants, although isomorphic trees may be counted many times. Then, c_1 is bounded by $(|\Sigma|+1)^{\frac{D^K-1}{D-1}}$ and thus the number of non-identical R^L is $O(n^{(|\Sigma|+1)^{\frac{D^K-1}{D-1}}})$. Although the degree of polynomial grows extremely fast, it is still a constant if K, D and $|\Sigma|$ are constants.

For tree T, let $T^{\geq K}$ denote the tree obtained by deleting nodes with height less than K, where we assume w.l.o.g. $K \geq 2$. Then, we have the following propositions and lemmas.

Proposition 3. *The number of leaves in $T^{\geq K}$ is at most n/K.*

Proof. We can obtain $T^{\geq K}$ by the following procedure:

```
**for** $k = 1$ **to** $K$ **do** Delete all leaves.
```

[4] Enumeration can still be done in polynomial time without using $B[b]$ if it is allowed to enumerate identical relevant sets multiple times.

[5] Maximal means that $T(u)$ is not counted by any k_j if u is a descendant of v and $T(v)$ is counted by some k_i.

Let L_i be the number of leaves deleted at the i-th step. Then, we clearly have

$$L_1 + L_2 + \cdots + L_K \leq n,$$
$$L_1 \geq L_2 \geq \cdots \geq L_K,$$

from which $L_K \leq n/K$ follows. □

Proposition 4. *The number of relevant sets for a tree with n nodes is maximized when there is no node v such that $chd(v) = \{u\}$ and $|chd(u)| > 1$.*

Proof. Assume that there exists such a node v in a tree T. We construct a tree T' by deleting v and adding a new node v' as a child of an arbitrary leaf of $T(v)$ (see Fig. 4). Then, we can verify that the number of relevant sets for T' is no less than that for T. □

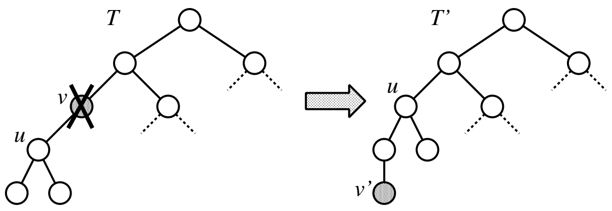


Fig. 4. Construction of T' from T in the proof of Proposition 4

Lemma 3. *The number of relevant sets in a tree T with n nodes is at most $2^L \cdot (n/L)^L$, where L is the number of leaves.*

Proof. From Proposition 4, the number is maximized when a tree has the shape shown in Fig. 5. Let L be the number of leaves of a tree T.

Each maximal connected component consisting of nodes with outdegree 1 and 0 is called a *hair*. Let the lengths of the hairs be $l_1, l_2, \ldots, l_L$. Clearly, $(l_1+1) + (l_2+1) + \cdots + (l_L+1) \leq n$. From Jensen's inequality, $(l_1+1) \times (l_2+1) \times \cdots \times (l_L+1)$ is maximized when $l_1 = l_2 = \cdots = l_L$. Therefore, we have

$$(l_1+1) \times (l_2+1) \times \cdots \times (l_L+1) \leq (n/L)^L,$$

which is the maximum number of relevant sets of nodes in hairs.

Since the number of nodes with outdegree > 1 is at most L, the number of combinations of selecting nodes not in any hair is at most 2^L.

Therefore, the number of relevant sets is bounded by $2^L \times (n/L)^L$. □

Lemma 4. *The number of non-identical relevant sets is $O(K^{(n/K)} \cdot 2^{(n/K)} \cdot n^{c_1})$.*

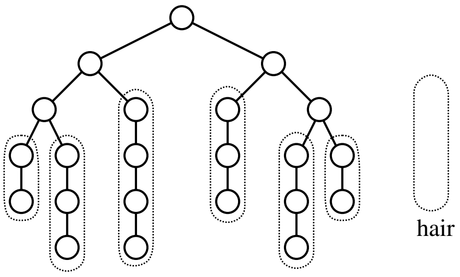


Fig. 5. The shape of the tree maximizing the number of relevant sets

Proof. The number of non-identical relevant sets is bounded by the number of R^Hs times the number of non-identical R^Ls.

From Lemma 3, the number of R^Hs is bounded by

$$f(L) = (n/L)^L \cdot 2^L,$$

where L is bounded by n/K from Proposition 3. We can show that $f(L)$ increases as L increases where n is fixed. To see this, let

$$g(L) = \log_2 f(L) = L(\log_2 n - \log_2 L) + L.$$

Since we are assuming $K \geq 2$ (i.e., $n/L \geq 2$), we have

$$\begin{aligned}\frac{dg(L)}{dL} &= (\log_2 n - \log_2 L) - \frac{1}{\ln 2} + 1 \\ &= (\log_2 \frac{n}{L}) - \frac{1}{\ln 2} + 1 \\ &\geq 2 - \frac{1}{\ln 2} > 0.\end{aligned}$$

Therefore, $f(L)$ is maximized when $L = n/K$.

Since the number of non-identical R^Ls is bounded by n^{c_1}, the required number is

$$O((n/L)^L \cdot 2^L \cdot n^{c_1}) \ \leq \ O(K^{(n/K)} \cdot 2^{(n/K)} \cdot n^{c_1}).$$

□

Theorem 2. *The edit distance between unordered trees of bounded degree and bounded alphabet can be computed in $O((1+\epsilon)^{n_1+n_2})$ time for any fixed $\epsilon > 0$.*

Proof. From Lemma 4, the time complexity of $UnordBounded(T_1, T_2)$ is

$$O(K^{(n_1+n_2)/K} \cdot 2^{(n_1+n_2)/K} \cdot poly(n_1, n_2)) \ = \ O(((2K)^{(1/K)})^{n_1+n_2} \cdot poly(n_1, n_2))$$

for fixed K, D and Σ. For any fixed ϵ,

$$(2K)^{(1/K)} < 1 + \epsilon$$

holds for sufficiently large K. Therefore, the theorem holds. □

It is to be noted that this algorithm is only of theoretical interest because of very high degree of a polynomial factor depending on K, D and $|\Sigma|$.

The conditions on the degree and alphabet in the above seem to be necessary as long as we use the same approach. Indeed, we can prove the following by using reductions similar to those in [8,9,14,15] (for Theorem 3) and in [10] (for Theorem 4).

Theorem 3. *The edit distance problem between unordered trees of height 3 is NP-hard for a ternary alphabet under the unit cost model.*

Theorem 4. *The edit distance problem between unordered trees is NP-hard for trees of outdegree at most four under the unit cost model even if branching nodes are either at height at most 1 or on a single path in a tree.*

5 Concluding Remarks

In this paper, we have presented an $O(1.26^{n_1+n_2})$ time algorithm for edit distance between two unordered trees, where the algorithm is a combination of dynamic programming, exhaustive search, and maximum weighted bipartite matching. In order to analyze the time complexity, combinatorial analysis was performed on the number of relevant sets of branching nodes. We have also shown that if inputs are restricted to bounded degree trees over a fixed alphabet, the problem can be solved in $O((1+\epsilon)^{n_1+n_2})$ time for any fixed $\epsilon > 0$. This result is achieved by avoiding duplicate calculations for identical subsets of small subtrees. Furthermore, we have shown hardness results that suggest the necessity of the conditions of bounded degree and fixed alphabet.

Although the former algorithm is much faster than a naive one, the base of the time complexity ($= 1.26$) is still high for practical applications. Therefore, improvement of this factor is left as an open problem. In the latter algorithm, the degree of polynomial grows extremely fast as a function of $1/\epsilon$. Therefore, significant reduction of the degree of polynomial is also left as an open problem.

Acknowledgments. We would like to thank Magnús M. Halldórsson for helpful discussions.

References

1. Akutsu, T., Fukagawa, D., Takasu, A., Tamura, T.: Exact algorithms for computing the tree edit distance between unordered trees. Theoret. Comput. Sci. 412, 352–364 (2011)
2. Akutsu, T., Mori, T., Tamura, T., Fukagawa, D., Takasu, A., Tomita, E.: An improved clique-based method for computing edit distance between unordered trees and its application to comparison of glycan structures. In: Proc. 5th International Conference on Complex, Intelligent and Software Intensive System, pp. 536–540. IEEE Press, New York (2011)

3. Akutsu, T., Fukagawa, D., Takasu, A.: Improved approximation of the largest common subtree of two unordered trees of bounded height. Inf. Proc. Lett. 109, 165–170 (2008)
4. Bille, P.: A survey on tree edit distance and related problem. Theoret. Comput. Sci. 337, 217–239 (2005)
5. Canzar, S., Elbassioni, K., Klau, G.W., Mestre, J.: On Tree-Constrained Matchings and Generalizations. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6755, pp. 98–109. Springer, Heidelberg (2011)
6. Demaine, E.D., Mozes, S., Rossman, B., Weimann, O.: An optimal decomposition algorithm for tree edit distance. ACM Trans. Algorithms 6, 1 (2009)
7. Fukagawa, D., Akutsu, T., Takasu, A.: Constant Factor Approximation of Edit Distance of Bounded Height Unordered Trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 7–17. Springer, Heidelberg (2009)
8. Halldórsson, M.M., Tanaka, K.: Approximation and special cases of common subtrees and editing distance. In: Nagamochi, H., Suri, S., Igarashi, Y., Miyano, S., Asano, T. (eds.) ISAAC 1996. LNCS, vol. 1178, pp. 75–84. Springer, Heidelberg (1996)
9. Hirata, K., Yamamoto, Y., Kuboyama, T.: Improved MAX SNP-Hard Results for Finding an Edit Distance between Unordered Trees. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 402–415. Springer, Heidelberg (2011)
10. Kilpeläinen, P., Mannila, H.: Ordered and unordered tree inclusion. SIAM J. Computing 24, 340–356 (1995)
11. Shasha, D., Wang, J.T.-L., Zhang, K., Shih, F.Y.: Exact and approximate algorithms for unordered tree matching. IEEE Trans. System, Man, and Cybernetics 24, 668–678 (1994)
12. Tai, K.-C.: The tree-to-tree correction problem. J. ACM 26, 4220–4433 (1979)
13. Tovey, C.A.: A simplified satisfiability problem. Disc. Appl. Math. 8, 85–89 (1984)
14. Zhang, K., Statman, R., Shasha, D.: On the editing distance between unordered labeled trees. Inf. Proc. Lett. 42, 133–139 (1992)
15. Zhang, K., Jiang, T.: Some MAX SNP-hard results concerning unordered labeled trees. Inf. Proc. Lett. 49, 249–254 (1994)

Fixed-Parameter Algorithms for Finding Agreement Supertrees*

David Fernández-Baca, Sylvain Guillemot, Brad Shutters, and Sudheer Vakati

Department of Computer Science, Iowa State University, Ames, IA 50011, USA
{fernande,sguillem,shutters,svakati}@iastate.edu

Abstract. We study the agreement supertree approach for combining rooted phylogenetic trees when the input trees do not fully agree on the relative positions of the taxa. Two approaches to dealing with such conflicting input trees are considered. The first is to contract a set of edges in the input trees so that the resulting trees have an agreement supertree. We show that this problem is NP-complete and give an FPT algorithm for the problem parameterized by the number of input trees and the number of edges contracted. The second approach is to remove a set of taxa from the input trees so that the resulting trees have an agreement supertree. An FPT algorithm for this problem when the input trees are all binary was given by Guillemot and Berry (2010). We give an FPT algorithm for the more general case when the input trees have arbitrary degree.

1 Introduction

A phylogeny, or evolutionary tree, is a tree representing the evolutionary history of a set of species. The leaves of the tree represent the current species (taxa), and the internal nodes of the tree represent the hypothetical ancestors. A fundamental problem in phylogenetics is to construct a supertree from smaller input trees with overlapping taxa in such a way that the inferred supertree complies as closely as possible with the topological information of the input trees. This problem is motivated by the biological and computational constraints on constructing large scale phylogenies. The supertree problem was introduced in [5], and a variety of supertree construction methods have been proposed. See [3,14,1,15,4] for more on supertrees.

In this paper we use the agreement supertree approach for combining rooted phylogenetic trees. The goal of this approach is to search for a supertree such that each of the input trees is a restriction of the supertree to a subset of its taxa. Formally, we have the following decision problem.

* This work was supported in part by the National Science Foundation under grants CCF-1017189 and DEB-0829674.

Agreement Supertree (AST)
Input: a collection $\mathcal{T}$ of k rooted phylogenetic trees on a set of n taxa.
Question: does there exist an agreement supertree for $\mathcal{T}$?

The answer to an instance of AST is "yes" if and only if the input trees fully agree on the relative positions of the taxa, in which case the input trees are said to agree and an agreement supertree can be found in polynomial time [13].

The input trees may fail to have an agreement supertree because of conflicts with respect to the relative positions of some taxa. In practice, such conflicts arise due to errors in the inference process, or due to biological processes, e.g., lateral gene transfer, gene duplication, and others [11,10]. These errors materialize as misplaced taxa and unnecessary edges in the input trees. We thus consider the following two approaches for dealing with conflicting input trees.

The first approach we consider for dealing with conflicting input trees is to find a subset of the edges of the input trees to contract so that the resulting collection of trees agree. Formally, we focus on the following parameterized problem.

Agreement Supertree Edge Contraction (AST-EC)
Input: a collection $\mathcal{T}$ of k rooted phylogenetic trees on a set of n taxa, and an integer p.
Question: can we contract at most p internal edges of $\mathcal{T}$ so that the trees in $\mathcal{T}$ agree?

The AST-EC problem does not seem to have been considered before. We give a proof of the NP-hardness of the AST-EC problem, and show that it is fixed-parameter tractable for parameters k and p.

The second approach we consider is to find a subset of the taxa to remove from the input trees so that the resulting collection of trees agree. Formally, we focus on the following parameterized problem.

Agreement Supertree Taxon Removal (AST-TR)
Input: a collection $\mathcal{T}$ of k rooted phylogenetic trees on a set of n taxa, and an integer p.
Question: can we remove at most p taxa so that the input trees agree?

The AST-TR problem is NP-hard [8,2], but was shown to be fixed-parameter tractable in k and p when restricted to the case when the input trees are all binary [6]. Our contribution is to show that the more general AST-TR problem, where the input trees are allowed to have arbitrary degree, is fixed-parameter tractable in k and p. It was also shown in [2] that if AST-TR is parameterized by only k or p, then the problem is fixed-parameter intractable. We also note that the optimization version of AST-TR, i.e., finding a minimum set of taxa to remove, is the dual of the Maximum Agreement Supertree (Smast) problem [2,8,9]. Exact algorithms for Smast on binary trees are known that run in time $O(6^k n^k)$ [6,7] and, when the maximum degree of the input trees is d, [7] gives an $O((kd)^{kd+3} 2^k n^k)$ time algorithm for Smast.

The rest of this paper proceeds as follows. In Section 3, we develop a characterization of when a set of input trees agree. We then use this characterization to develop an algorithm for testing agreement that solves the AST problem. We remark here that this algorithm could be easily modified to produce an agreement supertree when the set of input trees agree. If the algorithm answers in the negative, it returns a subset of the internal nodes of the trees in $\mathcal{T}$ encapsulating the taxa on which the trees in $\mathcal{T}$ disagree. In Section 4 we use these internal nodes to develop $O((2k)^p kn^2)$ time algorithms to solve the AST-EC and AST-TR problems. We also prove the NP-hardness of the AST-EC problem by giving a reduction from Multicut to AST-EC. We conclude in Section 5 with a brief discussion of some ideas for future research. Due to space limitations we omit some of the proofs.

2 Definitions

Let $\mathcal{T} = \{T_1, \ldots, T_k\}$ be a collection of rooted phylogenetic trees, and let T be some arbitrary tree in $\mathcal{T}$. We use $V(T)$, $E(T)$, $\hat{E}(T)$, and $r(T)$ to denote the vertices, edges, internal edges, and root vertex of T respectively. We use $L(T)$ to denote the set of labels mapped to the leaves of T, and we write $L(\mathcal{T})$ for $\bigcup_{i\in[k]} L(T_i)$ and $\hat{E}(\mathcal{T}) = \bigcup_{i\in[k]} \hat{E}(T_i)$, where $[k]$ stands for $\{1, \ldots, k\}$. For each $u \in V(T)$, we use parent(u), Ch(u), $T(u)$, and $L(u)$ to denote the parent of u, the children of u, the subtree of T rooted at u, and the set of labels mapped to the leaves of $T(u)$, respectively.

For a label set L, the *restriction* of T to L, denoted by $T|L$, is the minimal homeomorphic subtree of T connecting leaves with labels in L. For a set $L \subseteq L(\mathcal{T})$, we write $\mathcal{T}|L$ for the collection $\{T_1|L, \ldots, T_k|L\}$ of trees in $\mathcal{T}$ restricted to L. For a set $F \subseteq \hat{E}(T)$ we use T/F to denote the tree obtained from T by contracting the edges of F. For a set $F \subseteq \hat{E}(\mathcal{T})$, we denote the set $\{T_1/F, ..., T_k/F\}$ by $\mathcal{T}/F$. Given two trees S and T where $L(T) \subseteq L(S)$, T is an *induced subtree* of S if and only if $S|L(T) = T$. Note that all degree two vertices in $S|L(T)$ are assumed to be contracted. An *agreement supertree* for $\mathcal{T}$ is a tree S such that each T_i is an induced subtree of S. We say that the trees in $\mathcal{T}$ *agree* if and only if there is an agreement supertree for $\mathcal{T}$.

A *position* π in $\mathcal{T}$ is a tuple $(v_1, v_2, \ldots, v_k)$ where each vertex v_i is either from the tree T_i or the symbol $\perp$. A *reduced position* is a position where each component is an internal node or $\perp$. The *reduction of a position* π, denoted by $\pi\downarrow$, is derived by substituting every leaf vertex in π by $\perp$. We use $\pi_\top$, respectively $\pi_\perp$, to denote the *initial*, respectively *final*, positions where $v_i = r(T_i)$, respectively $v_i = \perp$, for each $i \in [k]$. We write $L(\pi)$ for $\bigcup_{i\in[k]} L(\pi[i])$. There is an agreement supertree for π if there is an agreement supertree for the collection of trees $\{T_1(\pi[1]), \ldots, T_k(\pi[k])\}$.

The graph $G(\mathcal{T}, \pi)$ is the graph whose vertex set consists of the children of all the vertices in π and there is an edge between two vertices u and v if and only if $L(u) \cap L(v) \neq \emptyset$. Note that the graph $G(\mathcal{T}, \pi)$ is only defined when π is a reduced position. In the rest of this paper, $G(\mathcal{T}, \pi)$ is denoted by $G = (V, E)$ and $V_i = \mathrm{Ch}(\pi[i])$ for each $i \in [k]$.

3 The Agreement Supertree Problem

In this section we give a characterization of when a collection of input trees has an agreement supertree. We then use this characterization to build an $O(kn^2)$ time algorithm that solves the AST problem.

3.1 Chararacterizing Agreement

The proofs of Lemmas 1, 2, and 3 are omitted.

Lemma 1. *The following statements hold:*

1. *There is an agreement supertree for $\mathcal{T}$ if and only if there is an agreement supertree for every position π of $\mathcal{T}$.*
2. *There is an agreement supertree for a position π of $\mathcal{T}$ if and only if there is an agreement supertree for $\pi\downarrow$.*

A subset $U \subseteq V$ is *nice* if, for each $i \in [k]$, U contains either zero, one, or all of the elements of V_i. A partition P of V is a *nice partition* of G if every set of P is nice, and, for every $\{C, C'\} \subseteq P$, C and C' are disconnected in G. The *successor of π w.r.t. a nice set U*, denoted by π_U, is defined as:

$$\pi_U[i] = \begin{cases} \bot & \text{if } V_i \cap U = \emptyset \\ p & \text{if } V_i \cap U = \{p\} \\ \pi[i] & \text{if } |V_i \cap U| \geq 2 \end{cases}$$

for each $i \in [k]$.

Lemma 2. *Let π be a reduced position such that $\pi \neq \pi_\bot$. The following statements are equivalent.*

1. *There is an agreement supertree for π.*
2. *There exists a nice partition P of G where P has at least two classes, and, for every $X \in P$, π_X has an agreement supertree.*

For partitions P and Q of V, we say P is *finer* than Q, denoted $P \sqsubseteq Q$, if and only if, for every $C \in P$, there exists a $D \in Q$ such that $C \subseteq D$. Let $\mathcal{P}$ represent the set of all nice partitions of G, and let $(\mathcal{P}, \sqsubseteq)$ represent the ordering of partitions of $\mathcal{P}$ under the operation $\sqsubseteq$. Note that $(\mathcal{P}, \sqsubseteq)$ is a poset.

Lemma 3. $(\mathcal{P}, \sqsubseteq)$ *has a unique minimal element.*

We call the unique minimal element of $(\mathcal{P}, \sqsubseteq)$ the *minimum nice partition* of G. Suppose that the minimum nice partition P of G is a singleton. Let $K = \{i \in [k] : \pi[i] \neq \bot\}$. We say that a set $I \subseteq V$ is *interesting* for a reduced position π of $\mathcal{T}$ if both $|I \cap V_i| = 2$ for each $i \in K$, and there is a set $F \subseteq E$ such that all of the following conditions hold: (i) $|F| \leq 2|K| - 1$; (ii) for each $v \in I$, there exists an $e \in F$ such that $v \in e$; and (iii) the subgraph of G induced by F has a minimum nice partition that is a singleton. Intuitively, a set of interesting vertices certifies there is no agreement supertree for π. If there is a set of interesting vertices for π, we say π is an *obstructing position* for $\mathcal{T}$.

3.2 Finding Successor Positions and Interesting Vertices

We now develop an algorithm GetSuccessors that takes as input a position π in a collection $\mathcal{T}$ of rooted phylogenetic trees, and finds the set Π of successor positions for each class in the minimum nice partition of G. In case the minimum nice partition of G is a singleton, the algorithm returns a set I of interesting vertices for π. An implementation of the algorithm and proofs of Theorems 1, 2 are omitted.

The central idea behind the GetSuccessors algorithm is that, for a given $\ell \in L(\pi)$, all of the vertices in the set $S_\ell = \{v \in V : \ell \in L(v)\}$ are connected, and hence, must be in the same class of the minimum nice partition. The algorithm examines, one by one, each label $\ell \in L(\pi)$, and builds a position π_ℓ by examing each vertex $v \in S_\ell$. If v is already covered by a position π', then π' will need to be merged with π_ℓ. If v is not already covered by some position, then we simply add v to π_ℓ. After merging a position π' with π_ℓ, it may be the case that π_ℓ contains two vertices from the same input tree. In such a case, the algorithm needs to merge with π_ℓ all of the positions covering any of the vertices from that input tree. Furthermore, since each $\ell \in L(\pi)$ can be in the subtree of at most one v per V_i, it follows that the first time two vertices from the same input tree end up in the same partition, those two vertices are unique and we add them to the set I of interesting vertices.

Theorem 1. GetSuccessors *can be implemented to run in $O(kn)$ time, and in the tuple (Π, I) returned, Π is exactly the successor positions of each class of the minimum nice partition P of G.*

If GetSuccessors determines that the minimum nice partition of G is a singleton, then the set Π returned has π as its only element. In this case, the set I of vertices returned by the algorithm is a set of interesting vertices for π.

Theorem 2. *If the set Π returned by* GetSuccessors *is a singleton with $\Pi = \{\pi\}$, then I is a set of interesting vertices for π.*

3.3 Testing for an Agreement Supertree

We conclude this section with an algorithm TestAgreement which takes as input a position π in a collection $\mathcal{T}$ of rooted phylogenetic trees, and tests whether there is an agreement supertree for π. If there is no agreement supertree for π, the algorithm returns an obstructing position π' and a set of interesting vertices for π'. Note that to test for the existence of an agreement supertree for $\mathcal{T}$, it suffices to call TestAgreement on the initial position $\pi_{\mathcal{T}}$. The set of interesting vertices returned by TestAgreement will be used in the remainder of the algorithms discussed in this paper. Theorem 3 states the runtime and correctness of the TestAgreement algorithm given in the listing of Algorithm 1. The runtime is proven in Lemma 4 and the correctness is proven in Lemma 5.

Lemma 4. *The implementation of* TestAgreement *given in Algorithm 1 runs in $O(kn^2)$ time.*

Proof. Since at each execution of a recursive call, the label sets in each position returned by GetSuccessors are disjoint, it follows that the recursion tree has $O(n)$ leaves. So there are $O(n)$ recursive calls to TestAgreement. Since each execution of the loop in line 5 results in a recursive call, and the body of the loop takes $O(1)$ time outside of the recursive call, it follows that the algorithm spends, over all recursive calls, a total of $O(n)$ time executing lines 5-7. Thus, it suffices to show that each recursive call spends at most $O(kn)$ time outside of lines 5-7. Clearly, lines 1 and 2 can be done in $O(k)$ time. The call to GetSuccessors takes $O(kn)$ time by Theorem 1. Line 4 and 8 can clearly be done in $O(1)$ time. □

Lemma 5. TestAgreement *correctly decides if there is an agreement supertree for a position π in $\mathcal{T}$, and in case there is no agreement supertree for π in $\mathcal{T}$,* TestAgreement *retuns a position π' in $\mathcal{T}$ for which there is no agreement supertree and I is a set of interesting vertices for π'.*

Proof. We show by induction on π, that TestAgreement$(\mathcal{T}, \pi)$ correctly decides if π has an agreement supertree, and, in case there is no agreement supertree for π, the position π'' and set I returned on line 7 are an obstructing position for $\mathcal{T}$ and a set of interesting vertices for π''.

Let P be the minimum nice partition of G.

There are two base cases: (i) when $\pi = \pi_\perp$; and (ii) when P has a single class. In case (i) there is an agreement supertree for π and the algorithm returns "yes" on line 2. In case (ii), by Lemma 2, there is no agreement supertree for π. By Theorem 1, the set Π returned in line 3 will be a singleton and π is an obstructing position. By Theorem 2, I is a set of interesting vertices for π. Line 4 returns π along with the set of interesting vertices returned by the call to GetSuccessors.

Now suppose that P has more than one class, Π is the set of successor positions returned by GetSuccessors, and, for each $\pi' \in \Pi$, TestAgreement$(\mathcal{T}, \pi')$ correctly decides whether there is an agreement supertree for π'. If there is an agreement supertree for each position in Π, then by Lemma 2, there is an agreement supertree for π and the algorithm returns "yes" on line 8. If there is no agreement supertree for some position $\pi' \in \Pi$, then by the inductive hypothesis, TestAgreement$(\mathcal{T}, \pi')$ will answer in the negative along with an obstructing position π'' and a set of interesting vertices for π''. By Lemma 1, π'' is an obstructing position for $\mathcal{T}$, so we return π'' along with the set I of interesting vertices returned by GetSuccessors. □

Theorem 3 follows directly from Lemmas 4 and 5.

Theorem 3. TestAgreement *can be implemented to run in $O(kn^2)$ time and correctly decides if there is an agreement supertree for a position π in $\mathcal{T}$. If there is no agreement supertree for π, it returns an obstructing position π' and a set I of interesting vertices for π'.*

```
Input: A position $\pi$ in a collection $\mathcal{T}$ of trees.
Output: A tuple $(B, \pi', I)$ where $B$ is a boolean indicating whether there is an
        agreement supertree for $\pi$, and, when $B$ is no, $\pi'$ is an obstructing
        position and $I$ is a set of interesting vertices for $\pi'$.
1 $\pi \leftarrow \pi\downarrow$
2 if $\pi = \pi_\perp$ then return $(\mathsf{yes}, \emptyset, \emptyset)$
3 $(\Pi, I) \leftarrow$ GetSuccessors$(\mathcal{T}, \pi)$
4 if $|\Pi| = 1$ then return $(\mathsf{no}, \pi, I)$
5 foreach $\pi' \in \Pi$ do
6     $(B, \pi'', I) \leftarrow$ TestAgreement$(\mathcal{T}, \pi')$
7     if $B = no$ then return $(\mathsf{no}, \pi'', I)$
8 return $(\mathsf{yes}, \emptyset, \emptyset)$
```

Algorithm 1. TestAgreement$(\mathcal{T}, \pi)$

4 FPT Algorithms

In this section we show that both the AST-EC and AST-TR problems are fixed-parameter tractable for parameters k and p, by giving $O((2k)^p kn^2)$ time algorithms for both problems.

We assume that there is no agreement supertree for the collection $\mathcal{T}$ of input trees, and that TestAgreement$(\mathcal{T}, \pi_\top)$ has returned the tuple (no, π, I). Thus, π is an obstructing position, and I is a set of interesting vertices for π.

4.1 An Auxiliary Algorithm

We define the *closure* of a set $C \subseteq V$ as the set $\langle C\rangle_G \subseteq V$ such that: (i) if $C \cap V_i = \emptyset$ then $\langle C\rangle_G \cap V_i = \emptyset$; (ii) if $C \cap V_i = \{v\}$, then $\langle C\rangle_G \cap V_i = \{v\}$; and (iii) if $|C \cap V_i| \geq 2$ then $\langle C\rangle_G \cap V_i = V_i$. Now, given G and I, we run the following algorithm, called Algorithm Merge. We maintain a partition P of I, initially P contains a class $\{v\}$ for each $v \in I$. At a given step, suppose that $P = \{C_1, \ldots, C_p\}$. If G contains a *transverse edge* joining $\langle C_i\rangle_G$ to $\langle C_j\rangle_G$ for some i, j, then we replace P by $P - \{C_i, C_j\} + \{C_i \cup C_j\}$, and we continue.

Lemma 6. *Let Q be the minimum nice partition of G. At a given step of Algorithm* Merge, *it holds that: for each $C \in P$, there is a component $K \in Q$ such that $\langle C\rangle_G \subseteq K$.*

The crucial property of Algorithm Merge is that, by starting with the interesting vertices, it will end with P consisting of a single class.

Lemma 7. *Suppose that* TestAgreement$(\mathcal{T}, \pi_\top)$ *has returned* (no, π, I). *Then Algorithm* Merge *run on G, I ends with the partition $\{I\}$.*

4.2 Solving the AST-EC Problem

The computational complexity of AST-EC does not seem to have been studied before. To motivate the development of a fixed-parameter algorithm to solve the AST-EC problem, we first prove the problem is NP-complete.

We use a recursive parenthesized notation for trees: if ℓ is a label, ℓ represents a tree with a single leaf labelled ℓ; if $T_1, \ldots, T_k$ are trees, then $(T_1, \ldots, T_k)$ represents the tree whose root is unlabelled and has $T_1, \ldots, T_k$ as child subtrees.

Theorem 4. AST-EC *is* NP-*hard.*

Proof. We reduce the MULTICUT problem to the AST-EC problem. Given a graph $G = (V, E)$ and a set of requests $R \subseteq V \times V$, the MULTICUT problem asks if there exists a set S of at most p edges in E, where, for every $uv \in R$, u and v are in different components of $G \backslash S$.

Given an instance $I = (G, R, p)$ of MULTICUT, we build an instance $I' = (\mathcal{T}, p)$ of the AST-EC problem as follows. The collection $\mathcal{T}$ is defined over the label set $V \cup \{x\}$. For every $uv \in E$, add the tree $((u, v), x)$ to $\mathcal{T}$, and, for every request $uv \in R$, add the tree (u, v, x) to $\mathcal{T}$. The answer to I is yes if and only if the answer to I' is yes. $\square$

In the remainder of this subsection, we show that AST-EC is fixed-parameter tractable in parameters k and p. Lemma 8 shows that if a call to TESTAGREEMENT$(\mathcal{T}, \pi_\top)$ answers negatively, we must contract at least one edge joining an interesting vertex to its parent.

Lemma 8. *Suppose that* TESTAGREEMENT$(\mathcal{T}, \pi_\top)$ *has returned a tuple* (no, π, I). *In order to obtain a collection having an agreement supertree, we need to contract one edge* $\{v,\ parent(v)\}$ *with* $v \in I$.

Proof. Assume that we have a set $S \subseteq \hat{E}(\mathcal{T})$ which contains none of the edges $\{v,\ parent(v)\}$ with $v \in I$; we show that $\mathcal{T}/S$ has no agreement supertree. By definition of S, each element of I is still a node of $\mathcal{T}/S$. Define the position π' in $\mathcal{T}/S$ as follows. If $\pi[i] = \perp$, then $\pi'[i] = \perp$. If $\pi[i] = u$ is the parent of two vertices $v, w \in I$, then $\pi'[i]$ is the common parent of v, w in T_i/S. Let $G' = G(\mathcal{T}/S, \pi') = (V', E')$. We show that G' has a unique nice component.

Let us consider an execution $\mathcal{E}$ of Algorithm MERGE on G and I. We claim that $\mathcal{E}$ can be simulated by an execution $\mathcal{E}'$ of Algorithm MERGE on G' and I. Let $P_{\mathcal{E}}, P_{\mathcal{E}'}$ denote the values of P during $\mathcal{E}, \mathcal{E}'$ respectively. We show by induction that at each step of $\mathcal{E}, \mathcal{E}'$, we have $P_{\mathcal{E}} = P_{\mathcal{E}'}$. This holds clearly at the beginning of $\mathcal{E}, \mathcal{E}'$. Suppose that this holds at the beginning of step s. Then $\mathcal{E}$ picks an edge induced by some label $\ell \in L(\langle C_i \rangle_G) \cap L(\langle C_j \rangle_G)$. The important observation is that given $C \subseteq I$, we have $L(\langle C \rangle_G) \subseteq L(\langle C \rangle_{G'})$ (as $L(\pi[i]) \subseteq L(\pi'[i])$ for every $i \in [k]$). Hence, $\ell \in L(\langle C_i \rangle_{G'}) \cap L(\langle C_j \rangle_{G'})$, and thus $\mathcal{E}'$ can pick an edge joining $\langle C_i \rangle_{G'}$ and $\langle C_j \rangle_{G'}$. This leads to the merge of C_i and C_j, and thus $P_{\mathcal{E}} = P_{\mathcal{E}'}$ at the end of step s.

Applying the induction hypothesis at the last step of $\mathcal{E}$, and using that $\mathcal{E}$ ends with the partition $\{I\}$ (Lemma 7), we obtain that $\mathcal{E}'$ ends with the partition $\{I\}$. By Lemma 6, if Q is the minimum nice partition of G', we have $\langle I \rangle_{G'} = V'$ in a same component of Q, and thus $\mathcal{T}/S$ has no agreement supertree by Lemmas 1 and 2. $\square$

Since TESTAGREEMENT returns a set of at most $2k$ interesting vertices, Lemma 8 leads to an FPT algorithm for AST-EC using a bounded search tree technique.

Theorem 5. AST-EC *can be solved in* $O((2k)^p kn^2)$ *time.*

Proof. We use a recursive algorithm SOLVEAST-EC$(\mathcal{T}, p)$. The algorithm answers "no" if $p < 0$. Otherwise, it runs TESTAGREEMENT$(\mathcal{T}, \pi^\top)$ to decide in $O(kn^2)$ if $\mathcal{T}$ has an agreement supertree. It answers "yes" in case in positive answer. In case of negative answer, it obtains a set I of nodes of $\mathcal{T}$; for each non-leaf vertex $v \in I$, it recursively calls SOLVEAST-EC$(\mathcal{T}/\{v, parent(v)\}, p-1)$. The algorithm then answers "yes" iff one of the recursive calls does. The correctness of the algorithm follows from Lemma 8, and the running time is $O((2k)^p kn^2)$. □

4.3 Solving the AST-TR Problem

We will say that a set $C \subseteq L(\mathcal{T})$ is a *conflict among* $\mathcal{T}$ if $\mathcal{T}|C$ has no agreement supertree. Lemma 9 shows that if TESTAGREEMENT$(\mathcal{T}, \pi_\top)$ answers negatively, we can obtain a conflict among $\mathcal{T}$ from the set of interesting vertices.

Lemma 9. *Suppose that* TESTAGREEMENT$(\mathcal{T}, \pi_\top)$ *has returned a tuple* (no, π, I)*. We can then obtain a set* $C \subseteq L$ *of size at most* $2k-1$ *such that* $\mathcal{T}|C$ *has no agreement supertree.*

Proof. Consider an execution $\mathcal{E}$ of Algorithm MERGE on G and I. For each edge $e = uv$ found by $\mathcal{E}$, pick a label $\ell_e \in L(u) \cap L(v)$, and let C be the resulting set of labels. Clearly $|C| \leq 2k-1$. Consider a vertex $v \in I \cap V_i$, then during $\mathcal{E}$ consider the first time that $\{v\}$ is merged with a component C; it corresponds to some label $\ell_v \in L(v) \cap C$. It follows that $\pi[i]$ is still a node of $T_i|C$, let v' denote the child of $\pi[i]$ in $T_i|C$ that contains ℓ_v. Let $I' = \{v' : v \in I\}$, and let $G' = G(\mathcal{T}|C, \pi) = (V', E')$. We show that G' has a unique nice component.

We claim that the execution $\mathcal{E}$ can be simulated by an execution $\mathcal{E}'$ of Algorithm MERGE on G' and I'. Let $P_\mathcal{E}, P_{\mathcal{E}'}$ denote the values of P during $\mathcal{E}, \mathcal{E}'$ respectively. We show by induction that at each step of $\mathcal{E}, \mathcal{E}'$, if $P_\mathcal{E} = \{C_1, \ldots, C_p\}$, then $P_{\mathcal{E}'} = \{C'_1, \ldots, C'_p\}$ with $C'_i = \{v' : v \in C_i\}$. This holds clearly at the beginning of $\mathcal{E}, \mathcal{E}'$. Suppose that this holds at the beginning of step s. Then $\mathcal{E}$ picks an edge $e = uv$ with $u \in \langle C_i \rangle_G, v \in \langle C_j \rangle_G$. Suppose that $u \in V(T_p)$ and $v \in V(T_q)$. In $T_p|C$ (resp. $T_q|C$), there is a child u' of $\pi[p]$ (resp. a child v' of $\pi[q]$) that contains ℓ_e. The induction hypothesis implies that $u' \in \langle C'_i \rangle_{G'}$ and $v' \in \langle C'_j \rangle_{G'}$, thus $\mathcal{E}'$ can pick the edge $e' = u'v'$ induced by label ℓ_e. This leads to merge C'_i and C'_j and thus the induction hypothesis holds at the end of step s.

Applying the induction hypothesis at the last step of $\mathcal{E}$, and since $\mathcal{E}$ ends with the partition $\{I\}$ (Lemma 7), we conclude that $\mathcal{E}'$ ends with the partition $\{I'\}$. By Lemma 6, if Q is the minimum nice partition of G', we have $\langle I' \rangle_{G'} = V'$ in a same component of Q, and thus $\mathcal{T}|C$ has no agreement supertree by Lemmas 1 and 2. □

We outline an algorithm FINDOBSTRUCTION that takes as input a set of interesting vertices and returns a conflict among $\mathcal{T}$ of size at most $2k-1$. Suppose that $I = \{v_1, \ldots, v_r\}$. We initialize components $C_1, \ldots, C_r$ with $C_i = \{v_i\}$, and

```
**Input**: A collection $\mathcal{T} = \{T_1, \dots, T_k\}$ of rooted trees, an obstructing position $\pi$
    in $\mathcal{T}$, a set $I = \{v_1, \dots, v_r\}$ of interesting vertices for $\pi$.
**Output**: A conflict among $\pi$.
if $2k \geq n$ then return $L$
$R \leftarrow \emptyset$ ; $J \leftarrow \{1, \dots, r\}$
for $i$ from $1$ to $r$ do
    $C_i \leftarrow \{v_i\}$
    foreach $\ell \in L(v_i)$ do $J(\ell) \leftarrow J(\ell) \cup \{i\}$
for $s$ from $1$ to $r-1$ do
    Pick $\ell \in L$ such that $|J(\ell)| \geq 2$, and choose $i, j \in J(\ell)$
    $R \leftarrow R \cup \{\ell\}$
    for $\ell \in L$ do
        if $j \in J(\ell)$ then $J(\ell) \leftarrow J(\ell) \backslash \{j\} \cup \{i\}$
    for $p$ from $1$ to $k$ do
        if $C_i \cap V_p \neq \emptyset$ and $C_j \cap V_p \neq \emptyset$ then
            foreach $\ell \in L(\pi[p])$ do $J(\ell) \leftarrow J(\ell) \cup \{i\}$
    $C_i \leftarrow C_i \cup C_j$, $J \leftarrow J \backslash \{j\}$
return $R$
```

Algorithm 2. FindObstruction($\mathcal{T}, \pi, I$)

we let $J = \{1, \dots, r\}$. We use a main loop which performs the $r-1$ steps of Algorithm Merge. At each step, we have $J \subseteq \{1, \dots, r\}$, and the current partition is represented by the components C_i ($i \in J$), which are called the *active* components. We maintain for each $\ell \in L$ a variable $J(\ell) = \{i \in J : \ell \in L(\langle C_i \rangle_G)\}$. At a given step, we have to find a label inducing an edge which joins the closure of two active components. This amounts to looking for an ℓ such that $|J(\ell)| \geq 2$. Once such an ℓ has been found, we pick two indices $i, j \in J(\ell)$, and we merge the components C_i, C_j, letting C_i be the newly created component, and updating the variables $J(\ell)$ accordingly. An implementation of FindObstruction is given in the listing of Algorithm 2.

Lemma 10. *Suppose that* TestAgreement($\mathcal{T}, \pi_\top$) *returns* (no, π, I). *Then Algorithm* FindObstruction($\mathcal{T}, \pi, I$) *returns in $O(kn)$ time a conflict among $\mathcal{T}$ of size at most $2k-1$.*

Proof. We first argue for the correctness. If $2k \geq n$, then the set L returned by the algorithm is a conflict, as by assumption $\mathcal{T}$ has no agreement supertree. Suppose now that $2k < n$. By Lemma 9, it suffices to show that an execution $\mathcal{E}$ of Algorithm 2 simulates an execution $\mathcal{E}'$ of Algorithm Merge. More precisely, we can show the following: at each step s,

1. there is an execution $\mathcal{E}'$ of s steps of Algorithm Merge which produces the set of components C_i ($i \in J$) and the set of labels R,
2. for each $\ell \in L$, $J(\ell) = \{i \in J : \ell \in L(\langle C_i \rangle_G)\}$.

This is shown by induction on s. The initialization of the variables C_i and $J(\ell)$ in Lines 3-5 ensure that this is true initially. Suppose that this holds at the beginning of step s. The choice of ℓ and i, j in Line 7 ensures that $\ell \in L(\langle C_i \rangle_G) \cap L(\langle C_j \rangle_G)$, and thus there exists an edge e between $\langle C_i \rangle_G$ and $\langle C_j \rangle_G$, induced by the label ℓ. The update of $C_i \leftarrow C_i \cup C_j$ and $J \leftarrow J \backslash \{j\}$ reflect the merge of C_i and C_j, and thus we can simulate step s of Algorithm MERGE which would choose edge e and merge C_i and C_j. This establishes Point 1, and the update of $J(\ell)$ in Lines 9-13 ensures that Point 2 is preserved.

We now justify the running time. Let us assume that $2k < n$, as otherwise the algorithm takes $O(1)$ time. We implement the sets $J(\ell)$ by bit arrays, allowing in constant time the following operations: (i) insertion or deletion of an element, (ii) obtaining the size of the set. It follows that Lines 3-5 take $O(rn) = O(kn)$ time. Let us now analyze the time taken by step s of the main loop. Let K_s denote the set of indices p for which the condition of Line 12 holds. Then Lines 7-10 take $O(n)$ time, Lines 11-13 take $O(k + |K_s|n)$ time, and Line 14 takes $O(k)$ time. Overall, Lines 7-14 take $O(n + |K_s|n)$ time as $k = O(n)$. Observe that the sets K_s are disjoint for $s = 1, \ldots, r-1$, and thus $\sum_s |K_s| = O(k)$. It follows that the loop of Lines 6-14 take $O(rn + kn) = O(kn)$ time, and we conclude that the whole algorithm runs in $O(kn)$ time. □

Theorem 6. AST-TR *can be solved in* $O((2k)^p kn^2)$ *time.*

Proof. We use a recursive algorithm SOLVEAST-TR$(\mathcal{T}, p)$. The algorithm answers "no" if $p < 0$. Otherwise, it runs TESTAGREEMENT$(\mathcal{T}, \pi^\top)$ to decide in $O(kn^2)$ if $\mathcal{T}$ has an agreement supertree. It answers "yes" in case of positive answer. In case of negative answer, it obtains a position π and a set I of interesting nodes for π. It calls FINDOBSTRUCTION$(\mathcal{T}, \pi, I)$ to obtain in $O(kn)$ time a conflict C among $\mathcal{T}$ of size at most $2k - 1$. Then, for each $\ell \in C$, it recursively calls SOLVEAST-TR$(\mathcal{T}|(L(\mathcal{T}) \setminus \{\ell\}), p-1)$, and it answers "yes" iff one of the recursive calls does. The correctness follows from Lemma 10, and the running time is $O((2k)^p kn^2)$. □

5 Concluding Remarks

We have given $O((2k)^p kn^2)$ time algorithms for both the AST-EC and AST-TR problems, thus showing they are fixed-parameter tractable for parameters k and p. We remark here that the bounds given for the cardinality of the obstruction sets for AST-EC of $2k$ (Lemma 8), and for AST-TR of $2k - 1$ (Lemmas 9 and 10), are both tight.

Our proof that AST-EC is NP-hard relies on a reduction from the parameterized MULTICUT problem to the AST-EC problem parameterized by p. As MULTICUT is fixed-parameter tractable [12], this leaves open the question of whether AST-EC could be fixed-parameter tractable in p only. It is known that AST-TR is fixed-parameter intractable for parameter p [2].

Our focus here was on agreement supertrees. A compatible supertree is one that contains a refinement of each of the input trees. There are natural analogs of AST-EC and AST-TR for compatible supertrees. For binary input trees, compatibility is equivalent to agreement, so the results of [6] imply fixed-parameter tractability. However, for input trees of arbitrary degree, we have established that any upper bound on the cardinality of an obstruction set is at least $c2^k$. Hence, the techniques given here are unlikely to imply efficient fixed-parameter tractability for the analogs of AST-TR and AST-EC to compatible supertrees.

There are also analogs of both AST-EC and AST-TR to unrooted trees. Although SMAST has been studied for unrooted trees [2,7], the AST-EC and AST-TR problems for unrooted trees do not seem to have been studied before.

References

1. Aho, A.V., Sagiv, Y., Szymanski, T.G., Ullman, J.D.: Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. SIAM J. Comput. 10(3), 405–421 (1981)
2. Berry, V., Nicolas, F.: Maximum agreement and compatible supertrees. J. Discrete Algorithms 5(3), 564–591 (2007)
3. Bininda-Emonds, O.R.P. (ed.): Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life. Ser. on Comp. Biol., vol. 4. Springer (2004)
4. Bryant, D.: Optimal Agreement Supertrees. In: Gascuel, O., Sagot, M.-F. (eds.) JOBIM 2000. LNCS, vol. 2066, pp. 24–31. Springer, Heidelberg (2001)
5. Gordon, A.D.: Consensus supertrees: The synthesis of rooted trees containing overlapping sets of labelled leaves. J. Classification 9, 335–348 (1986)
6. Guillemot, S., Berry, V.: Fixed-parameter tractability of the maximum agreement supertree problem. IEEE/ACM Trans. Comput. Biology Bioinform. 7(2), 342–353 (2010)
7. Hoang, V.T., Sung, W.K.: Improved algorithms for maximum agreement and compatible supertrees. Algorithmica 59(2), 195–214 (2011)
8. Jansson, J., Ng, J.H.K., Sadakane, K., Sung, W.K.: Rooted maximum agreement supertrees. Algorithmica 43(4), 293–307 (2005)
9. Kao, M.Y.: Encyclopedia of algorithms. Springer, New York (2007)
10. Linder, C.R., Rieseberg, L.H.: Reconstructing patterns of reticulate evolution in plants. Am. J. Bot. 91(10), 1700–1708 (2004)
11. Maddison, W.: Gene trees in species trees. Systematic Biology 46(3), 523–536 (1997)
12. Marx, D., Razgon, I.: Fixed-parameter tractability of multicut parameterized by the size of the cutset. In: STOC 2011, pp. 469–478. ACM (2011)
13. Ng, M., Wormald, N.: Reconstruction of rooted trees from subtrees. Discrete Appl. Math. 69(1-2), 19–31 (1996)
14. Scornavacca, C.: Supertree methods for phylogenomics. Ph.D. thesis, Univ. of Montpellier II, Montpellier, France (2009)
15. Steel, M.A.: The complexity of reconstructing trees from qualitative characters and subtrees. J. Classification 9, 91–116 (1992)

Computing the Rooted Triplet Distance between Galled Trees by Counting Triangles

Jesper Jansson[1,⋆] and Andrzej Lingas[2,⋆⋆]

[1] Laboratory of Mathematical Bioinformatics, Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan
jj@kuicr.kyoto-u.ac.jp
[2] Department of Computer Science, Lund University, 22100 Lund, Sweden
Andrzej.Lingas@cs.lth.se

Abstract. We consider a generalization of the rooted triplet distance between two phylogenetic trees to two phylogenetic networks. We show that if each of the two given phylogenetic networks is a so-called galled tree with n leaves then the rooted triplet distance can be computed in $o(n^{2.688})$ time. Our upper bound is obtained by reducing the problem of computing the rooted triplet distance to that of counting monochromatic and almost-monochromatic triangles in an undirected, edge-colored graph. To count different types of colored triangles in a graph efficiently, we extend an existing technique based on matrix multiplication and obtain several new related results that may be of independent interest.

1 Introduction

Phylogenetic trees and their generalization to non-treelike structures, *phylogenetic networks*, are commonly used by scientists to describe evolutionary relationships among a set of objects such as biological species or natural languages [2, 3, 6–8, 10–14]. Various metrics for measuring the (dis-)similarity of two given phylogenetic trees have been proposed and analyzed in the literature; see, e.g., [2] and the references therein. In this paper, we consider an extension of one particular, well-known method called the *rooted triplet distance* [2, 6] to the phylogenetic network model and describe how to compute it efficiently.

The rooted triplet distance between two phylogenetic trees provides an intuitive measure of their dissimilarity and exhibits many attractive mathematical properties [2, 6]. It counts the number of substructures (more precisely, subtrees induced by three leaves) that differ between the two trees. More formally, it is defined as follows. A *rooted phylogenetic tree* is an unordered, rooted tree in which every internal node has at least two children and all leaves are distinctly labeled. A rooted phylogenetic tree with three leaves is called a *rooted triplet*. A *non-binary* rooted triplet leaf-labeled by $\{a, b, c\}$ is called a *rooted fan triplet* and is denoted by $a|b|c$ (see the leftmost tree in Fig. 1), and a *binary* rooted triplet

⋆ Funded by The Hakubi Project and KAKENHI grant number 23700011.
⋆⋆ Research supported in part by VR grant 621-2008-4649.

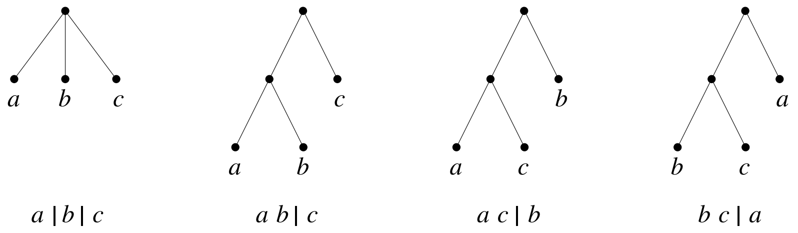


Fig. 1. The rooted fan triplet $a|b|c$ and the rooted proper triplets $ab|c$, $ac|b$, and $bc|a$

is called a *rooted proper triplet*; in the latter case, there are three possibilities, denoted by $ab|c$, $ac|b$, and $bc|a$, corresponding to the three possible topologies (see also Fig. 1). A rooted triplet t is said to be *consistent with* a rooted phylogenetic tree T if t is an embedded subtree of T. [1] Now, given two rooted phylogenetic trees T_1, T_2 with the same set L of leaf labels, the *rooted triplet distance* $d_{rt}(T_1, T_2)$ is the number of rooted triplets over L that are consistent with exactly one of T_1 and T_2.

The naive algorithm for computing $d_{rt}(T_1, T_2)$ between two trees T_1 and T_2 with a leaf label set of cardinality n runs in $O(n^3)$ time: Just preprocess T_1 and T_2 in $O(n)$ time so that lowest common ancestor queries can be answered in $O(1)$ time by the method in [9, 17], and then check each of the $O(n^3)$ possible rooted triplets for consistency with T_1 and T_2 in $O(1)$ time. Critchlow *et al.* [6] provided a more efficient algorithm for computing the rooted triplet distance between two *binary* phylogenetic trees with $O(n^2)$ running time, and Bansal *et al.* [2] extended the $O(n^2)$-time upper bound to two phylogenetic trees of *arbitrary* degrees.

Due to the recently increasing focus on phylogenetic networks (see, e.g., the two new textbooks [10, 13]), it is compelling to consider generalizations of the rooted triplet distance to the network case. For this case, it seems much harder to improve on the naive $O(n^3)$-time algorithm and to derive a subcubic upper bound on the running time. Therefore, one would like to know if any important special classes of phylogenetic networks such as the *galled trees* [8, 10] admit fast algorithms for the rooted triplet distance. Galled trees are structurally restricted phylogenetic networks in which all underlying cycles are vertex-disjoint (for a detailed definition, refer to Section 2.2 below). This kind of phylogenetic network was first considered by Wang *et al.* [18] and later by Gusfield *et al.* [8], and is also known in the literature as a *level*-1 *phylogenetic network* [4, 10]. Galled trees have turned out to be useful in certain settings where reticulation events do occur but are known to be rare. [2] As a consequence, a number of algorithms for building galled trees from different kinds of data have been published [3, 8, 10–12].

[1] There are several equivalent ways to define this. For example, for any two leaf labels x, y, let $lca^T(x, y)$ denote the lowest common ancestor in T of the leaves labeled by x and y. Then $a|b|c$ is *consistent with* T if $lca^T(a, b) = lca^T(a, c) = lca^T(b, c)$, and $ab|c$ is *consistent with* T if $lca^T(a, b)$ is a proper descendant of $lca^T(a, c) = lca^T(b, c)$. See also Section 2.1 below.

[2] See [8] for a discussion about the biological relevance of galled trees.

1.1 New Results

The main contribution of our paper is an $o(n^{2.688})$-time algorithm for computing the rooted triplet distance between two galled trees with n leaves each [Theorem 4]. From a computational complexity point of view, this is significant because it breaks the natural $O(n^3)$-time barrier for any kind of non-tree phylogenetic networks for the first time. The precise running time is $O(n^{(3+\omega)/2})$, where ω denotes the exponent in the running time of the fastest existing method for matrix multiplication. It is well known that $\omega < 2.376$ [5], and recent developments [16, 20] suggest slightly tighter bounds on ω.

Our main result is obtained in part by a reduction to the problem of counting monochromatic and "almost-monochromatic" triangles in an undirected graph with colored edges. To solve the latter efficiently, we strengthen a technique based on matrix multiplication used in [1] and [19] for *detecting* if a graph contains a triangle to also *count* the number of triangles in the graph. More exactly, we show that:

- The number of triangles in a connected, undirected graph with m edges can be computed in $O(m^{\frac{2\omega}{\omega+1}}) \leq o(m^{1.408})$ time [Theorem 1].
- If G is a connected, undirected, edge-colored graph with n vertices and C is a subset of the set of edge colors then the number of monochromatic triangles of G with colors in C can be computed in $O(n^{(3+\omega)/2}) \leq o(n^{2.688})$ time [Theorem 2].

We also need to relax the concept of a monochromatic triangle to what we call an R-chromatic triangle (see Section 3 for the definition), and obtain:

- If G is a connected, undirected, edge-colored graph with n vertices and R is a binary relation on the colors that is computable in $O(1)$ time then the number of R-chromatic triangles in G can be computed in $O(n^{(3+\omega)/2}) \leq o(n^{2.688})$ time [Theorem 3].

Our new results on counting triangles in a graph may be of general interest and could be useful in other applications unrelated to the main problem studied here.

2 Preliminaries

2.1 Basic Definitions

A (rooted) *phylogenetic network* U is a directed acyclic graph with a single root vertex and a set L of distinctly labeled leaves, and no vertices having both indegree 1 and outdegree 1. A vertex u is an *ancestor* of a vertex v (or, equivalently, v is a descendant of u) in U if and only if there is a directed path from u to v in U. In particular, u is an ancestor and descendant of itself. If the path from u to v has non-zero length then v is a *proper descendant* of u. Next, a vertex w is a *common ancestor* of vertices u and v in U if and only if w is an ancestor of both u and v in U. Furthermore, w is a *junction common ancestor* (jca) of u

and v in U if and only if there are two directed non-zero length paths from w to u and v, respectively, which are vertex disjoint but for the start vertex w. Finally, w is a *lowest common ancestor* (lca) of u and v in U if and only if: (1) w is a common ancestor of u and v; and (2) w has no proper descendant that is a common ancestor of u and v. As an example, in Fig. 2 (i), vertices w and z are two different jca's of a and c, w is an lca of a and c, and z is not an lca of a and c.

We now define rooted triplet consistency for a phylogenetic network U. Following [10, 11], for any three leaf labels a, b, c, say that the rooted proper triplet $ab|c$ is *consistent with* U if and only if U contains a junction common ancestor w of a and b as well as a junction common ancestor z of c and w such that there are four directed paths from w to a, from w to b, from z to w, and from z to c that are vertex disjoint except for in the vertices w and z. Secondly, say that the rooted fan triplet $a|b|c$ is *consistent with* U if and only if U contains a vertex w such that there are three directed paths from w to a, from w to b, and from w to c that are vertex-disjoint except for in the common start vertex w. Observe that in the special case where U is a tree, the concepts of a lowest common ancestor and a junction common ancestor between two leaves coincide, and the definitions of rooted triplet consistency thus reduce to the definitions in footnote 1.

Next, we define the rooted triplet distance between phylogenetic networks as:

Definition 1. *Let U_1, U_2 be two phylogenetic networks on the same leaf label set L. The* rooted triplet distance between U_1 and U_2*, denoted by $d_{rt}(U_1, U_2)$, is the number of rooted fan triplets and rooted proper triplets with leaf labels from L that are consistent with exactly one of U_1 and U_2.*

This definition of d_{rt} differs slightly from the one restricted to trees in [2, 6]. The definition in [2, 6] counts the number of "bad" cardinality-3 subsets L' of L for which the rooted triplet with leaf set L' consistent with U_1 differs from the rooted triplet with leaf set L' consistent with U_2. Therefore, when restricted to trees, our definition of d_{rt} is exactly two times d_{rt} from [2, 6] because each "bad" subset will contribute twice to our d_{rt} (once for the rooted triplet in U_1 and once for the rooted triplet in U_2); obviously, our definition of d_{rt} could be normalized by dividing by two but then d_{rt} would no longer always be an integer in the non-tree case. We believe that our definition is more suitable in the context of phylogenetic networks because it allows us to distinguish between cases such as: (i) $ab|c$ and $bc|a$ are consistent with U_1 whereas only $bc|a$ is consistent with U_2; and (ii) $ab|c$ is consistent with U_1 and $bc|a$ is consistent with U_2.

2.2 Galled Trees

Here, we recall the definition of the class of phylogenetic networks called the *galled tree* [8, 10], and investigate some of its properties.

A *reticulation vertex* of a phylogenetic network is any vertex of indegree greater than 1. For any phylogenetic network U, define *its underlying undirected graph* as the undirected graph obtained by replacing every directed edge in U by an undirected edge. A phylogenetic network U is called a *galled tree* if all cycles in its underlying undirected graph are vertex-disjoint [8, 10]. A *cycle* C in a galled tree is any set of vertices that induce a cycle in the underlying undirected graph, and the vertex of C in U that is an ancestor of all vertices on C is called the *root* of C. Thus, every cycle C in a galled tree has exactly one root and one reticulation vertex, and C consists of two directed paths from its root to its reticulation vertex. Also, any directed path from the root of the galled tree to a vertex on such a cycle must pass through the root of the cycle. The next lemma summarizes some useful properties of galled trees:

Lemma 1. *Let U be a galled tree with n leaves and let u, v be any two vertices in U. Then:*

1. *The lowest common ancestor in U of u and v is unique.*
2. *There are at most two different junction common ancestors of u and v.*
3. *If there are two junction common ancestors of u and v then both of them lie on the same cycle C in U. Furthermore, one of them is the root of C and the other one is the lowest common ancestor of u and v in U.*
4. *The number of vertices in U as well as the number of edges in U is $O(n)$.*
5. *All junction common ancestors of pairs of vertices in U can be listed in $O(n^2)$ time.*

Proof. To prove property 1, suppose there were two different lowest common ancestors w_1 and w_2 of u and v in U. Consider any path from w_1 to u in U and any path from w_2 to u in U. Since both paths lead to u, they must meet at some ancestor u' of u which then has indegree larger than 1, where u' is a proper descendant of w_1 and also a proper descendant of w_2. In the same way, there exists an ancestor v' of v with indegree larger than 1 which is a proper descendant of both w_1 and w_2, with $u' \neq v'$. Now let x be a lowest common ancestor of w_1 and w_2 in U. In the underlying undirected graph of U, there is a cycle containing x and u' and another cycle containing x and v', i.e., two non-vertex-disjoint cycles, contradicting the definition of a galled tree.

Next, we prove properties 2 and 3. For each cycle in U, arbitrarily term one of the two edges on C incident to the reticulation vertex as *the left reticulation edge* and the other one as *the right reticulation edge*. Let U_L be the tree obtained from U by removing all right reticulation edges in U and define U_R symmetrically. Then, every junction common ancestor of u and v in U is a lowest common ancestor of u and v in at least one of U_L and U_R. Property 2 follows. According to the definitions, if w is a lowest common ancestor of u and v in U then w is also a junction common ancestor of u and v in U, which yields property 3.

To upper-bound the number of vertices in U, construct a *binary* galled tree U' (where every vertex has outdegree at most 2) by repeatedly selecting any vertex w with outdegree larger than 2 and replacing any two of its outgoing edges (w, c_1) and (w, c_2) by a single edge (w, x) and two edges (x, c_1) and (x, c_2) where x is a newly created vertex, until no vertex with outdegree larger than 2 remains. This will not introduce any vertices having both indegree 1 and outdegree 1, and U' is still a galled tree with n leaves, but U' contains at least as many vertices as U. According to Lemma 3 in [4], the number of vertices in any binary galled tree U' with n leaves is $O(n)$, so this gives an upper bound for U as well. Furthermore, any vertex in a galled tree can have indegree at most 2 (otherwise, there would exist two non-vertex-disjoint cycles in the underlying undirected graph), so the total number of edges in U is $O(n)$. Thus, property 4 holds.

Finally, since the trees U_L, U_R can be preprocessed in linear time to answer ancestor or descendant queries as well as *lca* queries in constant time [9, 17], and lca's in a tree are unique, property 5 follows. □

When the phylogenetic network U is a galled tree, the definitions of consistency of a rooted proper triplet $ab|c$ or a rooted fan triplet $a|b|c$ with U can be expressed as in Lemma 2 and Lemma 3 below. (These two key lemmas are used by our main algorithm in Section 4 to efficiently count the number of shared rooted triplets in two galled trees.) See Fig. 2 for some examples.

A junction common ancestor z of two vertices u, v in U is said to *use* another vertex w if, after the removal of w from U, the vertex z is no longer a junction common ancestor of u and v.

Lemma 2. *Let U be a galled tree. For any three leaves a, b, c in the leaf label set of U, the rooted proper triplet $ab|c$ is consistent with U if and only if U contains a junction common ancestor w of a and b as well as a different junction common ancestor z of c and w such that if both w and z belong to the same cycle C of U then at least one of them does not use the reticulation vertex of C.*

Proof. The necessity of the condition stated in the lemma follows directly from the definition of consistency of $ab|c$ with U. It remains to show the sufficiency of this condition for galled trees.

The proof is by contradiction. First of all, the path from z to w crosses neither that from w to a or that from w to b since U is an acyclic directed graph. Next, if the path from z to c had to cross that from w to a (or b, respectively) in an inner vertex x then z and w would lie on a common cycle whose reticulation vertex is exactly x, and both would use x. We obtain a contradiction. □

Lemma 3. *Let U be a galled tree, and let a, b, c be three leaf labels in U. The rooted fan triplet $a|b|c$ is consistent with U if and only if there exists a vertex w in U such that: (1) w is a junction common ancestor of all three pairs of leaves $\{a, b\}, \{a, c\}, \{b, c\}$; and (2) w is the lowest common ancestor of at least two pairs of leaves among $\{a, b\}, \{a, c\}, \{b, c\}$.*

Proof. $\Rightarrow$) Suppose $a|b|c$ is consistent with U. Then U contains a vertex w such that there are three directed paths P_a, P_b, and P_c from w to a, b, and c, respectively, that are vertex-disjoint except for in the common start vertex w. Thus, property (1) always holds. Next, since U is a galled tree, at most two of the three paths P_a, P_b, and P_c overlap with edges from the same cycle in U. Clearly, if none of them overlap with the same cycle then $lca(a, b) = lca(a, c) = lca(b, c) = w$; on the other hand, if w.l.o.g. P_a and P_b overlap with the same cycle then no path from w to c can intersect P_a or P_b except for in the starting vertex w, so $lca(a, c) = lca(b, c) = w$.

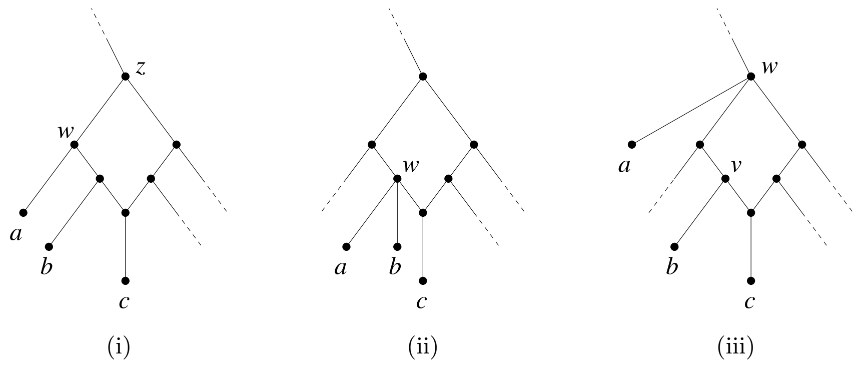


Fig. 2. Illustrating Lemmas 2 and 3. In (i), w is a jca of a and b that does not use the reticulation vertex, and z is a jca of c and w, so Lemma 2 gives us the rooted proper triplet $ab|c$. In (ii), w is a jca and also the lca for all three pairs $\{a, b\}, \{a, c\}, \{b, c\}$, so the network is consistent with $a|b|c$ according to Lemma 3. Similarly, in (iii), w is a jca for all three pairs $\{a, b\}, \{a, c\}, \{b, c\}$ and the lca for exactly two pairs $\{a, b\}, \{a, c\}$, so the network is consistent with $a|b|c$ according to Lemma 3. Note that in addition to the above, Lemma 2 also correctly identifies $bc|a$ in (i), $ab|c$ in (ii), and $bc|a$ in (iii).

$\Leftarrow$) Suppose there exists a vertex w that satisfies properties (1) and (2). There are two cases.

- First case: w is the lca of all three pairs of leaves in $\{a, b, c\}$, i.e., $w = lca(a, b) = lca(a, c) = lca(b, c)$. By the definition of a lowest common ancestor, no proper descendant of w can be an ancestor of any two of the three leaves $\{a, b, c\}$. Hence, there are three internally vertex-disjoint paths $w \leadsto a$, $w \leadsto b$, and $w \leadsto c$, i.e., $a|b|c$ is consistent with U. See also Fig. 2 (ii).
- Second case: w is the lca of exactly two pairs of leaves in $\{a, b, c\}$, say $w = lca(a, b) = lca(a, c)$ but $w \neq lca(b, c) = v$. Then there are two junction common ancestors of b and c, namely w and v, so by Lemma 1, both v and w lie on the same cycle C in U. Note that exactly one of the leaves b and c is a descendant of the reticulation vertex of C. Let P_b and P_c be two internally disjoint paths from w to b and c, respectively, where one of P_b and P_c passes through the reticulation vertex of C and the other one passes through v. Since $w = lca(a, b)$, there is no path from w to a that intersects P_b. In the same way, since $w = lca(a, c)$, there is no path from w to a that intersects P_c. Thus, there are three internally vertex-disjoint paths $w \leadsto a$, $w \leadsto b$, and $w \leadsto c$, so $a|b|c$ is consistent with U. See also Fig. 2 (iii). $\square$

3 Counting Monochromatic and Almost-Monochromatic Triangles

A *triangle* in an undirected graph is a cycle of length 3. In [1], Alon *et al.* showed how to determine if a connected, undirected graph with m edges contains a triangle, and if so, how to find a triangle in $O(m^{\frac{2\omega}{\omega+1}}) \leq o(m^{1.408})$ time. In the same paper, they also showed how to *count* the number of triangles in an undirected graph with n vertices in $O(n^{\omega}) \leq o(n^{2.376})$ time. We first improve their technique to count the number of triangles more efficiently in case $m \ll n^2$:

Theorem 1. *Let G be a connected, undirected graph with m edges. The number of triangles in G can be computed in $O(m^{\frac{2\omega}{\omega+1}}) \leq o(m^{1.408})$ time.*

Proof. First, we count the number of triangles in G whose three vertices all have degree at least t in G, where t is a threshold parameter that will be set later. To do this, we take the subgraph of G induced by all vertices of degree $\geq t$, and apply the triangle counting method from [1] which runs in $O(|V|^{\omega})$ time for any graph with $|V|$ vertices. Since the number of vertices with degree $\geq t$ is $O(\frac{m}{t})$, the aforementioned method takes $O(\frac{m^{\omega}}{t^{\omega}})$ time. Let N_{Δ} be the computed number of triangles in the subgraph.

Secondly, we count the number of triangles with at least one vertex of degree strictly less than t. For this purpose, we enumerate the set E_t of edges in G with at least one endpoint of degree $< t$, and for $i = 1, \ldots, |E_t|$, iterate the following:

- Pick an endpoint v of the i-th edge e_i in E_t of degree less than t; for each edge e incident to e_i at v, check if e_i and e induce a triangle in G which does not include any edge $e_j \in E_t$ where $j < i$; if yes then increase N_{Δ} by one.

The above steps can be implemented in $O(t)$ time, so counting the remaining triangles takes $O(mt)$ time. By solving the equation $\frac{m^{\omega}}{t^{\omega}} = mt$, we obtain and set $t = m^{\frac{\omega-1}{\omega+1}}$. □

Next, we similarly refine the part of Theorem 1.8 in [19] which states that a monochromatic triangle in a connected, undirected, edge-colored graph with n vertices can be found (if one exists) in $O(n^{(3+\omega)/2}) \leq o(n^{2.688})$ time. We obtain:

Theorem 2. *Let G be a connected, undirected, edge-colored graph with n vertices and let C be a subset of the set of edge colors. The number of monochromatic triangles of G with colors in C can be computed in $O(n^{(3+\omega)/2}) \leq o(n^{2.688})$ time.*

Proof. For each color $i \in C$, let E_i be the set of edges in G colored by i. Following [19], we say that i is *heavily used* if $|E_i| \geq n^{(\omega+1)/2}$. For each heavily used color, we count the number of monochromatic triangles by directly applying the triangle counting method from [1] to the subgraph induced by edges colored with i in $O(n^{\omega})$ time. This takes $O(n^2/n^{(\omega+1)/2}) \cdot O(n^{\omega}) = O(n^{2-(\omega+1)/2+\omega}) = O(n^{(3+\omega)/2})$ time in total.

To count the remaining monochromatic triangles, for each non-heavily used color $i \in C$, we apply the method of Theorem 1 above to the subgraph induced by the edges in E_i. This takes $O(|E_i|^{2\omega/(\omega+1)})$ time. As in the proof of Theorem 1.8 in [19], we observe that the total time taken by the non-heavily used colors $i \in C$ is maximized if $|E_i| = \Theta(n^{(\omega+1)/2})$ holds for each of them, and thus there are $\Theta(n^{2-(\omega+1)/2})$ of them. Since $O(n^{2-(\omega+1)/2}) \cdot O((n^{(\omega+1)/2})^{\frac{2\omega}{\omega+1}}) = O(n^{(3-\omega)/2}) \cdot O(n^{\omega}) = O(n^{(3+\omega)/2})$, this shows that the total time taken by counting the remaining monochromatic triangles is $O(n^{(3+\omega)/2})$, too. □

Finally, we consider a kind of relaxation of the concept of a monochromatic triangle to an "almost-monochromatic triangle" in an undirected, edge-colored graph G. Let R be a binary relation on the edge colors. A triangle in G with two edges of the same color i and the third one of color k such that iRk holds is called an *R-chromatic triangle* (e.g., if R stands for $<$ then k is simply required to be larger than i.). We need to extend Theorem 2 to count R-chromatic triangles. We begin with the following technical generalization of Theorem 1:

Lemma 4. *Suppose that an undirected graph G with colored edges is preprocessed so that for any color edge i, the subgraph induced by the edges of color i can be extracted in $O(m_i)$ time, where m_i is the number of edges with color i in G. Let R be a binary relation on the colors of G computable in constant time. The number of R-chromatic triangles with at least two edges of color i in G can be computed in $O(m_i^{\frac{2\omega}{\omega+1}})$ time.*

Proof. First, extract the subgraph G_i induced by the edges of G with color i in $O(m_i)$ time. Then, run the method of Theorem 1 on G_i with the following modifications which do not affect the asymptotic time complexity:

1. Once the square C_i of the adjacency matrix of the subgraph of G_i consisting of all vertices of degree at least t is computed then for each entry $C_i[k,l]$ we check if (k,l) is an edge of G whose color j is in the relation R with the color i, i.e., $R(i,j)$ holds. Only in this case we increase the count of triangles by the arithmetic value of $C[k,l]$ (in case (k,l) is an edge whose color is also i and $R(i,i)$ holds, we increase the count of triangles by $C[k,l]/3$ only).
2. When we scan the edges e of G_i with at least one vertex v of degree smaller than t, then for each edge e' of G incident to e at v, we check if these two edges induce an R-chromatic triangle that was not counted before. If so, we increase the count by one. □

We now generalize Theorem 2 to R-chromatic triangles by applying Lemma 4:

Theorem 3. *Let G be a connected, undirected graph with n vertices and colored edges, and let R be a binary relation on the colors of G computable in constant time. The number of R-chromatic triangles in G can be computed in $O(n^{(3+\omega)/2}) \leq o(n^{2.688})$ time.*

Proof. First construct the graphs G_i induced by the sets E_i of edges with color i. This takes $O(n^2)$ time in total. Next, proceed as in the proof of Theorem 2. For each heavily used color i, i.e., satisfying $|E_i| \geq n^{(\omega+1)/2}$, count the number of R-chromatic triangles with at least two edges with color i by squaring the adjacency matrix of G_i and testing, for each entry $C_i[k,l]$ of the resulting matrix, if (k,l) is an edge whose color is in the relation R with i (analogously as in (1) in the proof of Lemma 4). This takes $O(n^{2-(\omega+1)/2+\omega}) = O(n^{(3+\omega)/2})$ time in total. To count the remaining R-chromatic triangles, each with at least two edges colored with a non-heavily used color i, use Lemma 4, which takes time $O(|E_i|^{2\omega/(\omega+1)})$ for any given color i. By an argument analogous to one in the proof of Theorem 2, the total time to count the remaining monochromatic triangles is $O(n^{(3+\omega)/2})$. □

4 Computing the Rooted Triplet Distance between Galled Trees

In this section, we apply the triangle counting techniques from Section 3 to obtain a subcubic-time algorithm for computing the rooted triplet distance between two galled trees. We first explain how to compute the number of rooted fan triplets consistent with both networks in Section 4.1 and then the number of rooted proper triplets consistent with both networks in Section 4.2. Combining these two results gives us our main result (Theorem 4) in Section 4.3.

4.1 Counting the Number of Shared Rooted Fan Triplets

To count the number of rooted fan triplets consistent with two given galled trees, we use Theorems 2 and 3 as detailed below. As a warm-up, we first present a simple reduction from the problem of counting rooted fan triplets shared by two *trees* to the problem of counting monochromatic triangles in a graph.

Lemma 5. *Let U_1, U_2 be two trees on the same set L of n leaves. The number of rooted fan triplets consistent with both U_1 and U_2 can be computed in $O(n^{(3+\omega)/2}) \leq o(n^{2.688})$ time.*

Proof. Form an auxiliary undirected complete graph $G = (L, E)$ in which every edge is assigned a color of the form (v_1, v_2), where v_1 is a vertex of U_1 and v_2 is a vertex of U_2, as follows: For each edge $\{u, v\} \in E$, let j_i for $i = 1, 2$ be the unique junction common ancestor of u and v in U_i, and color the edge $\{u, v\}$ in G with the color (j_1, j_2). By Lemma 1, G can be constructed in $O(n^2)$ time.

For any $\{a, b, c\} \subseteq L$, the rooted fan triplet $a|b|c$ is consistent with U_1 if and only if the junction common ancestors in U_1 of a and b, of a and c, and of b and c are identical. The same holds for U_2. Therefore, $a|b|c$ is consistent with both U_1 and U_2 if and only if all three edges $\{a, b\}$, $\{a, c\}$, $\{b, c\}$ have the same color in G. It follows that the number of rooted fan triplets which are common to both trees equals the number of monochromatic triangles in G. By Theorem 2, we can compute the number of rooted fan triplets that are consistent with both U_1 and U_2 in $O(n^{(3+\omega)/2})$ time. □

Next, we adapt the reduction in the proof of Lemma 5 to the more complicated *galled tree* case:

Lemma 6. *Let U_1, U_2 be two galled trees on the same set L of n leaves. The number of rooted fan triplets consistent with both U_1 and U_2 can be computed in $O(n^{(3+\omega)/2}) \leq o(n^{2.688})$ time.*

Proof. By Lemma 3, we can distinguish two classes of rooted fan triplets in a galled tree U: those where for each of its three pairs of leaves, the lca is equal to the shared junction common ancestor as in the example in Fig. 2 (ii) (henceforth referred to as "class 1"), and those where the equality holds for two pairs only, as in Fig. 2 (iii) (henceforth referred to as "class 2"). For the sake of the proof, we need to consider a slightly different two-partition of rooted fan triplets in U. We shall say that a rooted fan triplet in U is of *type 1* iff it belongs to the class 1 and the unique lca of each pair of leaves in the triplet is also their lca in each of the trees U_L, U_R. All remaining rooted fan triplets in U are said to be of *type 2.*

For $i = 1, 2$, consider the trees $(U_i)_L$, $(U_i)_R$ defined as in the proof of Lemma 1. By Lemma 5, we can compute the number of shared rooted fan triplets between $(U_1)_A$ and $(U_2)_B$ for any $A, B \in \{L, R\}$ in $O(n^{(3+\omega)/2})$ time. Note that each rooted fan triplet of type 1 in U_1 occurs in both $(U_1)_L$ and $(U_1)_R$, while each rooted fan triplet of type 2 in U_1 occurs in only one of the trees. The reason for the distinction is that a rooted fan triplet of type 2 contains exactly one pair of leaves whose lca in U_1 relies on one of the two edges directed to a reticulation vertex of a cycle. Hence, the lca of the pair occurs in exactly one of the trees $(U_1)_L$ and $(U_1)_R$, and consequently the rooted fan triplet also occurs in exactly one of (not necessarily the same as above) $(U_1)_L$ and $(U_1)_R$. Analogous observations hold for U_2. Hence, if we sum the number of shared rooted fan triplets between $(U_1)_A$ and $(U_2)_B$ over all $A, B \in \{L, R\}$, then each rooted fan triplet that is of type 1 both in U_1 and U_2 is counted four times, while those that are of different types in U_1 and U_2 are counted twice, and finally those of type 2 both in U_1 and U_2 are counted only once. Hence, if for $p,\ q \in \{1, 2\}$, $T_{p,q}$ denotes the number of rooted shared fan triplets that are of type p in U_1 and of type q in U_2 then the computed sum equals $4T_{1,1} + 2T_{1,2} + 2T_{2,1} + T_{2,2}$.

In fact, we can also determine $T_{1,1}$ in $O(n^{(3+\omega)/2})$ time in the same way as we have done for a pair of trees in Lemma 5. While constructing the auxiliary complete graph, we require j_i to be both the lca of u and v in $(U_i)_L$ and $(U_i)_R$), as well as a junction common ancestor of u and v in U. Then, we use Theorem 2 to determine the number of monochromatic triangles analogously.

It remains to determine the number of shared rooted fan triplets for U_1 and U_2 that are of different types in U_1 and U_2 in order to cancel repetitions in the aforementioned sum, i.e., $T_{1,2} + T_{2,1}$. To compute, say $T_{2,1}$, we again form the auxiliary complete graph on the set L of leaves and color each edge $\{u, v\}$ as described next. Recall that lca's are unique in a galled tree. For $i = 1, 2$, let j_i be the unique lca of of u and v in U_i. If, for $i = 1, 2$, j_i is also a junction common ancestor of u and v in U_i and it is the lca of u and v in both trees $(U_i)_L$ and $(U_i)_R$, then $\{u, v\}$ is colored with (j_1, j_2) as before. Next, if for $i = 1, 2$, j_i is also a junction common ancestor of u and v in U_i, and j_1 is the lca of u and v in exactly one of the trees $(U_1)_L$ and $(U_1)_R$ while j_2 is the lca of u and v in both trees $(U_2)_L$ and $(U_2)_R$ then $\{u, v\}$ is colored with $((j_1)^*, j_2)$. Otherwise, $\{u, v\}$ is colored with the null color. To use Theorem 3, we define the relation R by:

- $(j_1, j_2)R(l_1, l_2)$ holds iff $l_1 = (k)^*$, where k is a proper descendant of j_1 or $j_1 = k$, and $j_2 = l_2$.

The trees $(U_i)_A$ for $i = 1, 2,\ A \in \{L, R\}$, can be preprocessed to support $O(1)$-time lca queries in $O(n)$ time [9, 17]. By using $(U_1)_L, (U_1)_R$, we can spend $O(n^2)$ time to build a data structure supporting $O(1)$-time proper descendant queries.

Now, we apply Theorem 3 to the auxiliary graph to obtain the number $T_{2,1}$ of rooted shared triplets of type 2 in U_1 and type 1 in U_2 in $O(n^{(3+\omega)/2})$ time.

The number $T_{1,2}$ of rooted shared triplets of type 1 in U_1 and type 2 in U_2 is obtained in $O(n^{(3+\omega)/2})$ time in the same way. $\square$

4.2 Counting the Number of Shared Rooted Proper Triplets

Lemma 7. *Let U_1, U_2 be two galled trees on the same set L of n leaves. The number of rooted proper triplets consistent with both U_1 and U_2 can be computed in $O(n^\omega) \leq o(n^{2.376})$ time.*

Proof. First, for $i = 1, 2$, for each pair of leaves in U_i, compute their junction common ancestors (if they exist) along with the information if the respective junction common ancestor is located on a cycle of U_i, if it is the root of the cycle, and if it uses the reticulation vertex of the cycle. By a straightforward modification of the proof of Lemma 1, this takes $O(n^2)$ time.

For each vertex v_i of U_i form two copies v_i^0, v_i^1. Next, for the set of pairs of distinct leaves in L, form the classes $C_{v_1^{b_1}, v_2^{b_2}}$, where $\{b_1, b_2\} \subset \{0, 1\}$ and v_i is a vertex of U_i for $i = 1, 2$, such that $(a, b) \in C_{v_1^{b_1}, v_2^{b_2}}$ if and only if the following three conditions hold for $i = 1, 2$: (1) v_i is a junction common ancestor of a and b in U_i; (2) v_i is located on a cycle of U_i and uses the reticulation vertex of the cycle iff $b_i = 1$; and (3) if v_i is the root of a cycle in U_i then there is no other junction common ancestor of a and b. By Lemma 1, any pair of leaves a, b in a galled tree can have at most two junction common ancestors. Moreover, if there are two then they are located on the same cycle and one of them will be the root of the cycle. Hence, from the point of view of a rooted triplet $ab|c$, it is sufficient to consider the junction common ancestor of a, b that is a descendant of the root vertex of the cycle in this case, since any path from an ancestor of the root of the cycle can be extended to reach the descendant junction common ancestor. This explains the third condition, which implies that the classes $C_{v_1^{b_1}, v_2^{b_2}}$ are pairwise disjoint. These classes can be formed in $O(n^2)$ time by integer sorting.

Furthermore, for $i = 1, 2$, form matrices M_i such that the rows of M_i correspond to the copies of vertices in U_i, the columns of M_i correspond to the leaves in L, and $M_i[v_i^{b_i}, c] = 1$ if and only if there is a junction common ancestor of v_i and c in U_i which in case $b_i = 1$ does not use the reticulation vertex of the cycle on which v_i lies in U_i. Importantly, if $M_i[v_i^{b_i}, c] = 1$ then c cannot occur in any pair in a class of the form $C_{v_1^{b_1}, v_2^{b_2}}$. Simply, in this case, an ancestor of c or c itself would be a reticulation vertex used by both v_i and any junction common ancestor of v_i and c. This in particular would imply $b_i = 1$. Hence, $M_i[v_i^{b_i}, c]$ would be set to 0, and we obtain a contradiction.

Next, compute $Q = M_1 \times M_2^t$ in $O(n^\omega)$ time. By the definitions of M_1 and M_2, the value of $Q[v_1^{b_1}, v_2^{b_2}]$ is exactly the number of leaves c in L that have a junction common ancestor with v_i in U_i not using the reticulation vertex of the cycle on which v_i lies if $b_i = 1$, for $i = 1, 2$. Note that for $\{b_1, b_2\} \neq \{b_1', b_2'\}$, $C_{v_1^{b_1}, v_2^{b_2}} \cap C_{v_1^{b_1'}, v_2^{b_2'}} = \emptyset$ and the aforementioned leaves c cannot occur in any pair in $C_{v_1^{b_1}, v_2^{b_2}}$. By Lemma 2, the sum $\sum_{\{b_1, b_2\} \subset \{0,1\}} |C_{v_1^{b_1}, v_2^{b_2}}| Q[v_1^{b_1}, v_2^{b_2}]$

equals the number of proper rooted triplets $ab|c$ consistent with both U_1 and U_2 that use v_i as a junction common ancestor of a and b in U_i, for $i = 1, 2$, with the exception of the case when v_1 or v_2 is the root vertex of a cycle in its galled tree and there is another junction common ancestor of a and b which is a descendant of the root vertex in the galled tree. Due to the latter, for different pairs of v_1, v_2, the sum counts different sets of the proper rooted triplets $ab|c$ consistent with both U_1 and U_2. Thus, it is sufficient to compute the sum $\sum_{v_1 \in U_1} \sum_{v_2 \in U_2} \sum_{\{b_1,b_2\} \subset \{0,1\}} |C_{v_1^{b_1}, v_2^{b_2}}| Q[v_1^{b_1}, v_2^{b_2}]$ to obtain the total number of rooted triplets consistent with both U_1 and U_2. This takes $O(n^2)$ time. □

4.3 Computing the Rooted Triplet Distance

By combining the results established in the previous two subsections, we obtain:

Theorem 4. *Let U_1, U_2 be two galled trees on the same set L of n leaves. The rooted triplet distance $d_{rt}(U_1, U_2)$ can be computed in $O(n^{(3+\omega)/2}) \leq o(n^{2.688})$ time.*

Proof. For $i = 1, 2$, let F_i denote the set of rooted fan triplets consistent with U_i, and let P_i denote the set of rooted proper triplets consistent with U_i. We have $d_{rt}(U_1, U_2) = \sum_{i=1}^{2}(|F_i| + |P_i|) - 2|F_1 \cap F_2| - 2|P_1 \cap P_2|$. Compute $|F_i \cap F_i| = |F_i|$ and $|F_1 \cap F_2|$ in $O(n^{(3+\omega)/2}) \leq o(n^{2.688})$ time by Lemma 6, and compute $|P_i \cap P_i| = |P_i|$ and $|P_1 \cap P_2|$ in $O(n^{\omega}) \leq o(n^{2.376})$ time by Lemma 7. □

5 Concluding Remarks

We have demonstrated that the rooted triplet distance can be computed in subcubic time for a well-known class of phylogenetic networks called galled trees [8, 10]. More precisely, we have presented a new $o(n^{2.688})$-time algorithm for computing the rooted triplet distance between two input galled trees with n leaves each [Theorem 4]. We have also derived three results on counting triangles in a graph [Theorems 1–3] that may have other applications. The first two triangle counting results are generalizations of their known (weaker) detection counterparts from [1] and [19], respectively.

Recently, Nielsen *et al.* [15] showed how to compute the *unrooted quartet distance* between two *unrooted* phylogenetic trees with n leaves in $o(n^{2.688})$ time. Interestingly, they also rely on matrix multiplication. Their method does not count triangles in an auxiliary graph as we have done here, but uses matrix multiplication to count so-called *shared* and *different butterflies* between the two input trees directly. In some sense, their problem seems inherently "easier" than ours as it does not involve cycles. A lot of the conceptual complexity in our paper stems from the non-uniqueness of junction common ancestors in galled trees; compare the proofs of Lemmas 5 and 6, for example.

It is an open question whether the problem of computing the rooted triplet distance $d_{rt}(U_1, U_2)$ between two galled trees U_1, U_2 admits a quadratic-time algorithm or not. Another important question is if our method can be enhanced to include even larger classes of phylogenetic networks than galled trees.

References

1. Alon, N., Yuster, R., Zwick, U.: Finding and counting given length cycles. Algorithmica 17(3), 209–223 (1997)
2. Bansal, M.S., Dong, J., Fernández-Baca, D.: Comparing and aggregating partially resolved trees. Theoretical Computer Science 412(48), 6634–6652 (2011)
3. Chan, H.-L., Jansson, J., Lam, T.-W., Yiu, S.-M.: Reconstructing an ultrametric galled phylogenetic network from a distance matrix. Journal of Bioinformatics and Computational Biology 4(4), 807–832 (2006)
4. Choy, C., Jansson, J., Sadakane, K., Sung, W.-K.: Computing the maximum agreement of phylogenetic networks. Theoretical Computer Science 335(1), 93–107 (2005)
5. Coppersmith, D., Winograd, S.: Matrix Multiplication via Arithmetic Progressions. Journal of Symbolic Computation 9, 251–280 (1990)
6. Critchlow, D.E., Pearl, D.K., Qian, C.: The triples distance for rooted bifurcating phylogenetic trees. Systematic Biology 45(3), 323–334 (1996)
7. Felsenstein, J.: Inferring Phylogenies. Sinauer Associates, Inc., Sunderland (2004)
8. Gusfield, D., Eddhu, S., Langley, C.: Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. Journal of Bioinformatics and Computational Biology 2(1), 173–213 (2004)
9. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13(2), 338–355 (1984)
10. Huson, D.H., Rupp, R., Scornavacca, C.: Phylogenetic Networks: Concepts, Algorithms and Applications. Cambridge University Press (2010)
11. van Iersel, L., Kelk, S.: Constructing the Simplest Possible Phylogenetic Network from Triplets. Algorithmica 60(2), 207–235 (2011)
12. Jansson, J., Nguyen, N.B., Sung, W.-K.: Algorithms for Combining Rooted Triplets into a Galled Phylogenetic Network. SIAM Journal on Computing 35(5), 1098–1121 (2006)
13. Morrison, D.: Introduction to Phylogenetic Networks. RJR Productions (2011)
14. Nakhleh, L., Warnow, T., Ringe, D., Evans, S.N.: A comparison of phylogenetic reconstruction methods on an Indo-European dataset. Transactions of the Philological Society 103(2), 171–192 (2005)
15. Nielsen, J., Kristensen, A.K., Mailund, T., Pedersen, C.N.S.: A sub-cubic time algorithm for computing the quartet distance between two general trees. Algorithms for Molecular Biology 6, Article 15 (2011)
16. Stothers, A.J.: On the Complexity of Matrix Multiplication. PhD thesis, University of Edinburgh (2010)
17. Tarjan, R.E.: Applications of path compression on balanced trees. Journal of the ACM 26(4), 690–715 (1979)
18. Wang, L., Ma, B., Li, M.: Fixed topology alignment with recombination. Discrete Applied Mathematics 104(1-3), 281–300 (2000)
19. Vassilevska, V., Williams, R., Yuster, R.: Finding Heaviest H-Subgraphs in Real Weighted Graphs, with Applications. ACM Transactions on Algorithms 6(3), Article 44 (2010)
20. Vassilevska Williams, V.: Breaking the Coppersmith-Winograd barrier. UC Berkely and Stanford University (2011) (manuscript)

Minimum Leaf Removal for Reconciliation: Complexity and Algorithms

Riccardo Dondi[1] and Nadia El-Mabrouk[2]

[1] Dipartimento di Scienze dei Linguaggi, della Comunicazione e degli Studi Culturali, Università degli Studi di Bergamo, Bergamo, Italy
[2] Départment d'Informatique et Recherche Opérationnelle, Université de Montréal, Montréal, Canada
riccardo.dondi@unibg.it, mabrouk@iro.umontreal.ca

Abstract. Reconciliation is a well-known method for studying the evolution of a gene family through speciation, duplication, and loss. Unfortunately, the inferred history strongly depends on the considered gene tree for the gene family, as a few misplaced leaves can lead to a completely different history, possibly with significantly more duplications and losses. It is therefore essential to develop methods that are able to preprocess and correct gene trees prior to reconciliation. In this paper, we consider a combinatorial problem, known as the Minimum Leaf Removal problem, that has been proposed to remove errors from a gene tree by deleting some of its leaves. We prove that the problem is APX-hard, even in the restricted case of a gene family with at most two copies per genome. On the positive side, we present fixed-parameter algorithms where the parameters are the size of the solution (minimum number of leaf removals) and the number of genomes containing multiple gene copies.

1 Introduction

The evolution of genomes is determined by a combination of micro-evolutionary events affecting their sequences, and macro-evolutionary events, involving rearrangement and content-modifying operations, affecting their overall gene content and organization. Among content-modifying operations, duplication is a fundamental process in the evolution of species, and a major source of gene innovation [24,14]. The consequence of duplications is that genes are not present in one, but in many copies, in the genome. In parallel to duplications, gene losses appear generally to maintain a minimum number of functional gene copies [5,10,11,20]. Using a local similarity search tool such as BLAST [2], genes can be clustered by sequence homology into *gene families*. From a conceptual evolutionary point of view, homologous gene copies originate from the same ancestral gene.

Understanding the evolution of gene families through duplication and loss is fundamental for many reasons. In particular, it allows distinguishing between two classes of gene homologs [21]: *orthologs* which are copies in different species that arose by speciation at their most recent point of origin, and *paralogs* which are gene copies in the same genome or in two different genomes that arose from a duplication at their most recent point of origin. While orthologs are, in essence, instances of the 'same gene' in different species, paralogs represent different copies of the ancestor that are likely to have independently evolved and diverged in their function. Consequently, identifying the "true" orthology relationship between genes is fundamental for functional annotation of genes, as well as phylogenetic inference and comparative genomics purposes.

Based on a micro-evolutionary model for sequences, a gene tree T that best explains the data can be constructed for a given gene family, by using a classical phylogenetic method. When a species tree S reflecting the speciation history of the genomes is known, then the macro-evolutionary events that gave rise to the data can be inferred by using a method known as *Reconciliation*. It consists in "embedding" T into S, and interpreting the disagreement between the two trees as a footprint of the evolution of the gene family through duplication and loss. This concept was pioneered by Goodman [15] and then widely accepted, utilized, and improved [3,6,7,8,10,13,26,28,29,30]. When no preliminary knowledge on the species tree is given, a natural problem, known as the *species tree inference problem*, is to infer, from a set of gene trees, a species tree leading to a parsimonious evolution scenario [4,8,22].

A major problem in the application of gene tree reconciliation is its high sensitivity to error-prone gene trees. Indeed, a few misplaced leaves can lead to a completely different history, possibly with significantly more duplications and losses [19,29]. Typically bootstrapping values are used as a measure of confidence in each edge of a phylogeny. How should the weak edges of a gene tree be handled? This problem has been addressed in [9,13,16] by exploring the space of gene trees obtained from the original one by performing rearrangements (such as NNIs) around weakly-supported edges and select the tree giving rise to the minimum duplications and losses. A different strategy that has been recently adopted for preprocessing a gene tree T prior to reconciliation or species tree inference, is to "remove" misplaced leaves (gene copies). Criteria for identifying such leaves were given in [8]. The duplication nodes of T with respect to a species tree S can be subdivided into apparent and non-apparent duplication (NAD) nodes, where the latter class has been flagged as potentially resulting from the misplacement of leaves in the gene tree. The reason is that each one of the NAD nodes reflects a phylogenetic contradiction with the species tree that is not due to the presence of duplicated gene copies. In [12], algorithmic results have been presented for the problem of removing, from a given gene tree, the minimum number of leaves leading to a tree without any NAD node (the Minimum Leaf Removal Problem). An exact polynomial-time algorithm has been described for two special classes of gene trees, and a polynomial-time heuristic with no guarantee of optimality, has been presented for the general case.

In this paper, we study the theoretical complexity of the Minimum Leaf Removal Problem. More precisely, we show in Section 3 that the problem is APX-hard, by reduction from the Minimum Vertex Cover problem on Cubic graph [1]. We then turn our attention in Section 4 to finding tractable versions of the problem under some biological meaningful parameterizations. The goal is to identify parameters that are small in practice, and to constraint the exponential explosion only to these parameters. We identify two fixed-parameter tractable versions of the problem and present exact polynomial-time algorithms constrained by: (1) the size of the solution (minimum number of leaf removal) and (2) the number of genomes containing multiple gene copies (paralogs). We begin in the next section by introducing the concepts and notations used in the rest of the paper. Due to space limitations some of the proofs are omitted.

2 Preliminary Definitions

2.1 Trees

Let $\Gamma = \{1, 2, \cdots, \gamma\}$ be a set of integers representing γ different species (genomes). We consider two kinds of rooted binary trees leaf-labelled by the elements of Γ: a *species tree* S is a tree where each element of Γ labels at most one leaf, while a *gene tree* T is a tree where each element of Γ may label more than one leaf (Figure 1 (a) and (b)). A gene tree represents a gene family, where each leaf labelled x represents a gene copy located on genome x.

Given a tree U, we denote by $L(U)$ the set of its leaves and by $V(U)$ the set of its nodes. Given an internal node x of U, we denote by x_l and x_r respectively, the left and right child of x, by $U(x)$ the subtree of U rooted at x, and by $\Gamma(U(x))$ the set of leaf-labels of $U(x)$. If there is no ambiguity on the tree being considered, we denote $\mathcal{C}(x) = \Gamma(U(x))$; $\mathcal{C}(x)$ is called the *cluster* of x. An *ancestor* of a node x of U is any node on the path from the root of U to x.

Given a tree U, a *leaf removal* consists in removing a given leaf l of U, and suppressing the resulting degree two node (that is the parent of l). If a tree U' is obtained from a tree U through a sequence of leaf removals, then U' is *included* in U. On the other hand a *subtree insertion* in U consists in creating a new node x on a branch (a, b) (joining node a to node b, b being the child of a), making b the left child of x, setting the parent of x to a, and grafting the subtree being inserted as the second child of x (create an edge from x to the root of the subtree). An *extension* of U is a tree obtained from U through a sequence of subtree insertions.

2.2 Reconciliation

Usually, the gene tree T obtained for a given gene family is different from the species tree S. Roughly speaking, a *reconciliation* between T and S is an extension $R(T, S)$ of T that is "consistent" with S, i.e. reflects the same phylogeny. A rigorous definition can be found in [8,12]. A history of duplications and losses can immediately be inferred from such a reconciliation. Different algorithms have been developed for recovering a reconciliation minimizing a duplication and/or loss cost [6,13,17,18,22,25,27,8], most of them based on a method called *LCA mapping*.

The LCA mapping between a gene tree T and a species tree S, denoted by $\text{lca}_{T,S}$, maps every node x of T to the Lowest Common Ancestor (LCA) of $\mathcal{C}(x)$

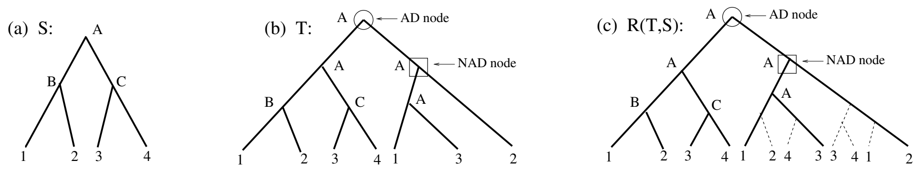


Fig. 1. (a) A species tree S for $\Gamma = \{1, 2, 3, 4\}$. The three internal nodes of S are named A, B and C; (b) A gene tree T. A leaf label g indicates a gene copy in genome g. Internal nodes are labelled according to the LCA mapping between T and S. Flagged nodes are duplication nodes of T with respect to S; (c) A reconciliation $R(T, S)$ of T and S. Dotted lines represent subtree insertions. This reconciliation reflects a history of the gene family with two gene duplications preceding the first speciation event, and 4 losses.

in S. Formally, $\text{lca}_{T,S}(x) = y$, where y is the node of S that has the minimum cluster such that $\mathcal{C}(x) \subseteq \mathcal{C}(y)$. A *duplication* occurs in a node x of T (or x is a duplication), if x and at least one of its children are mapped by $\text{lca}_{T,S}$ in the same node y of the species tree S. If x is not a duplication node, then x is a *speciation* (Figure 1).

2.3 Duplication Nodes and MD-trees

The notations of this section are those used in [8,12]. Let x be a node of a gene tree T verifying $\mathcal{C}(x_l) \cap \mathcal{C}(x_r) \neq \emptyset$. Then, for any species tree S, x is guaranteed to be a duplication node. Such a node x is called an *Apparent Duplication node* (*AD node* for short). Given a species tree S, a duplication node x which is not an AD node is called a *Non-Apparent Duplication node* (*NAD node* for short). A gene tree T is *MD-consistent* (MD holds for "Minimum Duplication") with a species tree S if and only if each node of T is either a speciation or an AD node.

As explained in [12], NAD nodes point to disagreement between a gene tree T and a species tree S that are not due to the presence of repeated leaf labels, i.e. duplicated gene copies (see Figure 1.(b)). It has therefore been suggested, and supported by simulations in [8], that NAD nodes may point at gene copies that are erroneously placed in the gene tree. It has to be noticed that a misplaced gene in a gene tree T does not necessarily lead to a NAD node. In other words, NAD nodes can only point to a subset of misplaced leaves. However, in the context of reconciliation, the damage caused by a misplaced leaf leading to a NAD node is to significantly increase the real duplication and/or loss cost of the tree. Following these observations, the Minimum Leaf Removal Problem, given bellow, has been considered in [12] for error-correction in gene trees.

Problem 1. ***Minimum Leaf Removal Problem****[MinLeafRem]*
Input: *A gene tree T and a species tree S, both leaf-labelled by Γ.*
Output: *A tree T^* MD-consistent with S such that T^* is obtained from T by a minimum number of leaf removals.*

3 Hardness of Minimum Leaf Removal

In this section we consider the computational (and approximation) complexity of the MinLeafRem problem. We show that MinLeafRem is APX-hard, even in the restricted case that each label is associated with at most two leaves of T. We denote this restriction of the problem by MinLeafRem(2).

We prove that MinLeafRem(2) is APX-hard, by giving an L-reduction from the MINIMUM VERTEX COVER PROBLEM on Cubic graphs (MVCC is known to be APX-hard [1]).

Problem 2. ***Minimum Vertex Cover Problem on Cubic graphs****[MVCC]*
Input*: A cubic graph $G = (V, E)$ where $V = \{v_1, \dots, v_n\}$ is the set of vertices and E the set of edges of G (in a cubic graph, each vertex has degree 3) .*
Output*: A minimum cardinality set $V' \subseteq V$, such that for each edge $e_{i,j} = \{v_i, v_j\} \in E$, at least one of v_i, v_j belongs to V'.*

Let $G = (V, E)$ be an instance of MVCC. We define an instance of MinLeafRem associated with G, consisting of a gene tree T and a species tree S, both leaf-labelled by Γ, defined as follows, where $t = 4|V| + |E| + 1$:

$$\Gamma = \{v_{i,l} : v_i \in V, 1 \leq l \leq 4\} \cup \{v_i^j : v_i \in V, \{v_i, v_j\} \in E\} \cup \{e_{i,j} : \{v_i, v_j\} \in E\} \cup \{z_i : 1 \leq i \leq t\} \cup \{\alpha\}.$$

We denote $Z = \{z_i : 1 \leq i \leq t\}$. Let U be a tree, which is either the gene tree T, the species tree S, or a tree included in T with a leaf labelled by α. We define the *spine* of U as the path from the root of U to the unique leaf of U labelled by α.

Next, we define an ordering on the edges E of G. Consider the edges $\{v_i, v_j\}$, with $i < j$, and $\{v_h, v_k\}$, with $h < k$, then $\{v_i, v_j\} < \{v_h, v_k\}$, iff $i \leq h$, and $j < k$ if $i = h$. Denote with $\{v_p, v_q\}$ the last edge in such ordering of E.

The gene tree T is defined as in Fig. 2. It contains the following kinds of subtrees: (1) a subtree T_{v_i}, for each vertex $v_i \in V$; (2) a subtree $T_{e_{ij}}$ and a leaf $e_{i,j}$, for each edge $e_{i,j} = \{v_i, v_j\} \in E$; (3) a tree T_Z, which is a caterpillar tree of size t with leaves uniquely leaf-labelled by the set Z. Notice that the order in which the subtrees $T_{e_{ij}}$ and the leaf $e_{i,j}$ appear in T, depends on the order of the corresponding edges of E.

The species tree S is defined in Fig. 3. It contains the three following kinds of subtrees : (1) a subtree S_{v_i}, for each vertex $v_i \in V$; (2) a single leaf labelled by $e_{i,j}$, for each edge $e_{i,j} = \{v_i, v_j\} \in E$; (3) a tree S_Z, which is a caterpillar tree of size t uniquely leaf-labelled by the set Z.

It is easy to see that S is a species tree uniquely leaf-labelled by Γ, and that T is a gene tree where each label in Γ is associated with at most two leaves of T. The following properties of T are directly deduced from the construction of T.

Remark 1. *The root of T_Z and all its ancestors are mapped (by the LCA mapping) to the root r of S. Consequently, all T_Z ancestors are duplication nodes.*

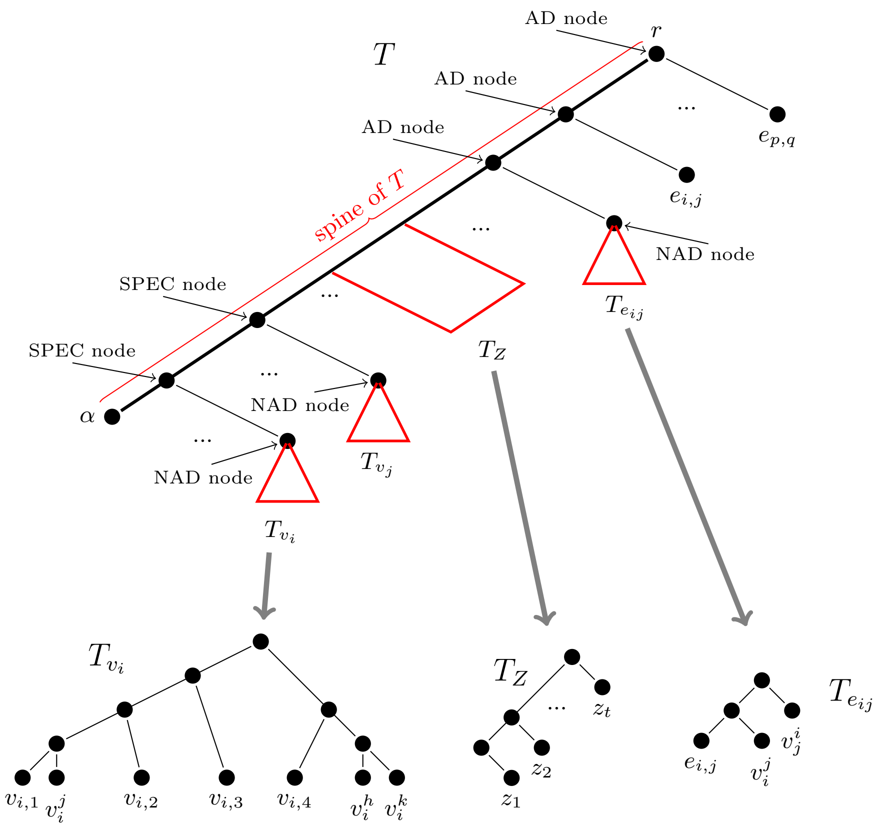


Fig. 2. The gene tree T, and the subtrees T_{v_i}, T_Z and $T_{e_{ij}}$ of T. Notice that $i < j$, hence T_{v_j} is closer to the root than T_{v_i}. Notice that a SPEC node is a speciation node.

Moreover, we deduce from the non-empty intersection of the left and right leaf sets that all these nodes are AD nodes.

Remark 2. *For each $e_{i,j} \in E$, the root of $T_{e_{i,j}}$ is a NAD node. Indeed, it is mapped to the same node of S than its left child, and it does not contain any duplicated leaf-label.*

Moreover, as each subtree T_{v_i} contains NAD nodes, any solution of MinLeafRem over instance (T, S) is obtained by removing appropriate leaves from each T_{v_i}. The following results give more details on the required removals.

Remark 3. *Let v_i be a vertex of G. Then: (1) the subtree of T_{v_i} obtained by removing the leaves with labels v_i^j, v_i^h, v_i^k is MD-consistent with S_{v_i}; (2) the subtree of T_{v_i} obtained by removing the leaves with labels $v_{i,1}$, $v_{i,2}$, $v_{i,3}$, $v_{i,4}$ is MD-consistent with S_{v_i}.*

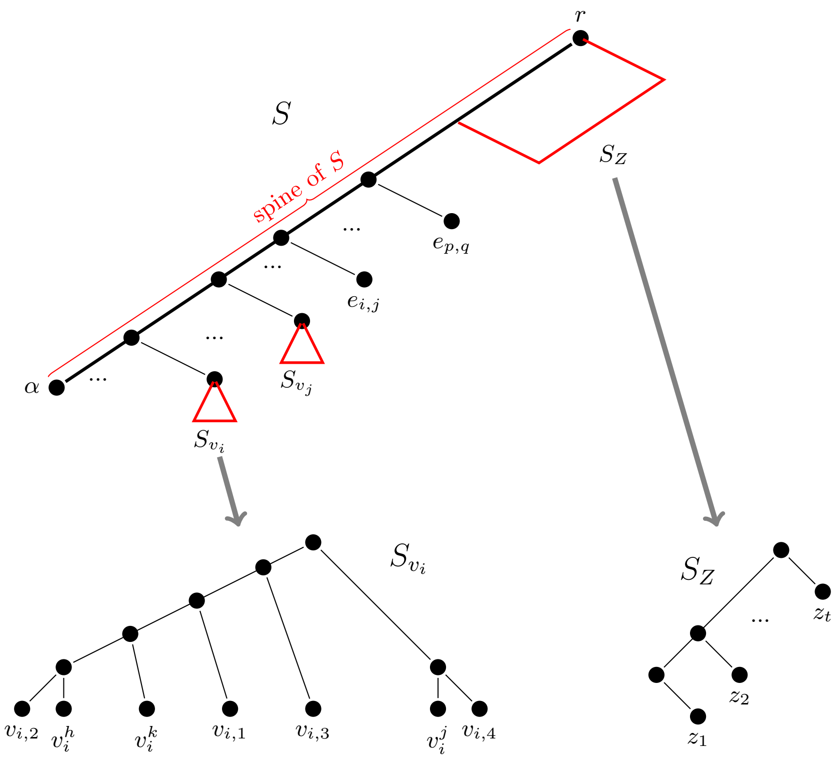


Fig. 3. The species tree S, and the subtrees S_{v_i}, S_Z of S. Notice that $i < j$, hence S_{v_j} is closer to the root than S_{v_i}.

Lemma 1. *Let v_i be a vertex of G. Then: (1) in a solution of MinLeafRem(2) over instance (T, S) at least three leaves are removed from T_{v_i}; (2) a solution of MinLeafRem(2) over instance (T, S) that contains a leaf of T_{v_i} with a label in $\{v_i^j, v_i^h, v_i^k\}$, contains at most three leaves of T_{v_i}.*

It follows from Remark 3 and Lemma 1 that a solution of MinLeafRem(2) over instance (T, S) is obtained by removing leaves from each T_{v_i} in essentially two possible ways: either remove the four leaves $\{v_{i,1}, v_{i,2}, v_{i,3}, v_{i,4}\}$, or remove the three leaves $\{v_i^j, v_i^h, v_i^k\}$. We will relate the former case to the vertex v_i being included in a vertex cover V' of G, and the latter case to the vertex v_i not included in V' (Lemma 4 and Lemma 5). We first give two preliminary lemmas.

Lemma 2. *Each solution of MinLeafRem(2) over instance (T, S) is obtained by removing at least one leaf from $T_{e_{ij}}$, for each $e_{i,j} \in E$.*

Proof. Direct corollary of Remark 2.

The following lemma will be used to show that the caterpillar tree T_Z is kept in a solution of MinLeafRem(2).

Lemma 3. *There is no optimal solution of MinLeafRem(2) over instance* (T, S) *that is obtained by removing less than* $4|V| + |E| + 1$ *leaves, one of them being a leaf of* T_Z.

Proof. Let T^* be a solution of MinLeafRem over instance (T, S) obtained from T by removing less than $4|V|+|E|+1$ leaves. Notice that, since $|Z| = 4|V|+|E|+1$, at least one leaf with a label in the set Z must be in T^*. Assume that a leaf with label z_h is removed from T^*. It is easy to see that inserting this leaf in T^* does not affect other nodes of T^*, that is the insertion of the leaf with label z_h does not cause any AD node to become a NAD node.

We are now ready to show the two main technical results of the reduction.

Lemma 4. *Let* $G = (V, E)$ *be an instance of MVCC and let* (T, S) *be the corresponding instance of MinLeafRem(2). Then, starting from a vertex cover* V' *of* G*, we can compute in polynomial time a solution of MinLeafRem(2) over instance* (T, S) *that is obtained by removing* $3|V| + |V'| + |E|$ *leaves from* T.

Proof. (Sketch) Let $V' \subseteq V$ be a vertex cover of $G = (V, E)$. Then we define a solution T^* by removing some leaves of the subtrees of T. We will denote by $T^*_{v_i}$ the subtree of T^* obtained from T_{v_i}, and by $T^*_{e_{i,j}}$ the subtree of T^* obtained from $T_{e_{ij}}$. The solution T^* is defined as follows:

- for each $v_i \in V'$, remove from the subtree T_{v_i} the set of leaves labelled by $\{v_{i,1}, v_{i,2}, v_{i,3}, v_{i,4}\}$ (hence the subtree $T^*_{v_i}$ has its leaf-set labelled by $\{v_i^j, v_i^h, v_i^k\}$);
- for each $v_i \in V \setminus V'$, remove from the subtree T_{v_i} the set of leaves labelled by $\{v_i^j, v_i^h, v_i^k\}$ (hence the subtree $T^*_{v_i}$ has its leaf-set labelled by $\{v_{i,1}, v_{i,2}, v_{i,3}, v_{i,4}\}$);
- for each $\{v_i, v_j\} \in E$, if $v_i \in V'$, then remove from $T_{e_{ij}}$ the leaf labelled by v_j^i (hence the subtree $T^*_{e_{i,j}}$ has its leaf-set labelled by $\{e_{i,j}, v_i^j\}$), else remove from $T_{e_{ij}}$ the leaf labelled by v_i^j (hence the subtree $T^*_{e_{i,j}}$ has its leaf-set labelled by $\{e_{i,j}, v_j^i\}$).

It is easy to see that the tree T^* is MD-consistent with S and that it is obtained by removing $3|V| + |E| + |V'|$ leaves from T.

Lemma 5. *Let* $G = (V, E)$ *be an instance of MVCC and let* (T, S) *be the corresponding instance of MinLeafRem(2). Then starting from a solution of MinLeafRem(2) over instance* (T, S) *that is obtained by removing at most* $3|V| + |E| + c$ *leaves from* T*, with* $1 \leq c \leq |V|$*, we can compute in polynomial time a vertex cover* V' *of* G *such that* $|V'| \leq c$.

Proof. (Sketch) Let T^* be a solution of MinLeafRem(2) over instance (T, S) obtained by removing at most $3|V| + |E| + c$ leaves from T, with $1 \leq c \leq |V|$. Let $T^*_{v_i}$, with $v_i \in V$, be the subtree of T^* obtained from T_{v_i} after removing appropriate leaves. Let $T^*_{e_{i,j}}$, with $\{v_i, v_j\} \in E$, be the subtree of T^* obtained from $T_{e_{ij}}$ after removing appropriate leaves.

We can show (using Remark 3 and Lemma 1) that $T^*_{v_i}$, for each $v_i \in V$, must be leaf-labelled either by the set $\{v_i^j, v_i^h, v_i^k\}$, or by the set $\{v_{i,1}, v_{i,2}, v_{i,3}, v_{i,4}\}$. Moreover, by Lemma 3, T^* contains all the leaves with labels in Z.

On the other hand, using Lemma 2, we can prove that $T^*_{e_{i,j}}$ must contain the leaf labelled by $e_{i,j}$ (otherwise the parent of the leaf labelled by $e_{i,j}$ on the spine of T, which is an AD node in T, becomes a NAD node) and exactly one leaf with label in $\{v_i^j, v_j^i\}$ (otherwise the parent of the subtree $T_{e_{i,j}}$ on the spine of T, which is an AD node in T, becomes a NAD node). Moreover, if $T^*_{e_{i,j}}$ contains a leaf labelled by v_i^j, then $T^*_{v_i}$ must be leaf-labelled by the set $\{v_i^j, v_i^h, v_i^k\}$ (otherwise the parent of the subtree $T_{e_{i,j}}$ on the spine of T becomes a NAD node), while if $T^*_{e_{i,j}}$ contains a leaf labelled by v_j^i, then T_{v_j} is leaf-labelled by the set $\{v_j^i, v_j^x, v_j^y\}$ (same reason as above).

It follows that the set

$$V' = \{v_i : T^*_{v_i} \text{ is leaf-labelled by } \{v_i^j, v_i^h, v_i^k\}\ \}$$

is a vertex cover of G of minimum size, as for each edge $e_{i,j} \in E$, exactly one of v_i and v_j is contained in V'. It is easy to see that $|V'| \leq c$.

Theorem 1. *MinLeafRem(2) is APX-hard.*

Proof. It follows from Lemma 4 and from Lemma 5, that we have designed an L-reduction from MVCC to MinLeafRem(2). Since MVCC is APX-hard [1], it follows that also MinLeafRem(2) is APX-hard.

4 Fixed-Parameter Algorithms

Since the MinLeafRem problem is APX-hard, it is interesting to see if the problem becomes tractable under some biological meaningful parameterizations (for an introduction to parameterized complexity see [23]). In this section we focus on the two following parameterizations: (1) the size of the solution of MinLeafRem (that is the number of leaves removed from T in order to obtain a tree MD-consistent with S), and (2) the number of labels in Γ associated with multiple leaves of T (i.e. the number of genomes containing multiple gene copies). We will give two fixed-parameter algorithms for MinLeafRem under these two parameterizations.

Notice that a third natural parameter would be the maximum number of leaves in T associated with a single label of Γ (i.e. the maximum number of gene copies in a given genome). However, we have already proved in the last section that the MinLeafRem problem is already APX-hard when each label has at most two occurrences in the gene tree T.

4.1 MinLeafRem Parameterized by the Number of Leaves Removed

In this section, we investigate the parameterized complexity of MinLeafRem, when the problem is parameterized by the size of the solution, that is the number of leaves removed from T. We present a fixed-parameter algorithm that is based on the depth-bounded search tree technique. Denote by c the size of the solution, that is the number of leaves that have to be removed from T in order to get a tree T^* which is MD-consistent with the species tree S.

If T does not contain NAD nodes, then T is MD-consistent with S and it requires no leaf removal. Hence in what follows we assume that T contains at least one NAD node.

Now, consider a NAD node v of T. Let s be the node of S where v is mapped. Since v is a NAD node, it follows that at least one of its children, denoted as v_l and v_r, is mapped by $\text{lca}_{T,S}$ in s. Assume w.l.o.g. that v_l is mapped in s, that is $\text{lca}_{T,S}(v_l) = s$. Denote by s_l and s_r the left child and the right child respectively of s. Since $\text{lca}_{T,S}(v_l) = s$, it follows that $\mathcal{C}(v_l) \subseteq \mathcal{C}(s)$, $\mathcal{C}(v_l) \cap \mathcal{C}(s_l) = X_1 \neq \emptyset$ and $\mathcal{C}(v_l) \cap \mathcal{C}(s_r) = X_2 \neq \emptyset$. It follows that either the leaves of $T(v_l)$ having labels in X_1, or the leaves of $T(v_l)$ having labels in X_2, or the leaves of $T(v_r)$ must be deleted from T. We formally prove this property in the following lemma.

Lemma 6. *Let v be a NAD node of a gene tree T, and let v_l, v_r be the children of v, such that $lca_{T,S}(v) = lca_{T,S}(v_l) = s$. Let s_l, s_r be the children of s. Then, there is no subtree included in T that is MD-consistent with S and that contains a leaf of $T(v_l)$ with a label in $X_1 = \mathcal{C}(v_l) \cap \mathcal{C}(s_l)$, a leaf of $T(v_l)$ with a label in $X_2 = \mathcal{C}(v_l) \cap \mathcal{C}(s_r)$, and a leaf of $T(v_r)$.*

Due to Lemma 6, we can design a fixed-parameter algorithm for MinLeafRem parameterized by c as follows. Let $Dup(T) = \langle v^1, \ldots, v^z \rangle$ be the ordered list of NAD nodes of T in a breadth-first visit of T. The algorithm at each step chooses the first node v^1 of $Dup(T)$. Let $\text{lca}_{T,S}(v^1) = s$, and let s_l and s_r be the two children of s. Consider a child v^1_x, with $v^1_x \in \{v^1_l, v^1_r\}$, of v^1 that is mapped in s, and let $v^1_{\bar{x}}$ be the other child of v^1. Let $\mathcal{C}(v^1_x) \cap \mathcal{C}(s_l) = X_1 \neq \emptyset$, $\mathcal{C}(v^1_x) \cap \mathcal{C}(s_r) = X_2 \neq \emptyset$.

Now, the algorithm branches in the following cases:

1. Remove the leaves of $T(v^1_x)$ with label in X_1 from $L(T)$ and suppress the resulting degree two nodes;
2. Remove the leaves of $T(v^1_x)$ with label in in X_2 from $L(T)$ and suppress the resulting degree two nodes;
3. Remove the subtree $T(v^1_{\bar{x}})$ from T, and suppress the resulting degree two node.

After the branching, the algorithm outputs a subtree T' of T. Then the lca mapping $\text{lca}_{T',S}$ between T' and S is computed (in polynomial time), and the ordered list $Dup(T')$ of NAD nodes of T' is computed (again in polynomial time). The algorithm stops either when it finds a subtree T' of T that is MD-consistent with S, or when there is no subtree included in T that can be obtained with c leaf removals.

Theorem 2. *The algorithm computes if there exists a solution of size at most c for MinLeafRem in time $O(3^c poly\ (|V(T)|, |V(S)|))$.*

Proof. The correctness of the algorithm follows from Lemma 6.

Now, we focus on the time complexity of the algorithm. At each step the algorithm branches in three possible cases, and for each of these cases at least one leaf is removed. As the depth of the search tree is bounded by c, the size of the search tree is bounded by 3^c. Since after each branching we require at most time $O(\text{poly }(|V(T)||V(S)|))$ to compute T', $\text{lca}_{T',S}$, and $Dup(T')$, it follows that the overall time complexity of the algorithm is $O(3^c\text{poly }(|V(T)||V(S)|))$.

4.2 MinLeafRem Parameterized by the Number of Labels with Multiple Copies

In this section we give a fixed-parameter algorithm for MinLeafRem, when the parameter is the number of labels associated with multiple leaves of T. Denote by $\Gamma_D \subseteq \Gamma$, the subset of labels associated with multiple leaves of T.

Let x be a node of T, having children x_l, x_r, and let y be a node of S, with children y_l, y_r. Given $\Gamma'_D \subseteq \Gamma_D$, we define $M[T(x), S(y), \Gamma'_D]$ as the minimum number of leaves that have to be removed to obtain a tree T' included in $T(x)$ such that (1) T' is MD-consistent with $S(y)$ and (2) the subset $\Gamma'_D \subseteq \Gamma(T')$. We can compute $M[T(x), S(y), \Gamma'_D]$ applying the following recurrence:

$$M[T(x), S(y), \Gamma'_D] = \min_{\substack{\Gamma'_{1,D} \subseteq \Gamma'_D,\\ \Gamma'_{2,D} \subseteq \Gamma'_D,\\ \Gamma'_{1,D} \cup \Gamma'_{2,D} = \Gamma'_D}} \begin{cases} M[T(x_l), S(y_l), \Gamma'_{1,D}] + M[T(x_r), S(y_r), \Gamma'_{2,D}] \\ \quad \text{if } \Gamma'_{1,D} \cap \Gamma'_{2,D} = \emptyset, \\ M[T(x_l), S(y_r), \Gamma'_{1,D}] + M[T(x_r), S(y_l), \Gamma'_{2,D}] \\ \quad \text{if } \Gamma'_{1,D} \cap \Gamma'_{2,D} = \emptyset, \\ M[T(x_l), S(y), \Gamma'_{1,D}] + M[T(x_r), S(y), \Gamma'_{2,D}] \\ \quad \text{if } \Gamma'_{1,D} \cap \Gamma'_{2,D} \neq \emptyset \\ M[T(x_l), S(y), \Gamma'_D] + |L(T(x_r))| \\ M[T(x_r), S(y), \Gamma'_D] + |L(T(x_l))| \\ M[T(x), S(y_l), \Gamma'_D] \\ M[T(x), S(y_r), \Gamma'_D] \end{cases} \tag{1}$$

Now, we define the basic cases of the recurrence, when each of $T(x)$ and $S(y)$ is a single leaf, with $\Gamma(T(x)) = \lambda_G$ and $\Gamma(S(y)) = \lambda_S$. If $\lambda_G = \lambda_S$, then $M[T(x), S(y), \Gamma'_D] = 0$ if $\Gamma'_D = \{\lambda_G\}$, $M[T(x), S(y), \Gamma'_D] = 0$ if $\Gamma'_D = \emptyset$, else $M[T(x), S(y), \Gamma'_D] = +\infty$. If $\lambda_G \neq \lambda_S$, then $M[T(x), S(y), \Gamma'_D] = 1$ if $\Gamma'_D = \emptyset$, else $M[T(x), S(y), \Gamma'_D] = +\infty$.

The correctness of Recurrence 1, is proved in the following lemma.

Lemma 7. *Let T be a gene tree, let S be a species tree, and let $\Gamma_D \subseteq \Gamma$ be the set of labels associated with multiple leaves of T. Let x be a node of T and y be a node of S, and consider a subset $\Gamma'_D \subseteq \Gamma_D$. Then $M[T(x), S(y), \Gamma'_D] = c$ if and only if there exists a tree T' included in $T(x)$ such that (i) T' is MD-consistent with $S(y)$; (ii) T' is obtained by removing c leaves; (iii) $\Gamma'_D \subseteq \Gamma(T')$.*

Theorem 3. *Given a gene tree T and a species tree S, let $\Gamma_D \subseteq \Gamma$ be the set of labels associated with multiple leaves of T. Then an optimal solution of MinLeaf over instance (T, S) can be computed in time $O(4^{|\Gamma_D|} poly(|V(T)||V(S)|))$.*

Proof. By Lemma 7 a solution of of MinLeaf over instance (T, S), is obtained looking for the minimum of the values $M[T(r_T), S(r_S), \Gamma'_D]$, for each $\Gamma'_D \subseteq \Gamma_D$, where r_T (r_S respectively) is the root of T (S respectively).

Now, we prove in the following that the time complexity of the algorithm is $O(4^{|\Gamma_D|}\ poly(|V(T)||V(S)|))$. It is easy to see that the time complexity to compute Recurrence 1 is dominated by case 3. The entries $M[T(x), S(y), \Gamma'_D]$ are $O(2^{|\Gamma_D|}|V(T)||V(S)|))$. For each pair of nodes $x \in V(T)$, $y \in V(S)$, we have to consider $O(4^{|\Gamma_D|})$ possible combinations. Indeed, the number of subsets $\Gamma'_{1,D}, \Gamma'_{2,D} \subseteq \Gamma'_D$, with $\Gamma'_D = \Gamma'_{1,D} \cup \Gamma'_{2,D}$, is $4^{|\Gamma_D|}$, since we have to consider all possible subsets Γ'_D of Γ_D and, for each subset Γ'_D, we have to consider all possible subsets $\Gamma'_{1,D}, \Gamma'_{2,D} \subseteq \Gamma'_D$, with $\Gamma'_D = \Gamma'_{1,D} \cup \Gamma'_{2,D}$. It follows that we have to consider $4^{|\Gamma_D|}$ combinations, since there are $4^{|\Gamma_D|}$ possible ways to split set Γ_D into four disjoint subsets (in this case the subsets are $\Gamma_D \setminus \Gamma'_D$, $\Gamma'_{1,D} \setminus \Gamma'_{2,D}$, $\Gamma'_{2,D} \setminus \Gamma'_{1,D}$, and $\Gamma'_{1,D} \cap \Gamma'_{2,D}$). For each combination, the recursion can be computed in constant time.

Finding the minimum value in the entries $M[T(r_G), S(r_S), \Gamma'_D]$ requires time $O(2^{|\Gamma_D|}|V(T)||V(S)|)$, hence the overall time complexity to find an optimal solution of MinLeafRem over instance (T, S), is $O(4^{|\Gamma_D|} poly(|V(T)||V(S)|))$.

5 Conclusion

We presented complexity results and gave two parameter tractable versions of the Minimum Leaf Removal Problem. This problem has been shown to be a natural one to consider for preprocessing gene trees prior to reconciliation [8]. Even though the problem is proved to be APX-hard, a polynomial-time heuristic, showing a good performance on simulated data sets, has already been developed [12]. The fixed-parameter algorithms presented in this paper nicely complement those in [12].

In the case of species tree inference, it has been shown in [8] that deciding whether a gene tree T is an MD-tree, i.e. a tree that is MD-consistent with at least one species tree, can be done in polynomial time and space, as well as computing a parsimonious species tree. In the case of a tree T being not an MD-tree, a natural extension of the Minimum Leaf Removal Problem would be to find the minimum number of leaves that have to be removed from a given gene tree T in order for T to be an MD-tree. Having appropriate solutions for this problem would give natural ways for correcting gene trees prior to species tree inference. We are presently studying the theoretical complexity of this problem.

Acknowledgements. We thank Krister M. Swenson for his careful reading of the proofs, and his advices on notations and presentation of the paper.

References

1. Alimonti, P., Kann, V.: Some APX-completeness results for cubic graphs. Theor. Comput. Sci. 237(1-2), 123–134 (2000)
2. Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D.: Basic local alignment search tool. J. Mol. Biol. 215(3), 403–410 (1990)
3. Arvestad, L., Berglung, A.C., Lagergren, J., Sennblad, B.: Gene tree reconstruction and orthology analysis based on an integrated model for duplications and sequence evolution. In: Gusfield, D. (ed.) RECOMB 2004, pp. 326–335. ACM, New York (2004)
4. Blin, G., Bonizzoni, P., Dondi, R., Rizzi, R., Sikora, F.: Complexity Insights of the Minimum Duplication Problem. In: Bieliková, M., Friedrich, G., Gottlob, G., Katzenbeisser, S., Turán, G. (eds.) SOFSEM 2012. LNCS, vol. 7147, pp. 153–164. Springer, Heidelberg (2012)
5. Blomme, T., Vandepoele, K., Bodt, S.D., Silmillion, C., Maere, S., van de Peer, Y.: The gain and loss of genes during 600 millions years of vertebrate evolution. Genome Biology 7, R43 (2006)
6. Bonizzoni, P., Della Vedova, G., Dondi, R.: Reconciling a gene tree to a species tree under the duplication cost model. Theoretical Computer Science 347, 36–53 (2005)
7. Chang, W.-C., Eulenstein, O.: Reconciling Gene Trees with Apparent Polytomies. In: Chen, D.Z., Lee, D.T. (eds.) COCOON 2006. LNCS, vol. 4112, pp. 235–244. Springer, Heidelberg (2006)
8. Chauve, C., El-Mabrouk, N.: New Perspectives on Gene Family Evolution: Losses in Reconciliation and a Link with Supertrees. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 46–58. Springer, Heidelberg (2009)
9. Chen, K., Durand, D., Farach-Colton, M.: Notung: Dating gene duplications using gene family trees. Journal of Computational Biology 7, 429–447 (2000)
10. Cotton, J., Page, R.: Rates and patterns of gene duplication and loss in the human genome. Proceedings of the Royal Society of London. Series B 272, 277–283 (2005)
11. Demuth, J., Bie, T.D., Stajich, J., Cristianini, N., Hahn, M.: The evolution of mammalian gene families. PLoS ONE 1, e85 (2006)
12. Doroftei, A., El-Mabrouk, N.: Removing Noise from Gene Trees. In: Przytycka, T.M., Sagot, M.-F. (eds.) WABI 2011. LNCS (LNBI), vol. 6833, pp. 76–91. Springer, Heidelberg (2011)
13. Durand, D., Haldórsson, B., Vernot, B.: A hybrid micro-macroevolutionary approach to gene tree reconstruction. Journal of Computational Biology 13, 320–335 (2006)
14. Eichler, E., Sankoff, D.: Structural dynamics of eukaryotic chromosome evolution. Science 301, 793–797 (2003)
15. Goodman, M., Czelusniak, J., Moore, G., Romero-Herrera, A., Matsuda, G.: Fitting the gene lineage into its species lineage, a parsimony strategy illustrated by cladograms constructed from globin sequences. Systematic Zoology 28, 132–163 (1979)
16. Górecki, P., Eulenstein, O.: A Linear Time Algorithm for Error-Corrected Reconciliation of Unrooted Gene Trees. In: Chen, J., Wang, J., Zelikovsky, A. (eds.) ISBRA 2011. LNCS, vol. 6674, pp. 148–159. Springer, Heidelberg (2011)
17. Gorecki, P., Tiuryn, J.: DLS-trees: a model of evolutionary scenarios. Theoretical Computer Science 359, 378–399 (2006)

18. Guigó, R., Muchnik, I., Smith, T.: Reconstruction of ancient molecular phylogeny. Molecular Phylogenetics and Evolution 6, 189–213 (1996)
19. Hahn, M.: Bias in phylogenetic tree reconciliation methods: implications for vertebrate genome evolution. Genome Biology 8(R141) (2007)
20. Hahn, M., Han, M., Han, S.G.: Gene family evolution across 12 *drosophilia* genomes. PLoS Genetics 3, e197 (2007)
21. Kristensen, D., Wolf, Y., Mushegian, A., Koonin, E.: Computational methods for gene orthology inference. Briefings in Bioinformatics 12(5), 379–391 (2011)
22. Ma, B., Li, M., Zhang, L.: From gene trees to species trees. SIAM J. on Comput. 30, 729–752 (2000)
23. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press, Oxford (2006)
24. Ohno, S.: Evolution by gene duplication. Springer, Berlin (1970)
25. Page, R.: Maps between trees and cladistic analysis of historical associations among genes, organisms, and areas. Systematic Biology 43, 58–77 (1994)
26. Page, R.: Genetree: comparing gene and species phylogenies using reconciled trees. Bioinformatics 14, 819–820 (1998)
27. Page, R., Charleston, M.: Reconciled trees and incongruent gene and species trees. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 37, 57–70 (1997)
28. Page, R., Cotton, J.: Vertebrate phylogenomics: reconciled trees and gene duplications. In: Pacific Symposium on Biocomputing, pp. 536–547 (2002)
29. Sanderson, M., McMahon, M.: Inferring angiosperm phylogeny from EST data with widespread gene duplication. BMC Evolutionary Biology 7, S3 (2007)
30. Vernot, B., Stolzer, M., Goldman, A., Durand, D.: Reconciliation with non-binary species trees. Journal of Computational Biology 15, 981–1006 (2008)

On the Closest String via Rank Distance

Liviu P. Dinu[1] and Alexandru Popa[2]

[1] University of Bucharest, Faculty of Mathematics and Computer Science, Academiei 14, RO-010014, Bucharest, Romania
ldinu@fmi.unibuc.ro

[2] Department of Communications & Networking, Aalto University School of Electrical Engineering, Aalto, Finland
alexandru.popa@aalto.fi

Abstract. Given a set S of k strings of maximum length n, the goal of the *closest substring problem (CSSP)* is to find the smallest integer d (and a corresponding string t of length $\ell \leq n$) such that each string $s \in S$ has a substring of length ℓ of "distance" at most d to t. The *closest string problem (CSP)* is a special case of CSSP where $\ell = n$. CSP and CSSP arise in many applications in bioinformatics and are extensively studied in the context of Hamming and edit distance. In this paper we consider a recently introduced distance measure, namely the rank distance. First, we show that the CSP and CSSP via rank distance are NP-hard. Then, we present a polynomial time k-approximation algorithm for the CSP problem. Finally, we give a parametrized algorithm for the CSP (the parameter is the number of input strings) if the alphabet is binary and each string has the same number of 0's and 1's.

1 Introduction

1.1 Motivation

In many important problems in computational biology a common task is to summarize the information shared by a collection of sequences (e.g. DNA, proteins). More specifically, given a set of DNA or protein sequences we want to design a new sequence that is similar to the input sequences. We mention several applications that involve this task: the design of genetic drugs with a structure similar to a set of existing sequences of RNA [12], PCR primer design [12, 11], genetic probe design [12], antisense drug design [5], finding unbiased consensus of a protein family [2], motif finding [20, 13].

The aforementioned task is formalized as the *closest string problem (CSP)*: given a set S of strings over an alphabet Σ, find the string that minimizes the longest distance (or radius) from the strings from S. The distance is defined to suit to the corresponding application. For example, the CSP was studied the first time in the area of coding theory, to determine the best encoding of a set of messages, and the measure used to compare the strings is the Hamming distance [10].

In computational biology the standard method for sequence comparison is by sequence alignment. Sequence alignment is the procedure of comparing two sequences (pairwise alignment) or more sequences (multiple alignment) by searching for a series of individual characters or character patterns that are in the same order in the sequences. The standard pairwise alignment method is based on dynamic programming: the algorithm compares every pair of characters of the two sequences and generates an alignment and a score (edit distance is based on a scoring scheme for insertion or deletion penalties). The sequence alignment procedure is by far too slow to compare a large number of sequences and therefore alternative approaches might be explored in bioinformatics if we can answer the following question: is it possible to design a sequence distance which is at the same time easily computable and non-trivial? This important problem, known also as DNA sequence comparison, is a major open problem in bioinformatics [21].

The standard distances with respect to the alignment principle are edit (Levenshtein) distance or ad-hoc variants. The study of rearrangement genome [17] is investigated also with Kendall tau distance (i.e. the minimum number of swaps needed to transform a permutation in other). To measure the similarity between strings Dinu proposes a new distance measure, termed *rank distance (RD)* [7], with applications in biology [9]. Rank distance can be computed fast and benefits from some features of the edit distance. The RD distance between two strings s_1 and s_2 is computed as follows:

1. For each character c, find its first occurrence in s_1 and the first occurrence in s_2 and store the difference between their positions. Repeat this procedure for the second occurrence, third occurrence etc. (if there is no kth occurrence of c in s_1 or s_2, we define the position of c to be zero).
2. Sum up the above values and obtain the rank distance.

In other words, rank distance measures the "gap" between the positions of a letter in the two given strings, and then sums up these values. Intuitively, rank distance gives us the total non-alignment score between two sequences.

Clearly, rank distance gives score zero only to the letters that are in the same position in both strings, as Hamming distance does (we recall that Hamming distance is the number of positions in which two strings of the same length differ). On the other hand, an important aspect is the reduced sensitivity of the rank distance with respect to deletions and insertions. Reduced sensitivity is of paramount importance, since it allows the *ad hoc extension to arbitrary strings*, without affecting the low computational complexity. In contrast, the extensions of Hamming distance are mathematically optimal but computationally too heavy, and lead to the *edit-distance*, which is the base of the standard alignment principle. Thus, the rank distance sides with Hamming distance rather than Levenshtein distance as far as computational complexity is concerned: a significant indicator is the fact that in the Hamming and rank distance case the median string problem is tractable [8], while in the edit distance case it is NP-hard [4].

RD is easy to implement, does not use the standard alignment principle, and has an extremely good computational behavior. Another advantage of RD is that it imposes minimal hardware demands: it runs in optimal conditions on modest computers, reducing the costs and increasing the number of possible users. For example, the time needed to compare a DNA string of $45,000$ nucleotides with other 150 DNA strings (with similar length), on a computer with 224 MB RAM and 1.4 GHz processor is no more than six seconds.

1.2 Previous Work

The first similarity measure used in the closest string problem is the Hamming distance and emerged from a coding theory application [10]. The CSP via Hamming distance is known to be NP-complete [10] and there exist a number of approximation algorithms and heuristics (see, for example, [12–14]).

As the CSP is used in many contexts, alternative distance measures were introduced. The most studied alternative approach is the edit distance. In many practical situations, the alphabet is of fixed constant size (in computational biology, the DNA and protein alphabets are of size 4 and 20, respectively). The closest string problem via edit distance is NP-hard even for binary alphabets [15, 16]. The existence of fast exact algorithms when the number of input strings is fixed is investigated in [15].

The study of genome rearrangement introduces new problems related to closest string via new distances. Recently, Popov [18] shows that the CSP via swap distance (or Kendal distance) and element duplication distance (i.e. the minimum number of element duplications needed to transform a string into another) is also NP-complete.

1.3 Our Contributions

In this paper we study the computational hardness of the *closest string problem via rank distance (CSRD)*. We show that the CSRD is NP-hard via a polynomial time reduction from 3-SAT (i.e. given a boolean formula in conjunctive normal form, where each clause has at most three literals, the goal of the 3-SAT problem is to decide if there exists a truth assignment of the variables such that the formula is satisfiable). Then, we present a k-approximation algorithm for the problem. The approximation algorithm has two steps: first we show a reduction of the CSRD problem to a matching problem and then we use an algorithm from [3]. Finally, we show a parametrized algorithm with respect to the number of input strings if the alphabet is binary and each string has the same number of 0's and 1's. We first prove that in this case RD is equivalent to Kendall Tau and then we use Schwarz's algorithm [19].

The rest of the paper is organized as follows. Section 2 introduces notation and preliminary notions. In Section 3 we show the hardness result for the CSRD. Then, in Section 4 we present the k-approximation algorithm. In Section 5 we describe the parametrized algorithm for binary alphabets. Section 6 is reserved for conclusions and open questions.

2 Preliminaries

In this section we introduce notation and preliminaries. We first introduce the rank distance and then we define closest string and closest substring problems.

Definition 1. *Let $\mathcal{U} = \{1, 2, \dots, m\}$ be a finite set of objects, named universe. A* ranking *over $\mathcal{U}$ is an ordered list $\tau = (x_1 > x_2 > ... > x_d)$, where $x_i \in \mathcal{U}$ for all $1 \leq i \leq d$, $x_i \neq x_j$ for all $1 \leq i \neq j \leq d$, and $>$ is a strict ordering relation on the set $\{x_1, x_2, ..., x_d\}$.*

A ranking defines a partial function on $\mathcal{U}$ where for each object $i \in \mathcal{U}$, $ord(\tau, i)$ represents the position of the object i in the ranking τ. The rankings that contain all the objects of an universe $\mathcal{U}$ are termed *full rankings*, while the others are *partial rankings*. By convention, if $x \in \mathcal{U} \setminus \sigma$, we have $ord(\sigma, x) = 0$.

Definition 2. *Given two partial rankings σ and τ over the same universe $\mathcal{U}$, we define the rank distance between them as:*

$$\Delta(\sigma, \tau) = \sum_{x \in \sigma \cup \tau} |ord(\sigma, x) - ord(\tau, x)|.$$

In [7] Dinu proves that Δ is a distance function. The rank distance is an extension of the Spearman footrule distance [6], defined below.

Definition 3. *If σ and τ are two permutations of the same order, then $\Delta(\sigma, \tau)$ is named the Spearman footrule distance.*

The rank distance is naturally extended to strings. The following observation is immediate: if a string does not contain identical symbols, it can be transformed directly into a ranking (the rank of each symbol is its position in the string). Conversely, each ranking can be viewed as a string, over an alphabet identical to the universe of the objects in the ranking. The next definition formalizes the transformation of strings into rankings.

Definition 4. *Let n be an integer and let $w = a_1 \dots a_n$ be a finite word of length n over an alphabet Σ. We define the extension to rankings of w, $\bar{w} = a_{1,i(1)} \dots a_{n,i(n)}$, where $i(j)$ is the number of occurrences of a_j in the string $a_1 a_2 \dots a_j$.*

Example 1. If $w = aaababbbac$ then $\bar{w} = a_1 a_2 a_3 b_1 a_4 b_2 b_3 b_4 a_5 c_1$.

Observe that given $\bar{w}$ we can obtain w by simply deleting all the indexes. Note that the transformation of a string into a ranking can be done in linear time (by storing for each symbol, in an array, the number of times it appears in the string). We extend the rank distance to arbitrary strings as follows:

Definition 5. *Given $w_1, w_2 \in \Sigma^*$, we define $\Delta(w_1, w_2) = \Delta(\bar{w}_1, \bar{w}_2)$.*

Example 2. Consider the following two strings $x = abcaa$ and $y = baacc$. Then, $\bar{x} = a_1b_1c_1a_2a_3$ and $\bar{y} = b_1a_1a_2c_1c_2$. The order of the characters in $\bar{x}$ and $\bar{y}$ is the following:

- a_1: 1 and 2;
- a_2: 4 and 3;
- a_3: 5 and 0 (as a_3 does not appear in $\bar{y}$, it has order 0);
- b_1: 2 and 1;
- c_1: 3 and 4;
- c_2: 0 and 5 (as c_2 does not appear in $\bar{x}$, it has order 0);

Thus, the rank distance between x and y is the sum of the absolute differences between the orders of the characters in $\bar{x}$ and $\bar{y}$

$$\Delta(x, y) = |1-2| + |4-3| + |5-0| + |2-1| + |3-4| + |0-5| = 14$$

The computation of the RD between two rankings can be done in linear time in the cardinality of the universe. Our universe has precisely $|w_1| + |w_2|$ objects and, thus, the RD between w_1 and w_2 can be computed in linear time.

Let χ_n be the space of all strings of size n over an alphabet Σ and let $p_1,\ p_2, \ldots, p_k$ be k strings from χ_n. The center string problem is to find the center of the sphere of minimum radius that includes all the k strings. An alternative formulation of the problem is to find a string from χ_n which minimizes the distance to all the input strings. We study the closest string problem under the metric defined by the rank distance.

Problem 1 (Closest string via rank distance). Let $P = \{p_1, p_2, \ldots, p_k\}$ be a set of k length n strings over an alphabet Σ. The *closest string problem via rank distance (CSRD)* is to find a minimal integer d (and a corresponding string t of length n) such that the maximum rank distance from t to any string in P is at most d. We say that t is the closest string to P and we name d the radius. Formally, the goal is to compute:

$$\min_{x \in \chi_n} \max_{i=1..k} \Delta(x, p_i) \tag{1}$$

Remark 1. The string t that minimizes (1) is not necessary unique. For example, if $P = \{(1, 2, 3),\ (3, 1, 2),\ (2, 3, 1)\}$ is a set of three length 3 permutations, then every length 3 permutation is a closest string to P.

Let P be the set of all closest strings with $CSRD(P)$:

$$CSRD(P) = \arg\min_{x \in \chi_n} \max_{i=1..k} \Delta(x, p_i) \tag{2}$$

The CSSP is a generalization of CSP in which the concern is to find a string similar to substrings of the input.

Problem 2 (Closest substring via rank distance). Let $P = \{p_1, p_2, \dots, p_k\}$ be a set of k length n strings over an alphabet Σ. The *closest substring problem via rank distance* is to find a minimal integer d (and a corresponding string t of length $\ell \leq n$) and a set $P' = \{p'_1, p'_2, \dots, p'_k\}$, where p'_i is a substring of p_i for all $1 \leq i \leq k$ such that the maximum rank distance from t to any string in P' is at most d. We say that t is the closest substring to P and we name d the radius. Formally, the goal is to compute:

$$\min_{x \in \chi_\ell} \max_{i=1..k} \min_{p'_i} \Delta(x, p'_i)$$

2.1 Pareto Optimality

In this subsection we prove that the CSRD satisfies the Pareto optimality criterion introduced by Arrow [1]. We use this criterion in the approximation algorithm from Section 4.

The Pareto optimality criterion says that given a set of rankings $\mathcal{T} = \{v_1, v_2, \dots, v_t\}$, such that t rankings agree on a pair $\{a, b\}$ (i.e. in all the rankings a is preferred to b or vice-versa), then the aggregation of $\mathcal{T}$ maintains their relative ranking (in our case the aggregation is the center string).

Lemma 1. *Let $\mathcal{T} = \{v_1, v_2, \dots v_t\}$ be a set of rankings and a pair of elements $\{a, b\}$ such that a is preferred to b in all the t rankings. There exists a closest string which satisfies the Pareto optimality criterion.*

Proof. Assume by contradiction that an optimal aggregation x permits $b < a$, even though $\forall v \in \mathcal{T}$ $a < b$ in v. We show that by swapping a and b in x, we obtain the same or a better result. Let y be the permutation obtained from x after swapping a and b. We show that y yields a score which is at most the score of x.

From the definition of CSRD, we know that x satisfies the following:

$$\min_{x \in \chi_n} \max_{v \in \mathcal{T}} \Delta(x, v)$$

We prove that

$$\Delta(y, v) \leq \Delta(x, v), \ \mathit{for\ all}\ v \in \mathcal{T}.$$

Let $v \in \mathcal{T}$ be a certain ranking. The rank distances $\Delta(x, v)$ and $\Delta(y, v)$ from v to x and to y, respectively are:

$$\sum_{c \notin \{a,b\}} (|ord(x,c) - ord(v,c)| + |ord(x,a) - ord(v,a)| + |ord(x,b) - ord(v,b)|) \quad (3)$$

$$\sum_{c \notin \{a,b\}} (|ord(y,c) - ord(v,c)| + |ord(y,a) - ord(v,a)| + |ord(y,b) - ord(v,b)|) \quad (4)$$

Since y is obtained from x only by swapping a and b, the first term from 3 is equal to first term from 4, hence, we have:

$$\begin{aligned}\Delta(x,v) - \Delta(y,v) &= |ord(x,a) - ord(v,a)| + |ord(x,b) - ord(v,b)| \\ &- |ord(y,a) - ord(v,a)| - |ord(y,b) - ord(v,b)| \\ &= |ord(y,b) - ord(v,a)| + |ord(y,a) - ord(v,b)| \\ &- |ord(y,a) - ord(v,a)| - |ord(y,b) - ord(v,b)| \\ &= |ord(y,b) - ord(v,a)| - |ord(y,b) - ord(v,b)| \\ &- (|ord(y,a) - ord(v,a)| - |ord(y,a) - ord(v,b)|) \\ &\geq 0\end{aligned}$$

To obtain the last inequality, we use :

- $ord(v,b) > ord(v,a)$ (from the hypothesis).
- $f(x) = |x - ord(v,a)| - |x - ord(v,b)|$ is an increasing function (this follows from Lemma 1 in [7]).
- $ord(y,b) > ord(y,a)$.

Thus, $\Delta(y,v) \leq \Delta(x,v),\ for\ all\ v \in \mathcal{T}$ and the theorem follows. □

Lemma 1 says that if an element a is preferred to another element b in all the rankings, there exists a center in which a is preferred to b. Can Lemma 1 be extended to multiple pairs? The following theorem affirmatively answers this question.

Theorem 1. *Let $T = \{v_1, v_2, \ldots v_t\}$ be a set of rankings and k pairs of elements $\{a_1, b_1\}, \{a_2, b_2\}, \ldots, \{a_k, b_k\}$ such that a_i is preferred to b_i for all $1 \leq i \leq k$ in all the t rankings. There exist a closest string which satisfies the Pareto optimality criterion for all the k pairs.*

Proof. Let (a_1, b_1) be the first pair of elements to whom we can apply Lemma 1. We obtain a ranking π where a_1 is preferred to b_1. If there exists a pair (a_i, b_i) such that b_i is preferred to a_i in π, then by interchanging a_i and b_i in π we obtain a ranking π_1 which is also in $CSRD(T)$ (according to the proof of Lemma 1) and a_i is preferred to b_i too. We continue until all the pairs satisfy the Pareto criterion. Thus, the theorem follows. □

3 Hardness of the Closest String via Rank Distance

In this section we prove that the CSRD problem is NP-hard via a reduction from the 3-SAT problem (a version of the SAT problem where each clause contains exactly three literals). This is formally stated in the next theorem.

Theorem 2. The closest string problem via rank distance *is NP-hard.*

Proof. Given a 3-SAT formula ϕ with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$ we construct an instance of the *CSRD* problem with $m + 4$ input strings, each of length $4n$. The alphabet Σ contains $2n$ characters, $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$, two for each variable. We prove that if we can solve *CSRD* in polynomial time, then we can decide if ϕ is satisfiable.

For a clause $c_i = (z_a \vee z_b \vee z_c)$, where z_a, z_b, z_c are the literals corresponding to the variables x_a, x_b, x_c (i.e. either the variables or their negations), we construct a string s_i:

– if z_a is a negation of a variable, then we set $s_i[4a-3] = x_a$, $s_i[4a-2] = y_a$, $s_i[4a-1] = x_a$, $s_i[4a] = y_a$. We proceed similarly for z_b and z_c.
– if z_a is not a negation of a variable (i.e. if $z_a = x_a$), then we set $s_i[4a-3] = y_a$, $s_i[4a-2] = x_a$, $s_i[4a-1] = y_a$, $s_i[4a] = x_a$. We proceed similarly for z_b and z_c.
– for all the other positions w different from a, b or c we set $s_i[4w-3] = x_w$, $s_i[4w-2] = y_w$, $s_i[4w-1] = y_w$, $s_i[4w] = x_w$

In the construction, we assume that a clause does not contain both a variable and its negation (these clauses can be easily removed since they are satisfiable by any assignment). We also construct four strings aux_1, aux_2, aux_3, aux_4. The string aux_1 has, for all $1 \leq j \leq n$, $aux_1[4j-3] = y_j$, $aux_1[4j-2] = x_j$, $aux_1[4j-1] = x_j$, $aux_1[4j] = y_j$. The string aux_2 has, for all $1 \leq j \leq n$, $aux_2[4j-3] = x_j$, $aux_2[4j-2] = y_j$, $aux_2[4j-1] = y_j$, $aux_2[4j] = x_j$. The string aux_3 has the first half like aux_1 and the second half like aux_2. The string aux_4 has the first half like aux_2 and the second half like aux_1.

We prove that ϕ has a satisfying assignment if and only if $\max_{i=1}^{m} \Delta(s_i, q) \leq 2(n-3) + 8$ and $\max_{i=1}^{4} \Delta(aux_i, q) \leq 2(n-3) + 8$, where q is the rank distance center.

If ϕ has a satisfying truth assignment, then we construct the center q as follows:

– if x_a is false, then $q[4a-3] = x_a$, $q[4a-2] = y_a$, $q[4a-1] = x_a$, $q[4a] = y_a$.
– if x_a is true, then we set $q[4a-3] = y_a$, $q[4a-2] = x_a$, $q[4a-1] = y_a$, $q[4a] = x_a$.

Now we show that $\Delta(q, s_i) \leq 2(n-3) + 8$, $\forall 1 \leq i \leq m$. For a string s_i and a variable x which is not in the clause c_i, the four positions that correspond to the variable x add 2 to the total rank distance (irrespective of the assignment of x, i.e. $\Delta(yxyx, xyyx) = \Delta(xyxy, xyyx) = 2$). If x (or its negation) is in the clause c_i, then the four positions corresponding to x add 4 to the total rank distance if x does not satisfy c_i (i.e. if x is true and appears negated in c_i or viceversa) or 0 otherwise. Since every clause can be satisfied, at most two variables from each clause add 4 to the total rank distance. Therefore, for all $i \geq 1$, we have $\Delta(q, s_i) \leq 2(n-3) + 8$. The $\Delta(aux_i, q)$, for all i, is equal to $2n$ and, thus, the first implication is true.

We now prove the reverse implication. As all the characters corresponding to a variable x_i are placed on the positions from $4i-3$ to $4i$ in all the strings, we know that the center has to contain these characters on the same positions too. We claim that for each variable x, the corresponding characters are placed either in the order $xyxy$ or $yxyx$, and thus we can recover a truth assignment: if the order is $xyxy$, then we set x to false; otherwise we set x to true. As the distance is less or equal than $2n+2$, this is a satisfying truth assignment (otherwise, if a clause is not satisfied, we have a string s_i with $\Delta(s_i, q) = 2n+6$).

If the order of the characters corresponding to a variable x is not $xyxy$ or $yxyx$, then the distance from q to one of the strings is greater than $2n+2$. This is enforced by the strings aux which do not allow the placement of strings $xyyx$ or $yxxy$ (this structure keeps a small distance from s_i, but has a big distance from one of the aux strings). $\square$

As the closest substring problem is a generalization of the closest string, the hardness of the closest substring problem follows.

Corollary 1. *The closest substring problem via rank distance is NP-hard.*

4 A *k*-Approximation Algorithm

In this section we present a k-approximation algorithm for the CSRD problem. First, we show that CSRD is equivalent to the problem of finding a minimum weighted perfect matching in a complete bipartite graph and then we apply a result from [3]. Berstein and Onn [3] prove that there exists a k-approximation algorithm for the minimum (under any norm ℓ_p) weighted perfect matching, where k is the number of different weights associated to the edges.

The reduction from CSRD to a multi-weighted complete bipartite graph is as follows. Without loss of generality, let the alphabet be $\Sigma = \{1, 2, \ldots, n\}$, and let $U = \{p_1, p_2, \ldots, p_k\}$ be a set of k full rankings over the alphabet Σ. Any set of k strings where each symbol appears the same number of times in any string can be transformed in a set of k full rankings (see Remark 2).

The nodes on the left side of the graph correspond to the characters of the alphabet. The nodes in the right set correspond to the n available positions. The graph is complete, since in the center string each character can be put on every position. Observe that a full ranking p over Σ can be seen as a perfect matching M of $K_{n,n}$, where edge $e = (i, j) \in M$ if and only if $j = ord(p, i)$. Therefore, the set of all full rankings over Σ is equal with the set of all perfect matchings of $K_{n,n}$. To every edge (i, j) of $K_{n,n}$ we add k integer weight functions, $w^d(i, j)$, $1 \le d \le k$, such that the weight $w^d(i, j)$, is the absolute difference between j and the position of i in p_d. In other words, $w^d(i, j)$ shows how much the RD between the center and p_d increases if the center has the character i on the j'th position. Each perfect matching has associated an n dimensional array and, thus, the center string has associated the array with the smallest $|| \cdot ||_\infty$.

Formally, the algorithm is presented below.

```
Input: $U = \{p_1, p_2, \ldots, p_k\}$, a set of $k$ full rankings over the alphabet
       $\Sigma = \{1, 2, \ldots, n\}$.

 1. Build a multi-objective complete bipartite graph $K_{n,n} = (N, P, W)$ as
    follows:
    (a) The nodes $N = \{1, 2, \ldots, n\}$ on the left side of the graph correspond
        to the characters $\Sigma$.
    (b) The nodes $P = \{1, 2, \ldots, n\}$ in the right set correspond to the $n$
        available positions.
    (c) Let $E = \{(i, j) | 1 \leq i, j \leq n\}$ be the set of edges of the complete
        bipartite graph $K_{n,n}$;
    (d) To every edge $(i, j) \in E$ are associated $k$ integer weights:
        $w^1(i, j), w^2(i, j), \ldots, w^k(i, j)$, where $w^d(i, j) = |j - ord(p_d, i)|$,
        $d = 1, 2, \ldots, k$.
 2. Apply Theorem 4.1 [3] with multi-objective function $f = || \cdot ||_\infty$ to the
    graph $K_{n,n}$ constructed at the previous steps.
 3. Let $M \subset E$ be the minimum perfect matching of $K_{n,n}$ obtained from
    the previous step.
 4. For every edge $(i, j) \in M$, define $ord(x, i) = j$.
 5. Output: $x$.
```

Algorithm 1: A k-approximation algorithm for CSRD problem

Theorem 3. *The ranking x returned by Algorithm 1 is a $k-$approximation of CSRD problem.*

Proof. First, observe that the graph constructed at Step 1, is a *multiobjective bipartite graph*, with the set of nodes $\Sigma = \{1, 2, \ldots, n\}$, the set of edges $E = \{(i, j) | 1 \leq i, j \leq n\}$, and $k \geq 1$ integer weight functions defined on E, $w^1, w^2, \ldots, w^k$. We prove now the equivalence with CSRD, namely that, for each ranking p, there exists a perfect matching M_p.

The rank distance between p and another string p_d:

$$\Delta(p, p_d) = \sum_{i=1}^{n} |ord(p_d, i) - ord(p, i)|$$

Since $|ord(p_d, i) - ord(p, i)| = |ord(p_d, i) - j|$ we have:

$$\sum_{(i,j) \in M_p} w^d(i, j) = \Delta(p, p_d). \tag{5}$$

Therefore, for a certain a perfect matching $M \subset E$, the following sums give the rank distance from the ranking associated to M to each ranking p_i, for $i = 1, 2, \ldots, k$:

$$\langle \sum_{(i,j) \in M_p} w^1(i, j), \sum_{(i,j) \in M_p} w^2(i, j), \ldots, \sum_{(i,j) \in M_p} w^k(i, j) \rangle \tag{6}$$

We define the *cost* of M as the maximum of the k values obtained in (6):

$$cost(M) = \max_{d=1,k} \sum_{(i,j)\in M} w^d(i,j) \tag{7}$$

The definition of CSRD and the equations (5), (6) and (7) imply that the CSRD is equal to problem of minimizing (7):

$$CSRD = \min_{M\subset E} cost(M) = \min_{M\subset E} \max_{d=1,k} \sum_{(i,j)\in M} w^d(i,j) \tag{8}$$

We observe that the objective function *cost* is in fact the infinity norm, $f = || \cdot ||_\infty$, and the hypothesis of Theorem 4.1 [3] holds. Thus, the Step 2 of Algorithm 1 returns a k-approximation of the minimum nonlinear bipartite matching problem. Finally, following Step 4, we obtain a ranking which is a k-approximation to the CSRD problem. $\square$

Remark 2. If a string does not contain any identical symbols, then it can be transformed into a ranking (see Definition 4). Similarly, each ranking can be viewed as a string, over an alphabet identical to the universe of the ranked objects.

Algorithm 1, which solves the CSRD problem on rankings (i.e. strings without repetitions), can be used to solve the CSRD problem on arbitrary strings (with repetitions and the same composition): we transform the set of strings into a set of rankings, we compute its CSRD, and finally we retransform these newly obtained rankings into strings by deleting the indexes.

This operation can be done due to Theorem 1 (Pareto optimality). Indeed, if Theorem 1 would not be satisfied, then after the indexing of strings and application of Algorithm 1 we can obtain a perfect matching that looks like this: $a_2a_1b_2b_1...$ (i.e. the second a of a string is in front of the first a, which is impossible). However, Theorem 1 assures us that if we obtain a perfect matching in which the second a is in front of the first a, then we can swap the two characters, since the CSRD satisfies the Pareto optimality. Therefore, we can retransform the obtained ranking in a string.

Remark 3. The classical bipartite matching problem is the special case where $k = 1$ and f is the identity function.

Example 3. Let $\Sigma = \{1, 2, 3\}$ and let $U = \{(1, 2, 3), (2, 1, 3), (3, 1, 2)\}$ be three rankings.

The graph associated to U (see Figure 1) is $K_{3,3}$, and to every edge (i, j), $1 \leq i, j \leq 3$, we associate three weights denoted, $w(i, j) = \langle w^1(i, j), w^2(i, j), w^3(i, j)\rangle$. The associated weights of $K_{3,3}$ are:

$$w(1,1) = \langle 0,1,1\rangle,\ w(1,2) = \langle 1,0,0\rangle,\ w(1,3) = \langle 2,1,1\rangle;$$

$$w(2,1) = \langle 1,0,2\rangle,\ w(2,2) = \langle 0,1,1\rangle,\ w(2,3) = \langle 1,2,0\rangle;$$

$$w(3,1) = \langle 2,2,0 \rangle,\ w(3,2) = \langle 1,1,1 \rangle,\ w(3,3) = \langle 0,0,2 \rangle.$$

Let $M = \{(1,1),(2,2),(3,3)\}$ be a perfect matching of $K_{3,3}$ (not necessary of minimum cost). The ranking $p = (1,2,3)$, associated to M has the following rank distances to U:

$$\Delta(p,(1,2,3)) = 0,\ \Delta(p,(2,1,3)) = 2,\ \Delta(p,(3,1,2)) = 4.$$

Thus, the cost of M is 4. In other words, p is the center of the sphere with radius $r = \max_{x \in U}(\Delta(p,x))$ in which are included all rankings of U.

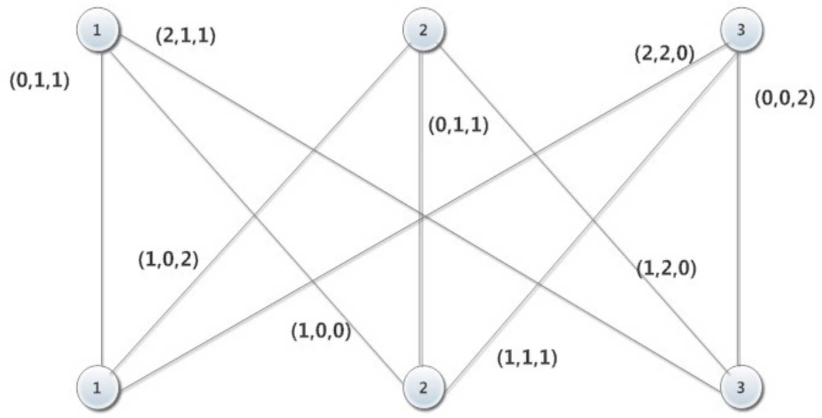


Fig. 1. The $K_{3,3}$ graph corresponding to the set of rankings $U = \{(1,2,3),(2,1,3),(3,1,2)\}$

5 A Polynomial Algorithm for Binary Alphabets

On full rankings (i.e. permutations), the RD (i.e. the footrule distance) is often compared to the Kendall-τ distance, which simply counts the minimum number of swaps between consecutive positions necessary to transform x into y.

In the next theorem we show that the RD between two binary strings is equal to the Kendall-τ distance.

Theorem 4. *Let u and v be two binary strings with the same number of 0's and 1's (i.e. $|u|_0 = |v|_0$, $|u|_1 = |v|_1$). The rank distance between u and v is equal to two times Kendal-τ distance between u and v.*

Proof. We prove by induction on the number of 1's that Kendall-τ between two strings is equal to total contribution of 1's in rank distance.

If $n = 1$, then τ is equal to the number of moves necessary to align the two strings, which, in turn, is equal to the absolute value of the difference between the positions of 1 in the two strings (i.e. rank distance).

Suppose the theorem is true for n 1's. We prove the theorem for $n+1$ 1's. Kendall-τ between two strings with $n+1$ 1's is equal to the absolute difference between the positions of the leftmost 1's in each string (as we have to align these two 1's) plus the Kendall-τ for the rest of the string. Thus, using the induction step, Kendall-τ is equal with the cost of 1's rank distance.

We can make a similar argument for the 0's.

Since rank distance between the two strings is equal to the cost of 0's added to the cost of 1's, we obtain that the rank distance is equal to two times Kendal-τ between u and v. □

Corollary 2. *The CSRD problem for k binary strings, $k \geq 2$, with the same number of* 0*'s and* 1*'s, is equivalent with the center problem via Kendal-τ distance.*

Proof. The proof follows immediately from the following observation. The CSRD problem for k binary strings $p_1, \dots, p_k$ is equivalent to

$$\min_{x \in \chi_n} \max_{i=1..k} \Delta(x, p_i),$$

and, using Theorem 4 we can substitute Δ with τ. □

In the general case, the closest string problem via Kendall-τ distance is an NP-hard problem [18]. However, there exists an algorithm for the center string problem via Kendall tau distance which has running time exponential in the number of input strings [19].

6 Conclusions

In this paper we study the closest string and the closest substring problems under a new distance measure, named the rank distance, recently introduced by Dinu[7]. We show that the CSP and CSSP are NP-hard via rank distance. Then, we present a k-approximation algorithm for the problem. Finally, we show a parametrized algorithm if the alphabet is binary and each string has the same number of 0's and 1's.

A natural open question is to derive approximation algorithms with better bounds or to prove hardness of approximation results.

Acknowledgements. The authors would like to thank the anonymous reviewers for their helpful comments. All authors contributed equally to this work. The research of Liviu P. Dinu was supported by the CNCS, IDEI - PCE project 311/2011.

References

1. Arrow, K.J.: Social Choice and Indivudual Values. Wiley, New York (1963)
2. Ben-Dor, A., Lancia, G., Perone, J., Ravi, R.: Banishing Bias from Consensus Sequences. In: Hein, J., Apostolico, A. (eds.) CPM 1997. LNCS, vol. 1264, pp. 247–261. Springer, Heidelberg (1997)
3. Berstein, Y., Onn, S.: Nonlinear bipartite matching. Disc. Optim. 5(1), 53–65 (2008)
4. de la Higuera, C., Casacuberta, F.: Topology of Strings: Median String is NP-Complete. Theor. Comput. Sci. 230(1-2), 39–48 (2000)
5. Deng, X., Li, G., Li, Z., Ma, B., Wang, L.: Genetic design of drugs without side-effects. SIAM J. Comput. 32(4), 1073–1090 (2003)
6. Diaconis, P., Graham, R.L.: Spearman's footrule as a measure of disarray. J. Royal Statist. Soc. Series B (Methodological) 39(2), 262–268 (1977)
7. Dinu, L.P.: On the classification and aggregation of hierarchies with different constitutive elements. Fundam. Inform. 55(1), 39–50 (2003)
8. Dinu, L.P., Manea, F.: An efficient approach for the rank aggregation problem. Theor. Comput. Sci. 359(1-3), 455–461 (2006)
9. Dinu, L.P., Sgarro, A.: A low-complexity distance for dna strings. Fundam. Inform. 73(3), 361–372 (2006)
10. Frances, M., Litman, A.: On covering problems of codes. Theory Comput. Syst. 30(2), 113–119 (1997)
11. Gramm, J., Huffner, F., Niedermeier, R.: Closest strings, primer design, and motif search. currents in computational molecular biology. In: RECOMB, pp. 74–75 (2002)
12. Lanctot, J.K., Li, M., Ma, B., Wang, S., Zhang, L.: Distinguishing string selection problems. Inf. Comput. 185(1), 41–55 (2003)
13. Li, M., Ma, B., Wang, L.: Finding similar regions in many sequences. J. Comput. Syst. Sci. 65(1), 73–96 (2002)
14. Liu, X., He, H., Sýkora, O.: Parallel Genetic Algorithm and Parallel Simulated Annealing Algorithm for the Closest String Problem. In: Li, X., Wang, S., Dong, Z.Y. (eds.) ADMA 2005. LNCS (LNAI), vol. 3584, pp. 591–597. Springer, Heidelberg (2005)
15. Nicolas, F., Rivals, E.: Complexities of the Centre and Median String Problems. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 315–327. Springer, Heidelberg (2003)
16. Nicolas, F., Rivals, E.: Hardness results for the center and median string problems under the weighted and unweighted edit distances. J. Disc. Alg. 3(2-4), 390–415 (2005)
17. Palmer, J., Herbon, L.: Plant mitochondrial dna evolves rapidly in structure, but slowly in sequence. J. Mol. Evol. 28, 87–89 (1988)
18. Popov, V.Y.: Multiple genome rearrangement by swaps and by element duplications. Theor. Comput. Sci. 385(1-3), 115–126 (2007)
19. Schwarz, N.: Rank aggregation by criteria. Minimizing the maximum Kendall-tau distance. Diplomarbeit, Jena (2009)
20. Wang, L., Dong, L.: Randomized algorithms for motif detection. J. Bioinf. and Comp. Biol. 3(5), 1039–1052 (2005)
21. Wooley, J.C.: Trends in computational biology: A summary based on a recomb plenary lecture. J. Comp. Biol. 6(3/4) (1999)

On Approximating String Selection Problems with Outliers

Christina Boucher[1], Gad M. Landau[2,3], Avivit Levy[4,5], David Pritchard[6], and Oren Weimann[2]

[1] Department of Computer Science, University of California, San Diego, USA
cboucher@eng.ucsd.edu
[2] Department of Computer Science, University of Haifa, Haifa 31905, Israel
landau@cs.haifa.ac.il
[3] Polytechnic Institute of NYU, Brooklyn, NY 11201-3840, USA
[4] Shenkar College for Engineering and Design, Ramat-Gan, Israel
[5] CRI, University of Haifa, Mount Carmel, Haifa 31905, Israel
[6] CEMC, University of Waterloo, Canada

Abstract. Many problems in bioinformatics are about finding strings that approximately represent a collection of given strings. We look at more general problems where some input strings can be classified as outliers. The *Close to Most Strings* problem is, given a set S of same-length strings, and a parameter d, find a string x that maximizes the number of "non-outliers" within Hamming distance d of x. We prove that this problem has no polynomial-time approximation scheme (PTAS) unless NP has randomized polynomial-time algorithms, correcting a decade-old mistake. The *Most Strings with Few Bad Columns* problem is to find a maximum-size subset of input strings so that the number of non-identical positions is at most k; we show it has no PTAS unless $\mathsf{P} = \mathsf{NP}$. We also observe *Closest to k Strings* has no efficient PTAS (EPTAS) unless the parameterized complexity hierarchy collapses. In sum, outliers help model problems associated with using biological data, but we show the problem of finding an approximate solution is computationally difficult.

1 Introduction

With the development of high-throughput next generation sequencing technologies, there has arisen large amounts of genomic data, and an increased need for novel ways to analyze this data. This has inspired numerous formulations of biological tasks as computational problems. In light of this observation, Lanctot et al. [18] initiated the study of *distinguishing string selection problems*, where we seek a representative string satisfying some distance constraints from each of the input strings. We will mostly have constraints in the form of an upper bound on the Hamming distance, but lower bounds on the Hamming distance, and substring distances, have also been considered [8, 14, 18].

Typically, the distance constraint must be satisfied for each of the input strings. However, biological sequence data is subject to frequent random mutations and errors, particularly in specific segments of the data; requiring that the solution fits the entire input data is problematic for many problems in bioinformatics. It would be preferable to find the similarity of a portion of the input strings, excluding a few bad reads that have been corrupted, rather than trying to fit the complete set of input and in doing so finding one that is distant from many or all of the strings.

What if we are given a measure of goodness (e.g., distance) the representative must satisfy, and want to choose the largest *subset* of strings with such a representative? Conversely, what if we specify the subset size and seek a representative that is as good as possible? Some results are known in this area with respect to fixed-parameter tractability [6]. Here, we prove results about the approximability of three *string selection problems with outliers*. For any two strings x and y of same length, we denote the Hamming distance between them as $d(x, y)$, which is defined as the number of mismatched positions. Our main results are about the three following NP optimization problems.

Definition 1. Close to Most Strings *(a.k.a.* Max Close String *[18, 24])*
Input: n strings $S = \{s_1, \ldots, s_n\}$ of length ℓ over an alphabet Σ, and $d \in \mathbf{Z}_+$.
Solution: a string s of length ℓ.
Objective: maximize the number of strings s_i in S that satisfy $d(s, s_i) \le d$.

Definition 2. Closest to k Strings
Input: n strings $S = \{s_1, \ldots, s_n\}$ of length ℓ over an alphabet Σ, and $k \in \mathbf{Z}_+$.
Solution: a string s of length ℓ and a subset S^* of S of size k.
Objective: minimize $\max\{d(s, s_i) \mid s_i \in S^*\}$.

In the special case $k = n$, *Closest to k Strings* becomes *Closest String* — an NP-hard problem [11] that has received significant interest in parameterized complexity and approximability [1, 2, 13, 18, 20, 26, 27].

We also consider a new problem where the "outliers" are considered to be positions ("columns") rather than strings ("rows"). Let $s(j)$ indicate the jth character of string s.

Definition 3. Most Strings with Few Bad Columns
Input: n strings $S = \{s_1, \ldots, s_n\}$ of length ℓ over an alphabet Σ, and $k \in \mathbf{Z}_+$.
Solution: a subset $S^* \subseteq S$ of strings such that the number $\{t \in [\ell] \mid \exists s_i^*, s_j^* \in S^* : s_i^*(t) \neq s_j^*(t)\}$ of bad columns is at most k.
Objective: maximize $|S^*|$.

A column t is *bad* when its entries are not-all-equal, among strings in S^*. As an example application, suppose we have a collection of DNA sequences from a heterogeneous population of two sub-groups: (1) a large collection of sequences that are identical except for k positions where mutations can occur, and (2) additional outliers. Then Most Strings with Few Bad Columns models the problem of separating the two groups. This problem also generalizes the problem of finding tandem repeats in a string [19].

1.1 Our Contributions

A *polynomial-time approximation scheme* (PTAS) for an optimization problem is an algorithm that takes an instance of the problem and a parameter $\epsilon > 0$ and, in time that is polynomial for any fixed ϵ, produces a solution that is within a factor $1 + \epsilon$ of being optimal. An *efficient PTAS* (EPTAS) further restricts the running time to be some function of ϵ times a constant-degree polynomial in the input size. A decision problem lies in ZPP if it has a randomized algorithm that is always correct, and whose expected running time is polynomial. A well-known equivalent characterization of ZPP is, for any fixed $0 < p < 1$, that the algorithm *always* runs in polynomial time, outputs either the correct answer or no answer, and gives the correct answer with probability at least p for every input. We defer a brief description of the parameterized complexity classes $\mathsf{W}[1]$ and FPT to Section 1.2.

We present several results on the computational hardness of efficiently finding an approximate solution to the above optimization problems. Specifically, we show the following:

- The Close to Most Strings Problem has no PTAS, unless $\mathsf{ZPP} = \mathsf{NP}$ (Theorem 1).
- The Most Strings with Few Bad Columns Problem has no PTAS, unless $\mathsf{P} = \mathsf{NP}$ (Theorem 2).
- We observe that the known PTAS [24] for the Closest to k Strings Problem cannot be improved to an EPTAS, unless $\mathsf{W}[1] = \mathsf{FPT}$.

Our first result corrects an error in prior literature. A problem is APX-*hard* if for some fixed $\epsilon > 0$, finding a $(1+\epsilon)$-approximation is NP-hard. A 2000 paper of Ma [24] claims that the Close to Most Strings problem is APX-hard; however, the reduction is erroneous. To explain, it is helpful to define one more problem, *Far from Most Strings*, which is the same as Close to Most Strings except that we want to maximize the number of strings s_i in S that satisfy $d(s, s_i) \geq d$ (rather than $\leq$). There is considerable experimental interest in heuristics for Far from Most Strings, mostly based on local search [23, 9, 10]. Far From Most Strings was introduced and studied by Lanctot et al. [18], and they (correctly) showed that for any fixed alphabet size greater than or equal to three, Far from Most Strings is at least as hard to approximate as Independent Set. Currently, Independent Set is known [17] to be inapproximable within a factor of $n/2^{\log^{3/4+\epsilon} n}$ unless $\mathsf{NP} \subset \mathsf{BPTIME}(2^{\log^{O(1)} n})$.

The main idea in Ma's approach was to consider a binary alphabet. In detail, the Far from Most Strings and Close to Most Strings Problem on alphabets $\Sigma = \{0, 1\}$ are basically the same problem, since a string s of length ℓ has $d(s_i, s) \leq d$ if and only if the 0-1 complement $\overline{s}$ of s satisfies $d(s_i, \overline{s}) \geq \ell - d$. The crucial error in [24] is that Ma mis-cited [18], assuming that their result held for binary alphabets. (One reason why the approach of [18] does not extend to binary alphabets in any obvious way is that the instances produced by their reduction satisfy $d = \ell$, whereas Far from Most Strings is easy to solve when $|\Sigma| = 2$ and $d = \ell$.)

From [18] and [24] we cannot conclude anything about the hardness of Close to Most Strings, nor can we say anything about the hardness of Far from Most Strings when $|\Sigma| = 2$. Our results close both of these gaps: the proof of Theorem 1 actually shows Close to Most Strings is hard over a binary alphabet, from which it follows that Far from Most Strings is, too. At the same time, the hardness that we are able to achieve is much more modest than the previous claim; we show only that there is no 1.001-approximation. We also require a randomized reduction. It is a very interesting open problem to determine whether this problem has any constant-factor approximation, even over a binary alphabet.

1.2 Brief Description of Parameterized Complexity

Some parameterized complexity concepts will arise in later sections, so we give a birds-eye view of this area. With respect to a parameter k, a decision algorithm with running time $f(k)n^{O(1)}$ (where n is the input length) is called *fixed parameter tractable* (FPT); the class FPT contains all parameterized problems with FPT algorithms. The corresponding reduction notion between two parameterized problems is an *FPT reduction*, which is FPT and also increases the parameter by some function that is independent of the instance size. The class $\mathsf{W}[1]$ is a superset of FPT closed under FPT-reductions. A problem is $\mathsf{W}[1]$-*hard* if any $\mathsf{W}[1]$ problem can be FPT-reduced to it, and $\mathsf{W}[1]$-*complete* if it is both in $\mathsf{W}[1]$ and $\mathsf{W}[1]$-hard. There are many natural $\mathsf{W}[1]$-complete problems, like Maximum Clique parameterized by clique size. It is widely hypothesized that $\mathsf{FPT} \subsetneq \mathsf{W}[1]$, but unproven, analogous to $\mathsf{P} \subsetneq \mathsf{NP}$.

1.3 Previous Work

The Closest String Problem can be viewed as a special case of the Close to Most Strings Problem where the number of outliers is equal to zero. This problem has been throughly studied and therefore, there exists numerous results concerning its complexity and approximability [2, 7, 13, 18, 20, 21, 24, 26–28]. It was shown NP-complete, even under the restriction that the alphabet is binary, in [11]. Lanctot et al. [18] gave a polynomial-time algorithm for the Closest String Problem that achieves a $\frac{4}{3} + o(1)$ approximation guarantee. Independently, Gąsieniec et al. [12] gave a $\frac{4}{3}$-approximation algorithm that uses a similar technique, which is based on a linear programming relaxation of an integer programming model of the problem. Using randomized rounding, Li et al. [20] proved the existence of a PTAS for this problem. The running time of the PTAS has since been improved by Andoni et al. [2], and Ma and Sun [26]. Currently, the PTAS with the best known running time is due to Ma and Sun, which runs in $O(n^{\Theta(\epsilon^{-2})})$-time.

Gramm et al. [13] demonstrated that the Closest String Problem is in FPT when parameterized by n, and when parameterized by d. Ma and Sun gave an $O(n|\Sigma|^{O(d)})$-time algorithm, which is a polynomial-time algorithm when $d = O(\log n)$ and Σ has constant size [26]; Chen et al. [7] and Zhao and Zhang [28] have improved upon the running time of this result. In [6], parameterized versions of Closest to k Strings (under the name Closest String With Outliers) were considered; it was shown that the problem is in FPT when parameterized by d and $n-k$, and when parameterized by $|\Sigma|$ and n, but W[1]-hard with respect to any combination of the parameters $\{\ell, d, k\}$, and any combination of the parameters $\{n-k, |\Sigma|\}$.

2 Approximation Hardness of Close to Most Strings

Theorem 1. *For some $\epsilon > 0$, if there is a polynomial-time $(1+\epsilon)$-approximation algorithm for the Close to Most Strings Problem, then* ZPP = NP.

Proof. We use a reduction from the *Max-2-SAT Problem*, which is to determine for a given 2-CNF formula, an assignment that satisfies the maximum number of clauses. Let $X = \{x_1, \ldots, x_n\}$ be a set of Boolean variables. In 2-CNF, each clause is a disjunction of two literals, each of which is either x_i or $\overline{x_i}$ for some i. Håstad [15] showed it is NP-hard to compute a 22/21-approximately optimal solution to Max-2-SAT, and this is the starting point for our proof. We will assume that $m \geq n$, i.e. the number of clauses is greater than or equal to the number of variables, which is without loss of generality since otherwise some variable appears in at most one clause and the instance can be reduced.

Input Instance		Output Instance	
$x_1 \vee x_2$			11 11 01 01
$\overline{x_1} \vee x_2$			00 11 01 01
$x_1 \vee \overline{x_3}$		$\ell = 2n$	11 01 00 01
$\overline{x_1} \vee \overline{x_3}$	$\longrightarrow$	$d = n$	00 01 00 01
$\overline{x_2} \vee x_3$			01 00 11 01
$x_3 \vee x_4$			01 01 11 11
$n = 4$ variables, $m = 6$ clauses		cm fixing strings, i.i.d. uniform from $\{01, 10\}^n$	$\left\{ \begin{array}{l} 01\ 10\ 01\ 01 \\ \vdots \end{array} \right.$
Input Solution		Output Solution	
value 5, $x = (\mathsf{true}, \mathsf{true}, \mathsf{false}, \mathsf{true})$	$\longleftrightarrow$	value $5 + cm$, $\widehat{x} = 11\ 11\ 00\ 11$	

Fig. 1. Overview of the reduction used to prove Theorem 1

We give a schematic overview of our reduction in Figure 1. The reduction will be randomized. It takes as input an instance of Max-2-SAT with m clauses and n variables. The reduction's output is an instance of Close to Most Strings with $cm + m$ strings of length $2n$ for some constant c, and the distance parameter of the instance is $d = n$. Of these strings, cm will be "fixing" strings to enforce a certain structure in near-optimal solutions, and the remaining m strings are defined from the clauses as follows. Given a 2-clause ω_j over the variables in X, we define the corresponding string $s_j = s_j(1) \dots s_j(2n)$ as follows:

$$s_j(2i-1)s_j(2i) = \begin{cases} 00 & \text{if } \omega_j \text{ contains the literal } \overline{x_i}, \\ 11 & \text{if } \omega_j \text{ contains the literal } x_i, \\ 01 & \text{otherwise.} \end{cases}$$

The fixing strings will all be elements of $\{01, 10\}^n$, selected independently and uniformly at random.

We now give a high-level explanation of the proof. For every variable assignment vector x define a string $\widehat{x}$ via

$$\widehat{x}(2i-1)\widehat{x}(2i) = \begin{cases} 11 & \text{if } x_i \text{ is true,} \\ 00 & \text{if } x_i \text{ is false.} \end{cases}$$

Notice that $\widehat{x}$ is at distance exactly $d = n$ from all of the fixing strings, and that $d(\widehat{x}, s_j) \leq n$ if and only if x satisfies clause ω_j. Hence, if x satisfies k clauses, the string $\widehat{x}$ is within distance d of $cm + k$ out of the $cm + m$ total strings. We will show conversely that with high probability, for all strings s within distance d of cm of the strings, we have $s \in \{00, 11\}^n$. Using this crucial structural claim, it follows that any sufficiently good approximation algorithm for Close to Most Strings must output s such that $s = \widehat{x}$ for some x. Then the claim will be complete via standard calculations.

Here is the precise statement of the structural property.

Lemma 1. *Fix $c \geq 20$. Let F be a set of cm strings selected uniformly and independently at random from $\{01, 10\}^n$ (with replacement), with $m \geq n$. Then with probability at least $1 - 0.9^n$, every string $s \in \{0,1\}^{2n} \setminus \{00, 11\}^n$ has distance greater than n from at least m strings in F.*

Proof. To explain the proof more simply, fix s and consider a particular $f \in F$. Our first claim is the following:

Claim 1. *For any $s \in \{0,1\}^{2n} \setminus \{00, 11\}^n$, if f is selected uniformly at random from $\{01, 10\}^n$, then* $\Pr[d(s, f) \geq n + 1] \geq 1/4$.

Proof. By hypothesis, for some i this s satisfies $s(2i-1) \neq s(2i)$, say $s(2i-1) = 0$ and $s(2i) = 1$ (the other case is symmetric). Let $\mathcal{E}$ denote the event $[f(2i-1) \neq s(2i-1)$ and $f(2i) \neq s(2i)]$. Since f is chosen uniformly at random from $\{01, 10\}^n$, we have $\Pr[\mathcal{E}] = 1/2$.

Next we show that $\Pr[d(s, f) \geq n + 1 \mid \mathcal{E}] \geq 1/2$. Observe that $d(s, f)$ is a sum of n independent random variables $d(s(2j-1)s(2j), f(2j-1)f(2j))$ for j from 1 to n; conditioning on $\mathcal{E}$ just fixes one of these variables at 2. The remaining ones are either always 1 (if $s(2j-1) = s(2j)$), or a uniformly random element of $\{0, 2\}$. The conditioned random variable $d(s, f) \mid \mathcal{E}$ is thus a shifted and scaled binomial distribution, in particular it is symmetric about $n + 1$. So

$\Pr[d(s,f) \geq n+1 \mid \mathcal{E}] = \Pr[d(s,f) \leq n+1 \mid \mathcal{E}]$ and since these two probabilities' sum is at least 1, $\Pr[d(s,f) \geq n+1 \mid \mathcal{E}] \geq 1/2$ follows.

Finally, unconditioning, $\Pr[d(s,f) \geq n+1] = \Pr[d(s,f) \geq n+1 \mid \mathcal{E}] \cdot \Pr[\mathcal{E}] \geq 1/2 \cdot 1/2 = 1/4$. □

Continuing with the proof of Lemma 1, we next use a Chernoff bound to reason about how a single s interacts with the entire collection F, and then will use a union bound to cover all possible s. Let $F = \{f_1, \ldots, f_{cm}\}$ and let X_i be an indicator variable for the event that $d(f_i, s) > n$. We have argued that each X_i is 1 with probability at least 1/4. Therefore, $E[\sum_i X_i] \geq cm/4$. We will use a Chernoff bound of the following form:

Claim 2 (Lower Chernoff bound)**.** *For any $\delta > 0$, if X is a sum of independent random variables that each only take on the values 0 and 1, then*

$$\Pr[X < (1-\delta)E[X]] < \exp(-E[X]\delta^2/2).$$

For a proof of this standard result, see a book such as [25]. We will apply it to $X = E[\sum_i X_i]$ and to δ chosen so that $(1-\delta)cm/4 = m$, i.e. $\delta = 1 - 4/c$. Then the Chernoff bound implies

$$\begin{aligned}\Pr[X < m] \leq \Pr[X < (1-\delta)E[X]] &< \exp(-E[X]\delta^2/2)\\ &\leq \exp(-cm/4 \cdot (1-4/c)^2/2) = \exp(\frac{-(c-4)^2}{8c}m).\end{aligned}$$

This shows that every s is very unlikely to falsify Lemma 1. We may now take a union bound over all $4^n - 2^n$ possible choices of s: the probability that a random choice of F admits *any* bad s is at most

$$(4^n - 2^n)\exp\Big(\frac{-(c-4)^2}{8c}m\Big) < 4^n \exp\Big(\frac{-(c-4)^2}{8c}n\Big) = \exp\Big(\big(\ln 4 - \frac{(c-4)^2}{8c}\big)n\Big),$$

where we used $m \geq n$ in the first inequality. Any large enough c makes this probability exponentially decreasing in n; it is straightforward to calculate that when $c = 20$ this is at most 0.9^n, as needed. □

Now that the proof of Lemma 1 is complete, we proceed with the proof of Theorem 1. Fix $c = 20$. Given a Max-2-SAT instance, we run the randomized reduction above to get an instance of Close to Most Strings. Let s_A be a $(1+\epsilon)$-approximation for this instance, where ϵ will be a small constant fixed later to satisfy two properties.

Let k^* be the maximum number of satisfiable clauses in the Max-2-SAT instance. As an important technicality, note that k^* is lower-bounded by $m/2$, since the expected number of clauses satisfied by a random assignment is at least $m/2$, by linearity of expectation. So the optimal solution to the Close to Most Strings instance has value at least $cm + m/2$.

First we want to use the structural lemma (Lemma 1). Assume for now the bad event with probability 0.9^n does not happen; so every $s \notin \{00, 11\}^n$ (i.e. not of the form $s = \widehat{x}$) is within distance d of at most cm of the $(c+1)m$ strings. Thus provided that ϵ is small enough to satisfy $1 + \epsilon < \frac{cm+m/2}{cm} = 1 + \frac{1}{2c}$, then s_A is of the form $\widehat{x}_A$ for some x_A.

Next we finish the typical calculations in a proof of APX-hardness. We know that s_A is within distance d of at least $(cm + k^*)/(1 + \epsilon)$ strings. If we can pick ϵ so that

$$\frac{cm + k^*}{1 + \epsilon} > cm + \frac{21}{22}k^* \tag{1}$$

then $\widehat{x}_A$ satisfies more than $\frac{21}{22}k^*$ clauses, which is NP-hard by Håstad's result. Using that $k^* \geq m/2$, it is easy to verify that (1) holds for all $\epsilon < 1/(21 + 44c)$.

Finally, we confirm that the randomized algorithm for Max-2-SAT coming from the reduction is ZPP-style. When the output s_A of the Close to Most Strings approximation algorithm satisfies $s_A \notin \{00, 11\}^n$ we output nothing. When $s_A \in \{00, 11\}^n$ we know for certain that $\widehat{x}_A$ is a 22/21-approximate solution for Max-2-SAT, as needed. □

3 Non-existence of an EPTAS for Closest to k Strings

Ma showed in [24] that the Closest to k Strings problem has a PTAS, which contrasts with the APX-hardness we obtain for the other problems in this paper. A natural question that comes up after a PTAS is obtained, is whether the running time can be improved to an EPTAS, or even further to a FPTAS (running time polynomial in the input length and ϵ^{-1}). We observe there does not exist an EPTAS for Closest to k Strings when the alphabet is unbounded, unless $\mathsf{W}[1] = \mathsf{FPT}$. To see this, we use a well-known fact relating fixed-parameter algorithms to the notion of an EPTAS, e.g. see [22], along with the fact that the decision version of Closest to k Strings is W[1]-hard when parameterized by d [6].

In detail, suppose for the sake of contradiction that we had an EPTAS for Closest to k Strings, i.e. that one could obtain a $(1 + \epsilon)$-approximation in time $f(\epsilon)s^{O(1)}$ where s is the input size. It is enough to prove that there is an FPT algorithm for the decision version of Closest to k Strings, with parameter d. Given an instance of this parameterized problem we need only call the EPTAS with any ϵ less than $(d + 1)/d$; notice the resulting algorithm takes FPT time with respect to d. To analyze this, let d_{ALG} be the distance value of the solution produced by the EPTAS algorithm, and d_{OPT} be the optimal distance value. If $d_{OPT} \leq d$, since $d_{OPT} \leq d_{ALG} \leq (1 + \epsilon)d_{OPT}$ and $d_{OPT}, d_{ALG} \in \mathbf{Z}$, we have $d_{OPT} = d_{ALG} \leq d$. Otherwise, $d_{ALG} \geq d_{OPT} > d$. So, we get an FPT algorithm just by comparing d_{ALG} to d.

Observation 1. *Closest to k Strings has no EPTAS unless* $\mathsf{W}[1] = \mathsf{FPT}$.

4 APX-Hardness of Most Strings with Few Bad Columns

In this section, we prove that the Most Strings with Few Bad Columns Problem is APX-hard, even in binary alphabets. To do this we reduce from the *Densest-k-Subgraph Problem*: given a graph $G = (V, E)$ and a parameter k, find a subset

$U \subseteq V$ with $|U| = k$ such that $|E[U]|$ is maximized — here $E[U]$ denotes the *induced edges* for U, meaning the set of all edges with both endpoints in U.

Our reduction will be approximation-preserving up to an additive $+1$ term. Given an instance $(G = (V, E), k)$ of Densest-k-Subgraph, we will generate an instance of Most Strings with Few Bad Columns with $|E| + 1$ strings, each of length $|V|$, and with the same values for the two parameters k (size of subgraph, maximum number of bad columns).

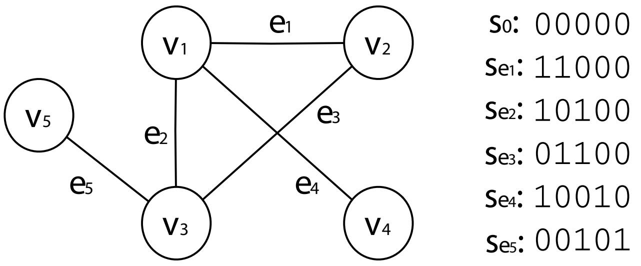


Fig. 2. Example of the reduction from an instance of Densest-k-Subgraph with G and $k = 3$ to an instance of Most Strings with Few Bad Columns with 6 strings of length 5

Let us define the set S of strings generated by the reduction; to do this, index $V = \{v_1, v_2, \dots\}$. For each edge $e = v_i v_j \in E$, let that edge's *0-1 incidence vector* $\chi(e)$ be the 0-1 string with 1s in positions i and j and 0 elsewhere; we put $\chi(e)$ into S. Finally, we put one more string into S, namely the all-zero string $\mathbf{0}$. This completes the description of the reduction; note it only takes polynomial time. See Figure 2 for an illustration of this reduction. We will use the following lemma in the proof of Claim 3.

Lemma 2. *Let $T \subseteq S$ be a subset of strings with at most k bad columns. Then there is a subset T' of S with at most k bad columns, $|T'| \geq |T|$, and $\mathbf{0} \in T'$.*

Proof of Lemma 2. Assume that $\mathbf{0} \notin T$, otherwise the lemma trivially follows. Also, assume $W = T \cup \{\mathbf{0}\}$ has more than k bad columns, otherwise we can take $T' = W$. Thus there must be a column that is not bad for T but that becomes bad when adding $\mathbf{0}$. I.e. T has a column that is entirely 1s. It follows that, viewed in the original graph setting, there exists a vertex v that is an end-point of all the edges corresponding to T. Pick any such edge arbitrarily, i.e. suppose $s = \chi(vw) \in T$. Since the input graph is simple, in column w, all entries of T are 0 except for $\chi(vw)$. Hence, $T' = T \setminus s \cup \{\mathbf{0}\}$ satisfies the lemma: compared with T it is bad in column v but not bad in column w. $\square$

Claim 3. *Let α be the optimal value for the Densest-k-Subgraph instance. Then the optimal value β for the new Most Strings with Few Bad Columns instance is $\beta = \alpha + 1$.*

Proof. First we show the easy direction, that $\beta \geq \alpha + 1$. Consider the optimal U for Densest-k-Subgraph, so that $|E[U]| = \alpha$ and $|U| = k$. Define a subset $\overline{T}$ of S by $\overline{T} = \{\mathbf{0}\} \cup \{\chi(e) \mid e \in E[U]\}$. Then the strings in $\overline{T}$ are all zero on any index corresponding to a node outside of U; the only bad columns are those corresponding to nodes in U, of which there are only k. So $\beta \geq |\overline{T}| = \alpha + 1$.

For the reverse direction, take a subset T of β strings that have at most k bad columns. We can assume without loss of generality that the string $\mathbf{0}$ is in T, as shown by Lemma 2. Using Lemma 2, we simply reverse the above reduction to show $\alpha \geq \beta - 1$. Take an optimal set S^* of strings with $|S^*| = \beta$ and such that S^* has at most k bad columns. By Lemma 2 we may assume $\mathbf{0} \in S^*$ — this implies that the set J of all non-bad columns for S^* satisfies $s(j) = 0$ for all $s \in S^*, j \in J$. Thus, each $\chi(uv) \in S^* \setminus \{\mathbf{0}\}$ has both of its 1s appearing at positions in $[\ell] \setminus J$, or equivalently each such uv is an element of $E[V \setminus J]$. So $V \setminus J$ is the required solution for Densest-k-Subgraph, and it has at least $\beta - 1$ induced edges. This ends the proof of Claim 3. □

This reduction yields our result:

Theorem 2. *The Most Strings with Few Bad Columns Problem is* NP*-hard, and* APX*-hard.*

Proof. Khot [16] showed that the Densest-k-Subgraph Problem is APX-hard. We need only to argue that our reduction can transform a PTAS for Most Strings with Few Bad Columns into a PTAS for Densest-k-Subgraph. Indeed, if we had a $(1+\delta)$-approximation algorithm for Most Strings with Few Bad Columns, then we get an algorithm for Densest-k-Subgraph that always returns a solution of value at least

$$(OPT+1)/(1+\delta)-1 = (OPT-\delta)/(1+\delta) \geq OPT(1-\delta)/(1+\delta) = OPT/(1+O(\delta))$$

where we used $OPT \geq 1$ in the middle inequality. □

While we ruled out a PTAS, it would also be out of the reach of current technology to obtain a constant or polylogarithmic factor for Most Strings with Few Bad Columns, because the best known approximation factor for the Densest-k-Subgraph Problem is $O(|V|^{1/4+\epsilon})$ [4].

5 Conclusions and Open Problems

Our results demonstrate that while outliers help model the problems associated with using biological data, such problems are computationally intractable to approximate. Here are the main open problems related to our results:

- Is there a constant-factor approximation for Close to Most Strings (even over a binary alphabet)?
- Is there a constant-factor approximation for Most Strings with Few Bad Columns (even over a binary alphabet)?

– Does Closest to k Strings have an EPTAS when the alphabet is binary? The reduction used in Section 3 needs an arbitrarily large alphabet. A more important problem is, does there exist an EPTAS for the Closest String Problem? Since the Closest String Problem is FPT with respect to d [13], the standard technique used in Section 3 cannot be used naively. The method presented by Boucher et al. [5] for proving the non-existence of an EPTAS may be applicable in this context.

Acknowledgments. The authors would like to thank Dr. Bin Ma for mentioning the error in his inapproximability proof and encouraging us to work on a correction. We thank Dr. Daniel Lokshtanov, Christine Lo, and the referees for their insights and comments. This work was supported by Natural Sciences and Engineering Research Council of Canada Post Doctoral Fellowship program, the Gerald Schwartz and Heather Reisman Foundation, the Israel Science Foundation grant (347/09), the National Science Foundation Award (0904246), and Grant Number 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG.

References

1. Amir, A., Paryenty, H., Roditty, L.: Approximations and Partial Solutions for the Consensus Sequence Problem. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 168–173. Springer, Heidelberg (2011)
2. Andoni, A., Indyk, P., Patrascu, M.: On the optimality of the dimensionality reduction method. In: Proc. of the 47th FOCS, pp. 449–456 (2006)
3. Arora, S.: Polynomial Time Approximation Schemes for Euclidean Travelling Salesman and other Geometric Problems. J. ACM 45(5), 753–782 (1998)
4. Bhaskara, A., Charikar, M., Chlamtac, E., Feige, U., Vijayaraghavan, A.: Detecting high log-densities: an $O(n^{1/4})$ approximation for densest k-subgraph. In: Proc. of the 42nd STOC, pp. 201–210 (2010)
5. Boucher, C., Lo, C., Lokshantov, D.: Outlier Detection for DNA Fragment Assembly. arXiv:1111.0376
6. Boucher, C., Ma, B.: Closest String with Outliers. BMC Bioinformatics 12(suppl.1), S55 (2011)
7. Chen, Z.-Z., Ma, B., Wang, L.: A Three-String Approach to the Closest String Problem. In: Thai, M.T., Sahni, S. (eds.) COCOON 2010. LNCS, vol. 6196, pp. 449–458. Springer, Heidelberg (2010)
8. Deng, X., Li, G., Li, Z., Ma, B., Wang, L.: Genetic design of drugs without side-effects. SIAM Journal on Computing 32(4), 1073–1090 (2003)
9. Festa, P.: On some optimization problems in molecular biology. Mathematical Biosciences 207(2), 219–234 (2007)
10. Festa, P., Pardalos, P.: Efficient solutions for the far from most string problem. Annals of Operations Research (December 2011) (published Online First)
11. Frances, M., Litman, A.: On covering problems of codes. Theoretical Computer Science 30(2), 113–119 (1997)
12. Gąsieniec, L., Jansson, J., Lingas, A.: Efficient approximation algorithms for the Hamming center problem. In: Proc. of the 10th SODA, pp. 905–906 (1999)

13. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for CLOSEST STRING and related problems. Algorithmica 37(1), 25–42 (2003)
14. Gramm, J., Guo, J., Niedermeier, R.: On Exact and Approximation Algorithms for Distinguishing Substring Selection. In: Proc. FST, pp. 195–209 (2003)
15. Håstad, J.: Some optimal inapproximability results. Journal of the ACM 48(4), 798–859 (2001)
16. Khot, S.: Ruling out PTAS for graph min-bisection, densest subgraph and bipartite clique. SIAM Journal on Computing 36(4), 1025–1071 (2006)
17. Khot, S., Ponnuswami, A.K.: Better Inapproximability Results for MaxClique, Chromatic Number and Min-3Lin-Deletion. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 226–237. Springer, Heidelberg (2006)
18. Lanctot, J.K., Li, M., Ma, B., Wang, S., Zhang, L.: Distinguishing string selection problems. In: Preliminary version appeared Proc. 10th SODA Information and Computation, pp. 41–55 (1999)
19. Landau, G.M., Schmidt, J.P., Sokol, D.: An algorithm for approximate tandem repeats. Journal of Computational Biology 8(1), 1–18 (2001)
20. Li, M., Ma, B., Wang, L.: Finding similar regions in many strings. Journal of Computer and System Sciences 65(1), 73–96 (2002)
21. Lokshtanov, D., Marx, D., Saurabh, S.: Slightly superexponential parameterized problems. In: Proc. of the 16th SODA, pp. 760–776 (2011)
22. Marx, D.: Parameterized complexity and approximation algorithms. Comput. J. 51(1), 60–78 (2008)
23. Meneses, C.N., Oliveira, C.A.S., Pardalos, P.M.: Optimization techniques for string selection and comparison problems in genomics. IEEE Engineering in Medicine and Biology Magazine 24(3), 81–87 (2005)
24. Ma, B.: A Polynomial Time Approximation Scheme for the Closest Substring Problem. In: Giancarlo, R., Sankoff, D. (eds.) CPM 2000. LNCS, vol. 1848, pp. 99–107. Springer, Heidelberg (2000)
25. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press (2000)
26. Ma, B., Sun, X.: More efficient algorithms for closest string and substring problems. SIAM Journal on Computing 39, 1432–1443 (2009)
27. Wang, L., Zhu, B.: Efficient Algorithms for the Closest String and Distinguishing String Selection Problems. In: Deng, X., Hopcroft, J.E., Xue, J. (eds.) FAW 2009. LNCS, vol. 5598, pp. 261–270. Springer, Heidelberg (2009)
28. Zhao, R., Zhang, N.: A more efficient closest string algorithm. In: Proc. of the 2nd BICoB, pp. 210–215 (2010)

The Parameterized Complexity of the Shared Center Problem

Zhi-Zhong Chen[1], Lusheng Wang[2], and Wenji Ma[2]

[1] Department of Information System Design, Tokyo Denki University, Hatoyama, Saitama 350-0394, Japan
zzchen@mail.dendai.ac.jp

[2] Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong SAR
lwang@cs.cityu.edu.hk

Abstract. Recently, the *shared center* (SC) problem has been proposed as a mathematical model for inferring the allele-sharing status of a given set of individuals using a database of confirmed haplotypes as reference. The problem was proved to be NP-complete and a ratio-2 polynomial-time approximation algorithm was designed for its minimization version (called the *closest shared center* (CSC) problem). In this paper, we consider the parameterized complexity of the SC problem. First, we show that the SC problem is $W[1]$-hard with parameters d and n, where d and n are the *radius* and the number of (diseased or normal) individuals in the input, respectively. Then, we present two asymptotically optimal parameterized algorithms for the problem.

Keywords: Haplotype inference, linkage analysis, pedigree, allele-sharing status, parameterized complexity, and parameterized algorithms.

1 Introduction

Linkage analysis is the first step to reduce the possible region for identifying a disease gene. Linkage studies have facilitated the identification of several hundred human genes that can harbor mutations leading to a disease phenotype. The fundamental problem in linkage analysis is to identify regions whose allele is shared by all or most affected members but by none or few unaffected family members. Most existing methods for linkage analysis are for families with clearly given pedigrees [3,8,9,15]. The pedigree information helps a lot for designing computational algorithms. Very few methods can handle the case when the sampled individuals are closely related but the real relationship is hidden (most of the times because of remote relationship). This situation occurs very often when the individuals share a common ancestor six or more generations ago.

With the new development of microarray techniques, high-density SNP genotype data can be used for large-scale and cost-effective linkage analysis [7,13]. Recently, the international HapMap project has produced enormous amount of haplotype data for individuals in some major populations. For example, there are 340 haplotypes in the group "Japanese in Tokyo"+"Han Chinese in Beijing". These new developments make it possible to propose new mathematical models for finding genes causing genetic diseases when the sampled individuals are closely related but their pedigree is unknown.

The real problem is as follows: We are given three sets $D = \{\hat{g}_1, \hat{g}_2, \ldots, \hat{g}_k\}$, $N = \{\hat{g}_{k+1}, \ldots, \hat{g}_n\}$, and $H = \{\hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m\}$, where D consists of *diseased* individuals represented by their genotype data on a *whole* chromosome C, N consists of *normal* individuals represented by their genotype data on C, and H consists of confirmed haplotype data on C of some individuals in the same (or similar) population. For convenience, we call H the *reference database.* Note that H can be obtained from any haplotype database for a set of individuals, e.g., the database of HapMap project is available. A *region* on a chromosome, denoted by $[a, b]$, is a set of consecutive SNP sites (positions) starting at position a and ending at position b. The objective here is to find the *true mutation regions* of C. Here, a true mutation region of C means a consecutive portion of C where all the diseased individuals share a common haplotype segment that is shared by none of the normal individuals. The true mutation regions defined here are based on the haplotype segments of all individuals. If we know the haplotype segments of all the individuals, the true mutation regions can be easily computed. Thus, the challenge is to infer the haplotypes of each individual based on the input genotype data as well as the reference database H.

The first strike to the problem was given by Ma *et al.* [10]. In order to tackle the problem, Ma *et al.* proposed the following strategy: First, divide the whole chromosome into a set of (disjoint) regions of the same length L. Then, classify the length-L regions into *valid* or *invalid* regions based on a mathematical model (called the *shared center* (SC) problem). Finally, design a heuristic to merge/refine the valid regions to get predicted mutation regions. For details, see Ma *et al.* [10]. The key computational technique used in the above method is the proposed mathematical model (namely, the SC problem) for inferring the allele-sharing status of a given set of individuals using a database of confirmed haplotypes as reference.

Ma *et al.* [10] show that the SC problem is NP-complete. They also consider the *closest shared center* (CSC) problem, which is the minimization counterpart of the SC problem. They propose a ratio-2 polynomial-time approximation algorithm for the CSC problem.

In this paper, we consider the parameterized complexity of the SC problem. First, we show that the SC problem is $W[1]$-hard. We then present two paremeterized algorithms for the SC problem. The two algorithms are asymptotically optimal as long as the following well-known conjecture is true:

– *Conjecture 1:* [6] There is no $O\left(2^{o(n)}\right)$-time algorithm for deciding whether a given boolean formula $C_1 \wedge C_2 \wedge \cdots \wedge C_m$ with n variables is satisfiable or not, where each C_i $(1 \leq i \leq m)$ is the disjunction of three literals.

2 Basic Definitions and Notations

For a finite set S, $|S|$ denotes the number of elements in S. Similarly, for a string s, $|s|$ denotes the length of s. A string s has $|s|$ positions, namely, 1, 2, ..., $|s|$. For convenience, if L is a positive integer, then we use $[1..L]$ to denote the set $\{1, 2, \ldots, L\}$. The letter of s at position $i \in [1..|s|]$ is denoted by $s[i]$. Thus, $s = s[1]s[2] \ldots s[|s|]$. For two integers i and j with $1 \le i \le j \le |s|$, $s[i..j]$ denotes $s[i]s[i+1] \cdots s[j]$. For a binary string s, $\bar{s}$ denotes the *complement string* of s, where $\bar{s}[i] \neq s[i]$ for every $i \in [1..|s|]$. For two strings s and t of the same length, $\{s \not\equiv t\}$ denotes the set of all positions $i \in [1..|s|]$ with $s[i] \neq t[i]$. For a string s and a subset P of $[1..|s|]$, $s|_P$ denotes the string obtained from s by deleting all letters at positions not in P.

At last, when an algorithm exhaustively tries all possibilities to find the right choice, we say that the algorithm *guesses* the right choice.

3 The SC Problem

An input to the SC problem is a quadruple (D, N, H, d), where $D = \{g_1, g_2, \ldots, g_k\}$ and $N = \{g_{k+1}, g_{k+2}, \ldots, g_n\}$ are sets consisting of genotype segments of the same length L, $H = \{h_1, h_2, \ldots, h_m\}$ is a set of haplotype segments of length L, and d (referred to as the *radius*) is a nonnegative integer. The segments in D are from diseased individuals while those in N are from normal individuals. Recall that a haplotype segment is a binary string, while a genotype segment is a string on alphabet $\{0, 1, 2\}$. A *haplotype pair* for a genotype segment g is a pair (h, h') of haplotype segments of the same length as g such that the following conditions hold for each position q:

1. If g has a 0 or 1 at position q, then both h and h' have the same letter as g does at position q.
2. If g has a 2 at position q, then one of h and h' has a 0 at position q while the other has a 1 at position q.

For convenience, for two binary strings s and t, we denote their Hamming distance by $dist(s, t)$. Given an input (D, N, H, d), the SC problem requires the computation of a *solution* to (D, N, H, d) which consists of a *center haplotype segment* s of length L, a *center index* $p \in \{1, 2, \ldots, m\}$, and a haplotype pair $(h_{i,1}, h_{i,2})$ for each $g_i \in D \cup N$ such that the following conditions hold:

C1. $dist(s, h_p) \le d$.
C2. For each $i \in \{1, 2, \ldots, k\}$, $h_{i,1} = s$ and there is an integer $x_i \in \{1, 2, \ldots, m\}$ such that $dist(h_{i,2}, h_{x_i}) \le d$.
C3. For each $i \in \{k+1, k+2, \ldots, n\}$ and for each $j \in \{1, 2\}$, the following hold:
- **C3a.** There is an integer $x_{i,j} \in \{1, 2, \ldots, m\} \setminus \{p\}$ with $dist(h_{i,j}, h_{x_{i,j}}) \le d$.
- **C3b.** There is at least one position q at which the letters of $h_{i,j}$ and s differ and the letter of some $g_\ell \in D$ is 0 or 1.

Note that the position q in Condition C3b depends not only on i and j but also on $h_{i,j}$, i.e., different i, j, or $h_{i,j}$ may yield different q.

For the reasons for Conditions C1 through C3, the reader is referred to [10].

4 The Hardness of the SC Problem

A *parameterized problem* Q over an alphabet Σ is a subset of $\Sigma^* \times \mathbf{N}$, where Σ^* is the set of all strings over Σ and $\mathbf{N}$ is the set of all nonnegative integers. A parameterized problem Q over an alphabet Σ is *fixed-parameter tractable* if for every $(x, k) \in \Sigma^* \times \mathbf{N}$, we can decide whether $(x, k) \in Q$ or not in time $O\left(f(k) \cdot |x|^c\right)$ for some constant c and computable function f.

Let FPT denote the set of all fixed-parameter tractable problems. There are a number of problems that do not seem to belong to FPT. So, certain complexity classes have been defined to include such problems in the literature. W[1] is one of them. Here, we omit the somewhat technical definition of W[1]. For a precise definition of W[1], the reader is referred to [4].

To give strong evidence that certain problems in W[1] are unlikely to belong to FPT, the theory of W[1]-hardness has been developed. At the heart of this theory is the notion of parameterized reduction. A *parameterized reduction* from a parameterized problem Q over an alphabet Σ to another parameterized problem Q' over an alphabet Γ is a function that maps each pair $(x, k) \in \Sigma^* \times \mathbf{N}$ to a pair $(x', k') \in \Gamma^* \times \mathbf{N}$ such that the following conditions hold:

– $(x, k) \in Q$ if and only if $(x', k') \in Q'$.
– k' is bounded from above by a function of k.
– (x', k') can be computed in time $O\left(f(k) \cdot |x|^c\right)$ for some constant c and function f.

A parameterized problem Q' is *W[1]-hard* if for every parameterized problem Q in W[1], there is a parameterized reduction from Q to Q'.

Theorem 1. *The SC problem is $W[1]$-hard with parameters d and n.*

Proof. We give a parameterized reduction from the binary closest-substring (BCSS) problem to the special case of the SC problem where all the individuals are diseased. Recall that an instance of the BCSS problem is a tuple $(s_1, \ldots, s_k, L, d)$, where $s_1, \ldots, s_k$ are binary strings each of length at least L and d is a nonnegative integer. Given $(s_1, \ldots, s_k, L, d)$, the BCSS problem asks if there is a binary string t of length L such that for all $1 \le i \le k$, s_i has a substring s'_i of length L with $dist(t, s'_i) \le d$. It is known that the BCSS problem is $W[1]$-hard with parameters d and k [12].

Let $(s_1, \ldots, s_k, L, d)$ be an instance of the BCSS problem. For each $1 \le i \le k$, let L_i be the length of s_i. For convenience, for a letter $\ell \in \{0, 1, 2\}$ and a nonnegative integer i, let ℓ^i denote the string consisting of i ℓs. Note that ℓ^0 is the empty string. We obtain $m = (L_1 - L + 1) + \sum_{i=1}^{k}(L_i - L + 1)$ strings h_1, $h_2, \ldots, h_m$ as follows:

1. For each $1 \le j \le L_1 - L + 1$, $h_j = s_1[j..j + L - 1]0^{(d+1)k}$.
2. For each $i \in \{1, \ldots, k\}$ and each $1 \le j \le L_i - L + 1$, $h_y = s_i[j..j + L - 1]0^{(d+1)(i-1)}1^{d+1}0^{(d+1)(k-i)}$, where $y = (L_1 - L + 1) + \sum_{z=1}^{i-1}(L_z - L + 1) + j$.

We further obtain k strings $g_1, \ldots, g_k$ as follows:

– For each $i \in \{1, \ldots, k\}$, $g_i = 2^L 0^{(d+1)(i-1)} 2^{d+1} 0^{(d+1)(k-i)}$.

Suppose that $(s_1, \ldots, s_k, L, d)$ has a solution t in the BCSS problem. Then, for each $1 \leq i \leq k$, there is an integer j_i with $1 \leq j_i \leq L_i - L + 1$ such that $dist(t, s_i[j_i..j_i + L - 1]) \leq d$. We next construct a solution for the instance $(\{g_1, \ldots, g_k\}, \emptyset, \{h_1, \ldots, h_m\}, d)$ of the SC problem as follows.

1. $s = t0^{(d+1)k}$. Note that $dist(s, h_{j_1}) \leq d$ because $dist(t, s_1[j_1..j_1 + L - 1]) \leq d$.
2. For each $i \in \{1, \ldots, k\}$, construct a haplotype pair $(h_{i,1}, h_{i,2})$ for g_i by setting $h_{i,1} = s$ and $h_{i,2} = \bar{t}0^{(d+1)(i-1)} 1^{d+1} 0^{(d+1)(k-i)}$. Note that for each $1 \leq i \leq k$, $dist(h_{i,2}, h_y) = dist(\bar{t}, \overline{s_i[j_i..j_i + L - 1]}) = dist(t, s_i[j_i..j_i + L - 1]) \leq d$, where $y = (L_1 - L + 1) + \sum_{z=1}^{i-1}(L_z - L + 1) + j$.

Conversely, suppose that the instance $(\{g_1, \ldots, g_k\}, \emptyset, \{h_1, \ldots, h_m\}, d)$ of the SC problem has a solution. Let s be the center haplotype segment in the solution. Let t be the prefix of s with $|t| = L$. We claim that t is a solution to $(s_1, \ldots, s_k, L, d)$ in the BCSS problem. To see this, first note that for each $1 \leq i \leq (d+1)k$, there is an integer $j \in \{1, \ldots, k\}$ such that the ith rightmost letter of g_j is a 0. This implies that the last $(d+1)k$ bits of s are 0s. So, the string h_j with $dist(s, h_j) \leq d$ has to be among $h_1, \ldots, h_{L_1 - L + 1}$ because there are $d + 1$ 1s in the last $(d + 1)k$ bits of each h_j with $L_1 - L + 2 \leq j \leq m$. Thus, $dist(t, s_1[j..j + L - 1]) \leq d$ for some $1 \leq j \leq L_1 - L + 1$. Moreover, for each $1 \leq i \leq k$, if we decompose g_i into two strings $h_{i,1}$ and $h_{i,2}$ with $h_{i,1} = s$, then $h_{i,2} = \bar{t}0^{(d+1)(i-1)} 1^{d+1} 0^{(d+1)(k-i)}$. Hence, for each $1 \leq i \leq k$, the string h_y with $1 \leq y \leq m$ and $dist(h_y, h_{i,2}) \leq d$ has to satisfy $(L_1 - L + 1) + \sum_{z=1}^{i-1}(L_z - L + 1) + 1 \leq y \leq (L_1 - L + 1) + \sum_{z=1}^{i-1}(L_z - L + 1) + (L_i - L + 1)$, because of the different locations of the $d + 1$ 1s in the last $(d+1)k$ bits of $h_1, \ldots, h_m$. Therefore, for some $1 \leq j \leq L_i - L + 1$, $dist(t, s_i[j..j + L - 1]) = dist(\bar{t}, \overline{s_i[j..j - L + 1]}) \leq d$. This completes the proof of the claim and hence that of the theorem. Q.E.D.

Corollary 1. *As long as Conjecture 1 is true, the SC problem cannot be solved in $O(f(d, k)\tilde{n}^{o(\log d)})$ time for any computable function f, where $\tilde{n}$ is the length of the input to the SC problem.*

Proof. Marx [12] shows that as long as Conjecture 1 is true, the BCSS problem cannot be solved in $O(f(d, k)\hat{n}^{o(\log d)})$ time for any computable function f, where $\hat{n}$ is the length of the input to the BCSS problem. In the reduction described in the proof of Theorem 1, we constructed an instance I of the SC problem from a length-$\hat{n}$ instance of the BCSS problem such that $|I| = O(\hat{n}^2)$. Moveover, the paremeters in the two instances are the same. Thus, the corollary holds. Q.E.D.

5 An Exact Algorithm for the SC Problem

Throughout this section, let $\mathcal{I} = (D, N, H, d)$ be an instance of the SC problem, where $D = \{g_1, g_2, \ldots, g_k\}$, $N = \{g_{k+1}, g_{k+2}, \ldots, g_n\}$, and $H = \{h_1, h_2, \ldots, h_m\}$.

Consider a genotype segment $g_i \in D \cup N$ and a haplotype pair $(h_{i,1}, h_{i,2})$ for g_i. A position of g_i with a letter 0 indicates that both $h_{i,1}$ and $h_{i,2}$ have a letter 0 at the position, while a position of g_i with a letter 1 indicates that both $h_{i,1}$ and $h_{i,2}$ have a letter 1 at the position. On the other hand, a position of g_i with a letter 2 indicates that one of $h_{i,1}$ and $h_{i,2}$ has a 0 at the position while the other has a 1 at the position. For convenience, we say that a position of g_i is *decided* if the letter of g_i at the position is 0 or 1, and is *undecided* otherwise.

For D, we define three sets as follows:

– The set of *decided positions associated with* D consists of all positions q in R such that q is a decided position of at least one string in D.
– The set of *undecided positions associated with* D consists of all positions q in R such that q is an undecided position for all strings in D.
– The set of *conflicting positions associated with* D consists of all positions q in R such that q is a decided position of two distinct $g_i \in D$ and $g_j \in D$ but the letters of g_i and g_j at position q differ.

We say that an integer b is a *valid radius* if the instance (D, N, H, b) to the SC problem has a solution. Our goal is to decide if d is a valid radius. Obviously, the following condition is necessary for d to be a valid radius:

A1. The set of conflicting positions associated with D is empty.

So, we hereafter assume that Condition A1 holds.

For convenience, we define the following notations:

– L is the common length of the strings in $D \cup N \cup H$.
– U (respectively, $\overline{U}$) is the set of undecided (respectively, decided) positions associated with D.
– For each $g_i \in N$, U_i (respectively, $\overline{U_i}$) is the set of undecided (respectively, decided) positions of g_i.

Now, Condition C3b in Section 1 can be concisely rewritten as follows:

C3b. $h_{i,j}|_{\overline{U}} \neq s|_{\overline{U}}$.

Since Condition A1 holds, we can define a letter ℓ_q for each $q \in \overline{U}$ as follows:

– If some segment in D is 0 at position q, then each of the other segments in D is 0 or 2 at position q and so we define $\ell_q = 0$.
– If some segment in D is 1 at position q, then each of the other segments in D is 1 or 2 at position q and so we define $\ell_q = 1$.

We call ℓ_q the *center letter* at position q.

Consider a $g_i \in N$. We say that g_i is *free* if there is a position in $\overline{U_i} \cap \overline{U}$ at which the center letter is different from the letter of g_i. On the other hand, we say that g_i is *dead* if (1) $|\overline{U} \setminus \overline{U_i}| \leq 1$ and (2) at every position q in $\overline{U_i} \cap \overline{U}$, the center letter is the same as the letter of g_i.

In [10], it is shown that L is a valid radius only if the following holds:

A2. No string $g_i \in N$ is dead.

Since d is a valid radius only when L is a valid radius, we hereafter assume that Condition A2 holds.

5.1 Decomposing $g_i \in N$

Throughout this subsection, fix a genotype segment $g_i \in N$ and two haplotype segments h_{j_1} and h_{j_2} in H. Note that it is possible that $j_1 = j_2$. Our goal is to decide if there is a haplotype pair $(h_{i,1}, h_{i,2})$ for g_i satisfying the following four conditions:

B1. $dist(h_{i,1}, h_{j_1}) \leq d$.
B2. $dist(h_{i,2}, h_{j_2}) \leq d$.
B3. $h_{i,1}|_{\overline{U}} \neq s|_{\overline{U}}$.
B4. $h_{i,2}|_{\overline{U}} \neq s|_{\overline{U}}$.

To reach the above goal, we first define several notations:

- $d_1 = dist(g_i|_{\overline{U}_i}, h_{j_1}|_{\overline{U}_i})$ and $d_2 = dist(g_i|_{\overline{U}_i}, h_{j_2}|_{\overline{U}_i})$.
- S is the set of positions $q \in U_i$ such that the letters of h_{j_1} and h_{j_2} at position q coincide.

Lemma 1. *If at least one of the following three conditions holds, then we can easily decide if there is a haplotype pair $(h_{i,1}, h_{i,2})$ for g_i satisfying Conditions B1 through B4.*

1. $d_1 = d$ or $d_2 = d$.
2. $d_1 + d_2 + |S| > 2d$.
3. $d_1 < d$, $d_2 < d$, $d_1 + d_2 + |S| \leq 2d$, and g_i is free or $(\overline{U} \cap U_i) \setminus S$ contains two positions q_1 and q_2 such that the letter of h_{j_1} at position q_1 is not the center letter at position q_1 and the letter of h_{j_2} at position q_2 is not the center letter at position q_2.

By Lemma 1 and Condition A2, we may assume that the following hold:

D1. $d_1 < d$ and $d_2 < d$.
D2. $d_1 + d_2 + |S| \leq 2d$.
D3. g_i is neither free nor dead.
D4. $(\overline{U} \cap U_i) \setminus S$ does not contain two positions q_1 and q_2 such that the letter of h_{j_1} at position q_1 is not the center letter at position q_1 and the letter of h_{j_2} at position q_2 is not the center letter at position q_2.

By Condition D3, $|\overline{U} \cap U_i| \geq 2$. Without loss of generality, we assume that the center letters at the positions in $\overline{U} \cap U_i$ are all 0s.

Lemma 2. *If at least one of the following two conditions holds, then there is no haplotype pair $(h_{i,1}, h_{i,2})$ for g_i satisfying Conditions B1 through B4.*

E1. *$|\overline{U} \cap U_i| = 2$, $\overline{U} \cap U_i \subseteq S$, the two letters of h_{j_1} at positions in $\overline{U} \cap U_i$ are different, and $d_1 = d_2 = d - 1$.*
E2. *$d_1 + d_2 + |S| \geq 2d - 1$, $\overline{U} \cap S = \emptyset$, and either the letters of h_{j_1} at the positions in $\overline{U} \cap U_i$ are all 0s or the letters of h_{j_2} at the positions in $\overline{U} \cap U_i$ are all 0s.*

Otherwise, we can find a haplotype pair $(h_{i,1}, h_{i,2})$ *for* g_i *satisfying Conditions B1 through B4 in* $O(L)$ *time.*

For convenience, we say that an integer $p \in \{1, \ldots, m\}$ is *valid* for a $g_i \in N$ if there is a haplotype pair $(h_{i,1}, h_{i,2})$ for g_i satisfying Condtion C3. Now, we are ready to state the key lemma in this subsection.

Lemma 3. *Given an integer* $p \in \{1, \ldots, m\}$*, we can decide in* $O(m^2L(n-k))$ *time if* p *is valid for every* $g_i \in N$*.*

5.2 Decomposing the Strings in D

Throughout this subsection, fix an integer $p \in \{1, 2, \ldots, m\}$ that is valid for every $g_i \in N$. Let s_p be the haplotype segment constructed by letting $s_p[q]$ be the center letter at position q for each $q \in \overline{U}$, and letting $s_p[q] = h_p[q]$ for each $q \in U$. Moreover, let d_p be the Hamming distance between $s_p|_{\overline{U}}$ and $h_p|_{\overline{U}}$.

Our task is to decide if we can modify at most $d - d_p$ letters of s_p at the positions in U and obtain a haplotype pair $(h_{i,1}, h_{i,2})$ for each $g_i \in D$ so that Condition C2 is satisfied. Obviously, no matter how we modify the letters of s_p at the positions in U, it always holds that for each $g_i \in D$, there is a unique haplotype pair $(h_{i,1}, h_{i,2})$ with $h_{i,1} = s_p$. Basically, we are not allowed to modify the letters of s_p at the positions in $\overline{U}$ and $h_{i,2}|_U$ is the complement string of $s_p|_U$. We hereafter use $\overline{s_p(i)}$ to denote this $h_{i,2}$.

Our task becomes much easier if we know the integer x_i for each $g_i \in D$ in Condition C2. So, we consider this case first. In this case, we want to decide if we can modify at most $d - d_p$ letters of s_p at the positions in U so that $dist(\overline{s_p(i)}, h_{x_i}) \le d$ for all $g_i \in D$. This case resembles the *binary closest string* (BCS) problem. Recall that an instance of the BCS problem is a pair $(\mathcal{S}, d)$, where $\mathcal{S}$ is a set of binary strings of the same length L and d is a nonnegative integer. Given $(\mathcal{S}, d)$, the BCS problem asks if there is a binary string t of length L such that $dist(t, s_i) \le d$ for all $s_i \in \mathcal{S}$. Known algorithms for the BCS problem can be found in [1,2,5,11,14,16]. All the algorithms indeed solve a more general problem (called the *extended BCS* problem). An input to the extended BCS problem contains not only $(\mathcal{S}, d)$ but also a triple (s, P, b), where s is a string of length L, P is a subset of $[1..L]$, and b is an integer less than or equal to d. The objective is to decide if we can modify at most b letters of s at the positions in $[1..L] \setminus P$ so that $dist(s, s_i) \le d$ for all $s_i \in \mathcal{S}$.

The correspondence between the extended BCS problem and the special case of the SC problem is as follows: s, b, $s_i \in \mathcal{S}$, and P in the former correspond to s_p, $d - d_p$, h_{x_i}, and $\overline{U}$ in the latter, respectively. A slight difference between the two is that the former tests if $dist(s, s_i) \le d$ for all $s_i \in \mathcal{S}$, while the latter tests if $dist(\overline{s_p(i)}, h_{x_i}) \le d$ for all $g_i \in D$. Based on this correspondence and difference, it is easy to modify the algorithm in [11] for the extended BCS problem so that it works for the special case of the SC problem. The resulting algorithm is called *Algorithm 1*.

```
Algorithm 1

Input: A triple $(s, P, b)$, where $s$ is a string of length $L$, $P$ is a subset of
    $[1..L]$, and $b$ is a nonnegative integer less than or equal to $d - d_p$.
Output: A string $t$ obtained by modifying at most $b$ positions of $s$ in $U \setminus P$
    such that $dist(\overline{t(i)}, h_{x_i}) \leq d$ for each $g_i \in D$, if such a $t$ exists; nothing,
    otherwise.
1. If $dist(h_{x_i}, \overline{s(i)}) \leq d$ for every $g_i \in D$, then output $s$ and stop immediately.
2. Select a $g_i \in D$ with $dist(h_{x_i}, \overline{s(i)}) > d$.
3. If $dist(\overline{s(i)}, h_{x_i}) - d > \min\{b, |\{\overline{s(i)} \not\equiv h_{x_i}\} \setminus P|\}$, then return.
4. Let $Q = \{\overline{s(i)} \not\equiv h_{x_i}\} \setminus P$ and $\ell = dist(\overline{s(i)}, h_{x_i}) - d$.
5. Guess a subset $Z$ of $Q$ with $\ell \leq |Z| \leq b$.
6. Obtain a string $s'$ by modifying $s$ by flipping the letters at the positions
    in $Z$.
7. Recursively call the algorithm on input $(s', P \cup Q, \min\{b - |Z|, |Z| - \ell\})$.
```

To solve our special case, it suffices to call Algorithm 1 on input $(s_p, \overline{U}, d-d_p)$. The correctness of Algorithm 1 relies on the following lemma whose proof is the same as that of Lemma 3.1 in [1].

Lemma 4. *Let (s, P, b) be an input to Algorithm 1. Assume that t is an output of Algorithm 1 on input (s, P, b). Suppose that $dist(\overline{s(i)}, h_{x_i}) > d$ for some $g_i \in D$. Let $\ell = dist(\overline{s(i)}, h_{x_i}) - d$, z be the number of positions $q \in \{\overline{s(i)} \not\equiv h_{x_i}\} \setminus P$ with $\overline{t(i)}[q] \neq \overline{s(i)}[q]$, and b' be the number of positions $q \in [1..L] \setminus (P \cup \{\overline{s(i)} \not\equiv h_{x_i}\})$ with $\overline{t(i)}[q] \neq \overline{s(i)}[q]$. Then, $b' \leq \min\{b - z, z - \ell\}$. Consequently, $b' \leq \frac{1}{2}(b - \ell)$.*

To see the correctness of Algorithm 1, first observe that Step 1 is clearly correct. To see that Step 3 is also correct, first note that $dist(h_{x_i}, \overline{s(i)}) = |\{h_{x_i} \not\equiv \overline{s(i)}\}| = d + \ell$. So, in order to satisfy $dist(h_{x_i}, \overline{s(i)}) \leq d$, we need to first select at least ℓ positions among the positions in $\{h_{x_i} \not\equiv \overline{s(i)}\}$ and then modify the letters at the selected positions. By definition, we are allowed to select at most b positions and the selected positions have to be in $Q = \{h_{x_i} \not\equiv \overline{s(i)}\} \setminus P$; so no solution exists if $\ell > \min\{b, |Q|\}$. The correctness of Step 7 is guaranteed by Lemma 4. This can be seen by viewing $|Z|$ in Algorithm 1 as z in Lemma 4, and viewing b' in Lemma 4 as the number of positions (outside $P \cup \{h_{x_i} \not\equiv \overline{s(i)}\} = P \cup Q$) of $\overline{s(i)}$ where the letters can be modified in order to transform $\overline{s(i)}$ into $\overline{t(i)}$. That is, $\min\{b - |Z|, |Z| - \ell\}$ in Step 7 corresponds exactly to $\min\{b - z, z - \ell\}$ in Lemma 4.

The execution of Algorithm 1 on input (s, P, b) can be modeled by a tree $\mathcal{T}$ in which the root corresponds to (s, P, b), each other node corresponds to a recursive call, and a recursive call A is a child of another call B if and only if B calls A directly. We call $\mathcal{T}$ the *search tree* on input (s, P, b). By the construction of Algorithm 1, each non-leaf node in $\mathcal{T}$ has at least two children. Thus, the number of nodes in $\mathcal{T}$ is at most twice the number of leaves in $\mathcal{T}$. Consequently, we can focus on how to bound the number of leaves in $\mathcal{T}$. For convenience, we define the *size* of $\mathcal{T}$ to be the number of its leaves.

```
Algorithm 2

Input: A triple $(s, P, b)$, where $s$ is a string of length $L$, $P$ is a subset of
    $[1..L]$, and $b$ is a nonnegative integer less than or equal to $d - d_p$.
Output: A string $t$ obtained by modifying at most $b$ positions of $s$ in
    $U \setminus P$ such that for each $g_i \in D$, there is an integer $x_i \in \{1, \ldots, m\}$ with
    $dist(\overline{t(i)}, h_{x_i}) \leq d$, if such a $t$ exists; nothing, otherwise.
1. If for every $g_i \in D$, there is a string $h_j \in H$ such that $dist(h_j, \overline{s(i)}) \leq d$,
    then output $s$ and stop immediately.
2. Select a $g_i \in D$ such that for every $h_j \in H$, $dist(h_j, \overline{s(i)}) > d$.
3. If all $h_j \in H$ satisfy $dist(\overline{s(i)}, h_j) - d > \min\{b, |\{\overline{s(i)} \not\equiv h_j\} \setminus P|\}$, then
    return.
4. Guess an $h_{x_i} \in H$ such that $dist(\overline{s(i)}, h_{x_i}) - d \leq \min\{b, |\{\overline{s(i)} \not\equiv h_{x_i}\} \setminus P|\}$.
5 - 7. Same as Steps 4, 5, and 6 in Algorithm 1, respectively.
8. Recursively call the algorithm on input $(s', P \cup Q, \min\{b - |Z|, |Z| - \ell\})$.
```

Let $T_1(d, d - d_p)$ be the size of the search tree of Algorithm 1 on input $(s_p, \overline{U}, d - d_p)$. Similar to Theorem 3.4 in [1], we can prove the next lemma:

Lemma 5. $T_1(d, d - d_p) \leq \frac{(6\sqrt{3})^d}{(\sqrt{3} \cdot \sqrt[3]{4})^{d_p}}$.

In the general case, we do not know the integer x_i for each $g_i \in D$ in Condition C2. To extend Algorithm 1 so that it works for the (general) SC problem, the idea is to *guess* x_i in Step 4. That is, we do not guess all of x_1, …, x_k in advance. Rather, we guess x_i dynamically. The algorithm is called *Algorithm 2*.

Obviously, if we do not guess h_{x_i} in Step 4 of Algorithm 2 but rather choose an arbitrary h_{x_i} in H there, then the search tree of Algorithm 2 on input (s, P, b) has the same size as the search tree of Algorithm 1 on input (s, P, b) does. So, to estimate the size of the search tree of Algorithm 2 on input (s, P, b), it suffices to find out how the guesses in Step 4 of Algorithm 2 expand the size of the search tree of Algorithm 1 on input (s, P, b). Clearly, the "guess" operation in Step 4 requires Algorithm 2 to try all $h_j \in H$ with $dist(\overline{s(i)}, h_j) - d \leq \min\{b, |\{\overline{s(i)} \not\equiv h_j\} \setminus P|\}$. A single "guess" expands the size of the search tree by a factor of at most m. Because of Lemma 4, the recursion depth of Algorithm 2 is at most $\lfloor \log_2(b + 1) \rfloor$. Thus, the size of the search tree of Algorithm 2 on input (s, P, b) is at most $m^{\lfloor \log_2(b+1) \rfloor}$ times that of the search tree of Algorithm 1 on input (s, P, b).

Now, let $T_2(d, d - d_p)$ be the size of the search tree of Algorithm 2 on input $(s_p, \overline{U}, d - d_p)$. Then,

Lemma 6. $T_2(d, d - d_p) \leq \frac{(6\sqrt{3})^d}{(\sqrt{3} \cdot \sqrt[3]{4})^{d_p}} \cdot m^{\lfloor \log_2(d - d_p + 1) \rfloor}$.

Theorem 2. *Algorithm 2 takes* $O\left(kmL + kmd \cdot \frac{(6\sqrt{3})^d}{\left(\sqrt{3}\cdot\sqrt[3]{4}\right)^{d_p}} \cdot m^{\lfloor \log_2(d-d_p+1)\rfloor}\right)$ *time.*

Proof. Obviously, excluding the recursive calls, each step of Algorithm 2 takes $O(kmL)$ time. To improve this time bound to $O(kmd)$, the idea is to perform an $O(kmL)$-time preprocessing. In the preprocessing, for each pair (i,j) with $i \in \{1,\dots,k\}$ and $j \in \{1,\dots,m\}$, we compute $\Delta_{i,j} = \{\overline{s_p(i)} \neq h_j\}$ and discard it if $|\Delta_{i,j}| > 2d - d_p$. Note that we modify only $O(d)$ letters of s_p at Step 7 of Algorithm 2 on input $(s_p, \overline{U}, d - d_p)$ and hence we can accordingly update each remaining $\Delta_{i,j}$ within $O(d)$ time. This is also true in subsequent recursive calls. So, by Lemma 6, the total time complexity of Algorithm 2 is as stated in the theorem.

By Lemma 3 and Theorem 2, we have the following corollary immediately:

Corollary 2. *The SC problem can be solved in time*

$$O\left(m^3L(n-k) + m^2Lk + kd \cdot \frac{(6\sqrt{3})^d}{\left(\sqrt{3}\cdot\sqrt[3]{4}\right)^{d'}} \cdot m^{\lfloor \log_2(d-d'+1)\rfloor+2}\right),$$

where $d' = \min_{p\in\{1,\dots,m\}} dist(s_p|_{\overline{U}}, h_p|_{\overline{U}})$.

As observed in [14], Algorithm 1 can be made faster by replacing Step 2 with the following step:

2'. If $b = d - d_p$, then select a $g_i \in D$ such that $dist(h_{x_i}, \overline{s(i)}) \geq dist(h_{x_j}, \overline{s(j)})$ for all $g_j \in D$. Otherwise, select an arbitrary $g_i \in D$ with $dist(h_{x_i}, \overline{s(i)}) > d$.

We call the modified algorithm *Algorithm 3*. The intuition behind Algorithm 3 is as follows. By Lemma 4, the larger ℓ is, the smaller b' is. Note that b' means the number of letters of $\overline{s(i)}$ we need to further modify. Thus, by maximizing ℓ, we can make the algorithm run faster.

Let $T_3(d, d - d_p)$ be the size of the search tree of Algorithm 1 on input $(s_p, \overline{U}, d - d_p)$. Similar to Theorem 4.3 in [1], we can prove the next lemma:

Lemma 7. $T_3(d, d-d_p) \leq \frac{8^d}{2^{d_p}}$.

We next obtain a slower version of Algorithm 3 by replacing Step 2' with the following step:

2". If $b = d - d_p$, then *guess* a $g_i \in D$ and make a copy s_0 of s and a copy i_0 of i. Otherwise, select an arbitrary $g_i \in D$ with $dist(h_{x_i}, \overline{s(i)}) > d$ and $dist(h_{x_{i_0}}, \overline{s_0(i_0)}) \geq dist(h_{x_i}, \overline{s_0(i)})$.

We call the modified algorithm *Algorithm 4*. Algorithm 4 will be useful later in this paper when we consider the general case where $x_1, \ldots, x_k$ are not known. Basically, if $b = d - d_p$, a string $g_i \in D$ in Step 2' is hard to find when $x_1, \ldots, x_k$ are not known. In this case, our idea is to guess this $g_i \in D$ and use i_0 and s_0 to memorize i and s (for later use), respectively. Note that Algorithm 4 does not verify that for all $g_j \in D$, $dist(h_{x_{i_0}}, \overline{s_0(i_0)}) \geq dist(h_{x_j}, \overline{s_0(j)})$. Indeed, only for those g_i selected in Step 2" of subsequent recursive calls, Algorithm 4 verifies that $dist(h_{x_{i_0}}, \overline{s_0(i_0)}) \geq dist(h_{x_i}, \overline{s_0(i)})$.

Algorithm 4 on input $(s_p, \overline{U}, d - d_p)$ is clearly correct, because (1) it performs the *guess* operation for $b = d - d_p$ by trying all $g_i \in D$ and (2) when trying the $g_i \in D$ with $dist(h_{x_i}, \overline{s_p(i)}) \geq dist(h_{x_j}, \overline{s_p(j)})$, it does the same as Algorithm 3. To estimate the running time of Algorithm 4 on input $(s_p, \overline{U}, d - d_p)$, let $T_4(d, b)$ be the size of the search tree of Algorithm 4 on input $(s_p, \overline{U}, d - d_p)$.

Lemma 8. $T_4(d, d - d_p) \leq k \cdot \frac{8^d}{2^{d_p}}$.

Proof. Suppose that we modify Algorithm 4 on input $(s_p, \overline{U}, d - d_p)$ by not *guessing* $g_i \in D$ but rather choosing a particular $g_i \in D$. Let $\mathcal{T}_i$ be the search tree of the modified algorithm on input $(s_p, \overline{U}, d - d_p)$. Obviously, $T_4(d, d - d_p)$ does not exceed the total size of $\mathcal{T}_1, \ldots, \mathcal{T}_n$. So, it suffices to show that for each $i \in \{1, \ldots, k\}$, the size of $\mathcal{T}_i$ is at most $8^d/2^{d_p}$.

Fix an $i_0 \in \{1, \ldots, k\}$. To show that the size of $\mathcal{T}_{i_0}$ is at most $8^d/2^{d_p}$, first note that for each non-root node μ of $\mathcal{T}_{i_0}$, the recursive call (of the modified algorithm) corresponding to μ selects a $g_i \in D$ in Step 2" such that $dist(h_{x_i}, \overline{s(i)}) > d$ and $dist(h_{x_{i_0}}, \overline{s_0(i_0)}) \geq dist(h_{x_i}, \overline{s_0(i)})$. Because of these two inequalities, we can use similar arguments to those in the proofs of Lemmas 4.1 and 4.2 and Theorem 4.3 in [1] to prove that the size of $\mathcal{T}_{i_0}$ is at most $8^d/2^{d_p}$. Q.E.D.

We next extend Algorithm 4 so that it works for the general case. The idea is the same as that used to obtain Algorithm 2 from Algorithm 1. So, as in Algorithm 2, we *guess* x_i dynamically in Step 4. The resulting algorithm is called *Algorithm 5*.

Let $T_5(d, d - d_p)$ be the size of the search tree of Algorithm 5 on input $(s_p, \overline{U}, d - d_p)$. Then,

Lemma 9. $T_5(d, d - d_p) \leq k \cdot \frac{8^d}{2^{d_p}} \cdot m^{\lfloor \log_2(d - d_p + 1) \rfloor}$.

Using Lemma 9, we can prove the next theorem (whose proof is similar to that of Theorem 2):

Theorem 3. *Algorithm 5 takes* $O\left(kmL + k^2 md \cdot \frac{8^d}{2^{d_p}} \cdot m^{\lfloor \log_2(d - d_p + 1) \rfloor}\right)$ *time.*

By Lemma 3 and Theorem 3, we have the following corollary immediately:

Corollary 3. *The SC problem can be solved in time*

$$O\left(m^3 L(n-k) + m^2 Lk + k^2 d \cdot \frac{8^d}{2^{d'}} \cdot m^{\lfloor \log_2(d - d' + 1) \rfloor + 2}\right),$$

where $d' = \min_{p \in \{1, \ldots, m\}} dist(s_p|_{\overline{U}}, h_p|_{\overline{U}})$.

```
Algorithm 5

Input: A triple $(s, P, b)$, where $s$ is a string of length $L$, $P$ is a subset of
    $[1..L]$, and $b$ is an integer less than or equal to $d - d_p$.
Output: A string $t$ obtained by modifying at most $b$ positions of $s$ in
    $U \setminus P$ such that for each $g_i \in D$, there is an integer $x_i \in \{1, \ldots, m\}$ with
    $dist(\overline{t(i)}, h_{x_i}) \leq d$, if such a $t$ exists; nothing, otherwise.
1. If for every $g_i \in D$, there is a string $h_j \in H$ such that $dist(h_j, \overline{s(i)}) \leq d$,
    then output $s$ and stop immediately.
2. If $b = d - d_p$, then guess a $g_i \in D$ and make a copy $s_0$ of $s$ and a copy $i_0$ of $i$.
    Otherwise, select a $g_i \in D$ such that for every $h_j \in H$, $dist(h_j, \overline{s(i)}) > d$.
3. If all $h_j \in H$ satisfy $dist(\overline{s(i)}, h_j) - d > \min\{b, |\{\overline{s(i)} \not\equiv h_j\} \setminus P|\}$, then
    return.
4. If $b = d - d_p$, guess an $h_{x_{i_0}} \in H$ such that $dist(\overline{s(i_0)}, h_{x_{i_0}}) - d \leq$
    $\min\{b, |\{\overline{s(i_0)} \not\equiv h_{x_{i_0}}\} \setminus P|\}$. Otherwise, guess an $h_{x_i} \in H$ such that
    $dist(\overline{s(i)}, h_{x_i}) - d \leq \min\{b, |\{\overline{s(i)} \not\equiv h_{x_i}\} \setminus P|\}$ and $dist(\overline{s_0(i_0)}, h_{x_{i_0}}) \geq$
    $dist(\overline{s_0(i)}, h_{x_i})$.
5 – 7. Same as Steps 4, 5, and 6 in Algorithm 1, respectively.
8. Recursively call the algorithm on input $(s', P \cup Q, \min\{b - |Z|, |Z| - \ell\})$.
```

Roughly speaking, Algorithm 5 is faster than Algorithm 2 if and only if $d \geq \log_b k$, where $b = \frac{3\sqrt{3}}{4}$.

6 Concluding Remarks

We have presented two asymptotically optimal parameterized algorithms for the SC problem. It remains to implement the algorithms in a programming language and apply them to linkage analysis of real biological data. Although the algorithms are asymptotically optimal, their complexity still looks high and they can be very slow when applied to real data. Thus, an interesting question is to ask if we can design faster algorithms for the SC problem. If no faster exact algorithms can be designed for the problem, then another question is to ask if we can design faster heuristic algorithms for the problem that works well for real biological data.

Acknowledgments. This work is supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 121608].

References

1. Chen, Z.-Z., Wang, L.: Fast exact algorithms for the closest string and substring problems with application to the planted (L,d)-motif model. IEEE/ACM Transactions on Computational Biology and Bioinformatics 8(5), 1400–1410 (2011)
2. Chen, Z.-Z., Ma, B., Wang, L.: A three-string approach to the closest string problem. Journal of Computer and System Sciences 78, 164–178 (2012)
3. Doi, K., Li, J., Jiang, T.: Minimum Recombinant Haplotype Configuration on Tree Pedigrees. In: Benson, G., Page, R.D.M. (eds.) WABI 2003. LNCS (LNBI), vol. 2812, pp. 339–353. Springer, Heidelberg (2003)
4. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Monogr. Comput. Sci. Springer, New York (1999)
5. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for closest string and related problems. Algorithmica 37, 25–42 (2003)
6. Impagliazzo, R., Paturi, R., Zane, F.: Which problems have strongly exponential complexity? J. Comput. System Sci. 63, 512–530 (2001)
7. Leykin, I., Hao, K., Cheng, J., Meyer, N., Pollak, M.R., Smith, R.J.H., Wong, W.H., Rosenow, C., Li, C.: Comparative linkage analysis and visualization of high-density oligonucleotide snp array data. BMC Genetics 6, 7 (2005)
8. Li, J., Jiang, T.: An exact solution for finding minimum recombinant haplotype configurations on pedigrees with missing data by integer linear programming. In: Proceedings of Symposium on Computational Molecular Biology (RECOMB), pp. 20–29 (2004)
9. Li, J., Jiang, T.: Computing the minimum recombinant haplotype configuration from incomplete genotype data on a pedigree by integer linear programming. Journal of Computational Biology 12(6), 719–739 (2005)
10. Ma, W., Yang, Y., Chen, Z.-Z., Wang, L.: Mutation region detection for closely related individuals without a known pedigree. IEEE/ACM Transactions on Computational Biology and Bioinformatics 9, 499–510 (2012)
11. Ma, B., Sun, X.: More efficient algorithms for closest string and substring problems. SIAM Journal on Computing 39, 1432–1443 (2009)
12. Marx, D.: Closest substring problems with small distances. SIAM Journal on Computing 38, 1382–1410 (2008)
13. Sellick, G., Longman, C., Tolmie, J., Newbury-Ecob, R., Geenhalgh, L., Hughes, S., Whiteford, M., Carrett, C., Houlston, R.: Genomewide linkage searches for mendelian disease loci can be efficiently conducted using high-density snp genotyping arrays. Nucleic Acids Res 32(20), e164 (2004)
14. Wang, L., Zhu, B.: Efficient Algorithms for the Closest String and Distinguishing String Selection Problems. In: Deng, X., Hopcroft, J.E., Xue, J. (eds.) FAW 2009. LNCS, vol. 5598, pp. 261–270. Springer, Heidelberg (2009)
15. Xiao, J., Liu, L., Xia, L., Jiang, T.: Fast elimination of redundant linear equations and reconstruction of recombination-free mendelian inheritance on a pedigree. In: Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 655–664 (2007)
16. Zhao, R., Zhang, N.: A more efficient closest string algorithm. In: Proceedings of the 2nd International Conference on Bioinformatics and Computational Biology (BICoB), pp. 210–215 (2010)

Author Index