28th Annual Symposium on Combinatorial Pattern Matching

CPM 2017, July 4–6, 2017, Warsaw, Poland

Edited by

Juha Kärkkäinen
Jakub Radoszewski
Wojciech Rytter

LIPIcs

Editors

Juha Kärkkäinen
Department of Computer Science
University of Helsinki, Finland
`juha.karkkainen@cs.helsinki.fi`

Jakub Radoszewski
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw, Poland
`jrad@mimuw.edu.pl`

Wojciech Rytter
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw, Poland
`rytter@mimuw.edu.pl`

ACM Classification 1998
E.1 Data Structures, E.2 Data Storage Representations, E.4 Coding and Information Theory, F. Theory of Computation, G.2 Discrete Mathematics, H. Information Systems, I.7 Document and Text Processing

ISBN 978-3-95977-039-2

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-039-2.

Publication date
July, 2017

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.CPM.2017.0

ISBN 978-3-95977-039-2 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

ISSN 1868-8969

http://www.dagstuhl.de/lipics

To all algorithmic stringologists in the world

Contents

Preface
Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter 0:ix

Invited Talks

Wheeler Graphs: Variations on a Theme by Burrows and Wheeler
Giovanni Manzini 1:1–1:1

Recompression of SLPs
Artur Jeż 2:1–2:1

Shortest Superstring
Marcin Mucha 3:1–3:1

Regular Papers

Document Listing on Repetitive Collections with Guaranteed Performance
Gonzalo Navarro 4:1–4:13

Path Queries on Functions
Travis Gagie, Meng He, and Gonzalo Navarro 5:1–5:15

Deterministic Indexing for Packed Strings
Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen 6:1–6:11

Representing the Suffix Tree with the CDAWG
Djamal Belazzougui and Fabio Cunial 7:1–7:13

Position Heaps for Parameterized Strings
Diptarama, Takashi Katsura, Yuhei Otomo, Kazuyuki Narisawa, and Ayumi Shinohara 8:1–8:13

On-Line Pattern Matching on Similar Texts
Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari 9:1–9:14

A Family of Approximation Algorithms for the Maximum Duo-Preservation String Mapping Problem
Bartłomiej Dudek, Paweł Gawrychowski, and Piotr Ostropolski-Nalewaja 10:1–10:14

Revisiting the Parameterized Complexity of Maximum-Duo Preservation String Mapping
Christian Komusiewicz, Mateus de Oliveira Oliveira, and Meirav Zehavi 11:1–11:17

Clique-Based Lower Bounds for Parsing Tree-Adjoining Grammars
Karl Bringmann and Philip Wellnitz 12:1–12:14

Communication and Streaming Complexity of Approximate Pattern Matching
Tatiana Starikovskaya 13:1–13:11

The Longest Filled Common Subsequence Problem
Mauro Castelli, Riccardo Dondi, Giancarlo Mauri, and Italo Zoppis 14:1–14:13

Lempel-Ziv Compression in a Sliding Window
Philip Bille, Patrick Hagge Cording, Johannes Fischer, and Inge Li Gørtz 15:1–15:11

Time-Space Trade-Offs for Lempel-Ziv Compressed Indexing
Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj .. 16:1–16:17

From LZ77 to the Run-Length Encoded Burrows-Wheeler Transform, and Back
Alberto Policriti and Nicola Prezza 17:1–17:10

Longest Common Extensions with Recompression
Tomohiro I 18:1–18:15

Fast and Simple Jumbled Indexing for Binary Run-Length Encoded Strings
Luís Cunha, Simone Dantas, Travis Gagie, Roland Wittler, Luis Kowada, and Jens Stoye 19:1–19:9

Faster STR-IC-LCS Computation via RLE
Keita Kuboi, Yuta Fujishige, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda 20:1–20:12

Gapped Pattern Statistics
Philippe Duchon, Cyril Nicaud, and Carine Pivoteau 21:1–21:12

Computing All Distinct Squares in Linear Time for Integer Alphabets
Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl 22:1–22:18

Palindromic Length in Linear Time
Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur 23:1–23:12

Tight Bounds on the Maximum Number of Shortest Unique Substrings
Takuya Mieno, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda 24:1–24:11

Can We Recover the Cover?
Amihood Amir, Avivit Levy, Moshe Lewenstein, Ronit Lubin, and Benny Porat ... 25:1–25:15

Approximate Cover of Strings
Amihood Amir, Avivit Levy, Ronit Lubin, and Ely Porat 26:1–26:14

Beyond Adjacency Maximization: Scaffold Filling for New String Distances
Laurent Bulteau, Guillaume Fertin, and Christian Komusiewicz 27:1–27:17

On the Weighted Quartet Consensus Problem
Manuel Lafond and Celine Scornavacca 28:1–28:18

Optimal Omnitig Listing for Safe and Complete Contig Assembly
Massimo Cairo, Paul Medvedev, Nidia Obscura Acosta, Romeo Rizzi, and Alexandru I. Tomescu 29:1–29:12

Dynamic Elias-Fano Representation
Giulio Ermanno Pibiri and Rossano Venturini 30:1–30:14

Synergistic Solutions on MultiSets
Jérémy Barbay, Carlos Ochoa, and Srinivasa Rao Satti 31:1–31:14

Preface

The Annual Symposium on Combinatorial Pattern Matching is 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 combinatorial properties of such structures and to exploit these properties in order to achieve more efficient algorithms for the corresponding computational problems. The meeting deals with problems in bioinformatics and computational biology, coding and data compression, combinatorics on words, data mining, information retrieval, natural language processing, pattern discovery, string algorithms, string processing in databases, symbolic computing, and text searching and indexing.

This volume contains the papers presented at the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017) held on July 4-6, 2017 in Warsaw, Poland.

The conference programme included 28 contributed papers and three invited talks by Artur Jeż (University of Wrocław, Poland), Giovanni Manzini (University of Eastern Piedmont and IIT-CNR, Italy), and Marcin Mucha (University of Warsaw, Poland). Contributions of the invited lectures are also included in this volume.

The contributed papers were selected out of 49 submissions, corresponding to an acceptance ratio of about 57%. Each submission received at least three reviews. We thank the members of the Programme Committee and all the additional external reviewers that are listed below for their hard and invaluable work that resulted in an excellent scientific programme.

The Annual Symposium on Combinatorial Pattern Matching started in 1990, and has since then 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 (Ontario, Canada), Pisa, Lille, New York, Palermo, Helsinki, Bad Herrenalb, Moscow, Ischia, and Tel Aviv. From the 3rd to the 26th meeting, all proceedings were published in the LNCS (Lecture Notes in Computer Science) series. The 27th meeting in 2016 was the first to have its proceedings appear in the LIPIcs (Leibniz International Proceedings in Informatics) series, as volume 54.

The whole submission and review process was carried out with the help of the EasyChair conference system. We thank the CPM Steering Committee for supporting Warsaw as the site for CPM 2017 and for their advice and help in different issues. We thank Tomasz Kociumaka and Tomasz Waleń from the University of Warsaw for their extensive involvement in the organisation of the conference and Hanna Bargieł and Monika Goszczycka from Global Congress, Poland for the local arrangements. We would like to thank the Warsaw Center of Mathematics and Computer Science for providing generous financial support to the conference.

Juha Kärkkäinen
Jakub Radoszewski
Wojciech Rytter

Programme Committee

Juha Kärkkäinen (Co-Chair)	University of Helsinki, Finland
Jakub Radoszewski (Co-Chair)	King's College London, UK and University of Warsaw, Poland
Wojciech Rytter (Co-Chair)	University of Warsaw, Poland
Hideo Bannai	Kyushu University, Japan
Philip Bille	Technical University of Denmark, Denmark
Maxime Crochemore	King's College London, UK and Université Paris-Est, France
Gabriele Fici	University of Palermo, Italy
Johannes Fischer	TU Dortmund, Germany
Jan Holub	Czech Technical University in Prague, Czech Republic
Stepan Holub	Charles University in Prague, Czech Republic
Moshe Lewenstein	Bar Ilan University, Israel
Gonzalo Navarro	University of Chile, Chile
Kunsoo Park	Seoul National University, South Korea
Marcin Piątkowski	Nicolaus Copernicus University in Toruń, Poland
Nadia Pisanti	University of Pisa, Italy and Erable Team INRIA, France
Simon Puglisi	University of Helsinki, Finland
Eric Rivals	CNRS and Université de Montpellier, France
Cenk Sahinalp	Indiana University, Bloomington, USA
Rahul Shah	Louisiana State University, USA
Ayumi Shinohara	Tohoku University, Japan
Arseny Shur	Ural Federal University, Russia
Tatiana Starikovskaya	Paris Diderot University, France
Gabriel Valiente	Technical University of Catalonia, Spain

External Reviewers

Badkobeh, Golnaz

Belazzougui, Djamal

Brubach, Brian

Canovas, Rodrigo

Cardona, Gabriel

Chakraborty, Diptarka

Chateau, Annie

Christiansen, Anders Roy

Cording, Patrick Hagge

Didier, Gilles

Faro, Simone

Ferrada, Hector

Freydenberger, Dominik

Gagie, Travis

Ganguly, Arnab

Georgiadis, Loukas

Hon, Wing-Kai

I, Tomohiro

Inenaga, Shunsuke

Jansson, Jesper

Kaniecki, Mariusz

Kempa, Dominik

Kockan, Can

Konow, Roberto

Kosolobov, Dmitry

Kucherov, Gregory

Lecroq, Thierry

Lozano, Antoni

Malikic, Salem

Manea, Florin

Mantaci, Sabrina

Marcus, Shoshana

Messeguer, Xavier

Mikulski, Łukasz

Nekrich, Yakov

Patil, Manish

Pereira, Alberto Ordóñez

Pibiri, Giulio Ermanno

Ponty, Yann

Prezza, Nicola

Raffinot, Mathieu

Rosone, Giovanna

Russo, Luis M. S.

Serna, Maria

Smyczyński, Sebastian

Sugimoto, Shiho

Tabei, Yasuo

Tarhio, Jorma

Tomescu, Alexandru I.

Vojtěchovský, Petr

Weller, Mathias

Zhang, Qin

Zhu, Binhai

Zhu, Kaiyuan

List of Authors

Amir, Amihood (25, 26)
Bannai, Hideo (20, 22, 24)
Barbay, Jérémy (31)
Belazzougui, Djamal (7)
Bille, Philip (6, 15, 16)
Borozdin, Kirill (23)
Bringmann, Karl (12)
Bulteau, Laurent (27)
Cairo, Massimo (29)
Castelli, Mauro (14)
Cording, Patrick Hagge (15)
Cunha, Luís (19)
Cunial, Fabio (7)
Dantas, Simone (19)
Diptarama (8)
Dondi, Riccardo (14)
Duchon, Philippe (21)
Dudek, Bartłomiej (10)
Ettienne, Mikko Berggren (16)
Fertin, Guillaume (27)
Fischer, Johannes (15)
Fujishige, Yuta (20)
Gagie, Travis (5, 19)
Gawrychowski, Paweł (10)
Grossi, Roberto (9)
Gørtz, Inge Li (6, 15, 16)
He, Meng (5)
I, Tomohiro (18)
Iliopoulos, Costas (9)
Inenaga, Shunsuke (20, 22, 24)
Jeż, Artur (2)
Katsura, Takashi (8)
Komusiewicz, Christian (11, 27)
Kosolobov, Dmitry (23)
Kowada, Luis (19)
Kuboi, Keita (20)
Köppl, Dominik (22)
Lafond, Manuel (28)
Levy, Avivit (25, 26)
Lewenstein, Moshe (25)
Liu, Chang (9)
Lubin, Ronit (25, 26)
Manzini, Giovanni (1)
Mauri, Giancarlo (14)
Medvedev, Paul (29)
Mieno, Takuya (24)
Mucha, Marcin (3)
Narisawa, Kazuyuki (8)
Navarro, Gonzalo (4, 5)
Nicaud, Cyril (21)
Obscura Acosta, Nidia (29)
Ochoa, Carlos (31)
Oliveira, Mateus de Oliveira (11)
Ostropolski-Nalewaja, Piotr (10)
Otomo, Yuhei (8)
Pibiri, Giulio Ermanno (30)
Pisanti, Nadia (9)
Pissis, Solon (9)
Pivoteau, Carine (21)
Policriti, Alberto (17)
Porat, Benny (25)
Porat, Ely (26)

Prezza, Nicola (17)

Retha, Ahmad (9)

Rizzi, Romeo (29)

Rosone, Giovanna (9)

Rubinchik, Mikhail (23)

Satti, Srinivasa Rao (31)

Scornavacca, Celine (28)

Shinohara, Ayumi (8)

Shur, Arseny (23)

Skjoldjensen, Frederik Rye (6)

Starikovskaya, Tatiana (13)

Stoye, Jens (19)

Takeda, Masayuki (20, 24)

Tomescu, Alexandru I. (29)

Vayani, Fatima (9)

Venturini, Rossano (30)

Versari, Luca (9)

Vildhøj, Hjalte Wedel (16)

Wellnitz, Philip (12)

Wittler, Roland (19)

Zehavi, Meirav (11)

Zoppis, Italo (14)

Wheeler Graphs: Variations on a Theme by Burrows and Wheeler

Giovanni Manzini

Computer Science Institute, DiSIT, University of Eastern Piedmont, Alessandria, Italy; and
IIT-CNR, Pisa, Italy
`giovanni.manzini@uniupo.it`

Abstract

The famous Burrows-Wheeler Transform was originally defined for single strings but variations have been developed for sets of strings, labelled trees, de Bruijn graphs, alignments, etc. In this talk we propose a unifying view that includes many of these variations and that we hope will simplify the search for more.

Somewhat surprisingly we get our unifying view by considering the Nondeterministic Finite Automata related to different pattern-matching problems. We show that the state graphs associated with these automata have common properties that we summarize with the concept of a Wheeler graph.[1] Using the notion of a Wheeler graph, we show that it is possible to process strings efficiently even if the automaton is nondeterministic. In addition, we show that Wheeler graphs can be compactly represented and traversed using up to three arrays with additional data structures supporting efficient rank and select operations. It turns out that these arrays coincide with, or are substantially equivalent to, the output of many Burrows-Wheeler Transform variants described in the literature.

This is joint work with Travis Gagie and Jouni Sirén.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems, H.3 Information Storage and Retrieval

Keywords and phrases compressed data structures, pattern matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.1

Category Invited Talk

[1] On many occasions Mike Burrows stated that the original idea of the transformation is due to David Wheeler. We therefore decided to name this graph class after this pioneer of computer science.

Recompression of SLPs

Artur Jeż

Institute of Computer Science, University of Wrocław, Wrocław, Poland

Abstract

In this talk I will survey the recompression technique in case of SLPs. The technique is based on applying simple compression operations (replacement of pairs of two different letters by a new letter and replacement of maximal repetition of a letter by a new symbol) to strings represented by SLPs. To this end we modify the SLPs, so that performing such compression operations on SLPs is possible. For instance, when we want to replace ab in the string and SLP has a production $X \to aY$ and the string generated by Y is bw, then we alter the rule of Y so that it generates w and replace Y with bY in all rules. In this way the rule becomes $X \to abY$ and so ab can be replaced, similar operations are defined for the right sides of the nonterminals. As a result, we are interested mostly in the SLP representation rather than the string itself and its combinatorial properties. What we need to control, though, is the size of the SLP. With appropriate choices of substrings to be compressed it can be shown that it stays linear.

The proposed method turned out to be surprisingly efficient and applicable in various scenarios: for instance it can be used to test the equality of SLPs in time $\mathcal{O}(n \log N)$, where n is the size of the SLP and N the length of the generated string; on the other hand it can be used to approximate the smallest SLP for a given string, with the approximation ratio $\mathcal{O}(\log(n/g))$, where n is the length of the string and g the size of the smallest SLP for this string, matching the best known bounds.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Straight Line Programs, smallest grammar problem, compression, processing compressed data, recompression

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.2

Category Invited Talk

Shortest Superstring

Marcin Mucha

University of Warsaw, Warsaw, Poland

Abstract

In the *Shortest Superstring* problem (SS) one has to find a shortest string s containing given strings $s_1, \ldots, s_n$ as substrings. The problem is NP-hard, so a natural question is that of its approximability.

One natural approach to approximately solving SS is the following *GREEDY* heuristic: repeatedly merge two strings with the largest overlap until only a single string is left. This heuristic is conjectured to be a 2-approximation, but even after 30 years since the conjecture has been posed, we are still very far from proving it. The situation is better for non-greedy approximation algorithms, where several approaches yielding 2.5-approximation (and better) are known.

In this talk, we will survey the main results in the area, focusing on the fundamental ideas and intuitions.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases shortest superstring, approximation algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.3

Category Invited Talk

Document Listing on Repetitive Collections with Guaranteed Performance[*]

Gonzalo Navarro

Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Santiago, Chile
`gnavarro@dcc.uchile.cl`

Abstract

We consider document listing on string collections, that is, finding in which strings a given pattern appears. In particular, we focus on repetitive collections: a collection of size N over alphabet $[1, \sigma]$ is composed of D copies of a string of size n, and s single-character edits are applied on the copies. We introduce the first document listing index with size $\tilde{O}(n+s)$, precisely $O((n \lg \sigma + s \lg^2 N) \lg D)$ bits, and with useful worst-case time guarantees: Given a pattern of length m, the index reports the *ndoc* strings where it appears in time $O(m^2 + m \lg N (\lg D + \lg^\epsilon N) \cdot \mathit{ndoc})$, for any constant $\epsilon > 0$.

1998 ACM Subject Classification E.1 Data Structures, E.4 Coding and Information Theory, H.3 Information Storage and Retrieval

Keywords and phrases repetitive string collections, document listing, grammar compression, range minimum queries, succinct data structures

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.4

1 Introduction

Document retrieval on general string collections is an area that has recently attracted attention [24]. On the one hand, it is a natural generalization of the basic Information Retrieval tasks carried out on search engines [1, 4], many of which are also useful on Far East languages, collections of genomes, code repositories, multimedia streams, etc. It also enables phrase queries on natural language texts. On the other hand, it raises a number of algorithmic challenges that are not easily addressed with classical pattern matching approaches.

In this paper we focus on one of the simplest document retrieval problems, *document listing* [22]. Let $\mathcal{D}$ be a collection of D documents of total length N. We want to build an index on $\mathcal{D}$ such that, later, given a search pattern P of length m, we report the identifiers of all the *ndoc* documents where P appears. Given that P may occur $\mathit{occ} \gg \mathit{ndoc}$ times in $\mathcal{D}$, resorting to pattern matching, that is, finding all the *occ* occurrences and then listing the distinct documents where they appear, can be utterly inefficient. Optimal $O(m + \mathit{ndoc})$ time document listing solutions appeared only in 2002 [22], although they use too much space. There are also more recent statistically compressed indices [29, 15] with a small time penalty.

In particular, we are interested in *highly repetitive* string collections [23], which are formed by a few distinct documents and a number of near-copies of those. Such collections arise, for example, when sequencing the genomes of thousands of individuals of a few species, when managing versioned collections of documents like Wikipedia, and in versioned software repositories. Although many of the fastest-growing datasets are indeed repetitive, this is

[*] Supported in part by Fondecyt grant 1-170048 and Basal Funds FB0001, Conicyt, Chile.

an underdeveloped area: most succinct indices for string collections are based on statistical compression, and these fail to exploit repetitiveness [19].

1.1 Our contribution

There are few document listing indices that profit from repetitiveness. A simple model to analyze them is as follows [21, 12, 23]: Assume there is a single document of size n on alphabet $[1, \sigma]$, and $D-1$ copies of it, on which s single-character edits are arbitrarily distributed, forming a collection of size $N \approx nD$. This models, for example, collections of genomes and their single-point mutations. The gold standard to measure space usage on repetitive collections is the size of the *Lempel-Ziv parsing* [20]. If we parse the concatenation of the strings in such a repetitive collection, we obtain at most $z = n/\lg_\sigma n + O(s) \ll N$ phrases. Therefore, while a statistical compressor would require basically $N \lg \sigma$ bits if the base document is incompressible [19], we can aim to reach as little as $O(n \lg \sigma + s \lg N)$ bits by exploiting repetitiveness via Lempel-Ziv compression.

This might be too optimistic for an index, however, as there is no known way to extract substrings efficiently from Lempel-Ziv compressed text. Instead, *grammar compression* allows extracting any text symbol in logarithmic time using $O(r \lg N)$ bits, where r is the size of the grammar [3, 31]. It is possible to obtain a grammar of size $r = O(z \lg(N/z))$ [5, 16], which using standard methods [28] can be tweaked to $r = n/\lg_\sigma N + s \lg N$ under our repetitiveness model. Thus the space we might aim at for indexing is $O(n \lg \sigma + s \lg^2 N)$ bits.

Although they perform reasonably well in practice, none of the preceding structures for document listing on repetitive collections [8, 12] offer good worst-case time guarantees combined with space guarantees that are appropriate for repetitive collections, that is, growing with $n+s$ rather than with N. Those offering search times of the form $O(\mathrm{poly}(m, \lg N) \cdot ndoc)$ require space of the form $O(N/\mathrm{poly}(\lg N))$. In this paper we present the *first index offering good guarantees in space and time*. Namely, our index

1. uses $O((n \lg \sigma + s \lg^2 N) \lg D)$ bits of space, and
2. performs document listing in time $O(m^2 + m \lg N (\lg D + \lg^\epsilon N) \cdot ndoc)$, for any constant $\epsilon > 0$.

That is, our index is an $O(\lg D)$ space factor away from what could be hoped from a grammar-based index. We actually build on a grammar-based document listing index [8] that stores lists of the documents where each nonterminal appears, and strengthen it by rearranging the nonterminals in different orders, following a wavelet tree [13] deployment that guarantees that only $O(m \lg r)$ ranges of lists have to be merged at query time. We do not store the lists themselves in various orders, but just succinct range minimum query (RMQ) data structures [11] that allow implementing document listing on ranges of lists [29]. Those RMQ structures are further compressed because their underlying data has long increasing runs, so the structures are reduced with techniques analogous to those developed for the ILCP data structure [12]. The space reduction brings new issues, however, because we cannot afford storing the underlying RMQ sequences. These problems are circumvented with a new, tailored, technique to extract the distinct documents in a range.

2 Related work

The first optimal-time and linear-space solution to document listing is due to Muthukrishnan [22], who solves the problem in $O(m + ndoc)$ time using an index of $O(N \lg N)$ bits of space. Later solutions [29, 15] improved the space to essentially the statistical entropy of $\mathcal{D}$, at the price of multiplying the times by low-order polylogs of N (e.g., $O(m + \lg N \cdot ndoc)$ time with $O(N)$ bits on top of the entropy). As said, however, statistical entropy does not capture repetitiveness well [19], and thus these solutions are not satisfactory in repetitive collections.

There has been a good deal of work on pattern matching indices for repetitive string collections, building on various principles (see [26, Sec 13.2]). However, there has been little work on document retrieval structures for repetitive string collections.

One precedent is Claude and Munro's index based on grammar compression [8]. It builds on a grammar-based pattern-matching index [10] and adds an *inverted index* that explicitly indicates the documents where each nonterminal appears; this inverted index is also grammar-compressed. To obtain the answer, an unbounded number of those lists of documents must be merged. No relevant worst-case time or space guarantees are offered.

Another precedent is ILCP [12], where it is shown that the longest common prefix array (LCP) of repetitive collections has long increasing runs. Then an index of size bounded by the runs in the suffix array [21] and in the LCP array performs document listing in time $O(\mathsf{search}(m) + \mathsf{lookup}(N) \cdot ndoc)$, where search and lookup are the search and lookup time, respectively, of a run-length compressed suffix array [21]. Yet, there are only average-case bounds for the size of the structure in terms of s: $O(n \lg N + s \lg^2 N)$ bits. A more serious problem is that, to obtain $\mathsf{lookup}(N)$ time per document, a suffix array sampling of $O(N \lg N/\mathsf{lookup}(N))$ bits must be stored.

The last previous work is PDL [12], which stores inverted lists at sampled nodes in the suffix tree of $\mathcal{D}$, and then grammar-compresses the set of inverted lists. For a sampling step b, it requires $O((N/b) \lg N)$ bits plus the (unbounded) space of the inverted lists. Searches that lead to the sampled nodes have their answers precomputed, whereas the others cover a suffix array range of size $O(b)$ and are solved by brute force in time $O(b \cdot \mathsf{lookup}(N))$. Again, the suffix array sampling of $O(N \lg N/\mathsf{lookup}(N))$ bits is necessary.

3 Basic Concepts

3.1 Listing the different elements in a range

Let $A[1, t]$ be an array of integers in $[1, D]$. Muthukrishnan [22] gives a structure that, given a range $[i, j]$, lists all the $ndoc$ distinct elements in $A[i, j]$ in time $O(ndoc)$. He defines an array $C[1, t]$ storing in $C[k]$ the largest position $l < k$ where $A[l] = A[k]$, or $C[k] = 0$ if no such position exists. Note that the leftmost positions of the distinct elements in $A[i, j]$ are exactly those k where $C[k] < i$. He then stores a data structure supporting range-minimum queries (RMQs) on C, $\textsc{rmq}_C(i, j) = \mathrm{argmin}_{i \le k \le j} C[k]$ [11]. Given a range $[i, j]$, he computes $k = \textsc{rmq}_C(i, j)$. If $C[k] < i$, then he reports $A[k]$ and continues recursively on $A[i, k-1]$ and $A[k+1, j]$. Whenever it turns out that $C[k] \ge i$ for an interval $[x, y]$, there are no leftmost occurrences of $A[i, j]$ within $A[x, y]$, so this interval can be abandoned. It is easy to see that the algorithm takes $O(ndoc)$ time and uses $O(t \lg t)$ bits of space; the RMQ structure uses just $2t + o(t)$ bits and answers queries in constant time [11].

Furthermore, the RMQ structure does not even access C, so we can replace C by a bitvector $V[1, D]$ to mark which elements have been reported. We set V initially to all zeros and replace the test $C[k] < i$ by $V[A[k]] = 0$, that is, the value $A[k]$ has not yet been reported (these tests are equivalent only if we recurse left and then right in the interval [24]). If so, we report $A[k]$ and set $V[A[k]] \leftarrow 1$. Overall, we need only $O(t + D)$ bits of space on top of A, and still run in $O(ndoc)$ time [29] (V can be reset to zeros by rerunning the query or through lazy initialization). Hon et al. [15] further reduce the extra space to $o(t)$ bits, yet increasing the time, via sampling the array C.

3.2 Wavelet trees

A wavelet tree [13] is a sequence representation that supports, in particular, two-dimensional orthogonal range queries [6, 25]. Let $(1, y_1), (2, y_2), \ldots, (r, y_r)$ be a sequence of points with $y_i \in [1, r]$, and let $S = y_1 y_2 \ldots y_r$ be the y coordinates in order. The wavelet tree is a perfectly balanced binary tree where each node handles a range of y values. The root handles $[1, r]$. If a node handles $[a, b]$ then its left child handles $[a, \mu]$ and its right child handles $[\mu + 1, b]$, with $\mu = \lfloor (a + b)/2 \rfloor$. The leaves handle individual y values. If a node handles range $[a, b]$, then it represents the subsequence $S_{a,b}$ of y coordinates that belong to $[a, b]$. Thus at each level the strings $S_{a,b}$ form a permutation of S. What is stored for each such node is a bitvector $B_{a,b}$ so that $B_{a,b}[i] = 0$ iff $S_{a,b} \leq \mu$, that is, if that value is handled in the left child of the node. Those bitvectors are provided with support for rank and select queries: $\mathsf{rank}_v(B, i)$ is the number of occurrences of bit v in $B[1, i]$, whereas $\mathsf{select}_v(B, j)$ is the position of the jth occurrence of bit v in B. The wavelet tree has height $\lg r$, and its total space requirement for all the bitvectors $B_{a,b}$ is $r \lg r$ bits. The extra structures for rank and select add $o(r \lg r)$ further bits and support the queries in constant time [7]. With the wavelet tree one can recover any y_i value by tracking it down from the root to a leaf, but let us describe a more general procedure.

Let $[x_1, x_2] \times [y_1, y_2]$ be a query range. The number of points that fall in the range can be counted in $O(\lg r)$ time as follows. We start at the root with the range $S[x_1, x_2] = S_{1,r}[x_1, x_2]$. Then we project the range both left and right, towards $S_{1,\mu}[\mathsf{rank}_0(B_{1,r}, x_1 - 1) + 1, \mathsf{rank}_0(B_{1,r}, x_2)]$ and $S_{\mu+1,r}[\mathsf{rank}_1(B_{1,r}, x_1 - 1) + 1, \mathsf{rank}_1(B_{1,r}, x_2)]$, respectively, with $\mu = \lfloor (r + 1)/2 \rfloor$. If some of the ranges is empty, we stop the recursion on that node. If the interval $[a, b]$ handled by a node is disjoint with $[y_1, y_2]$, we also stop. If the interval $[a, b]$ is included in $[y_1, y_2]$, then all the points in the x range qualify, and we simply sum the length of the range to the count. Otherwise, we keep splitting the ranges recursively. It is well known that the range $[y_1, y_2]$ is covered by $O(\lg r)$ wavelet tree nodes, and that we traverse $O(\lg r)$ nodes to reach them. If we also want to report all the corresponding y values, then instead of counting the points found, we track each one individually towards its leaf, in $O(\lg r)$ time. At the leaves, the y values are sorted, so in particular if they are a permutation of $[1, r]$, we know that the ith left-to-right leaf is the value $y = i$. Thus, extracting the $nocc$ results takes time $O((1 + nocc) \lg r)$.

3.3 Range minimum queries on arrays with runs

Let $A[1, t]$ be an array that can be cut into ρ runs of nondecreasing values. Then it is possible to solve RMQs in $O(\lg \lg t)$ time plus $O(1)$ accesses to A using $O(\rho \lg(t/\rho))$ bits. The idea is that the possible minima (breaking ties in favor of the leftmost) in $A[i, j]$ are either $A[i]$ or the positions where runs start in the range. Then, we can use a sparse bitvector $M[1, t]$ marking with $M[k] = 1$ the run heads. We also define an array $A'[1, \rho]$, so that if $M[k] = 1$ then $A'[\mathsf{rank}_1(M, k)] = A[k]$. We do not store A', but just an RMQ structure on it. Hence, the minimum of the run heads in $A[i, j]$ can be found by computing the range of run heads involved, $i' = \mathsf{rank}_1(M, i - 1) + 1$ and $j' = \mathsf{rank}_1(M, j)$, then finding the smallest value among them in A' with $k' = \textsc{rmq}_{A'}(i', j')$, and mapping it back to A with $k = \mathsf{select}_1(M, k')$. Finally, the RMQ answer is either $A[i]$ or $A[k]$, so we access A twice to compare them.

This idea was used by Gagie et al. [12, Sec 3.2] for runs of equal values, but it works verbatim for runs of nondecreasing values. They show how to store M in $\rho \lg(t/\rho) + O(\rho)$ bits so that it solves rank in $O(\lg \lg t)$ time and select in $O(1)$ time, by enriching a sparse bitvector representation [27]. This dominates the space and time of the whole structure.

The idea was used even before by Barbay et al. [2, Thm. 2], for runs of nondecreasing values. They represented M using $\rho \lg(t/\rho) + O(\rho) + o(t)$ bits so that the $O(\lg \lg t)$ time becomes $O(1)$, but we are not be able to afford the $o(t)$ extra bits in this paper.

3.4 Grammar compression

Let $T[1, N]$ be a sequence of symbols over alphabet $[1, \sigma]$. Grammar compressing T means finding a context-free grammar that generates T and only T. The grammar can then be used as a substitute for T, which provides good compression when T is repetitive. We are interested, for simplicity, in grammars in Chomsky normal form, where the rules are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B, and C are nonterminals and $a \in [1, \sigma]$ is a terminal symbol. For every grammar, there is a proportionally sized grammar in this form.

A Lempel-Ziv parse [20] of T cuts T into z phrases, so that each phrase $T[i, j]$ appears earlier in $T[i', j']$, with $i' < i$. It is known that the smallest grammar generating T must have at least z rules [28, 5], and that it is possible to convert a Lempel-Ziv parse into a grammar with $r = O(z \lg(N/z))$ rules [28, 5, 30, 17, 18]. Furthermore, such grammars can be balanced, that is, the parse tree is of height $O(\lg N)$. By storing the length of the string to which every nonterminal expands, it is easy to access any substring $T[i, j]$ from its compressed representation in time $O(j - i + \lg N)$ by tracking down the range in the parse tree. This can be done even on an unbalanced grammar [3]. The total space used by this representation, with a grammar of r rules, is $O(r \lg N)$ bits.

3.5 Grammar-based indexing

The pattern-matching index of Claude and Navarro [9] builds on a grammar in Chomsky normal form that generates a text $T[1, N]$, with $r + 1$ rules. Let $s(A)$ be the string generated by nonterminal A. Then they collect the strings $s(A)$ for all those nonterminals, except the initial symbol S. Let $C_1, \ldots, C_r$ be the nonterminals sorted lexicographically by $s(A)$ and let $B_1, \ldots, B_r$ be the nonterminals sorted lexicographically by the reverse strings, $s(A)^{rev}$. They create a set of points in $[1, r] \times [1, r]$ so that (i, j) is a point (corresponding to nonterminal A) if the rule that defines A is $A \rightarrow B_i C_j$. Those points are stored in a wavelet tree.

To search for a pattern $P[1, m]$, they first find the primary occurrences, that is, those that appear when B is concatenated with C in a rule $A \rightarrow BC$. The secondary occurrences, which appear when A is used elsewhere, are found in a way that does not matter for this paper. To find the primary occurrences, they cut P into two nonempty parts $P = P_1 P_2$, in the $m - 1$ possible ways. For each cut, they binary search for P_1^{rev} in the sorted set $s(B_1)^{rev}, \ldots, s(B_r)^{rev}$ and for P_2 in the sorted set $s(C_1), \ldots, s(C_r)$. Let $[x_1, x_2]$ be the interval obtained for P_1 and $[y_1, y_2]$ the one obtained for P_2. Then all the points in $[x_1, x_2] \times [y_1, y_2]$, for all the $m - 1$ partitions of P, are the primary occurrences.

To search for P_1^{rev} or for P_2, the grammar is used to extract the required substrings of T in time $O(m + \lg N)$, so the overall search time to find the *nocc* primary occurrences is $O(m \lg r (m + \lg N) + \lg r \cdot nocc)$. The space used by the structure is $O(r \lg N)$ bits. Within this space one can store Patricia trees on the strings $s(B_i^{rev})$ and $s(C_i)$, to speed up binary searches and reduce the time to $O(m(m+\lg N)+\lg r \cdot nocc)$. Also, one can use the structure of Gasieniec et al. [14] that, within $O(r \lg N)$ further bits, allows extracting any prefix/suffix of any nonterminal in constant time per symbol (see also [10]). Since in our search we only access prefixes/suffixes of whole nonterminals, this further reduces the time to $O(m^2 + \lg r \cdot nocc)$.

Claude and Munro [8] extend this structure to support document listing on a collection $\mathcal{D}$ of D string documents, which are concatenated into a text $T[1, N]$. To each nonterminal

A they associate the increasing list $\ell(A)$ of the identifiers of the documents (integers in $[1, D]$) where A appears. To perform document listing, they find all the primary occurrences $A \to BC$ of all the partitions of P, and merge their lists. There is no useful worst-case time bound for this operation other than $O(nocc \cdot ndoc)$, where $nocc$ can be much larger than $ndoc$. To reduce space, they also grammar-compress the sequence of all the r lists $\ell(A)$. They also give no worst-case space bound for the compressed lists (other than $O(rD \lg D)$ bits).

4 Our Document Listing Index

We build on the basic structure of Claude and Munro [8]. Our main idea is to take advantage of the fact that the *nocc* primary occurrences to detect in Section 3.5 are found as points in the two-dimensional structure, along $O(\lg r)$ ranges within wavelet tree nodes (recall Section 3.2) for each partition of P. Instead of retrieving the *nocc* individual lists, decompressing and merging them [8], we will use the techniques to extract the distinct elements of a range seen in Section 3.1. This will drastically reduce the amount of merging necessary, and will provide useful upper bounds on the document listing time.

4.1 Structure

We store the grammar of T in a way that it allows direct access for pattern searches, as well as the wavelet tree for the points (B_i, C_j), the Patricia trees, and extraction of prefixes/suffixes of nonterminals, all in $O(r \lg N)$ bits.

Consider any sequence $S_{a,b}[1, q]$ at a wavelet tree node handling the range $[a, b]$ (recall that those sequences are not explicitly stored). Each element $S_{a,b}[k] = A_k$ corresponds to a point (i, j) associated with a nonterminal $A_k \to B_i C_j$. Then let $L_{a,b} = \ell(A_1) \cdot \ell(A_2) \cdots \ell(A_q)$ be the concatenation of the inverted lists associated with the nonterminals in $S_{a,b}$, and let $M_{a,b} = 10^{|\ell(A_1)|-1} 10^{|\ell(A_2)|-1} \ldots 10^{|\ell(A_q)|-1}$ mark where each list begins in $L_{a,b}$. Now let $C_{a,b}$ be the C-array corresponding to $L_{a,b}$, as described in Section 3.1. As in that section, we do not store $L_{a,b}$ nor $C_{a,b}$, but just the RMQ structure on $C_{a,b}$, which together with $M_{a,b}$ will be used to retrieve the unique documents in a range $S_{a,b}[i, j]$.

Since $M_{a,b}$ has only r 1s out of (at most) rD bits across all the wavelet tree nodes of the same level, it can be stored with $O(r \lg D)$ bits per level [27], and $O(r \lg r \lg D)$ bits overall. On the other hand, as we will show, $C_{a,b}$ is formed by a few increasing runs, say ρ across the wavelet tree nodes of the same level, and therefore we represent its RMQ structure using the technique of Section 3.3. The total space used by those RMQ structures is then $O(\rho \lg r \lg(rD/\rho))$ bits.

Finally, we store the explicit lists $\ell(B_i)$ aligned to the wavelet tree leaves, so that the list of any element in any sequence $S_{a,b}$ is reached in $O(\lg r)$ time by tracking down the element. Those lists, of maximum total length rD, are grammar-compressed as well, just as in the basic scheme [8]. If the grammar has r' rules, then the total compressed size is $O(r' \lg(rD))$ bits to allow for direct access in $O(\lg(rD))$ time, see Section 3.4.

In total, our structure uses $O(r \lg N + r \lg r \lg D + \rho \lg r \lg(rD/\rho) + r' \lg(rD))$ bits.

4.2 Document listing

A document listing query proceeds as follows. We cut P in the $m - 1$ possible ways, and for each way identify the $O(\lg r)$ wavelet tree nodes (and ranges) where the desired points lie. Overall, we have $O(m \lg r)$ ranges and need to take the union of the inverted lists of all the points inside those ranges. We extract the distinct documents in each range and then compute their union. If a range has only one element, then we can track it to the leaves, where its list $\ell(\cdot)$ is stored, and recover it by decompressing the whole list.

Otherwise, we use in principle the document listing technique of Section 3.1. Let $S_{a,b}[i,j]$ be a range from where to obtain the distinct documents. We compute $i' = \mathsf{select}_1(M_{a,b}, i)$ and $j' = \mathsf{select}_1(M_{a,b}, j+1) - 1$, and obtain the distinct elements in $L_{a,b}[i',j']$, by using RMQs on $C_{a,b}[i',j']$. Recall that, as in Section 3.3, we use a run-length compressed RMQ structure on $C_{a,b}$. With this arrangement, every RMQ operation takes time $O(\lg\lg(rD))$ plus the time to accesses two cells in $C_{a,b}$. Those accesses are made to compare a run head with the leftmost element of the query interval, $C_{a,b}[i']$. The problem is that we have not represented the cells of $C_{a,b}$, and cannot easily compute them on the fly.

Barbay et al. [2, Thm. 3] give a sophisticated representation that determines the position of the minimum in $C_{a,b}[i',j']$ without the need to perform the two accesses on $C_{a,b}$. They need $\rho\lg(rD) + \rho\lg(rD/\rho) + O(\rho) + o(rD)$ bits, which unfortunately is too high for us[1].

Instead, we modify the way the distinct elements are obtained, so that comparing the two cells of $C_{a,b}$ is unnecessary. In the same spirit of Sadakane's solution (see Section 3.1) we use a bitvector $V[1,D]$ where we mark the documents already reported. Given a range $S_{a,b}[i,j] = A_i \ldots A_j$, we first track A_i down the wavelet tree, recover and decompress its list $\ell(A_i)$, and mark all of its documents in V. Note that all the documents in the list $\ell(\cdot)$ are different. Now we do the same with A_{i+1}, decompressing $\ell(A_{i+1})$ left to right and marking the documents in V, and so on, until we decompress a document $\ell(A_{i+d})[k]$ that is already marked in V. Only now we use the RMQ technique of Section 3.3 on the interval $C_{a,b}[i',j']$, where $i' = \mathsf{select}_1(M_{a,b}, i+d) - 1 + k$ and $j' = \mathsf{select}_1(M_{a,b}, j+1) - 1$, to obtain the next document to report. This technique, as explained, yields two candidates: one is $L_{a,b}[i'] = \ell(A_{i+d})[k]$ itself, and the other is some run head $L_{a,b}[k']$ whose identity we can obtain from the wavelet tree leaf. But we know that $L_{a,b}[i']$ was already reported, so we act as if the RMQ was always $L_{a,b}[k']$: If the RMQ answer was $L_{a,b}[i']$ then, since it is already reported, we should stop. But in this case, $L_{a,b}[k']$ is also already reported and we do stop anyway. Hence, if $L_{a,b}[k']$ is already reported we stop, and otherwise we report it and continue recursively on the intervals $C_{a,b}[i', k'-1]$ and $C_{a,b}[k'+1, j']$. On the first, we can continue directly, as we still know that $L_{a,b}[i']$ is already reported. On the second interval, instead, we must restore the invariant that the leftmost element was already reported. So we find out with M the list and position $\ell(A_t)[u]$ corresponding to $C_{a,b}[k'+1]$ (i.e., $t = \mathsf{rank}_1(M_{a,b}, k'+1)$ and $u = k'+1-\mathsf{select}_1(M,t)+1$), track A_t down to its leaf in the wavelet tree, and traverse $\ell(A_t)$ from position u onwards, reporting documents until finding one that has been reported. The correctness of this document listing algorithm is proved in Appendix A.

The $m-1$ searches for partitions of P take time $O(m^2)$. In the worst case, extracting each distinct document in the range requires an RMQ computation without access to $C_{a,b}$ ($O(\lg\lg(rD))$ time), tracking an element down the wavelet tree ($O(\lg r)$ time), and extracting an element from its grammar-compressed list $\ell(\cdot)$ ($O(\lg(rD)$ time). This adds up to $O(\lg(rD))$ time per document extracted in a range. In the worst case, however, the same documents are extracted over and over in all the $O(m\lg r)$ ranges, and therefore the final search time is $O(m^2 + m\lg r\lg(rD)\cdot ndoc)$.

[1] Even if we get rid of the $o(rD)$ component, the $\rho\lg(rD)$ term becomes $O(s\lg^3 N)$ in the final space, which is larger than what we manage to obtain. Also, using it does not make our solution faster.

5 Analysis in a Repetitive Scenario

Our structure uses $O(r \lg N + r \lg r \lg D + \rho \lg r \lg(rD/\rho) + r' \lg(rD))$ bits, and performs document listing in time $O(m^2 + m \lg r \lg(rD) \cdot ndoc)$. We now specialize those formulas under our repetitiveness model. Note that our index works on any string collection; we use the simplified model of the $D-1$ copies of a single document of length n, plus the s edits, to obtain analytical results that are easy to interpret in terms of repetitiveness. We also assume a particular strategy to generate the grammars to show that it is possible to obtain the complexities we give; the actual index may use more sophisticated ones.

5.1 Space

We assume $s \geq D-1$, since otherwise there will be identical documents, and this is easily reduced to a smaller collection with multiple identifiers per document. The documents are concatenated into $T[1,N]$, where $N \leq nD + s$. Let us make our grammar for T contain the $N^{1/3}$ nonterminals that generate all the strings of length $\frac{1}{3}\lg_\sigma N$. Then it replaces the first document with $O(n/\lg_\sigma N)$ such nonterminals, and builds a balanced parse tree of height $h = O(\lg n)$ on top of them, with nonterminal symbol S at the root. On the copies, it first covers them with $D-1$ copies of S. Now, for each edit that occurs on a copy, let $A_1, \ldots, A_h$ be the nonterminals from the leaf (where the edit is applied) to the root $A_h = S$. We create new nonterminals $A'_1, \ldots, A'_{h'}$ so that $h' \leq h+1$ and $A'_{h'} = S'$ generates the modified document. All the other nonterminals can be reused. Therefore, the maximum height h' of the final nonterminal S' rooting a modified document is $O(\lg(n+s))$, which is reached when many of the edits apply to a single copy. The final grammar size is then $r = O(N^{1/3} + n/\lg_\sigma N + s\lg(n+s)) = O(n/\lg_\sigma N + s \lg N)$, where we used that either n or s is $\Omega(\sqrt{N})$ because $N \leq nD + s \leq n(s+1) + s$. Once all the edits are applied, we add a balanced tree on top of those r symbols, which asymptotically does not change r (we may also avoid this final tree and access the documents individually, since our accesses never cross document borders).

Let us now bound ρ. If there are no edits, then every nonterminal appears in all the documents, so all the lists are of the form $\ell(A) = 1, 2, \ldots, D$. Therefore, all the corresponding C values are $C[k] = k - D$, and C has just one nondecreasing run (the first D values are 0, and thus included in the run too). Let us consider the effect of an edit operation at some document d. When we update the upward path $A_1, A_2, \ldots, A_h$ and create nonterminals $A'_1, A'_2, \ldots, A'_{h'}$ to reflect the edit, document d may disappear from all the lists $\ell(A_i)$. Each of those (up to) h' disappeared documents produces a change in C, where the cell that pointed to the disappeared position now points earlier, and this may break one run. There are other h' updates due to the creation of the lists for the nonterminals A'_i. Overall, array C undergoes $O(\lg N)$ run breaks per edit, and therefore it has a total of $\rho = O(s \lg N)$ runs.

The analysis of r' is analogous. When there are no edits and $\ell(A) = 1, 2, \ldots, D$ for all nonterminals A, we can represent the lists with a grammar of $O(D)$ symbols generating one list from nonterminal U, and then $r-1$ copies of U. Now, an edit in a document d that removes d from the lists of nonterminals $A_1, \ldots, A_h$ produces $O(\lg N)$ edits in the lists $\ell(A_1), \ldots, \ell(A_h)$ (and new lists $\ell(A'_i)$ as well). As done for the text, the grammar needs to add $O(\lg D)$ nonterminals to modify the copy of U of each list $\ell(A_i)$, from the point where d disappears to the root U. The new lists $\ell(A'_i)$ also fit within the same space. Therefore, the final grammar is of size $r' = O(D + s \lg N \lg D) = O(s \lg N \lg D)$. Instead of adding a balanced grammar tree over the r resulting nonterminals U', we retain direct pointers to those roots. As a result, the lists, of maximum length D, need $O(r' \lg D)$ bits and can be accessed in time $O(\lg D)$. From the wavelet tree, however, we still have to pay also the $O(\lg r)$ time needed to identify the list to access.

Therefore, the total size of the index can be expressed as follows. The $O(r \lg r \lg D)$ bits coming from the sparse bitvectors M, is $O(r \lg N \lg D)$ (since $\lg r = \Theta(\lg(ns)) = \Theta(\lg N)$), and thus it is $O(n \lg \sigma \lg D + s \lg^2 N \lg D)$. This subsumes the $O(r \lg N)$ bits of the grammar and the wavelet tree. The $O(\rho \lg r \lg(rD/\rho))$ bits of the structures C are monotonically increasing with ρ, so since $\rho = s \lg N \leq r$, we can upper bound it by replacing ρ with r, obtaining $O(r \lg r \lg D)$ as in the space for M. Finally, the $O(r' \lg D)$ bits of the explicit inverted lists are $O(s \lg N \lg^2 D)$. Overall, the structures add up to $O((n \lg \sigma + s \lg^2 N) \lg D)$ bits. Note that we can also analyze the space required by Claude and Munro's structure [8], which is $O(r \lg N)$ bits plus the inverted lists, $O(n \lg \sigma + s \lg N (\lg N + \lg^2 D))$ bits. Although smaller than ours, their search time has no useful bounds.

5.2 Time

Our search time is $O(m^2 + m \lg r \lg(rD) \cdot ndoc) = O(m^2 + m \lg^2 N \cdot ndoc)$. The $O(\lg(rD))$ cost corresponds to accessing a list $\ell(A)$ from the wavelet tree, and includes the $O(\lg r)$ time to reach the leaf and the $O(\lg D)$ time to access a position in the grammar-compressed list. It is possible to reduce the $O(\lg r)$ wavelet tree time by spending more space. The trick is to track the positions upwards to the root, not downwards to the leaves, and associate the lists $\ell(A)$ aligned to the root order. It is possible to reach the root position of a symbol in time $O((1/\epsilon) \lg^\epsilon r)$ by using $O((1/\epsilon) r \lg r)$ bits [6, 25], for any $\epsilon > 0$. By using a constant ϵ we obtain our main result.

▶ **Theorem 1.** *Let collection $\mathcal{D}$, of total size N, be formed by an initial document of length n plus $D-1$ copies of it, with s single-character edit operations applied on the copies. Then $\mathcal{D}$ can be represented within $O((n \lg \sigma + s \lg^2 N) \lg D)$ bits, so that the ndoc documents where a pattern of length m appears can be listed in time $O(m^2 + m \lg N (\lg D + \lg^\epsilon N) \cdot ndoc)$, for any constant $\epsilon > 0$.*

We can also obtain other tradeoffs. For example, with $\epsilon = 1/\lg\lg r$ we obtain $O((n \lg \sigma + s \lg^2 N)(\lg D + \lg\lg N))$ bits of space and $O(m^2 + m \lg N (\lg D + \lg\lg N) \cdot ndoc)$ search time.

6 Conclusions

We have presented the first document listing index with worst-case space and time guarantees that are useful for repetitive collections. On a collection of size N formed by an initial document of length n and $D-1$ copies it, with s single-character edits applied on the copies, our index uses $O((n \lg \sigma + s \lg^2 N) \lg D)$ bits and lists the *ndoc* documents where a pattern of length m appears in time $O(m^2 + m \lg N (\lg D + \lg^\epsilon N) \cdot ndoc)$, for any constant $\epsilon > 0$. We also prove a slightly lower space bound on a previous index that had not been analyzed [8], but which has no useful worst-case time bounds for listing.

The space of our index is an $O(\lg D)$ factor away from what can be expected from a grammar-based index. This is the price paid for storing the inverted lists of the nonterminals. An important question is whether this space factor can be removed, that is, if the inverted lists can be represented within the grammar-compressed size of the text itself.

Another interesting question is whether there exists an index (or a better analysis of this index) whose space and time can be bounded on more general repetitiveness measures of the collection, for example in terms of the number z of Lempel-Ziv phrases into which it can be parsed. In our model it holds $z \leq n/\lg_\sigma n + O(s)$, but other kinds of plausible repetitive collections have $s \gg z$, for example if the edits apply to ranges of documents, or if they involve blocks of text inserted, deleted, or moved around. Typical grammar-based pattern matching indices [9, 10] require $O(r \lg N) = O(z \lg^2 N)$ bits in general; it would be good to obtain the same in the document-listing grammar-based indices.

Finally, there is the question of how much of the theoretical improvements over previous work [8] can be translated into practice. This is also a subject of future work. On one hand, our upper bounds are utterly pessimistic when they assume that the same documents will be reported $O(m \lg r)$ times; the average case should be much better. On the other hand, practical improvements are possible over the basic theoretical ideas presented, which should allow us effectively avoid the cases where the previous index deviates significantly from our worst-case time guarantees, without ruining the cases where it performs well. For example, we can list the documents by brute force when the wavelet tree ranges are short, and use the document listing algorithm only on the long ones, where it is worth applying.

References

1 Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval: The Concepts and Technology behind Search*. Addison-Wesley Professional, 2011. URL: `http://www.mir2ed.org/`.

2 Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theor. Comput. Sci.*, 459:26–41, 2012. `doi:10.1016/j.tcs.2012.08.010`.

3 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015. `doi:10.1137/130936889`.

4 Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010. URL: `http://mitpress.mit.edu/books/information-retrieval`.

5 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005. `doi:10.1109/TIT.2005.850116`.

6 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988. `doi:10.1137/0217026`.

7 David R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996. URL: `http://hdl.handle.net/10012/64`.

8 Francisco Claude and J. Ian Munro. Document listing on versioned documents. In Oren Kurland, Moshe Lewenstein, and Ely Porat, editors, *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE 2013)*, volume 8214 of *LNCS*, pages 72–83. Springer, 2013. `doi:10.1007/978-3-319-02432-5_12`.

9 Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundam. Inform.*, 111(3):313–337, 2010. `doi:10.3233/FI-2011-565`.

10 Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE 2012)*, volume 7608 of *LNCS*, pages 180–192. Springer, 2012. `doi:10.1007/978-3-642-34109-0_19`.

11 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. `doi:10.1137/090779759`.

12 Travis Gagie, Kalle Karhu, Gonzalo Navarro, Simon J. Puglisi, and Jouni Sirén. Document listing on repetitive collections. In Johannes Fischer and Peter Sanders, editors, *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *LNCS*, pages 107–119. Springer, 2013. `doi:10.1007/978-3-642-38905-4_12`.

13 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In Martin Farach-Colton, editor, *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 841–850. ACM/SIAM, 2003. URL: `http://dl.acm.org/citation.cfm?id=644108.644250`.

14 Leszek Gąsieniec, Roman M. Kolpakov, Igor Potapov, and Paul Sant. Real-time traversal in grammar-based compressed files. In James A. Storer and Martin Cohn, editors, *Proceedings of the 2005 Data Compression Conference (DCC 2005)*, page 458. IEEE Computer Society, 2005. `doi:10.1109/DCC.2005.78`.

15 Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Space-efficient framework for top-k string retrieval problems. In Daniel A. Spielman, editor, *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2009)*, pages 713–722. IEEE Computer Society, 2009. `doi:10.1109/FOCS.2009.19`.

16 Danny Hucke, Markus Lohrey, and Carl Philipp Reh. The smallest grammar problem revisited. In Shunsuke Inenaga, Kunihiko Sadakane, and Tetsuya Sakai, editors, *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, volume 9954 of *LNCS*, pages 35–49. Springer, 2016. `doi:10.1007/978-3-319-46049-9_4`.

17 Artur Jeż. Approximation of grammar-based compression via recompression. *Theor. Comput. Sci.*, 592:115–134, 2015. `doi:10.1016/j.tcs.2015.05.027`.

18 Artur Jeż. A really simple approximation of smallest grammar. *Theor. Comput. Sci.*, 616:141–150, 2016. `doi:10.1016/j.tcs.2015.12.032`.

19 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013. `doi:10.1016/j.tcs.2012.02.006`.

20 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Trans. Inf. Theory*, 22(1):75–81, 1976. `doi:10.1109/TIT.1976.1055501`.

21 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010. `doi:10.1089/cmb.2009.0169`.

22 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In David Eppstein, editor, *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 657–666. ACM/SIAM, 2002. URL: `http://dl.acm.org/citation.cfm?id=545381.545469`.

23 Gonzalo Navarro. Indexing highly repetitive collections. In S. Arumugam and W. F. Smyth, editors, *Proceedings of the 23rd International Workshop on Combinatorial Algorithms (IWOCA 2012)*, volume 7643 of *LNCS*, pages 274–279. Springer, 2012. `doi:10.1007/978-3-642-35926-2_29`.

24 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):52:1–52:47, 2014. `doi:10.1145/2535933`.

25 Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014. `doi:10.1016/j.jda.2013.07.004`.

26 Gonzalo Navarro. *Compact Data Structures: A practical approach*. Cambridge University Press, 2016. `doi:10.1017/CBO9781316588284`.

27 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In David Applegate and Gerth Stølting Brodal, editors, *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*. SIAM, 2007. `doi:10.1137/1.9781611972870.6`.

28 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

29 Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007. doi:10.1016/j.jda.2006.03.011.

30 Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005. doi:10.1016/j.jda.2004.08.016.

31 Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In Johannes Fischer and Peter Sanders, editors, *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *LNCS*, pages 247–258. Springer, 2013. doi:10.1007/978-3-642-38905-4_24.

A Proof of Correctness

We prove that our new document listing algorithm is correct. We remind that the algorithm proceeds as follows, to find the distinct elements in $A[sp, ep]$. It starts recursively with $[i, j] = [sp, ep]$ and remembers the documents that have already been reported, globally. To process interval $[i, j]$, it considers $A[i], A[i + 1], \ldots$ until finding an already reported element at $A[d]$. Then it finds the minimum $C[k]$ in $C[d, j]$. If $A[k]$ has been reported already, it stops; otherwise it reports $A[k]$ and proceeds recursively in $A[d, k - 1]$ and $A[k + 1, j]$, in this order. (The algorithm does this without noticing the cases where $k = d$, but this is correct, as explained in Section 4.2).

▶ **Lemma 2.** *The described algorithm reports the ndoc distinct elements in $A[sp, ep]$ in $O(ndoc)$ steps.*

Proof. We prove that the algorithm reports the leftmost occurrence in $A[sp, ep]$ of each distinct element. In particular, we prove by induction on i (and, upon ties, on $j - i$) that, when run on any subrange $[i, j]$ of $[sp, ep]$, (1) every leftmost occurrence in $A[sp, i - 1]$ is already reported before processing $[i, j]$ and (2) every leftmost occurrence in $A[sp, j]$ is reported after processing $[i, j]$. Invariant (1) holds for $[i, j] = [sp, ep]$, and the recursive procedure always produces intervals with nondecreasing values of i. Then the base case $i = j$ is trivial: the algorithm checks $A[i]$ and reports it if it was not reported before. On a larger interval $[i, j]$, the algorithm first reports $d - i$ occurrences of distinct elements in $A[i, d - 1]$. Since these were not reported before, by invariant (1) they must be leftmost occurrences in $[sp, ep]$, and thus after doing this the invariant (1) holds for any range starting at d.

Now, we compute the position k with minimum $C[k]$ in $C[d, j]$. Note that $A[k]$ is a leftmost occurrence iff $C[k] < sp$. In this case, it has not been reported before and thus it must be reported by the algorithm. The algorithm then recurses on $A[d, k - 1]$, reports $A[k]$, and finally recurses on $A[k + 1, j]$.[2] Since those subintervals are inside $[i, j]$, we can apply induction. In the call on $A[d, k - 1]$, the invariant (1) holds and thus by induction we have that after the call the invariant (2) holds, so all the leftmost occurrences in $A[sp, k - 1] = A[sp, d - 1] \cdot A[d, k - 1]$ have been reported. After we report $A[k]$ too, the invariant (1) also holds for the call on $A[k + 1, j]$, so by induction all the leftmost occurrences in $A[sp, j]$ have been reported when the call returns.

In case $C[k] \geq sp$, $A[k]$ is not a leftmost occurrence in $A[sp, ep]$, and moreover there are no leftmost occurrences in $A[d, j]$, so we can stop since all the leftmost occurrences in

[2] Since $A[k]$ does not appear in $A[d, k - 1]$, the algorithm also works if $A[k]$ is reported before the recursive calls, which makes it real-time.

$A[sp,j] = A[sp,d-1] \cdot A[d,j]$ are already reported. Indeed, if the leftmost occurrence of $A[k]$ is in $A[sp,d-1]$, then we had already reported it by invariant (1), so the algorithm stops.

Then the algorithm is correct. As for the time, clearly the algorithm never reports the same element twice. The sequential part reports $d-i$ documents in time $O(d-i+1)$. The extra $O(1)$ can be charged to the caller, as well as the $O(1)$ cost of the subranges that do not produce any result. Each calling procedure reports at least one element $A[k]$, so it can absorb those $O(1)$ costs, for a total cost of $O(ndoc)$. ◀

Path Queries on Functions*

Travis Gagie[1], Meng He[2], and Gonzalo Navarro[3]

1 **CeBiB – Center for Biotechnology and Bioengineering, University of Chile, Santiago, Chile; and**
School of Computer Science and Telecommunications, Diego Portales University, Santiago, Chile
`travis.gagie@gmail.com`

2 **Faculty of Computer Science, Dalhousie University, Halifax, Canada**
`mhe@cs.dal.ca`

3 **CeBiB – Center for Biotechnology and Bioengineering and Department of Computer Science, University of Chile, Chile**
`gnavarro@dcc.uchile.cl`

Abstract

Let $f : [1..n] \to [1..n]$ be a function, and $\ell : [1..n] \to [1..\sigma]$ indicate a label assigned to each element of the domain. We design several compact data structures that answer various queries on the labels of *paths* in f. For example, we can find the minimum label in $f^k(i)$ for a given i and any $k \geq 0$ in a given range $[k_1..k_2]$, using $n \lg n + O(n)$ bits, or the minimum label in $f^{-k}(i)$ for a given i and $k > 0$, using $2n \lg n + O(n)$ bits, both in time $O(\lg n / \lg \lg n)$. By using $n \lg \sigma + o(n \lg \sigma)$ further bits, we can also count, within the same time, the number of labels within a range, and report each element with such labels in $O(1 + \lg \sigma / \lg \lg n)$ additional time. Several other possible queries are considered, such as top-t queries and τ-majorities.

1998 ACM Subject Classification E.1 Data Structures, E.4 Coding and Information Theory

Keywords and phrases succinct data structures, integer functions, range queries, trees and permutations

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.5

1 Introduction

We focus on the representation of *integer functions* where the domain coincides with the image, $f : [1..n] \to [1..n]$. This kind of functions were studied by Munro et al. [10], who focused on how to compute efficiently *powers* of functions. A positive power is $f^k(i)$, for a given $i \in [1..n]$ and $k \geq 0$, whereas a negative power returns all the elements in the set $f^{-k}(i) = \{j, f^k(j) = i\}$, for a given $i \in [1..n]$ and $k > 0$. They show that f can be represented within $n \lg n + O(n)$ bits so that any positive power $f^k(i)$ is computed in time $O(1)$, and any negative power $f^{-k}(i)$ is listed in time $O(|f^{-k}(i)|)$. The main idea of Munro et al. is summarized in their metaphor "functions are just hairy permutations", in the sense that the directed graph $G(V,E)$ where $V = [1..n]$ and $E = \{(i, f(i)), i \in [1..n]\}$ has the form of a set of cycles, where a tree may sprout from each node in each cycle (permutations, instead, are decomposed into just a set of cycles).

In this article we go beyond the goal of simply listing the elements of powers of permutations. Instead, we seek to compute *summaries* on the elements belonging to *paths* in G. We consider three kinds of paths P:

* Supported with Basal Funds FB0001, Conicyt, Chile, and by Fondecyt grant 1-140796, Chile.

1. A *positive path* is formed by the distinct elements in $f^{k_1..k_2}(i) = \{f^k(i), k \in [k_1..k_2]\}$ for a given $i \in [1..n]$ and $0 \leq k_1 \leq k_2$.
2. A *negative path* is formed by the distinct elements in $f^{-k_1..-k_2}(i) = \{j \in f^{-k}(i), k \in [k_1..k_2]\}$ for a given $i \in [1..n]$ and $0 < k_1 \leq k_2$.
3. A *negative path point* is a particular case of a negative path, formed by the elements in $f^{-k}(i)$, for a given $i \in [1..n]$ and $k > 0$.

In turn, we consider various kinds of summarizations. For maximum generality, let us assume that the elements are assigned a label $\ell : [1..n] \rightarrow [1..\sigma]$, and we perform summary queries on the labels. We consider the following queries on paths P: (1) *Minimum or maximum queries:* Return $\min\{\ell(j), j \in P\}$ or $\max\{\ell(j), j \in P\}$. (2) *Selection queries:* Return the element of P with the rth smallest or largest label, including queries where the value of r is relative to $|P|$ such as median queries. (3) *Top-t queries:* Return a set $M \subseteq P$ formed by t elements with smallest or largest labels in P. (4) *τ-Majority queries:* Return a set of labels whose relative frequency in P is over τ, for a given $0 \leq \tau < 1$. (5) *Range queries:* Let $R = \{j \in P, \ell(j) \in [\ell_1, \ell_2]\}$, given $1 \leq \ell_1 \leq \ell_2 \leq \sigma$. A counting query asks for $|R|$, whereas a reporting query requires listing all the elements in R.

As an application of summary queries on paths, suppose we are simulating a system to prepare for situations in which we need to react quickly, e.g., natural disasters or conflicts or critical-equipment failures. We run our simulation through some finite set of states and want to store the traces such that later, given a start state in that set and a number of time-steps, we can quickly return statistics about the states the simulation passes through from that state in that many steps. Of course, we could precompute all the possible answers, but this could take space quadratic in the number of states; we could iterate through all the relevant states at query time, but this could take linear time. If our simulation is deterministic, our problem reduces to storing a function (from states to states, with each state labelled by satellite data) compactly such that we can efficiently answer path queries on it.

The case of positive paths is the easiest. We build on the recent results of He et al. [8] and Chan et al. [3], who give succinct (and also larger) structures for various path queries on trees. Then a relatively simple unfolding and doubling of the cycles in the graph G allows us to directly apply their results to positive paths, with a small extra time penalty to map from the domain of f to the nodes of G. For example, we can solve minimum or maximum queries using $n \lg n + O(n)$ bits and $O(\lg n / \lg \lg n)$ time, range queries in $n \lg n + n \lg \sigma + O(n) + o(n \lg \sigma)$ bits and $O(\lg n / \lg \lg n)$ time per returned element, and selection queries in $n \lg n + 2n \lg \sigma + O(n) + o(n \lg \sigma)$ bits and $O(\lg n / \lg \lg n)$ time.

For negative path points, we unroll the cycles in a way that all the desired nodes in any $f^{-k}(i)$ belong to a contiguous range within a single level of the tree. Then an appropriate layout of the data associated with the node allows us to reduce queries on negative path points to array range queries. Since array ranges are particular cases of tree paths, all the complexities obtained for positive paths are inherited by negative path points, but in addition we can perform other queries that have good solutions on array ranges. For example, we can solve top-t queries [14] using $n \lg n + O(n \lg T)$ bits, where T is the maximum t value permitted, in time $O(t + \lg n / \lg \lg n)$. As another example, we can solve τ-majority queries using $n \lg n + (1 + \epsilon) n \lg \sigma$ bits, for any constant $\epsilon > 0$, in time $O(\lg n / \lg \lg n + 1/\tau)$ [1].

The hardest case is the general negative paths. Our queries in this case are mapped into a three-dimensional space, and thus the structures require $O(n \lg n)$ space in order to offer polylogarithmic times. Still, there is no previous result in this case, and thus it is left open whether those queries can be solved efficiently within linear space.

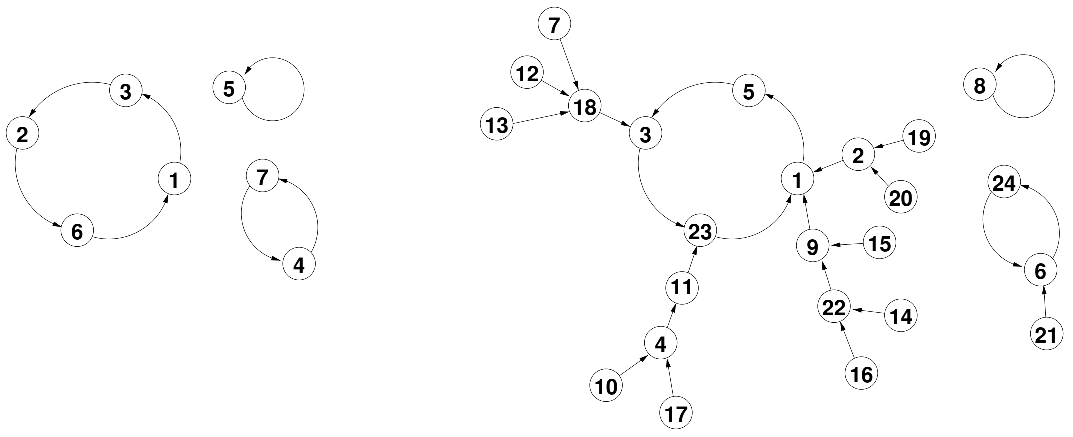


Figure 1 On the left, the representation of a permutation as a directed graph. On the right, the permutation is extended into a function.

2 Background

2.1 Rank and select on bitvectors

A bitvector $B[1..n]$ can be represented in $n + o(n)$ bits so as to perform operations *rank* and *select* in constant time [5]. Operation $rank_b(B, i)$, for $b \in \{0, 1\}$ and $i \in [1..n]$, is the number of occurrences of bit b in $B[1..i]$. Operation $select_b(B, j)$, with $b \in \{0, 1\}$ and $j \in [1..rank_b(B, n)]$, is the position of the jth occurrence of bit b in B.

2.2 Permutations and functions

Munro et al. [10] regard a permutation π on $[1..n]$ as a directed graph $G = (V, E)$, where $V = [1..n]$ and $E = \{(i, \pi(i)),\ i \in [1..n]\}$. This graph turns out to be a set of simple cycles, which correspond to the cycle decomposition of π. Figure 1 (left) shows the graphical representation of permutation $\pi = (3\ 6\ 2\ 7\ 5\ 1\ 4)$, which is decomposed into the cycles $(1\ 3\ 2\ 6)$, (5), and $(4\ 7)$. A function $f : [1..n] \to [1..n]$ is then regarded as an extension of permutations, where a general tree may sprout from each node of the cycles. Figure 1 (right) illustrates the case of $f(1..24) = (5, 1, 23, 11, 3, 24, 18, 8, 1, 4, 23, 18, 18, 22, 9, 22, 4, 3, 2, 2, 6, 9, 1, 6)$, which extends the cycles of our example π.

From the results that are interesting to us, Munro et al. obtain two representations for permutations π. The first uses $\lg n! + o(n)$ bits and computes any $\pi(i)$ in time $O(1)$ and any $\pi^{-1}(i)$ in time $O(\lg n / \lg \lg n)$. The second uses $\lg n! + O((n/t) \lg n)$ bits, for any $t \leq \lg n$, and computes any $\pi(i)$ in time $O(1)$ and any $\pi^{-1}(i)$ in time $O(t)$. For functions, they can compute any positive power $f^k(i)$, with $k \geq 0$, or negative power $f^{-k}(i) = \{j,\ f^k(j) = i\}$, with $k > 0$, in time $O(t)$ and $O(t + |f^{-k}(i)|)$, respectively, using $n \lg n(1 + 1/t)$ bits of space, for any $t \leq \lg n$.

2.3 Path queries on trees

He et al. [8] and Chan et al. [3] showed how to represent a tree where the nodes have labels (or weights) in succinct space so as to support various queries on the paths of the tree. Let us regard the trees as acyclic connected graphs $G(V, E)$; then a *path* is a sequence of nodes

$v_1, v_2, \dots, v_p$, such that every $(v_k, v_{k+1}) \in E$, and it can be specified by giving v_1 and v_p.[1] Given a general ordinal tree of n nodes, where each node v has a label $\ell(v) \in [1..\sigma]$, they support the following queries on paths P of the tree, among others:

1. Minimum/maximum queries, that is, find a node with the smallest or largest label in P, are solved in time $\alpha(m, n)$ with a structure using $O(m)$ bits of space on top of the raw data, for any $m \geq n$, where α is the inverse of the Ackermann function [3].
2. Selection, that is, find the node holding the rth smallest label in P, is solved in time $O(\lg \sigma / \lg\lg \sigma)$, with a structure using $nH(\ell) + o(n \lg \sigma) + O(n)$ bits of space. Here $H(\ell) \leq \lg \sigma$ is the entropy of the distribution of the values $\ell(v)$ over all the nodes v [8].
3. Range queries include counting, that is, how many nodes in P have labels in $[\ell_1..\ell_2]$, and reporting, that is, reporting all those nodes, given ℓ_1 and ℓ_2. Both are solved within $nH(\ell) + o(n \lg \sigma) + O(n)$ bits of space, supporting counting in time $O(1 + \lg \sigma / \lg\lg n)$ and reporting of r results in time $O((r + 1)(1 + \lg \sigma / \lg\lg n))$ [8]. By using more space, it is possible to match the same results of two-dimensional range queries [3].

Those structures include an $O(n)$-bit representation of the tree topology. There are several alternatives (see [11, Ch. 8]) using $2n + o(n)$ bits and supporting a wide set of navigation operations on trees. For positive paths, it turns out that the representations for the path queries used in this section [8, 3] support in constant time a few queries that will be useful:

- Mapping from each tree node v to a unique identifier $id(v) \in [1..n]$, and from an identifier $i \in [1..n]$ to the tree node, $node(i)$.
- Level ancestor queries, that is, given a node v and a distance d, $anc(v, d)$ is the ancestor of v at distance d (e.g., $anc(v, 0)$ is v and $anc(v, 1)$ is the parent of v).
- The depth of a node, $depth(v)$, where the depth of the root is 0.
- The leftmost leaf of the subtree of a node, $leftmost(v)$.
- The lowest common ancestor of two nodes, $lca(u, v)$.

For negative paths, instead, we will use the Fully-Functional (FF) representation [15], which represents the tree using $2n$ parentheses: the tree is traversed in depth-first order, writing an opening parenthesis when we reach a node and a closing one when we leave it. Within $2n + o(n)$ bits it supports in constant time all of the above operations, plus $fwd(x, d)$ and $bwd(x, d)$, defined as follows. Let $excess(y)$ be the number of opening minus closing parentheses up to position y in the parentheses sequence. Then $fwd(x, d)$ (resp. $bwd(x, d)$) finds the closest position y to the right (resp. to the left) of x where $excess(y) = excess(x) + d$. For example, if there is an opening (resp. closing) parenthesis at x, its corresponding closing (resp. opening) parenthesis is at $close(x) = fwd(x, -1)$ (resp. $open(x) = bwd(x, 0) + 1$).

2.4 Range queries on arrays

A much better studied particular case of path queries is that of range queries on an array $A[1..n]$ of labels in $[1..\sigma]$. The following is a brief selection from a number of results reported in the literature:

1. Minimum queries, where it is possible to find the position of a minimum in any range $A[i..j]$ in $O(1)$ time with a structure that uses $2n + o(n)$ bits and does not access A [6]. An analogous result holds for maximum queries.
2. Selection queries, where we can set at construction time a maximum value R of r that can be used in queries, and then a structure using $O(n \lg R)$ bits, without accessing A,

[1] They actually handle undirected graphs, supporting paths between any two nodes u and v. Those can be easily decomposed into two directed paths, from u to $lca(u, v)$ and from v to $lca(u, v)$, where lca is the lowest common ancestor operation.

can answer queries in optimal time $O(1 + \lg r / \lg\lg n)$ [14]. Note that we can set $R = n$ for maximum generality.

3. Top-t queries, that is, finding t elements in $A[i..j]$ with largest labels, can be answered in optimal time $O(t)$ with a structure that uses $O(n \lg T)$ bits and does not access A, where T is an upper bound on the values of t that can be queried [14].
4. τ-majority queries, that is, finding the labels whose relative frequencies in $A[i..j]$ are above τ. This can be solved in optimal time $O(1/\tau)$ and $O(1+\epsilon)nH(\ell) + o(n)$ bits, for any constant $\epsilon > 0$; this representation contains A in compressed form. The space can be reduced to $nH(\ell)(1 + o(1)) + o(n)$ bits, and still obtain any time in $\omega(1/\tau)$ [1].
5. Range counting can be performed in $O(1 + \lg\sigma / \lg\lg n)$ time, and reporting of the r results can be done in time $O((r + 1)(1 + \lg\sigma / \lg\lg n))$, using $n \lg\sigma + o(n \lg\sigma)$ bits [2].

2.5 Range queries in two dimensions

When the ranges are two-dimensional and the points have weights, most of the queries require linear and even super-linear space. Some examples in the literature follow.

1. The top-t elements in a two-dimensional range of an $n \times n$ grid with points having weights in $[1..\sigma]$ can be computed in time $O((t + \lg n) \lg^\epsilon n)$, for any constant $\epsilon > 0$, with a data structure that uses $O(n \lg n)$ bits [12, Lem. 7.1]. With $t = 1$, this gives a structure for range minima or maxima.
2. The rth largest element in a two-dimensional range can be obtained in time $O(\ell \lg n \lg_\ell \sigma)$ with a structure using $n \lg n \lg_\ell \sigma + O(n \lg\sigma)$ bits, for any $\ell \in [2, \sigma]$ [13].
3. The same structure of the previous point can be used to find the τ-majorities in a range in time $O((1/\tau)\ell \lg n \lg_\ell \sigma)$ [13].
4. Range counting queries in three dimensions (or in two dimensions and labels) can be carried out in time $O((\lg n / \lg\lg n)^2)$ with a structure that uses $O(n \lg^2 / \lg\lg n)$ bits of space [9]. Within that space, each point can be reported in time $O((\lg n / \lg\lg n)^2)$ [9]. By raising the space to $O(n \lg^{2+\epsilon} n)$ bits, for any constant $\epsilon > 0$, the time to report r points is reduced to $O(r + \lg\lg n)$ [4].

3 Positive Paths

A positive path of the form $f^{k_1..k_2}(i)$ can be handled by converting the graph G that represents f (recall Figure 1 (right)) into a single tree. The transformation is as follows:

1. We cut each cycle $v_1 \to v_2 \to \ldots \to v_c \to v_1$ at an arbitrary position, say removing the edge $v_c \to v_1$. The result is a directed tree rooted at v_c (with arrows pointing from children to parents) where the cycle edges form the leftmost path.
2. We add a new leaf per cycle, which will be the leftmost child of v_1.
3. We add an artificial root, which will be the parent of the roots v_c of all the cycles.
4. We represent the resulting tree using the data structures of Section 2.3, for whichever query we want to answer. The representation must support in constant time the operations *id*, *node*, *anc*, *depth*, *leftmost*, and *lca*.
5. We store a bitvector $B[1..n + l + 1]$, where $l \leq n$ is the number of leaves added, or equivalently the number of cycles in f, so that $B[i] = 1$ iff the tree node with identifier i is one of the original nodes of G. We give *rank* and *select* support to B, so as to map the tree node identifiers in $[1..n + l + 1]$ of the nodes that are in G to the interval $[1..n]$.
6. We store a permutation π that goes from the mapped node identifiers in $[1..n]$ to the corresponding domain elements, using the representation of Section 2.2.

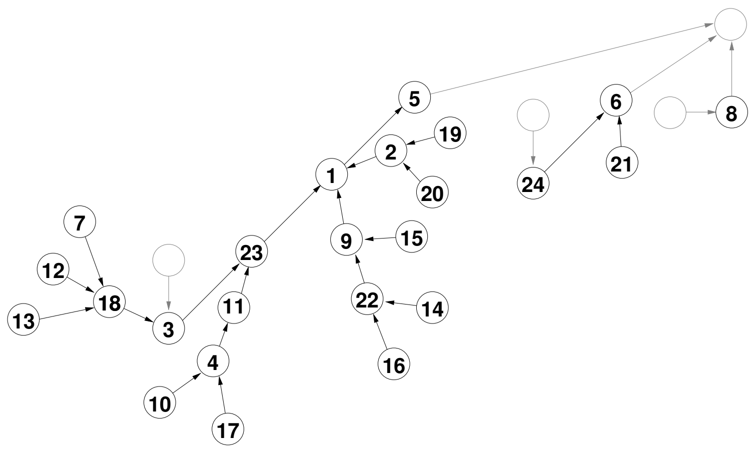


Figure 2 Our transformation to solve positive queries on functions using path queries on trees.

Figure 2 exemplifies our construction on the function of Figure 1. The permutation π is displayed in the form of numbers associated with the nodes. Note how we have broken the cycle $3 \to 23 \to 1 \to 5 \to 3$, for example.

Consider now a positive path query $f^{k_1..k_2}(i)$. In the simplest case, we proceed as follows:

1. We compute $v = node(select_1(B, \pi^{-1}(i)))$, the node where the path query will start.
2. We compute the path extremes $v_s = anc(v, k_1)$ and $v_e = anc(v, k_2)$.
3. We carry out the desired query on the tree path from v_s to v_e.
4. Any node u returned by the query is mapped back to a domain value in constant time using $\pi(rank_1(B, id(u)))$.

In our example, we can compute a query on $f^{1..3}(4) = (11, 23, 1)$ with this technique. However, consider $f^{1..4}(15)$. Our technique maps the path to the domain elements $(9, 1, 5, ?)$, whereas the correct domain elements to include were $(9, 1, 5, 3)$. This is because the path goes through the node v_c where we have cut the cycle. In general, both k_1 and k_2 may be several times larger than the cycle length.

To handle this situation, we use the cycle as follows. First, if $k_1 \geq depth(v)$, then we set $v_s \leftarrow anc(v_1, (k_1 - depth(v)) \bmod c)$, where v_1 is the lowest node of the cycle and c is the cycle length. Similarly, if $k_2 \geq depth(v)$, we set $v_e \leftarrow anc(v_1, (k_2 - depth(v)) \bmod c)$. For this we compute $v_c = anc(v, depth(v) - 1)$, then $v_1 = anc(leftmost(v_c), 1)$ and $c = depth(v_1)$.

However, v_e might not be an ancestor of v_s after this transformation, that is, $depth(v_e) > depth(v_s)$ or $anc(v_s, depth(v_s) - depth(v_e)) \neq v_e$. This means that the positive path is cut into two tree paths: one from v_s to v_c, and the other from v_1 to v_e. In our example, $f^{1..4}(15)$ is cut into the paths $(9, 1, 5)$ and (3).

This can be handled if the query is *decomposable*, that is, we can obtain the answer from the results on the two paths. For example, range counting and reporting are obviously decomposable, whereas range minima (if we do not store the labels, as in the solution of Section 2.3) and selection queries are not decomposable.

A final issue is that, if $k_2 - k_1 \geq c$, we may visit the same domain values several times along the positive path. Since we want to consider each distinct element only once, we can solve this problem by splitting the query into up to three paths: one inside the tree where v belongs that sprouts from the cycle, and two on the cycle. We first compute $v' = lca(v, v_1)$, to find the cycle node where the tree of v sprouts. Then a first path to consider, if $k_1 < d = depth(v) - depth(v')$, is the one corresponding to $[k_1, \min(k_2, d - 1)]$.

If $k_2 \geq d$, we then consider paths on the cycle, starting at node v' and with the range $[k'_1, k'_2] = [\max(0, k_1 - d), k_2 - d]$. If $k'_2 - k'_1 \geq c - 1$, we simply include the whole cycle, with the path from v_1 to v_c. Otherwise, we proceed as before.

Algorithm 1, in Appendix A, gives the complete procedure. We have then Theorem 1, where the extra time is the one spent to compute $\pi^{-1}(i)$ and the extra space is that of storing π and B.

▶ **Theorem 1.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Let there be a tree representation that performs in constant time the operations id, node, anc, depth, leftmost, and lca, and in addition it solves a certain decomposable path query on n-node trees with labels in $[1..\sigma]$ in $T(n, \sigma)$ time, using in total $S(n, \sigma)$ bits of space. Then, there exists a data structure using $n \lg n + O(n) + S(n, \sigma)$ bits that answers the same query on the positive paths of f in time $O(\lg n/ \lg\lg n) + T(n, \sigma)$. There exists another data structure using $n \lg n(1 + 1/t) + S(n, \sigma)$ bits that answers the query in time $O(t) + T(n, \sigma)$, for any $t \leq \lg n$.*

By considering the range queries of He et al. [8] (Section 2.3), we derive Corollary 2.

▶ **Corollary 2.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n \lg n + nH(\ell) + O(n) + o(n \lg \sigma)$ bits that answers counting queries on the positive paths of f in time $O(\lg n/ \lg\lg n)$, and also reports those r results in time $O(\lg n/ \lg\lg n + r(1 + \lg\sigma/ \lg\lg n))$, where $H(\ell) \leq \lg\sigma$ is the entropy of the distribution of the values in ℓ.*

Non-decomposable path queries

When the query is not decomposable, we cannot allow splitting paths. Instead, we unroll the cycles twice, as illustrated in Figure 3. More formally:

1. We cut each cycle $v_1 \to v_2 \to \ldots \to v_c \to v_1$ as before, removing the edge $v_c \to v_1$ and leaving a tree rooted at v_c.
2. We add a leaf as the leftmost child of v_1, as before.
3. We add an upward path per cycle, starting at each tree root v_c, which repeats the cycle with copies of the nodes. That is, we add edges $v_c \to v'_1 \to v'_2 \to \ldots \to v'_{c-1}$. Each of the new nodes v'_i is assigned the same label of v_i.
4. We add an artificial root, which will be the parent of all the nodes v'_{c-1} (or of the node $v_1 = v_c$ for cycles of length 1, since in those cases no v'_i nodes are added).
5. We represent the resulting tree using the data structures of Section 2.3, as before.
6. We store a bitvector $B[1..n + g]$, where $g \leq n + 1$ is the number of nodes added, so that $B[i] = 1$ iff the tree node with identifier i is one of the original nodes of G. As before, we give *rank* and *select* support to B.
7. We store a permutation π that goes from the mapped node identifiers in $[1..n]$ to the corresponding domain elements, as before.

We can now compute $v'_{c-1} = anc(v, depth(v) - 1)$, $v_1 = anc(leftmost(v'_{c-1}), 1)$, $c = (depth(v_1) + 1)/2$, and $v_c = anc(v_1, c - 1)$. We also compute $v' = lca(v_1, v)$ as before. There are two cases. The first is that the path starts inside the subtree of v', that is, if $k_1 < d = depth(v) - depth(v')$. In this case, we set $v_s = anc(v, k_1)$. Then, if $k_2 - d < c$, we set $v_e = anc(v, k_2)$; otherwise we set $v_e = anc(v', c - 1)$. Finally, we run the tree path query from v_s to v_e.

The other case is that the path lies completely on the cycle, that is, $k_1 \geq d$. We can first exclude the condition $k_2 - k_1 \geq c$, as in this case we simply query the path from v_1 to v_c.

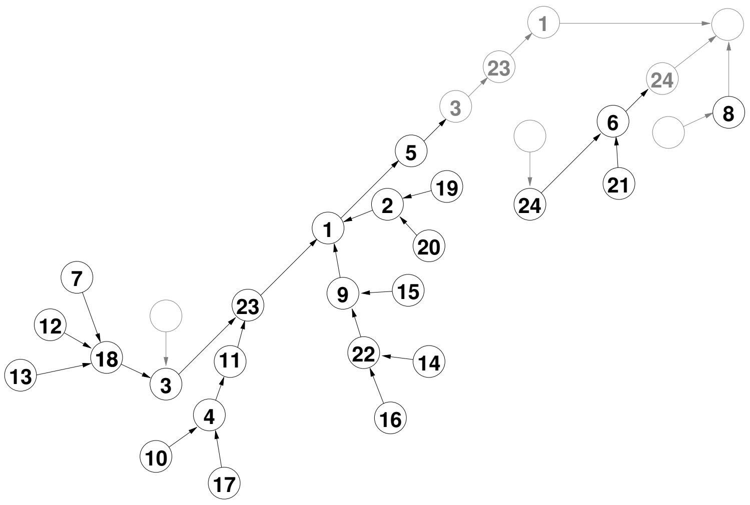


Figure 3 Our transformation to solve non-decomposable positive queries on functions using path queries on trees.

If $k_2 - k_1 < c$, we find v_s inside the path that goes from v_1 to v_c: If $depth(v) - k_1 \geq c$, we set $v_s = anc(v, k_1)$; otherwise we set $v_s = anc(v_1, ((c-1) - (depth(v) - k_1)) \bmod c)$. We then do the same to compute v_e with k_2. Finally, if v_e is deeper than v_s, we recompute $v_e = anc(v_e, c)$. Now we can safely run the tree path query from v_s to v_e.

A final issue is how to map back the nodes $u = v'_i$ that the algorithm may return. Note that we know the cycle where the query was performed, so we know c and v_1. Thus, if $depth(u) < c$, we know that u is a created node, and replace it with $anc(v_1, (c-1) - depth(u))$ before mapping it to the domain of f. Algorithm 2, in Appendix A, gives the pseudocode.

Since we have up to n newly created nodes for which we have to store labels, we have Theorem 3.

▶ **Theorem 3.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Let there be a tree representation that performs in constant time the operations id, node, anc, depth, leftmost, and lca, and it addition it solves a certain non-decomposable path query on n-node trees with labels in $[1..\sigma]$ in $T(n, \sigma)$ time, using in total $S(n, \sigma)$ bits of space. Then, there exists a data structure using $n \lg n + O(n) + S(2n+1, \sigma)$ bits that answers the same query on the positive paths of f in time $O(\lg n / \lg \lg n) + T(2n+1, \sigma)$. There exists another data structure using $n \lg n(1 + 1/t) + S(2n+1, \sigma)$ bits that answers the query in time $O(t) + T(2n+1, \sigma)$, for any $t \leq \lg n$.*

By considering the minimum/maximum and the selection queries of He et al. [8] (Section 2.3), we derive Corollaries 4 and 5.

▶ **Corollary 4.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n \lg n + O(n)$ bits that answers minimum/maximum queries on the positive paths of f in time $O(\lg n / \lg \lg n)$. There exists another structure using $(1+\epsilon) n \lg n + O(m)$ bits, for any constant $\epsilon > 0$ and any $m \geq n$, that answers the queries in time $\alpha(m, n)$.*

▶ **Corollary 5.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n \lg n + 2nH(\ell) + O(n) + o(n \lg \sigma)$ bits that answers selection queries on the positive paths of f in time $O(\lg n / \lg \lg n)$, where $H(\ell) \leq \lg \sigma$ is the entropy of the distribution of the values in ℓ.*

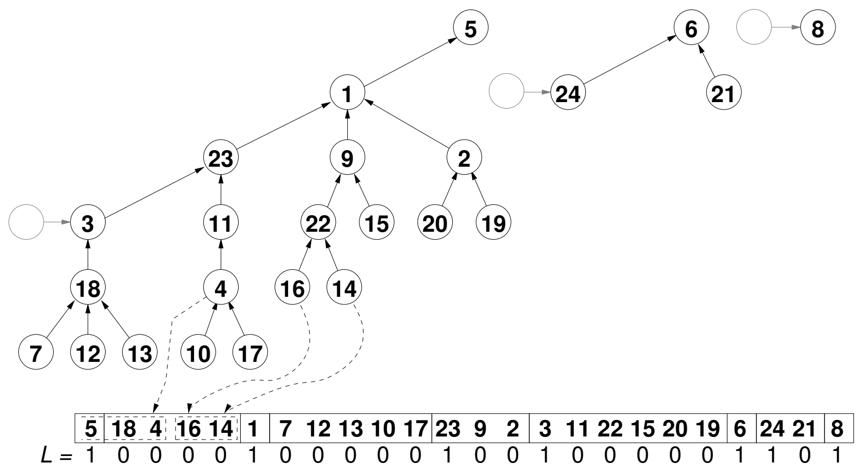


Figure 4 The same tree for decomposable positive queries (without the extra root), showing how the levels are deployed to aid in negative path points.

4 Negative Path Points

Figure 4 shows the same tree of Figure 2, now showing clearly the resulting levels of the tree, and without the extra root. The result is a forest, which we will store with the FF representation [15]. The figure illustrates an important point: all the nodes in $f^{-k}(i)$ correspond to the descendants at distance k of the node corresponding to i. For example $f^{-2}(1) = \{3, 11, 22, 15, 20, 19\}$. These form a range if we deploy the nodes in levelwise order.

Just as for positive paths, we will store a bitvector B indicating which nodes are originally in G (i.e., not the added leaves) and a permutation π on $[1..n]$ mapping from the identifiers of those nodes in G (after being mapped to $[1..n]$ using B) to domain elements. The information on the nodes (such as the labels) will be stored in levelwise order, with a permutation ρ on $[1..n]$ mapping from the levelwise deployment to the tree identifier of the node. Let $v = \mathit{node}(\mathit{select}_1(B, \pi^{-1}(i)))$ be the node corresponding to domain element i, and assume v is not on the cycle of its component in G. Then the elements of $f^{-k}(i)$ are the descendants of v at distance k. The leftmost such descendant is found with $v_1 = \mathit{fwd}(v, k)$, whereas the rightmost one is $v_2 = \mathit{open}(\mathit{bwd}(\mathit{close}(v), k + 1) + 1)$. Then the range of values where the information on the elements of $f^{-k}(i)$ is stored is $[\rho^{-1}(\mathit{rank}_1(B, \mathit{id}(v_1))), \rho^{-1}(\mathit{rank}_1(B, \mathit{id}(v_2)))]$. Note that any element at position j in the levelwise deployment can be converted into a domain element with $\pi(\rho(j))$. Figure 4 shows how $f^{-2}(9)$ is mapped to the range containing $(16, 14)$, which is within the level containing $(18, 4, 16, 14)$ (disregard for now bitvector L and the way levels are interlaced in the array).

When v is on a cycle (of length c), then we can go to its predecessor in the cycle (taking the arrow backwards) and collect the descendants at distance $k - 1$ in its sprouting tree, then to its predecessor and collect its descendants at distance $k - 2$, and so on. Given the way we have converted G into a tree, all these nodes are indeed the descendants of v at distance k; consider again $f^{-2}(1)$ in Figure 4. However, the situation can be more complicated because, if the trees sprouting from the cycle are tall enough, then we could run over the whole cycle in backward direction and return again to v, now looking for descendants at distance $k - c$. Therefore, not only we have to include the descendants of v at distance k, but also all the elements in the whole tree where v belongs at depths $\mathit{depth}(v) + k - c$, $\mathit{depth}(v) + k - 2c$, and so on.

To handle this case, we will store the levelwise information on the nodes of each tree of the forest in an interlaced order of the levels: levels 1, $c+1$, $2c+1$, and so on, then levels 2, $c+2$, $2c+2$, and so on, until levels c, $2c$, $3c$, and so on. A bitvector $L[1..n]$ with *rank* and *select* support will mark, in the levelwise ordered domain, the first node at a level of the form $l+tc$ in each tree, for all $1 \leq l \leq c$. Figure 4 shows the levelwise deployment. The nodes of the first tree are listed as $5, 18, 4, 16, 14$ for $l=1$, then $1, 7, 12, 13, 10, 17$ for $l=2$, then $23, 9, 2$ for $l=3$, and finally $3, 11, 22, 15, 20, 19$ for $l=4$. The following two trees are then listed as $6, 24, 21$ and 8. The bitvector L marks the beginnings of the change in tree or in l.

With this arrangement, we only have to find as before $p_2 = \rho^{-1}(rank_1(B, id(v_2)))$, the second endpoint of the range, and then $p_1 = select_1(L, rank_1(L, p_2))$, the beginning of the nodes of the tree of v_2 with its same l value. Figure 4 shows how v_2 is found for $f^{-2}(23)$, and then the range includes up to the beginning of $l=1$ in its tree, to contain $(5, 18, 4)$.

The final issue is how to determine if v is or not on the cycle. We can do this by computing, similarly to the positive paths, $v_c = anc(v, depth(v))$, $v_0 = leftmost(v_c)$ as the leftmost leaf,[2] and then v is in the cycle iff v_0 descends from v, that is, $v \leq v_0 \leq close(v)$.

Finally, we can build on the levelwise deployment of the node data any array range query data structure we desire. Algorithm 3, in Appendix A, shows the pseudocode.

The time per query is that of the array range query, plus the time needed to compute π^{-1} and ρ^{-1} a constant number of times; answers are converted back to domain values by computing ρ and π in constant time. Apart from the array range query structures, we are storing two permutations and some bitvectors. We then have Theorem 6.

▶ **Theorem 6.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Let there be an array range query data structure that, on an array $A[1..n]$ of values in $[1..\sigma]$, answers queries in time $T(n,\sigma)$ using $S(n,\sigma)$ bits of space. Then, there exists a data structure using $2n\lg n + O(n) + S(n,\sigma)$ bits that answers the same query on the negative path points of f in time $O(\lg n/\lg\lg n) + T(n,\sigma)$. There exists another data structure using $2n\lg n(1+1/t) + S(n,\sigma)$ bits that answers the query in time $O(t) + T(n,\sigma)$, for any $t \leq \lg n$.*

By considering the various array range queries of Section 2.4, we can derive Corollaries 7 to 10, among others.

▶ **Corollary 7.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $2n\lg n + O(n)$ bits that finds the elements of $f^{-k}(i)$ with the minimum and the maximum labels, for any $i \in [1..n]$ and $k > 0$, in time $O(\lg n/\lg\lg n)$. There exists another data structure using $2n\lg n(1+1/t) + O(n)$ bits that answers the query in time $O(t)$, for any $t \leq \lg n$.*

▶ **Corollary 8.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $2n\lg n + O(n\lg R)$ bits that finds the element with the rth largest label in $f^{-k}(i)$, for any $i \in [1..n]$, $k > 0$, and $1 \leq r \leq R$, in time $O(\lg n/\lg\lg n)$. It can also list the r elements with the largest or smallest values in $f^{-k}(i)$ in time $O(r + \lg n/\lg\lg n)$.*

▶ **Corollary 9.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $2n\lg n + (1+$*

[2] This operation can be computed in this representation with $select_{)}(rank_{)}(v_c)+1)-1$ on the sequence of parentheses, which has *rank* and *select* support.

$\epsilon)nH(\ell) + O(n)$ bits, where $\epsilon > 0$ is any constant and $H(\ell)$ is the entropy distribution of the labels, that finds the τ-majorities in the labels of $f^{-k}(i)$, for any $i \in [1..n]$, $k > 0$, and $0 < \tau < 1$, in time $O(1/\tau + \lg n/\lg\lg n)$.

▶ **Corollary 10.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $2n \lg n + nH(\ell) + o(n \lg \sigma) + O(n)$ bits, where $H(\ell)$ is the entropy distribution of the labels, that counts the number of labels of $f^{-k}(i)$ within a range, for any $i \in [1..n]$, $k > 0$, and range of labels, in time $O(\lg n/\lg\lg n)$. It can then list those r elements in time $O(\lg n/\lg\lg n + r(1 + \lg\sigma/\lg\lg n))$.*

5 Negative Paths

For ranges of negative values of k, $f^{-[k_1..k_2]}(i)$, our solution maps the queries into two-dimensional ranges, which require more space and/or time than previous ones. We preserve the same tree as in Section 4, but this time the mapping from nodes v is done to pairs $(\mathit{preorder}(v), \mathit{depth}(v))$. Here $\mathit{preorder}(v) = \mathit{id}(v)$ is the preorder of the node in the FF representation. Therefore, once we have mapped the domain element i to a tree node v, and determined that v is not on the cycle, we have that the query encompasses the two dimensional range $[\mathit{preorder}(v) \mathinner{..} \mathit{preorder}(v) + \mathit{subtreesize}(v) - 1] \times [\mathit{depth}(v) + k_1 \mathinner{..} \mathit{depth}(v) + k_2]$. All these operations are supported in constant time with the FF representation [15]. We now perform the desired query on a structure that handles two-dimensional points (possibly with labels). The returned points (p, d) are then mapped to the nodes with preorder p, $\mathit{node}(p)$, which is also supported in constant time.

For the case where v is on the cycle, we will use another arrangement. Note that we want to consider, in addition to the previous range, all the nodes in the tree of v with a depth that is between $d_1 = \mathit{depth}(v) + k_1$ and $d_2 = \mathit{depth}(v) + k_2$, modulo c, but not reaching the range $[d_1..d_2]$, as that one is already handled. To this end, we will map the nodes v to pairs $(\mathit{depth}(v) \text{ div } c, \mathit{depth}(v) \bmod c)$, and will query for the points in the range $[0..d_2 \text{ div } c - 1] \times [d_1 \bmod c \mathinner{..} d_2 \bmod c]$. If, however, $d_1 \bmod c > d_2 \bmod c$, then we split the second range into $[d_1 \bmod c \mathinner{..} c - 1]$ and $[0 \mathinner{..} d_2 \bmod c]$.

An exception occurs if $k_2 - k_1 \geq c$, since then the two types of ranges overlap and we could count points twice. In this case we take, in this second arrangement, the range $[0..d_2 \text{ div } c - 1] \times [0..c - 1]$, and reduce the range within the subtree of v to $[\mathit{preorder}(v)..\mathit{preorder}(v) + \mathit{subtreesize}(v) - 1] \times [(d_2 \text{ div } c) \cdot c \mathinner{..} d_2]$.

Note that in this case we have to complete the query from the results of up to 3 two-dimensional ranges, so the query must be decomposable. We then obtain Theorem 11.

▶ **Theorem 11.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Let there be a two-dimensional range query data structure that, on an $n \times n$ grid with values in $[1..\sigma]$, answers decomposable queries in $T(n, \sigma)$ time using in $S(n, \sigma)$ bits of space. Then, there exists a data structure using $2n \lg n + O(n) + S(n, \sigma)$ bits that answers the same query on the negative paths of f in time $O(\lg n/\lg\lg n) + T(n, \sigma)$. There exists another data structure using $2n \lg n(1 + 1/t) + S(n, \sigma)$ bits that answers the query in time $O(t) + T(n, \sigma)$, for any $t \leq \lg n$.*

We can combine the theorem with various results on querying two-dimensional grids of points with labels (or weights); recall Section 2.5. We obtain Corollaries 12 and 13.

▶ **Corollary 12.** *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $O(n \lg n)$ bits* *that answers minima and maxima queries on the negative paths of f in time $O(\lg^{1+\epsilon} n)$, and top-t queries in time $O((t + \lg n) \lg^{\epsilon} n)$, for any constant $\epsilon > 0$.*

▶ **Corollary 13.** *Let $f : [1..n] \rightarrow [1..n]$ be a function and $\ell : [1..n] \rightarrow [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $O(n \lg^2 n / \lg \lg n)$ bits that answers range counting queries on the negative paths of f in time $O((\lg n / \lg \lg n)^2)$, and reports the r values in time $O((r + 1)(\lg n / \lg \lg n)^2)$. By using slightly more space, $O(n \lg^{2+\epsilon} n)$ bits for any constant $\epsilon > 0$, the time to report is reduced to $O(r + \lg \lg n)$.*

6 Conclusions

Munro et al. [10] studied how to represent an integer function $f : [1..n] \rightarrow [1..n]$ so as to efficiently find all the elements of positive and negative powers of f. We have now considered, for the first time, queries on ranges of positive or negative powers of f. For positive powers, we essentially retain optimal storage space and almost match the best results of path queries on trees [8, 3]. Negative powers lead to a set of domain values. For a single negative power, we basically double the space while almost retaining the performance of the corresponding array range query. For a range of negative powers, we resort to three-dimensional range queries, where time and space are essentially multiplied by $O(\lg n)$.

Our results consider queries on arbitrary labels on $[1, \sigma]$ attached to the domain elements. Appendix B gives a few improved results for the simpler case where the queries are run over the domain elements themselves.

This is the first study on this problem, and it is not clear whether the results can be improved, in particular it is not clear if queries on ranges of negative powers of f must resort to three-dimensional range queries.

References

1 Djamal Belazzougui, Travis Gagie, J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Range majorities and minorities in arrays, 2016. `arXiv:1606.04495`.

2 Prosenjit Bose, Meng He, Anil Maheshwari, and Pat Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In Frank K. H. A. Dehne, Marina L. Gavrilova, Jörg-Rüdiger Sack, and Csaba D. Tóth, editors, *Proceedings of the 11th International Symposium on Algorithms and Data Structures (WADS 2009)*, volume 5664 of *LNCS*, pages 98–109. Springer, 2009. `doi:10.1007/978-3-642-03367-4_9`.

3 Timothy M. Chan, Meng He, J. Ian Munro, and Gelin Zhou. Succinct indices for path minimum, with applications to path reporting. In Andreas S. Schulz and Dorothea Wagner, editors, *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA 2014)*, volume 8737 of *LNCS*, pages 247–259. Springer, 2014. `doi:10.1007/978-3-662-44777-2_21`.

4 Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry (SoCG 2011)*, pages 1–10. ACM, 2011. `doi:10.1145/1998196.1998198`.

5 David R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996. URL: `http://hdl.handle.net/10012/64`.

6 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. `doi:10.1137/090779759`.

7 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In Martin Farach-Colton, editor, *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 841–850. ACM/SIAM, 2003. URL: `http://dl.acm.org/citation.cfm?id=644108.644250`.

8 Meng He, J. Ian Munro, and Gelin Zhou. Succinct data structures for path queries. In Leah Epstein and Paolo Ferragina, editors, *Proceedings of the 20th Annual European Symposium on Algorithms (ESA 2012)*, volume 7501 of *LNCS*, pages 575–586. Springer, 2012. `doi:10.1007/978-3-642-33090-2_50`.

9 Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In Rudolf Fleischer and Gerhard Trippen, editors, *Proceedings of the 15th International Symposium on Algorithms and Computation (ISAAC 2004)*, volume 3341 of *LNCS*, pages 558–568. Springer, 2004. `doi:10.1007/978-3-540-30551-4_49`.

10 J. Ian Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012. `doi:10.1016/j.tcs.2012.03.005`.

11 Gonzalo Navarro. *Compact Data Structures: A practical approach*. Cambridge University Press, 2016. `doi:10.1017/CBO9781316588284`.

12 Gonzalo Navarro and Yakov Nekrich. Top-k document retrieval in optimal time and linear space. In Yuval Rabani, editor, *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, pages 1066–1077. SIAM, 2012. `doi:10.1137/1.9781611973099.84`.

13 Gonzalo Navarro, Yakov Nekrich, and Luís M. S. Russo. Space-efficient data-analysis queries on grids. *Theor. Comput. Sci.*, 482:60–72, 2013. `doi:10.1016/j.tcs.2012.11.031`.

14 Gonzalo Navarro, Rajeev Raman, and Srinivasa Rao Satti. Asymptotically optimal encodings for range selection. In Venkatesh Raman and S. P. Suresh, editors, *Proceedings of the 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*, volume 29 of *LIPIcs*, pages 291–301. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2014. `doi:10.4230/LIPIcs.FSTTCS.2014.291`.

15 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014. `doi:10.1145/2601073`.

A Pseudocodes

We give detailed pseudocodes for the main procedures described in the paper. In Algorithm 1, it is possible to reduce the case of three paths to two, since those of lines 9 and 23 can be concatenated into one, but we opt for simplicity.

B Functions Without Labels

In the simple case where the function has no assigned labels, or said another way, we may assume $\ell(i) = i$ for the queries, we can do better than Corollaries 2 and 5. Both path query structures [8] store the sequence of labels (now domain elements) in node identifier order, and represent it with a wavelet tree [7]. This structure allows us, with a query similar to *select*, to find the occurrence of element i, thus effectively computing $\pi^{-1}(i)$, in time $O(\lg n/\lg\lg n)$. Instead of returning the node identifier, they may return the label, that is, the domain element, by accessing the wavelet tree in the same time. Therefore, they do not require the permutation to map from elements to nodes. In the case of Corollary 5, where we have duplicated nodes v_i', we may use the *select*-like operation to find the two places where an element is mentioned in the labels, and choose the one with largest depth to avoid starting

Algorithm 1: Computing decomposable queries on positive paths.

```
Proc $Positive(i, k_1, k_2)$
    $v \leftarrow node(select_1(B, \pi^{-1}(i)))$
    $v_c \leftarrow anc(v, depth(v) - 1)$
    $v_1 \leftarrow anc(leftmost(v_c), 1)$
    $c \leftarrow depth(v_1)$
    $v' \leftarrow lca(v_1, v)$
    $d \leftarrow depth(v) - depth(v')$
    if $k_1 < d$ then
        Compute path query from $anc(v, k_1)$ to $anc(v, \min(k_2, d-1))$
    if $k_2 \geq d$ then
        $k'_1 \leftarrow \max(0, k_1 - d)$
        $k'_2 \leftarrow k_2 - d$
        if $k'_2 - k'_1 \geq c - 1$ then
            Compute path query from $v_1$ to $v_c$
        else
            if $k'_1 < depth(v')$ then $v_s \leftarrow anc(v', k'_1)$
            else $v_s \leftarrow anc(v_1, (k'_1 - depth(v')) \bmod c)$
            if $k'_2 < depth(v')$ then $v_e \leftarrow anc(v', k'_2)$
            else $v_e \leftarrow anc(v_1, (k'_2 - depth(v')) \bmod c)$
            if $depth(v_s) \geq depth(v_e)$ and $anc(v_s, depth(v_s) - depth(v_e)) = v_e$ then
                Compute path query from $v_s$ to $v_e$
            else
                Compute path query from $v_s$ to $v_c$
                Compute path query from $v_1$ to $v_e$
    Return the composition of all the path queries performed; resulting nodes $u$ are
      converted into domain values $\pi(rank_1(B, id(u)))$
```

the query from a node v'_i. Since the wavelet tree has each distinct element mentioned once or twice, its entropy is essentially maximal, and we have the following results for this case.

▶ **Corollary 14.** *Let $f : [1..n] \rightarrow [1..n]$ be a function. Then there exists a data structure using $n \lg n + o(n \lg n)$ bits that answers counting queries on the positive paths of f in time $O(\lg n / \lg \lg n)$, and also reports those r results in time $O((r+1) \lg n / \lg \lg n)$.*

▶ **Corollary 15.** *Let $f : [1..n] \rightarrow [1..n]$ be a function. Then, there exists a data structure using $2n \lg n + o(n \lg n)$ bits that answers selection queries on the positive paths of f in time $O(\lg n / \lg \lg n)$.*

We can also simplify Corollary 10, where the structures used perform the equivalent to *select* queries on the sequence of labels. Here, we can find where the domain value i appears in the sequence, and then map it to the tree using ρ and B. Then there is no need for permutation π, and we can subtract $n \lg n$ bits to the space in this corollary.[3]

[3] The same happens in Corollary 9, but the query makes no sense if the labels are all unique.

Algorithm 2: Computing non-decomposable queries on positive paths.

```
Proc $Positive(i, k_1, k_2)$
    $v \leftarrow node(select_1(B, \pi^{-1}(i)))$
    $v'_{c-1} \leftarrow anc(v, depth(v) - 1)$
    $v_1 \leftarrow anc(leftmost(v'_{c-1}), 1)$
    $c \leftarrow (depth(v_1) + 1)/2$
    $v' \leftarrow lca(v_1, v)$
    $d \leftarrow depth(v) - depth(v')$
    if $k_1 < d$ then
        $v_s \leftarrow anc(v, k_1)$
        if $k_2 - d < c$ then $v_e \leftarrow anc(v, k_2)$
        else $v_e \leftarrow anc(v', c - 1)$
    else if $k_2 - k_1 \geq c$ then
        $v_s \leftarrow v_1$
        $v_e \leftarrow v_c$
    else
        if $depth(v) - k_1 \geq c$ then $v_s \leftarrow anc(v, k_1)$
        else $v_s \leftarrow anc(v_1, ((c - 1) - (depth(v) - k_1)) \bmod c)$
        if $depth(v) - k_2 \geq c$ then $v_e \leftarrow anc(v, k_2)$
        else $v_e \leftarrow anc(v_1, ((c - 1) - (depth(v) - k_2)) \bmod c)$
        if $depth(v_s) < depth(v_e)$ then
            $v_e \leftarrow anc(v_e, c)$
    Compute path query from $v_s$ to $v_e$
    Return the answers; resulting nodes $u$ are converted into domain values
      $\pi(rank_1(B, id(u)))$, but if $depth(u) < c$ we first set $u \leftarrow anc(v_1, (c - 1) - depth(u))$
```

Algorithm 3: Computing queries on negative path points.

```
Proc $Negative(i, k)$
    $v \leftarrow node(select_1(B, \pi^{-1}(i)))$
    $v_c \leftarrow anc(v, depth(v))$
    $v_0 \leftarrow leftmost(v_c)$
    $v_2 = open(bwd(close(v), k + 1) + 1)$
    $p_2 = \rho^{-1}(rank_1(B, id(v_2)))$
    if $v \leq v_0 \leq close(v)$ then
        $p_1 \leftarrow select_1(L, rank_1(L, p_2))$
    else
        $v_1 \leftarrow fwd(v, k)$
        $p_1 \leftarrow \rho^{-1}(rank_1(B, id(v_1)))$
    Compute array range query on $[p_1, p_2]$
    Return the answers; resulting positions $j$ are converted into domain values $\pi(\rho(j))$.
```

▶ **Corollary 16.** *Let $f : [1..n] \to [1..n]$ be a function. Then, there exists a data structure using $2n \lg n + o(n \lg n)$ bits, that counts the number of elements of $f^{-k}(i)$ within a range, for any $i \in [1..n]$, $k > 0$, and range of elements, in time $O(\lg n / \lg \lg n)$. It can then list those r elements in time $O((r + 1) \lg n / \lg \lg n)$.*

Deterministic Indexing for Packed Strings

Philip Bille*[1], Inge Li Gørtz†[2], and Frederik Rye Skjoldjensen‡[3]

1 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
`phbi@dtu.dk`
2 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
`inge@dtu.dk`
3 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
`fskj@dtu.dk`

Abstract

Given a string S of length n, the classic string indexing problem is to preprocess S into a compact data structure that supports efficient subsequent pattern queries. In the *deterministic* variant the goal is to solve the string indexing problem without any randomization (at preprocessing time or query time). In the *packed* variant the strings are stored with several character in a single word, giving us the opportunity to read multiple characters simultaneously. Our main result is a new string index in the deterministic *and* packed setting. Given a packed string S of length n over an alphabet σ, we show how to preprocess S in $O(n)$ (deterministic) time and space $O(n)$ such that given a packed pattern string of length m we can support queries in (deterministic) time $O\left(m/\alpha + \log m + \log\log\sigma\right)$, where $\alpha = w/\log\sigma$ is the number of characters packed in a word of size $w = \Theta(\log n)$. Our query time is always at least as good as the previous best known bounds and whenever several characters are packed in a word, i.e., $\log\sigma \ll w$, the query times are faster.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems, H.3.1 Content Analysis and Indexing

Keywords and phrases suffix tree, suffix array, deterministic algorithm, word packing

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.6

1 Introduction

Let S be a string of length n over an alphabet of size σ. The *string indexing problem* is to preprocess S into a compact data structure that supports efficient subsequent pattern queries. Typical queries include *existential queries* (decide if the pattern occurs in S), *reporting queries* (return all positions where the pattern occurs), and *counting queries* (returning the number of occurrences of the pattern).

The string indexing problem is a classic well-studied problem in combinatorial pattern matching and the standard textbook solutions are the suffix tree and the suffix array (see e.g., [9, 10, 11, 14]). A straightforward implementation of suffix trees leads to an $O(n)$ preprocessing time and space solution that given a pattern of length m supports existential and counting queries in time $O(m\log\sigma)$ and reporting queries in time $O(m\log\sigma + \mathrm{occ})$, where occ is the number of occurrences of the pattern. The suffix array implemented with additional arrays storing longest common prefixes leads to a solution that also uses $O(n)$ preprocessing time and space while supporting existential and counting queries in time

* Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).
† Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).
‡ Supported by the Danish Research Council (DFF – 1323-00178).

$O(m + \log n)$ and reporting queries in time $O(m + \log n + \text{occ})$. If we instead combine suffix trees with perfect hashing [7] we obtain $O(n)$ *expected* preprocessing time and $O(n)$ space, while supporting existential and counting queries in time $O(m)$ and reporting queries in time $O(m + \text{occ})$. The above bounds hold assuming that the alphabet size σ is polynomial in n. If this is not the case, additional time for sorting the alphabet is required [5]. For simplicity, we adopt this convention in all of the bounds throughout the paper.

In the *deterministic* variant the goal is to solve the string indexing problem without any randomization. In particular, we cannot combine suffix trees with perfect hashing to obtain $O(m)$ or $O(m + \text{occ})$ query times. In this setting Cole et al. [4] showed how to combine the suffix tree and suffix array into the *suffix tray* that uses $O(n)$ preprocessing time and space and supports existential and counting queries in $O(m + \log \sigma)$ time and reporting queries in $O(m + \log \sigma + \text{occ})$ time. Recently, the query times were improved by Fischer and Gawrychowski [6] to $O(m + \log \log \sigma)$ and $O(m + \log \log \sigma + \text{occ})$, respectively.

In the *packed* variant the strings are given in a *packed representation*, with several characters in a single word [3, 2, 1, 13]. For instance, DNA-sequences have an alphabet of size 4 and are therefore typically stored using 2 bits per character with 32 characters in a 64-bit word. On packed strings we can read multiple characters in constant time and hence potentially do better than the immediate $\Omega(m)$ or $\Omega(m + \text{occ})$ lower bound for existential/counting queries and reporting queries, respectively. In this setting Takagi et al. [13] recently introduced the *packed compact trie* that stores packed strings succinctly and also supports dynamic insertion and deletions of strings. In a static and deterministic setting their data structure implies a linear space and superlinear time preprocessing solution that uses $O(\frac{m}{\alpha} \log \log n)$ and $O(\frac{m}{\alpha} \log \log n + \text{occ})$ query time, respectively.

In this paper, we consider the string indexing problem in the deterministic and packed setting simultaneously, and present a solution that improves all of the above bounds.

1.1 Setup and result

We assume a standard unit-cost word RAM with word length $w = \Theta(\log n)$, and a standard instruction set including arithmetic operations, bitwise boolean operations, and shifts. All strings in this paper are over an alphabet Σ of size σ. The *packed representation* of a string A is obtained by storing $\alpha = w/\log \sigma$ characters per word thus representing A in $O(|A| \log \sigma / w)$ words. If A is given in the packed representation we simply say that A is a *packed string*.

Throughout the paper let S be a string of length n. Our goal is to preprocess S into a compact data structure that given a packed pattern string P supports the following queries.

- Count(P): Return the number of occurrence of P in S.
- Locate(P): Report all occurrences of P in S.
- Predecessor(P): Returns the predecessor of P in S, i.e., the lexicographically largest suffix in S that is smaller than P.

We show the following main result.

▶ **Theorem 1.** *Let S be a string of length n over an alphabet of size σ and let $\alpha = w/\log \sigma$ be the number of characters packed in a word. Given S we can build an index in $O(n)$ deterministic time and space such that given a packed pattern string of length m we can support Count and Predecessor in time $O(\frac{m}{\alpha} + \log m + \log \log \sigma)$ and Locate in time $O(\frac{m}{\alpha} + \log m + \log \log \sigma + \text{occ})$ time.*

Compared to the result of Fischer and Gawrychowski [6], Thm 1 is always at least as good and whenever several characters are packed in a word, i.e., $\log \sigma \ll w$, the query times are faster. Compared to the result of Takagi et al. [13], our query time is a factor $\log \log n$ faster.

Technically, our results are obtained by a novel combination of previous techniques. Our general tree decomposition closely follows Fischer and Gawrychowski [6], but different ideas are needed to handle packed strings efficiently. We also show how to extend the classic suffix array search algorithm to handle packed strings efficiently.

2 Preliminaries

2.1 Deterministic hashing and predecessor

We use the following results on deterministic hashing and predecessor data structures.

▶ **Lemma 2** (Ružić [12, Theorem 3]). *A static linear space dictionary on a set of k keys can be deterministically constructed in time $O(k(\log\log k)^2)$, so that lookups to the dictionary take time $O(1)$.*

Fischer and Gawrychowski [6] use the same result for hashing characters. In our context we will apply it for hashing words of packed characters.

▶ **Lemma 3** (Fischer and Gawrychowski [6, Proposition 7]). *A static linear space predecessor data structure on a set of k keys from a universe of size u can be constructed deterministically in $O(k)$ time and $O(k)$ space such that predecessor queries can be answered deterministically in time $O(\log\log u)$.*

2.2 Suffix tree

The suffix tree T_S of S is the compacted trie over the n suffixes from the string S. We assume that the special character $\$ \notin \Sigma$ is appended to every suffix of S such that each string is ending in a leaf of the tree. The edges are sorted lexicographically from left to right. We say that a leaf *represents* the suffix that is spelled out by concatenating the labels of the edges on the path from the root to the leaf. For a node v in T_S, we say that the *subtree* of v is the tree induced by v and all proper descendants of v. We distinguish between implicit and explicit nodes: implicit nodes are conceptual and refer to the original non branching nodes from the trie without compacted paths. Explicit nodes are the branching nodes in the original trie. When we refer to nodes that are not specified as either explicit or implicit, then we are always referring to explicit nodes. The lexicographic ordering of the suffixes represented by the leaves corresponds to the ordering of the leaves from left to right in the compacted trie. For navigating from node to child, each node has a predecessor data structure over the first characters of every edge going to a child. With the predecessor data structure from Lemma 3 navigation from node to child takes $O(\log\log\sigma)$ time and both the space and the construction time of the predecessor data structure is linear in the number of children.

2.3 Suffix array

Let $S_1, S_2, \ldots, S_n$ be the n suffixes of S from left to right. The suffix array SA_S of S gives the lexicographic ordering of the suffixes such that $S_{\mathsf{SA}_S[i]}$ refers to the ith lexicographically largest suffix of S. This means that for every $1 < i \leq n$ we have that $S_{\mathsf{SA}_S[i-1]}$ is lexicographically smaller than $S_{\mathsf{SA}_S[i]}$. For simplicity we let $\mathsf{SA}_S[i]$ refer to the suffix $S_{\mathsf{SA}_S[i]}$ and we say that $\mathsf{SA}_S[i]$ represents the suffix $S_{\mathsf{SA}_S[i]}$. Every suffix from S with pattern P as a prefix will be located in a consecutive range of SA_S. This range corresponds to the range of consecutive leaves in the subtree spanned by the explicit or implicit node that represents P in T_S. We can find the range of SA_S where P prefixes every suffix by performing binary search twice over SA_S. A naïve binary search takes $O(m \log n)$ time: We maintain the boundaries, L and R, of the current search interval and in each iteration we compare the median string from the range L to R in SA_S, with P, and update L and R accordingly. This can be improved to $O(m + \log n)$ time if we have access to additional arrays storing the value of the longest common prefixes between a selection of strings from SA_S. We construct the suffix array from the suffix tree in $O(n)$ time.

3 Deterministic index for packed strings

In this section we describe how to construct and query our deterministic index for packed strings. This structure is the basis for our result in Thm 1. For short patterns where $m < \log_\sigma n - 1$ we store tabulated data that enables us to answer queries fast. We construct the tables in $O(n)$ time and space and answer queries in $O(\log\log\sigma + \text{occ})$ time. For long patterns where $m \geq \log_\sigma n - 1$ we use a combination of a suffix tree and a suffix array that we construct in $O(n)$ time and space such that queries take $O(m/\alpha + \log\log n + \text{occ})$ time. For $m \geq \log_\sigma n - 1$ we have that $\log\log n = \log(\frac{\log n}{\log\sigma}\log\sigma) = \log\log_\sigma n + \log\log\sigma \leq \log(\log_\sigma n - 1) + 1 + \log\log\sigma \leq \log m + 1 + \log\log\sigma$. This gives us a query time of $O(m/\alpha + \log m + \log\log\sigma + \text{occ})$ for the deterministic packed index. We need the following connections between T_S and SA_S: For each explicit node t in T_S we store a reference to the range of SA_S that corresponds to the leaves spanned by the subtree of t and for each index in SA_S we store a reference to the corresponding leaf in T_S that represents the same string.

We first describe our word accelerated algorithm for matching patterns in SA_S that we need for answering queries on long patterns. Then we describe how to build and use the data structures for answering queries on short and long patterns.

3.1 Packed matching in SA_S

We now show how to word accelerate the suffix array matching algorithm by Manber and Myers [10]. They spend $O(m)$ time reading P but by reading α characters in constant time we can reduce this to $O(m/\alpha)$. We let $\text{LCP}(i, j)$ denote the length of the longest common prefix between the suffixes $\mathsf{SA}_S[i]$ and $\mathsf{SA}_S[j]$ and obtain the result in Lemma 4.

▶ **Lemma 4.** *Given the suffix array SA_S over the packed string S and a data structure for answering the relevant* LCP *queries, we can find the lexicographic predecessor of a packed pattern P of length m in SA_S in $O(m/\alpha + \log n)$ time where α is the number of characters we can pack in a word.*

In the algorithm by Manber and Myers we maintain the left and right boundaries of the current search interval of SA_S denoted by L and R and the length of the longest common prefix between $\mathsf{SA}_S[L]$ and P, and between $\mathsf{SA}_S[R]$ and P, that we denote by l and r, respectively. Initially the search interval is the whole range of SA_S such that $L = 1$ and $R = n$. In an iteration we do as follows: If $l = r$ we start comparing $\mathsf{SA}_S[M]$ with P from index $l + 1$ until we find a mismatch and update either L and l, or R and r, depending on whether $\mathsf{SA}_S[M]$ is lexicographically larger or smaller than P. Otherwise, when $l \neq r$, we perform an LCP query that enable us to either halve the range of SA_S without reading from P or start comparing $\mathsf{SA}_S[M]$ with P from index $l+1$ as in the $l = r$ case. When $l > r$ there are three cases: If $\text{LCP}(L, M) > l$ then P is lexicographically larger than $\mathsf{SA}_S[M]$ and we set L to M and continue with the next iteration. If $LCP(L, M) < l$ then P is lexicographically smaller than $\mathsf{SA}_S[M]$ and we set R to M and set r to $\text{LCP}(L, M)$ and continue with the next iteration. If $LCP(L, M) = l$ then we compare $\mathsf{SA}_S[M]$ and P from index $l + 1$ until

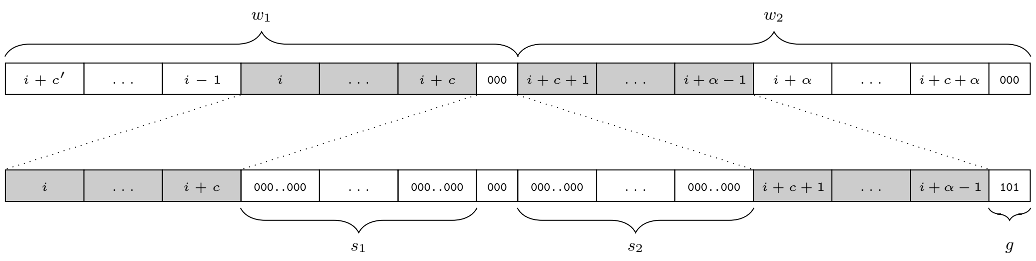


Figure 1 Alignment of α characters that extends over a word boundary where $c' = c + 1 - \alpha$. The relevant part of the lower word w_1 and upper word w_2 is combined with bitwise shifts, a bitwise or and the g bits on the right is set to 0.

we find a mismatch. Let that mismatch be at index $l + i$. If the mismatch means that P is lexicographically smaller than $\mathsf{SA}_S[M]$ then we set R to M and set r to $l + i - 1$ and continue with the next iteration. If the mismatch means that P is lexicographically larger than $\mathsf{SA}_S[M]$ then we set L to M and set l to $l + i - 1$ and continue with the next iteration. Three symmetrical cases exists when $r > l$.

We generalize their algorithm to work on word packed strings such that we can compare α characters in constant time. In each iteration where we need to read from P we align the next α characters from P and $\mathsf{SA}_S[M]$ such that we can compare them in constant time: Assume that we need to read the range from i to $i + \alpha - 1$ in P. If this range of characters is contained in one word we do not need to align. Otherwise, we extract the relevant parts of the words that contain the range with bitwise shifts and combine them in w_{align} with a bitwise or. See Figure 1. We align the α characters from $\mathsf{SA}_S[M]$ in the same way and store them in w'_{align}.

We use a *bitwise exclusive or* operation between w_{align} and w'_{align} to construct a word where the most significant set bit is at a bit position that belong to the mismatching character with the lowest index. We obtain the position of the most significant set bit in constant time with the technique of Fredman and Willard [8]. From this we know exactly how many of the next α characters that match and we can increase i accordingly. Since every mismatch encountered result in a halving of the search range of SA_S we can never read more than $O(\log n)$ incomplete chunks. The number of complete chunks we read is bounded by $O(m/\alpha)$. Overall we obtain a $O(m/\alpha + \log n)$ time algorithm for matching in SA_S. This result is summarized in Lemma 4.

3.2 Handling short patterns

Now we show how to answer count, locate and lexicographic predecessor queries on short patterns. We store an array containing an index for every possible pattern P where $m < \log_\sigma n - 1$ and at the index we store a pointer to the deepest node in T_S that prefixes P. We call this node d_P. We use d_P as the basis for answering every query on short patterns. We assume that the range in SA_S spanned by d_P goes from l to r. We answer predecessor queries as follows: If P is lexicographically smaller than $\mathsf{SA}_S[0]$ then P has no predecessor in SA_S. Otherwise, we find the predecessor as follows: If d_P represents P then the predecessor of P is located at index $l - 1$ of SA_S. Otherwise, we assume that d_P prefixes P with i characters and need to decide whether P continues on an edge out of d_P or P deviates from T_S in d_P. We do this by querying the predecessor data structure over the children of d_P with the character at position $i + 1$ of P. If this query does not return an edge, then $P[i + 1]$ is lexicographically smaller than the first character of every edge out of d_P, and the predecessor of P is the string located at index $l-1$ of SA_S. If this query returns an edge e_{pred} then there are two cases.

Case 1: The first character of e_{pred} is not identical to $P[i+1]$. Then the predecessor of P is the lexicographically largest string in the subtree under e_{pred}.

Case 2: The first character on e_{pred} is identical to $P[i+1]$. In this case, if there exists an edge e'_{pred} out of d_P on the left side of e_{pred}, then the predecessor of P is the lexicographically largest string in the subtree under e'_{pred} and otherwise the predecessor is the string at index $l-1$ of SA_S.

We report the node in T_S that represents the predecessor of P.

We let e_{pred} be defined as above and answer count queries as follows: If d_P represents P we return the number of leaves spanned by d_P in T_S. If P instead continues and ends on e_{pred} we report the number of leaves spanned by the subtree below e_{pred}. We answer locate queries in the same way but instead of reporting the range we report the strings in the range.

We find d_P in $O(1)$ time and e_{pred} in $O(\log\log\sigma)$ time. In total we answer predecessor and count queries in $O(\log\log\sigma)$ time and locate queries in $O(\log\log\sigma + \text{occ})$ time

Since $m < \log_\sigma n - 1$ there exists $\sigma + \sigma^2 + \ldots + \sigma^{\lfloor \log_\sigma n-1 \rfloor} \leq \sigma^{\lfloor \log_\sigma n \rfloor} \leq \sigma^{\log_\sigma n} = n$ short patterns and we compute them in $O(n)$ time by performing a preorder traversal of T_S bounded to depth $\log_\sigma n - 1$. Let d_P be the node we are currently visiting and let d_{next} be the node we visit next. When we visit d_P we fill the tabulation array for every string that is lexicographically larger than or equal to the string represented by d_P and lexicographically smaller than the string represented by d_{next}. Every short string can be stored in a word of memory and therefore we can index the tabulation array with the numerical value of the word that represent the string. We fill each of these indices with a pointer to d_P since d_P is the deepest node in T_S that represents a string that prefixes these strings. We can store the tabulation array in $O(n)$ space.

3.3 Handling long patterns

Now we show how to answer count, locate and lexicographic predecessor queries on long patterns. We first give an overview of our solution followed by a detailed description of the individual parts. In T_S we distinguish between *light* and *heavy* nodes. If a subtree under a node spans at least $\log^2 \log n$ leaves, we call the node heavy, otherwise we call it light. A node is a heavy branching node if it has at least two heavy children and all the heavy nodes constitute a subtree that we call the heavy tree. We decompose the heavy tree into micro trees of height α and we augment every micro tree with a data structure that enables navigation from root to leaf in constant time. For micro trees containing a heavy branching node we do this with deterministic hashing and for micro trees without a heavy branching node we just compare the relevant part of P with the one unique path of the heavy tree that goes through the micro tree. To avoid navigating the light nodes we in each light node store a pointer to the range of SA_S that the node spans. We construct two predecessor data structures for each micro tree: The *light predecessor* structure over the strings represented by the light nodes that are connected to the heavy nodes in the micro tree and the *heavy predecessor* structure over the heavy nodes in the micro tree. We answer queries on P as follows: We traverse the heavy tree in chunks of α characters until we are unable to traverse a complete micro tree. This means that P either continues in a light node, ends in the micro tree or deviates from T_S in the micro tree. We can decide if P continues in a light node with the light predecessor structure and if this is the case we answer the query with the packed matching algorithm on the range of SA_S spanned by the light node. Otherwise, we use the heavy predecessor structure for finding d_P in the micro tree and use d_P for answering the

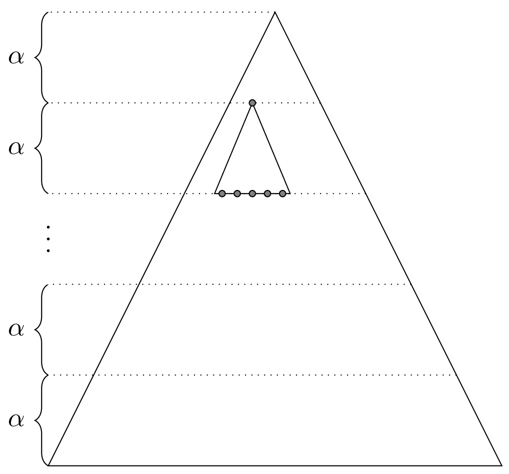


Figure 2 The decomposition of HT_S in micro trees of height α. One micro tree is shown with the root at string depth α and the boundary nodes at string depth 2α.

query as in section 3.2. The following sections describes in more detail how we build our data structure and answer queries.

3.3.1 Data structure

This section describes our data structure in details. If a subtree under a node in T_S spans at least $\log^2 \log n$ leaves, we call the node heavy. The heavy tree HT_S is the induced subgraph of all the the heavy nodes in T_S. We decompose HT_S into *micro trees* of string depth α. This decomposition into micro trees of height α was also employed by Takagi et al. [13]. A node, explicit or implicit, is a boundary node if its string depth is a multiple of α. Except for the original root and leaves of HT_S, each boundary node belongs to two micro trees i.e., a boundary node at depth $d\alpha$ is the root in a micro tree that starts at string depth $d\alpha$ and is a leaf in a micro tree that starts at string depth $(d-1)\alpha$. Figure 2 shows the decomposition of HT_S into micro trees of string depth α.

We augment every micro tree with information that enables us to navigate from root to leaf in constant time. To avoid using too much space we promote only some of the implicit boundary nodes to explicit nodes. We distinguish between three kinds of micro trees:

- **Type 1.** At least one heavy branching node exists in the micro tree: We promote the root and leaves to explicit nodes and use deterministic hashing to navigate the micro tree from root to leaf. Because the micro tree is of height α, each of the strings represented by the leaves in the micro tree fits in a word and can be used as a key for hashing. We say that the root is a hashing node and the leaves are hashed nodes. We will postpone the analysis of time and space used by the micro trees that use hashing for navigation.
- **Type 2.** No heavy branching node exists in the micro tree: When the micro tree does not contain a heavy branching node, the micro tree is simply a path from root to leaf. Here we distinguish between two cases:
  - **Type 2a.** The micro tree contains an explicit non branching heavy node: We promote the root and leaf to explicit nodes. Navigating from root to leaf takes constant time by comparing the string represented by the leaf with the appropriate part of P. We charge the space increase from the promotion of the root and leaf to the explicit non branching heavy node. Since there are at most n explicit non branching heavy nodes we never promote more than $2n$ implicit nodes from type 2a micro trees.

- **Type 2b.** The micro tree does not contain an explicit heavy node: Let t be a micro tree with no explicit heavy nodes. If the root of t is a leaf in a micro tree that contains an explicit heavy node, we promote the root of t to an explicit node and store a pointer to the root of the nearest micro tree below t that contains an explicit heavy node. The path from root to root corresponds to a substring in S that we navigate by comparing this string to the appropriate part of P. We charge the space increase from the promotion of the root to the heavy node descendant. Since we have at most n explicit heavy nodes we promote no more than n implicit nodes from type 2b micro trees. If the root of t is a leaf in a micro tree without an explicit heavy node we do not promote the root of t.

We say that a node in T_S is a heavy leaf if it is a heavy node with no heavy children. We want to bound the number of heavy branching nodes and heavy leaves. Every heavy leaf spans at least $\log^2 \log n$ leaves of T_S. This means we can have at most $n/\log^2 \log n$ heavy leaves in T_S. Since we have at most one branching heavy node per heavy leaf the number of heavy branching nodes is at most $n/\log^2 \log n$.

We want to bound the number of implicit nodes that are promoted to explicit hashed nodes. This number is critical for constructing all hash functions in $O(n)$ time. We bound the number of promoted hashed nodes by associating each with the nearest descendant that is either a heavy branching node or a heavy leaf: Let l be a promoted hashed node in a micro tree that contain a heavy branching node h. Then every promoted hashed node above l is associated with h or a node above h in the tree. Hence, no other promoted node can be associated with the first encountered heavy branching or leaf node below l. Since we have at most $O(n/\log^2 \log n)$ heavy branching and heavy leaf nodes we also have at most $O(n/\log^2 \log n)$ implicit nodes that are promoted to explicit hashed nodes.

With deterministic hashing from Lemma 2 the total time for constructing the explicit hashing nodes are given as follows. Here H is the set of all the hash functions and we bound the elements in every hash function h to $n/\log^2 \log n$.

$$O\left(\sum_{h\in H} |h| \log^2 \log |h|\right) = O\left(\sum_{h\in H} |h| \log^2 \log(n/\log^2 \log n)\right)$$
$$= O\left(\log^2 \log(n/\log^2 \log n) \cdot \sum_{h\in H} |h|\right) = O\left(\log^2 \log(n/\log^2 \log n) \frac{n}{\log^2 \log n}\right) = O(n)$$

Summing the elements of every hash function is bounded by the maximum number of promoted nodes, i.e. $O(n/\log^2 \log n)$. To conclude, we spend linear time constructing the hash functions in the micro trees that contain a heavy branching node.

We associate two predecessor data structures with each micro tree that contains a heavy node: The first predecessor structure contains every light node that is a child of a heavy node in the micro tree. We call this predecessor data structure for the *light predecessor structure* of the micro tree. The key for each light node is the string on the path from the root of the micro tree to the node itself padded with character \$ such that every string has length α. These keys are ordered lexicographically in the predecessor data structure and a successful query yields a pointer to the node. The second predecessor structure is similar to the first but contains every heavy node in the micro tree. We call this predecessor structure for the *heavy predecessor structure.* We use Lemma 3 for the predecessor structures. The total size of every light and heavy predecessor structures is $O(n)$ and a query in both takes $O(\log \log n)$ because the universe is of size $(\sigma + 1)^{\alpha}$.

For each light node that are a child of a heavy node we additionally store pointers to the range of SA_S that corresponds to the leaves in T_S that the light node spans.

3.3.2 Answering queries

We answer queries on long patterns as follows. First we search for the deepest micro tree in HT_S where the root prefixes P. We do this by navigating the heavy tree in chunks of α characters starting from the root. Assuming that we have already matched a prefix of P consisting of i chunks of α characters we need to show how to match the $(i+1)$th chunk: If the micro tree is of type 1 and P has length at least $(i+1)\alpha$, we try to hash the substring $P[i\alpha,(i+1)\alpha]$. If we obtain a node v from the hash function we continue matching chunk $P[(i+1)\alpha,(i+2)\alpha]$ from v. If the micro tree is of type 2 we compare α sized chunks of P with the string on the unique path from root to the first micro tree with an explicit root and continue matching from here. We have found the deepest micro tree where the root prefixes P when we are unable to match a complete chunk of α characters or are unable to reach a micro tree with an explicit root. From this micro tree we need to decide whether the query is answered by searching SA_S from a light node or answered by finding d_P in the micro tree, where d_P is defined as in Section 3.2, i.e. the deepest node in T_S that prefixes P. We check if P continues in a light node by querying the light predecessor structure of the micro tree with the next unmatched α characters from P and pad with character \$ if less than α characters remain unmatched in P. If the light node returned by the query represents a string that prefixes P we answer the query by searching the range of SA_S spanned by the light node with the packed matching algorithm.

When P does not continue in a light node we instead find and use d_P for answering the query: If the micro tree is of type 2b or the root of the micro tree represents P then d_P is the root of the micro tree. Otherwise, we find d_P with a technique, very similar to a technique used by Fredman and Willard [8], that queries the heavy predecessor structure three times as follows: We call the remaining part of P, padded to length α with character \$, for p_0. We first query the predecessor structure with p_0 which yields a node that represents a string n_0. We then construct a string, p_1, that consists of the longest common prefix of p_0 and n_0, and as above, padded to length α. We query the predecessor structure with p_1 which yields a new node that represents a string n_1. We then construct a string, p_2, that consists of the longest common prefix of p_0 and n_1, again padded to length α. At last, we query the predecessor structure with p_2 which returns d_P. Given d_P, we answer count, locate and lexicographic predecessor queries exactly as we did in Section 3.2.

Now we prove the correctness of our queries. First we prove that if P continues in a light node then the query in the light predecessor structure returns that light node: Assume that P goes through the light node l_P that has a heavy parent in the micro tree T_p and that we query the light predecessor structure with the string Q_α. Let L_{pred} be the string that represents l_P in the light predecessor structure. Since P goes through l_P then L_{pred} is identical or lexicographically smaller than Q_α. Let L'_{pred} be the successor of L_{pred} in the light predecessor structure. Since L_{pred} is lexicographically smaller than L'_{pred} and has a longer common prefix with Q_α than L'_{pred} has with Q_α, then L'_{pred} must be lexicographically larger than Q_α. Since Q_α is identical or lexicographically larger than L_{pred} and lexicographically smaller than L'_{pred}, a query on Q_α in the light predecessor structure will return l_P.

We now prove that the queries in the heavy predecessor structure always returns d_P: Because P is not prefixed by a leaf of the micro tree or a light node from the light predecessor structure we know that d_P is a heavy node in the micro trie. In Figure 3, d_P is depicted and P either ends on or deviates from the edge e that leads to the tree T_2. The trees T_1,

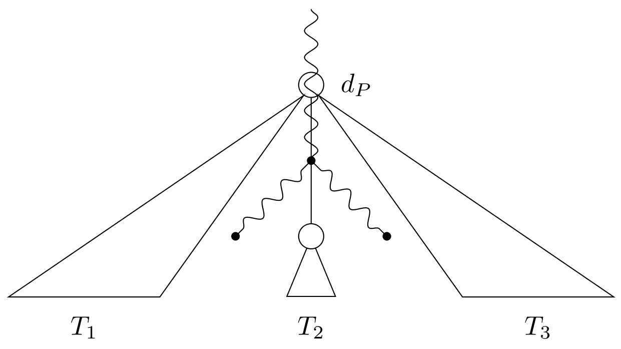


Figure 3 Searching for a prefix of P in HT_S.

T_2 and T_3 combined with d_P and the edge e constitute the subtree of d_P. If P deviates to the left or ends on e then P is lexicographically smaller than every string represented in T_2. If P deviates to the right then P is lexicographically larger than every string represented in T_2. Assume that P deviates to the right on e. Then the query to the heavy predecessor structure with pattern p_0 will yield n_0 that represents the lexicographically largest string in T_2. The pattern p_1 will then be represented by the implicit node from where P deviates from e. The pattern p_1 is lexicographically smaller than every string represented in T_2 and a query will yield n_2 as the lexicographically largest node in T_1 or, if T_1 is empty, the node d_P. Either way, the query on p_2 will yield the node d_P. We can make similar arguments for the other cases where P ends on e, deviates left from e, ends at d_P or goes through d_P without following e.

The following gives an analysis of the running time of our queries. We spend at most $O(m/\alpha)$ time traversing the heavy tree. Both predecessor structures contains strings over a universe of size n such that a query takes $O(\log\log n)$ time using Lemma 3. Each light node spans at most $\log^2\log n$ leaves which corresponds to an interval of length $\log^2\log n$ in SA_S that we search in $O(m/\alpha + \log\log\log n)$ time with the word accelerated algorithm for matching in SA_S. Overall, we spend $O(m/\alpha + \log\log n)$ time for answering count and lexicographic predecessor queries and $O(m/\alpha + \log\log n + \mathrm{occ})$ time for answering locate queries. Since we only query this data structure for patterns where $m \geq \log_\sigma n - 1$ we have that $\log\log n = \log(\frac{\log n}{\log\sigma}\log\sigma) = \log\log_\sigma n + \log\log(\sigma) \leq \log(\log_\sigma n - 1) + 1 + \log\log(\sigma) \leq \log(m) + 1 + \log\log(\sigma)$, such that we answer count and lexicographic predecessor queries in $O(m/\alpha + \log m + \log\log\sigma)$ time and locate queries in $O(m/\alpha + \log m + \log\log\sigma + \mathrm{occ})$ time. Combined with our solution for patterns where $m < \log_\sigma n - 1$, that answer the queries in $O(\log\log\sigma)$ and $O(\log\log\sigma + \mathrm{occ})$ time, respectively, we can for patterns of *any* length answer count and lexicographic predecessor queries in $O(m/\alpha + \log m + \log\log\sigma)$ time and locate queries in $O(m/\alpha + \log m + \log\log\sigma + \mathrm{occ})$ time. This is our main result which is summarized in Thm 1.

References

1 Djamal Belazzougui. Worst-case efficient single and multiple string matching on packed texts in the word-RAM model. *J. Discrete Algorithms*, 14:91–106, 2012. `doi:10.1016/j.jda.2011.12.011`.

2 Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. Towards optimal packed string matching. *Theor. Comput. Sci.*, 525:111–129, 2014. `doi:10.1016/j.tcs.2013.06.013`.

3 Philip Bille. Fast searching in packed strings. *J. Discrete Algorithms*, 9(1):49–56, 2011. `doi:10.1016/j.jda.2010.09.003`.

4 Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Proceedings of the 33rd International Colloquium on Automata, Languages, and Programming (ICALP 2006)*, volume 4051 of *LNCS*, pages 358–369. Springer, 2006. `doi:10.1007/11786986_32`.

5 Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. `doi:10.1145/355541.355547`.

6 Johannes Fischer and Paweł Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 160–171. Springer, 2015. `doi:10.1007/978-3-319-19929-0_14`.

7 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. `doi:10.1145/828.1884`.

8 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. `doi:10.1016/0022-0000(93)90040-4`.

9 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. `doi:10.1017/CBO9780511574931`.

10 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. `doi:10.1137/0222058`.

11 Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. `doi:10.1145/321941.321946`.

12 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Proceedings of the 35th International Colloquium on Automata, Languages, and Programming (ICALP 2008)*, volume 5125 of *LNCS*, pages 84–95. Springer, 2008. `doi:10.1007/978-3-540-70575-8_8`.

13 Takuya Takagi, Shunsuke Inenaga, Kunihiko Sadakane, and Hiroki Arimura. Packed compact tries: A fast and efficient data structure for online string processing. In Veli Mäkinen, Simon J. Puglisi, and Leena Salmela, editors, *Proceedings of the 27th International Workshop on Combinatorial Algorithms (IWOCA 2016)*, volume 9843 of *LNCS*, pages 213–225. Springer, Springer, 2016. `doi:10.1007/978-3-319-44543-4_17`.

14 Peter Weiner. Linear pattern matching algorithms. In H. Raymond Strong, editor, *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE Computer Society, 1973. `doi:10.1109/SWAT.1973.13`.

Representing the Suffix Tree with the CDAWG

Djamal Belazzougui[1] and Fabio Cunial[2]

1 **CERIST (Research Centre for Scientific and Technical Information), Algiers, Algeria**
`dbelazzougui@cerist.dz`

2 **Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany**
`cunial@mpi-cbg.de`

Abstract

Given a string T, it is known that its suffix tree can be represented using the compact directed acyclic word graph (CDAWG) with e_T arcs, taking overall $O(e_T + e_{\overline{T}})$ words of space, where $\overline{T}$ is the reverse of T, and supporting some key operations in time between $O(1)$ and $O(\log\log n)$ in the worst case. This representation is especially appealing for highly repetitive strings, like collections of similar genomes or of version-controlled documents, in which e_T grows sublinearly in the length of T in practice. In this paper we augment such representation, supporting a number of additional queries in worst-case time between $O(1)$ and $O(\log n)$ in the RAM model, without increasing space complexity asymptotically. Our technique, based on a heavy path decomposition of the suffix tree, enables also a representation of the suffix array, of the inverse suffix array, and of T itself, that takes $O(e_T)$ words of space, and that supports random access in $O(\log n)$ time. Furthermore, we establish a connection between the reversed CDAWG of T and a context-free grammar that produces T and only T, which might have independent interest.

1998 ACM Subject Classification E.1 Data Structures, E.4 Coding and Information Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases CDAWG, suffix tree, heavy path decomposition, maximal repeat, context-free grammar

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.7

1 Introduction

Given a string T of length n, the compressed suffix tree [21, 18] and the compressed suffix array can take an amount of space that is bounded by the k-th order empirical entropy of T, but such measure of redundancy is known not to be meaningful when T is very repetitive [11], e.g. a collection of similar genomes. The space taken by such compressed data structures also includes a $o(n)$ term, typically $O(n/\text{polylog}(n))$, which can become an obstacle when T is very compressible. Rather than compressing the suffix array, we could compress a *differentially encoded suffix array* [12], which stores at every position the difference between two consecutive positions of the suffix array. Previous approaches have compressed such differential array using grammar or Lempel-Ziv compression [12], and the same methods can be used to compress the suffix tree topology and the LCP array [1, 17]. Such heuristics, however, have either no theoretical guarantee on their performance [1, 17], or weak ones [12].

In previous research [4] we described a representation of the suffix tree of T that takes space proportional to the size of the compact directed acyclic word graph (CDAWG) of T, and that supports a number of operations in time between $O(1)$ and $O(\log\log n)$ in the worst case (see Table 2). If T is highly repetitive, the size of the CDAWG of T is known to grow

Table 1 Time complexity of the operations on the suffix tree of a string T described in this paper ($n = |T|$).

	`leftmostLeaf` `rightmostLeaf`	`selectLeaf`, `lca` SA$[i]$, ISA$[i]$, LCP$[i]$ PLCP$[i]$, $T[i]$	SA$[i..j]$ ISA$[i..j]$ LCP$[i..j]$	$T[i..j]$	`depth` `ancestor` `strAncestor`
1	$O(1)$	$O(\log n)$	$O(\log n + j - i)$	$O(\log n + \frac{j-i}{\log_\sigma n})$	$O(\log n)$
2	$O(1)$	$O(\log n)$	$O(\log n + j - i)$	$O(\log n + \frac{j-i}{\log_\sigma n})$	

Table 2 Complexity of the operations on the suffix tree of a string T described in [4] ($n = |T|$).

	Space (words)	`stringDepth` `nLeaves`, `height` `locateLeaf` `firstChild`, `child`	`isAncestor` `leafRank`	`parent` `nextSibling`	`suffixLink`	`weinerLink`
1	$O(e_T + e_{\overline{T}})$	$O(1)$	$O(1)$	$O(\log\log n)$	$O(\log\log n)$	$O(\log\log n)$
2	$O(e_T)$	$O(1)$		$O(\log\log n)$	$O(1)$	

sublinearly in the length of T in practice (see e.g. [4]). Being related to maximal repeats, the size of the CDAWG is also a natural measure of redundancy for very repetitive strings. Moreover, since the difference between consecutive suffix array positions is the same inside isomorphic subtrees of the suffix tree, and since such isomorphic subtrees are compressed by the CDAWG, the CDAWG itself can be seen as a grammar that produces the differential suffix array, and the suffix tree can be seen as the parse tree of such grammar: this provides a formal substrate to heuristics that grammar-compress the differential suffix array.

In this paper we further exploit the compression of isomorphic subtrees of a suffix tree induced by the CDAWG, augmenting the representation of the suffix tree described in [4] with a number of additional operations that take between $O(1)$ and $O(\log n)$ time in the worst case (see Table 1), without increasing space complexity asymptotically. We also describe CDAWG-based representations of the suffix array, of the inverse suffix array, of the LCP array, and of T itself, with $O(\log n)$ random access time.

Our approach is related to the work of Bille et al [7], in which a straight-line program (effectively a DAG) that produces the balanced parentheses representation of a tree with n nodes, is used to support operations on the topology of the tree in $O(\log n)$ time. Applying such compression to the suffix tree achieves the space bounds of this paper, but it only supports operations on the topology of the tree, and it supports each operation in $O(\log n)$ time, whereas we achieve either constant or $O(\log\log n)$ time for some key primitives.

2 Preliminaries

We work in the RAM model with word length at least $\log n$ bits, where n is the length of a string that is implicit from the context, and we index strings and arrays starting from one.

2.1 Graphs

We assume the reader to be familiar with the notions of tree and of directed acyclic graph (DAG). By `lca`(u, v) we denote the lowest common ancestor of nodes u and v in a tree. By *weighted tree* we mean a tree with nonnegative weights on the edges, and we use $\omega(u, v)$ to denote the weight of edge (u, v). Weighted DAGs are defined similarly. In this paper we only deal with *ordered* trees and DAGs, in which there is a total order among the out-neighbors of every node. The i-th leaf of a tree is its i-th leaf in depth-first order, and to every node v of a tree we assign the compact interval $[\mathtt{sp}(v)..\mathtt{ep}(v)]$, in depth-first order, of all leaves that belong to the subtree rooted at v. In this paper we use the expression DAG also for directed acyclic *multigraphs*, allowing distinct arcs to have the same source and destination nodes. In what follows we consider just DAGs with exactly one source and one sink.

We denote by $\mathcal{T}(G)$ the tree generated by DAG G with the following recursive procedure: the tree generated by the sink of G consists of a single node; the tree generated by a node v of G that is not the sink, consists of a node whose children are the roots of the subtrees generated by the out-neighbors of v in G, taken in order, and connected to their parent by edges whose weight, if any, is identical to the weight of the corresponding arc of G. Note that: (1) every node of $\mathcal{T}(G)$ is generated by exactly one node of G; (2) a node of G different from the sink generates one or more internal nodes of $\mathcal{T}(G)$, and the subtrees of $\mathcal{T}(G)$ rooted at all such nodes are isomorphic; (3) the sink of G can generate one or more leaves of $\mathcal{T}(G)$; (4) there is a bijection, between the set of root-to-leaf paths in $\mathcal{T}(G)$ and the set of source-to-sink paths in G, such that every path $v_1, \ldots, v_k$ in $\mathcal{T}(G)$ is mapped to a path $v'_1, \ldots, v'_k$ in G, and such that $\omega(v_i, v_{i+1}) = \omega(v'_i, v'_{i+1})$ for all $i \in [1..k-1]$ if $\mathcal{T}(G)$ is weighted. Symmetrically, given any tree T, merging all subtrees with identical topology and edge weights produces a DAG G such that $\mathcal{T}(G) = T$: we denote such DAG by $\mathcal{G}(T)$. Clearly $\mathcal{G}(\mathcal{T}(G)) = G$.

Given nodes v and w of $\mathcal{T}(G)$ such that v is an ancestor of w, let $\mathtt{nLeaves}(v)$ be the number of leaves in the subtree rooted at v, and let $\mathtt{left}(v, w)$ (respectively, $\mathtt{right}(v, w)$) be the number of leaves in the subtree rooted at v that precede (respectively, follow) in depth-first order the leaves in the subtree rooted at w. A *heavy path decomposition* of $\mathcal{T}(G)$ [14] is the following marking: for every node u, we mark exactly one edge (u, v) as *heavy* if $\mathtt{nLeaves}(v)$ is the largest among all children of u, with ties broken arbitrarily (Figure 1a). We call *light* an edge that is not heavy, and we call *heavy path* a maximal sequence of nodes $v_1, \ldots, v_k$ such that (v_i, v_{i+1}) is heavy for all $i \in [1..k-1]$. Note that v_k is a leaf, every node of $\mathcal{T}(G)$ belongs to exactly one heavy path, distinct heavy paths are connected by light edges, and every path from the root to a leaf contains $O(\log N)$ light edges, or equivalently intersects $O(\log N)$ heavy paths, where N is the number of leaves of $\mathcal{T}(G)$. Heavy paths are disjoint in $\mathcal{T}(G)$, but their corresponding paths in G form a spanning tree $\tau(G)$, with $O(n)$ nodes and edges, rooted at the sink of G, where n is the number of nodes of G (Figure 1b).

2.2 Strings

Let $\Sigma = [1..\sigma]$ be an integer alphabet, let $\# = 0 \notin \Sigma$ be a separator, and let $T \in [1..\sigma]^{n-1}\#$ be a string. Given a string $W \in [1..\sigma]^k$, we call the *reverse of* W the string $\overline{W}$ obtained by reading W from right to left. For a string $W \in [1..\sigma]^k\#$ we abuse notation, and we denote by $\overline{W}$ the string $\overline{W[1..k]}\#$. Given a substring W of T, let $\mathcal{P}_T(W)$ be the set of all starting positions of W in the circular version of T. A *repeat* W is a string that satisfies $|\mathcal{P}_T(W)| > 1$. We denote by $\Sigma^{\ell}_T(W)$ the set of characters $\{a \in [0..\sigma] : |\mathcal{P}_T(aW)| > 0\}$ and by $\Sigma^{r}_T(W)$ the set of characters $\{b \in [0..\sigma] : |\mathcal{P}_T(Wb)| > 0\}$. A repeat W is *right-maximal* (respectively, *left-maximal*) iff $|\Sigma^{r}_T(W)| > 1$ (respectively, iff $|\Sigma^{\ell}_T(W)| > 1$). It is well known that T can have at most $n-1$ right-maximal repeats and at most $n-1$ left-maximal repeats. A *maximal repeat* of T is a repeat that is both left- and right-maximal. It is also well known that a maximal repeat $W \in [1..\sigma]^m$ of T is the equivalence class of all the right-maximal strings $\{W[1..m], \ldots, W[k..m]\}$ such that $W[k+1..m]$ is left-maximal, and $W[i..m]$ is not left-maximal for all $i \in [2..k]$.

For reasons of space we assume the reader to be familiar with the notion of *suffix tree* ST_T of T (see e.g. [13] for an introduction), which we do not define here. We denote by $\ell(\gamma)$, or equivalently by $\ell(u,v)$, the string label of edge $\gamma = (u,v) \in E$, and we denote by $\ell(v)$ the string label of node $v \in V$. It is well known that a substring W of T is right-maximal iff $W = \ell(v)$ for some internal node v of the suffix tree. We assume the reader to be familiar with the notion of *suffix link* connecting a node v with $\ell(v) = aW$ for some $a \in [0..\sigma]$ to a node w with $\ell(w) = W$. Here we just recall that inverting the direction of all suffix links yields the so-called *explicit Weiner links*.

Finally, we assume the reader to be familiar with the notion and uses of the Burrows-Wheeler transform of T (see e.g. [10]). In this paper we use BWT_T to denote the BWT of T, and we use $\mathtt{range}(W) = [\mathtt{sp}(W)..\mathtt{ep}(W)]$ to denote the lexicographic interval of a string W in a BWT that is implicit from the context. As customary, we denote by $C[0..\sigma]$ the array such that $C[a]$ equals the number of occurrences of characters lexicographically smaller than a in T. For a node v of ST_T, we use the shortcut $\mathtt{range}(v) = [\mathtt{sp}(v)..\mathtt{ep}(v)]$ to denote $\mathtt{range}(\ell(v))$. We say that $\mathsf{BWT}_T[i..j]$ is a *run* iff $\mathsf{BWT}_T[k] = c \in [0..\sigma]$ for all $k \in [i..j]$, and moreover if any substring $\mathsf{BWT}_T[i'..j']$ such that $i' \leq i$, $j' \geq j$, and either $i' \neq i$ or $j' \neq j$, contains at least two distinct characters. We denote by $\mathcal{R}_T$ the set of all triplets (c,i,j) such that $\mathsf{BWT}_T[i..j]$ is a run of character c. Given a string $T \in [1..\sigma]^{n-1}\#$, we call *run-length encoded BWT* (RLBWT_T) any representation of BWT_T that takes $O(|\mathcal{R}_T|)$ words of space, and that supports the well known rank and select operations: see for example [15, 16, 23]. It is easy to implement a version of RLBWT_T that supports rank in $O(\log\log n)$ time and select in $O(\log\log n)$ time [4].

2.3 CDAWG

The *compact directed acyclic word graph* of a string T (denoted by CDAWG_T in what follows) is the minimal compact automaton that recognizes the suffixes of T [8, 9]. We denote by e_T the number of arcs in CDAWG_T. The CDAWG of T can be seen as the minimization of ST_T, in which all leaves are merged to the same node (the sink) that represents T itself, and in which all nodes except the sink are in one-to-one correspondence with the maximal repeats of T [20]. Every arc of CDAWG_T is labeled by a substring of T, and the out-neighbors $w_1,\ldots,w_k$ of every node v of CDAWG_T are sorted according to the lexicographic order of the distinct labels of arcs $(v,w_1),\ldots,(v,w_k)$. Since there is a bijection between the nodes of CDAWG_T and the maximal repeats of T, the node v' of CDAWG_T with $\ell(v') = W$ is the equivalence class of the nodes $\{v_1,\ldots,v_k\}$ of ST_T such that $\ell(v_i) = W[i..|W|]$ for all $i \in [1..k]$, and such that $v_k, v_{k-1},\ldots,v_1$ is a maximal unary path of explicit Weiner links. The subtrees of ST_T rooted at all such nodes are isomorphic, and $\mathcal{T}(\mathsf{CDAWG}_T) = \mathsf{ST}_T$ (Figure 1b). It follows that the set of right-maximal strings that belong to the equivalence class of a maximal repeat can be represented by a single integer k, and a right-maximal string can be identified by the maximal repeat W it belongs to, and by the length of the corresponding suffix of W. Similarly, a suffix of T can be identified by a length relative to the sink of CDAWG_T.

In BWT_T, the right-maximal strings in the same equivalence class of a maximal repeat enjoy the following properties:

▶ **Property 1** ([4]). *Let $\{W[1..m],\ldots,W[k..m]\}$ be the right-maximal strings that belong to the equivalence class of maximal repeat $W \in [1..\sigma]^m$ of a string T, and let $\mathtt{range}(W[i..m]) = [p_i..q_i]$ for $i \in [1..k]$. Then: (1) $|q_i - p_i + 1| = |q_j - p_j + 1|$ for all i and j in $[1..k]$; (2) $\mathsf{BWT}_T[p_i..q_i] = W[i-1]^{q_i-p_i+1}$ for $i \in [2..k]$. Conversely, $\mathsf{BWT}_T[p_1..q_1]$ contains at least two distinct characters. (3) $p_{i-1} = C[c] + \mathtt{rank}_c(\mathsf{BWT}_T, p_i)$ and $q_{i-1} = p_{i-1} + q_i - p_i$ for $i \in [2..k]$, where $c = W[i-1] = \mathsf{BWT}_T[p_i]$. (4) $p_{i+1} = \mathtt{select}_c(\mathsf{BWT}_T, p_i - C[c])$ and*

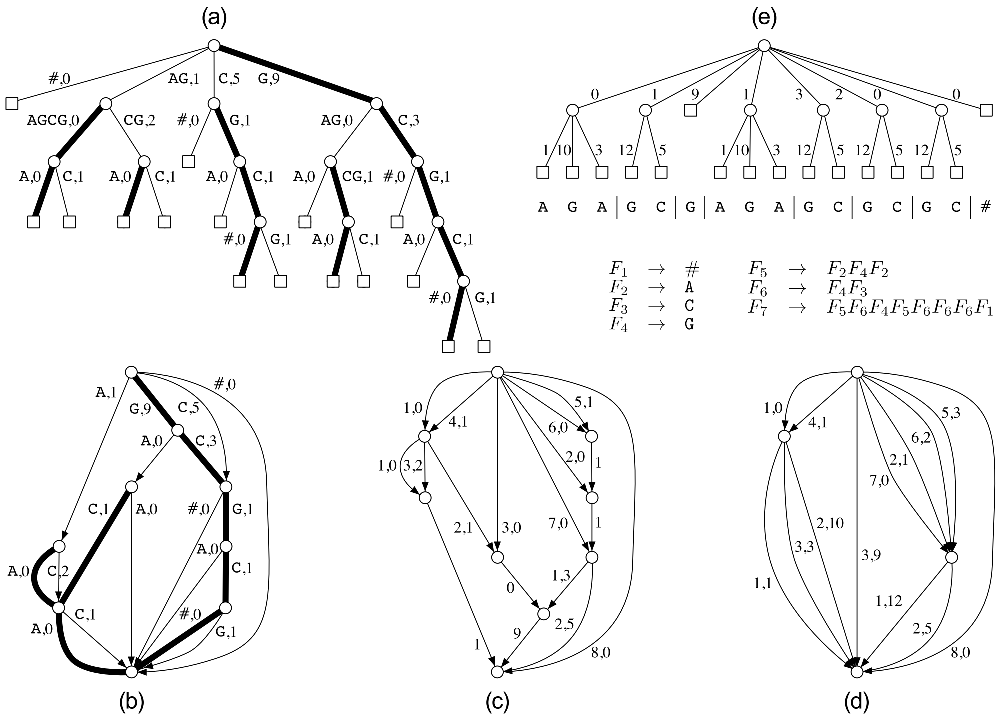


Figure 1 The data structures used in this paper for string $T =$ AGAGCGAGAGCGCGC#. (a) The suffix tree of T. Edges to leaves are labelled by just the first character of their string. The weight of edge (u, v) is $\texttt{sp}(v) - \texttt{sp}(u)$. Heavy edges according to the number of leaves are bold. (b) The CDAWG of T. Just the first character of each arc label is shown. Arc weights are from (a). Arcs in the spanning tree τ are bold. (c) The reverse CDAWG. Arc (u, v) is labelled by pair (x, y), where x is the order of v among the out-neighbors of u, and y is the weight in (b). (d) The compacted version of (c). (e) The weighted tree generated from (d), and the corresponding grammar.

$q_{i+1} = p_{i+1} + q_i - p_i$ for $i \in [1..k-1]$, where $c = W[i]$ is the character that satisfies $C[c] < p_i \leq C[c+1]$. (5) Let $c \in [0..\sigma]$, and let $\texttt{range}(W[i..m]c) = [x_i..y_i]$ for $i \in [1..k]$. Then, $x_i = p_i + x_1 - p_1$ and $y_i = p_i + y_1 - p_1$.

Character c in Property 1.4 can be computed in $O(\log\log n)$ time using a predecessor data structure that uses $O(\sigma)$ words of space [26]. Moreover, the equivalence class of a maximal repeat is related to the equivalence classes of its in-neighbors in the CDAWG in the following way:

▶ **Property 2** ([4]). *Let w be a node in CDAWG_T with $\ell(w) = W \in [1..\sigma]^m$, and let $\mathcal{S}_w = \{W[1..m], \ldots, W[k..m]\}$ be the right-maximal strings that belong to the equivalence class of node w. Let $\{v^1, \ldots, v^t\}$ be the in-neighbors of w in CDAWG_T, and let $\{V^1, \ldots, V^t\}$ be their labels. Then, $\mathcal{S}_w$ is partitioned into t disjoint sets $\mathcal{S}^1_w, \ldots, \mathcal{S}^t_w$ such that $\mathcal{S}^i_w = \{W[x^i+1..m], W[x^i+2..m], \ldots, W[x^i+|\mathcal{S}_{v^i}|..m]\}$, and the right-maximal string $V^i[p..|V^i|]$ labels the parent of the locus of the right-maximal string $W[x^i+p-1..m]$ in ST_T.*

Property 2 applied to the sink v of CDAWG_T partitions T into x left-maximal factors, where x is the number of in-neighbors of v (Figure 1e). Moreover, by Property 2, it is natural to say that in-neighbor v^i of node w is smaller than in-neighbor v^j of node w iff $x^i < x^j$, or equivalently if the strings in $\mathcal{S}^i_w$ are longer than the strings in $\mathcal{S}^j_w$. We call $\overline{\mathsf{CDAWG}}_T$ the ordered DAG obtained by applying this order to the reverse of CDAWG_T, i.e. to the DAG obtained by inverting the direction of all arcs of CDAWG_T (Figure 1c). Note that $\overline{\mathsf{CDAWG}}_T$ is not the same as $\mathsf{CDAWG}_{\overline{T}}$, although there is a bijection between their sets of nodes. Note also that some nodes of $\overline{\mathsf{CDAWG}}_T$ can have just one out-neighbor: for brevity we denote by $\overline{\mathsf{CDAWG}}_T$ the graph obtained by collapsing every such node v, i.e. by adding the weight (if any) of the only outgoing arc from v to the weights of all incoming arcs to v, and by redirecting such incoming arcs to the out-neighbor of v (Figure 1d). This can be done in linear time by an inverse topological sort of $\overline{\mathsf{CDAWG}}_T$ that starts from its sink.

The source of $\overline{\mathsf{CDAWG}}_T$ is the sink of CDAWG_T, which is the equivalence class of all suffixes of T in string order, and there is a bijection between the distinct paths of $\overline{\mathsf{CDAWG}}_T$ and the suffixes of T. It follows that:

▶ **Property 3.** *The i-th leaf of $\mathcal{T}(\overline{\mathsf{CDAWG}}_T)$ in depth-first order corresponds to the i-th suffix of T in string order.*

Thus, $\mathcal{T}(\overline{\mathsf{CDAWG}}_T)$ can be seen as the parse tree of a context-free grammar that generates T and only T, and $\overline{\mathsf{CDAWG}}_T$ can be seen as such grammar (Figure 1e). This implies a lower bound on the size of the CDAWG:

▶ **Lemma 1.** *Let f be the function that maps the length of a string to the size of its CDAWG, and let g be the function that maps the length of a string T to the size of the smallest grammar that produces T and only T. Then, $f \in \Omega(g)$.*

In some classes of strings the size of the CDAWG is asymptotically the same as the size of the smallest grammar that produces the string, but in other classes the ratio between the two sizes reaches its maximum, $O(n/\log n)$: see Section 2.1 in [4].

Let G be an ordered DAG, let $\gamma = (v, w)$ be an edge of $\mathcal{T}(G)$, and assume that we assign to γ a weight equal to the offset $\mathtt{sp}(w) - \mathtt{sp}(v)$ between the first leaf in the leaf interval of w and the first leaf in the leaf interval of v (Figure 1a). Thus, we can compute the depth-first order of a leaf of $\mathcal{T}(G)$ by summing the weights of all edges in its root-to-leaf path. Note that edges (v, w) and (v', w') in $\mathcal{T}$ such that v and v' correspond to the same node v'' in G, and such that w and w' correspond to the same node w'' in G, have the same weight: in the case of CDAWG_T and ST_T, this is equivalent to Property 1.5, and weights are offsets between the starting positions of nested BWT intervals (Figure 1b). Assume that every such weight is stored inside arc (v'', w'') of CDAWG_T, and that weights are preserved when building $\overline{\mathsf{CDAWG}}_T$. Then, one plus the sum of all weights in the source-to-sink path of $\overline{\mathsf{CDAWG}}_T$ that corresponds to suffix $T[i..|T|]$ is the lexicographic rank of suffix $T[i..|T|]$ (see e.g. Figures 1d and 1e). Equivalently:

▶ **Property 4.** *Let arc (u, v) of CDAWG_T be weighted by $\mathtt{sp}(v') - \mathtt{sp}(u')$, where v' (respectively, u') is a node of ST_T that belongs to the equivalence class of v (respectively, u), and v' is a child of u' in ST_T. Then, the lexicographic rank of suffix $T[i..|T|]$ is one plus the sum of all weights in the path from the root of $\mathcal{T}(\overline{\mathsf{CDAWG}}_T)$ to the i-th leaf of $\mathcal{T}(\overline{\mathsf{CDAWG}}_T)$ in depth-first order.*

2.4 Representing the suffix tree with the CDAWG

It is known that Properties 1 and 2 enable two encodings of ST_T that take $O(e_T + e_{\overline{T}})$ words of space each, and that support the operations in Table 2 with the specified time complexities [4]. Since the rest of this paper builds on the representation described in [4], we summarize it here for completeness.

It is known that $|\mathcal{R}_T|$ is at most the number of arcs in CDAWG_T [4], thus augmenting CDAWG_T with RLBWT_T does not increase space asymptotically. For every node v of

CDAWG_T, we store: $|\ell(v)|$ in a variable v.`length`; the number v.`size` of right-maximal strings that belong to its equivalence class; the interval $[v.\texttt{first}..v.\texttt{last}]$ of $\ell(v)$ in BWT_T; a linear-space predecessor data structure [26] on the boundaries induced on the equivalence class of v by its in-neighbors (Property 2); and pointers to the in-neighbor that corresponds to the interval associated with each boundary. For every arc $\gamma = (v,w)$ of CDAWG_T, we store the first character of $\ell(\gamma)$ in a variable γ.`char`, and the number of characters of the right-extension implied by γ in a variable γ.`right`. We also add to the CDAWG all arcs (v,w,c) such that w is the equivalence class of the destination of a Weiner link from v labeled by character c in ST_T, and the reverse of all explicit Weiner link arcs. We represent a node v of ST_T as a tuple $\texttt{id}(v) = (v', |\ell(v)|, i, j)$, where v' is the node in CDAWG_T that corresponds to the equivalence class of v, and $[i..j]$ is the interval of $\ell(v)$ in BWT_T. Implementing operations $\texttt{stringDepth}(\texttt{id}(v))$, $\texttt{nLeaves}(\texttt{id}(v))$ (which returns the number of leaves of the subtree of ST_T rooted at a given node), $\texttt{isAncestor}(\texttt{id}(v), \texttt{id}(w))$ (which returns true iff a node v of ST_T is an ancestor of another node w of ST_T), $\texttt{suffixLink}(\texttt{id}(v))$, $\texttt{weinerLink}(\texttt{id}(v))$, $\texttt{locateLeaf}(\texttt{id}(v))$ (which returns the position in T of a leaf v of ST_T) and $\texttt{leafRank}(\texttt{id}(v))$ (which returns the position of a leaf v of ST_T in lexicographic order) is straightforward using Properties 1.3 and 1.4, and implementing $\texttt{parent}(\texttt{id}(v))$, $\texttt{child}(\texttt{id}(v))$ and $\texttt{nextSibling}(\texttt{id}(v))$ is easy using Properties 2 and 1.5.

Removing all implicit Weiner link arcs from our data structure achieves $O(e_T)$ words of space, and still supports all queries except following implicit Weiner links. We can further drop RLBWT_T and remove from $\texttt{id}(v)$ the interval of $\ell(v)$ in BWT_T, still supporting most of the original queries in the same amount of time, and `suffixLink` in constant time. The data structure after such removals corresponds to the second row of Table 2. Conversely, storing also the RLBWT of $\overline{T}$, and the interval in such RLBWT of the reverse of the maximal repeat that corresponds to every node of the CDAWG, allows one to also read the label of an edge γ of ST_T in $O(\log\log n)$ time per character, for the same asymptotic space complexity.

3 Additional suffix tree operations

In this paper we augment the representation of the suffix tree described in Section 2.4, enabling it to support a number of additional suffix tree operations in $O(\log n)$ time without increasing space complexity asymptotically. At the core of our methods lies a heavy path decomposition of CDAWG_T along the lines of [7], which we summarize in what follows to keep the paper self-contained.

▶ **Definition 2** (Smooth function). Let T be a tree, let $v_1, v_2, \ldots, v_N$ be its N leaves in depth-first order, let f be a function that assigns a real number to every leaf, and let $F[1..N]$ be the array that stores at position i the value of $f(v_i)$. We say that f is *smooth with respect to T* iff $F[\texttt{sp}(v)..\texttt{ep}(v)] = F[\texttt{sp}(w)..\texttt{ep}(w)]$ for every pair of internal nodes v, w of T that are generated by the same node of $\mathcal{G}(T)$.

For example, let T be the parse tree of a string S generated by a context-free grammar: the function that assigns character $T[i]$ to every position i of T is smooth.

▶ **Lemma 3** ([7]). *Let G be a DAG with n arcs such that every node has exactly two out-neighbors, let f be a smooth function with respect to $\mathcal{T}(G)$, and let N be the number of leaves of $\mathcal{T}(G)$. There is a data structure that, given a number $i \in [1..N]$, returns $f(u_i)$ in $O(\log N)$ time, where u_i is the i-th leaf of $\mathcal{T}(G)$ in depth-first order. Moreover, given two integers $1 \leq i \leq j \leq N$, the data structure returns in $O(\log N)$ time the node of G that corresponds to $\texttt{lca}(u_i, u_j)$, and it returns in $O(\log N + j - i)$ time the sequence of values*

$f(u_i), f(u_{i+1}), \dots, f(u_j)$, where u_h is the h-th leaf of $\mathcal{T}(G)$ in depth-first order. Such data structure takes $O(n)$ words of space.

Proof Sketch. For each heavy path $v_1, \dots, v_k$ of $\mathcal{T}(G)$, we store at v_1 values $\mathtt{nLeaves}(v_1)$, $\mathtt{left}(v_1, v_k)$, $f(v_k)$, a predecessor data structure on the set of values $\{\mathtt{left}(v_1, v_i) : i \in [2..k]\}$, and a predecessor data structure on the set of values $\{\mathtt{right}(v_1, v_i) : i \in [2..k]\}$. If we query v_1 with the position i_1 of a leaf in the subtree rooted at v_1, such data structures allow us to detect the largest $j \in [1..k]$ such that v_j is an ancestor of the query leaf. If $j = k$ we return $f(v_k)$, otherwise we take the light edge (v_j, w) and we recur on w, which is itself the first node of a heavy path. This solution takes $O(\log N)$ queries to prefix-sum data structures, but the total size of all prefix-sum data structures can be $O(N^2)$.

Note that a predecessor query on the left and right predecessor data structures stored at the first node v_1 of a heavy path of $\mathcal{T}(G)$ can be implemented with a *weighted ancestor* query[1] on $\tau(G)$, if we assign to each arc (v, w) of G that also belongs to $\tau(G)$ a left weight equal to zero if w is the left successor of v, and equal to the number of leaves in the left successor of v otherwise (the right weight is defined similarly). Using a suitable data structure for weighted ancestor queries allows one to achieve $O(n)$ words of space and overall $O(\log N \cdot \log\log N)$ query time after $O(n)$ preprocessing of G. More advanced data structures that implement weighted ancestor queries on $\tau(G)$ allow one to achieve the claimed bounds [7].

Given $\mathcal{T}(G)$, we proceed as follows to extract the values of all leaves in a depth-first interval $[i..j]$. Inside every node v of a heavy path, we store an auxiliary right pointer to the closest descendant of v in the heavy path whose right child is light. We symmetrically store an auxiliary left pointer. Then, we traverse $\mathcal{T}(G)$ top-down as described above, but searching for both the i-th leaf u_i and the j-th leaf u_j at the same time: when the nodes w and w' of G that result from such searches are different, we know that one is a descendant of the other in $\tau(G)$, and the node of G that corresponds to $\mathtt{lca}(u_i, u_j)$ in $\mathcal{T}(G)$ is the one whose number of leaves equals $\max\{\mathtt{nLeaves}(w), \mathtt{nLeaves}(w')\}$. Then we continue the search for the two leaves separately: during the search for u_i (respectively, u_j) we follow all right (respectively, left) auxiliary pointers in all heavy paths, and we concatenate the corresponding nodes in a left (respectively, right) linked list. The size of such lists is $O(j - i)$, and computing sequence $f(u_i), \dots, f(u_j)$ from the lists takes $O(j - i)$ time. The same approach can be applied to G, at the cost of $O(n)$ preprocessing time and space. ◀

Since a node v of $\mathcal{T}(G)$ can be uniquely identified by an interval of leaves in depth-first order, Lemma 3 effectively implements a map from the identifier of a node in $\mathcal{T}(G)$ to the identifier of its corresponding node in G.

▶ **Lemma 4.** *Lemma 3 holds also for a DAG in which all nodes have out-degree* at least *two.*

Proof. We expand every node v with out-degree $d > 2$ into a binary directed tree, with $d-1$ artificial internal nodes, whose d leaves are the out-neighbors of v in G. We also store in each artificial internal node w a pointer $w.\mathtt{real} = v$. The size of such expanded DAG G' is still $O(n)$, where n is the number of arcs of G, $\mathcal{T}(G')$ is a binary tree with the same number of leaves as $\mathcal{T}(G)$, there is a bijection between the leaves of $\mathcal{T}(G)$ and the leaves of $\mathcal{T}(G')$ such that the i-th leaf in depth-first order in $\mathcal{T}(G)$ corresponds to the i-th leaf in depth-first order in $\mathcal{T}(G')$, and the extension of f to the leaves of $\mathcal{T}(G')$ induced by such bijection is

[1] A *weighted ancestor query* (v, k) on a tree with weights on the edges asks for the lowest ancestor u of a node v such that the sum of weights in the path from u to v is at least k [2].

smooth with respect to $\mathcal{T}(G')$. Note that, if Lemma 3 returns an artificial node w as the result of a lowest common ancestor query, it suffices to return w.`real` instead. ◀

Lemma 3 can be adapted to support queries on another class of functions:

▶ **Definition 5** (Telescoping function)**.** Let f be a function that assigns a real number to any path of any weighted graph. We say that f is *telescoping* iff:

1. Given a path $P = v_1, v_2, \dots, v_k$, $f(P) = g(\omega(v_1, v_2)) \circ \dots \circ g(\omega(v_{k-1}, v_k))$, where $\omega(v_i, v_j)$ is the weight of edge or arc (v_i, v_j), g is a function that can be computed in constant time, and $x \circ y$ is a binary associative operator with identity element $\mathbb{I}$ that can be computed in constant time.
2. $f(v_1, \dots, v_k) \geq f(v_1, \dots, v_i)$ for all $i < k$, and $f(v_1, \dots, v_k) \geq f(v_i, \dots, v_k)$ for all $i > 1$.
3. For every path $v_1, \dots, v_i, \dots, v_j, \dots, v_k$, $f(v_i, \dots, v_j)$ can be computed in constant time given $f(v_1, \dots, v_i)$ and $f(v_1, \dots, v_j)$, or given $f(v_i, \dots, v_k)$ and $f(v_j, \dots, v_k)$.

We call y the *inverse* of x with respect to $\circ$ iff $x \circ y = y \circ x = \mathbb{I}$. For example, the sum of edge weights in a path is telescoping, $\mathbb{I} = 0$, and the inverse of x is $-x$. Note that a telescoping function is not necessarily smooth.

▶ **Lemma 6.** *Let G be a weighted DAG with n arcs in which every node has at least two out-neighbors, let f be a telescoping function, and let N be the number of leaves of $\mathcal{T}(G)$. There is a data structure that, given a number $i \in [1..N]$, evaluates f in $O(\log N)$ time on the path from the root of $\mathcal{T}(G)$ to the i-th leaf in depth-first order. Moreover, given two numbers $1 \leq i \leq j \leq N$, the data structure:*

1. *Evaluates f in $O(\log N)$ time on the path from the root of $\mathcal{T}(G)$ to $\mathtt{lca}(u_i, u_j)$, where u_i and u_j are the i-th and j-th leaf of $\mathcal{T}(G)$ in depth-first order.*
2. *Returns in $O(\log N + j - i)$ time the sequence of values $f(u_i), f(u_{i+1}), \dots, f(u_j)$, where $f(u_h)$ is the value of function f evaluated on the path from the root of $\mathcal{T}(G)$ to the h-th leaf in depth-first order.*
3. *If $[i..j]$ is the identifier of node v in $\mathcal{T}(G)$, given a nonnegative number k, returns in $O(\log N)$ time the node of G that corresponds to the highest ancestor w of v in $\mathcal{T}(G)$ such that f, evaluated on the path from the root of $\mathcal{T}(G)$ to w, is at least k* (weighted ancestor query)*.*

Such data structure takes $O(n)$ words of space.

Proof. If a node v in the DAG has out-degree greater than two, we expand it as described in Lemma 4, assigning weight $\mathbb{I}$ to all arcs that end in an artificial internal node of the expanded DAG, and assigning the weight of arc (v, w) to the arc that connects an artificial internal node to out-neighbor w of v in G. We also store a pointer to v inside each artificial internal node. Let G' be the expanded version of G. At every node v of G' we store variable v.`count` $= f(P(v))$, where $P(v)$ is the path from v to the sink of G' that uses only arcs in the spanning tree $\tau(G')$. We traverse G' as described in Lemma 3: at the current node u, we compute its highest ancestor v in $\tau(G')$ that lies in the path, from the source of G' to the sink of G', that corresponds to the i-th leaf of $\mathcal{T}(G')$. We use u.`count` and v.`count` to evaluate f in constant time on the path from u to v along $\tau(G')$, and we cumulate such value to the output. For each arc (v, w) that does not belong to $\tau(G')$, we compute $g(\omega(v, w))$ and we cumulate it to the output.

To evaluate f on the path from the root of $\mathcal{T}(G)$ to $\mathtt{lca}(v_i, v_j)$, we follow the extraction strategy described in Lemma 3, using in the last step u.`count` and v.`count`, where u is the current node and v is the (possibly artificial) node of G' that corresponds to $\mathtt{lca}(v_i, v_j)$ in $\mathcal{T}(G')$. We use the extraction strategy of Lemma 3 also to evaluate f on all leaves of $\mathcal{T}(G)$

in the depth-first interval $[i..j]$: every time we take a right pointer or a left pointer (u, v), we cumulate weight $u.\texttt{count} \circ y$ to the current value of f, where y is the inverse of $v.\texttt{count}$, and we start from such value of f when visiting the subgraph of G' that starts at v.

To support weighted ancestor queries on f and $\mathcal{T}(G)$, we build a data structure that supports *level ancestor queries* on $\tau(G')$: given a node v and a path length d, such data structure returns the ancestor u of v in $\tau(G')$ such that the path from the root of $\tau(G')$ to u contains exactly d nodes. The level ancestor data structure described in [5, 6] takes $O(n)$ words of space and it answers queries in constant time. We search again for the i-th and j-th leaf in parallel, cumulating f using the weights of light arcs and of heavy paths as done before. Let u be the current node in this search, and let x be the current value of f: if $x < k$, but the value of f is at least k at the next node v such that the path from u to v in G' belongs to $\tau(G')$, we binary search the nodes w on the path from u to v, using level ancestor queries from u and comparing $x \circ u.\texttt{count} \circ y$ to k, where y is the inverse of $w.\texttt{count}$. The result of the binary search is not an artificial node. ◀

Let $[i..j]$ be the identifier of a node of $\mathcal{T}(G)$, and let $[i'..j']$ be the identifier of its weighted ancestor. Since it is easy to transform the node of G that corresponds to $[i'..j']$ into interval $[i'..j']$ itself, Lemma 6 effectively implements a map from $[i..j]$ to $[i'..j']$ in $O(\log N)$ time.

Applying Lemma 6 to CDAWG_T is all we need to support the additional operations in Table 1 efficiently:

▶ **Theorem 7.** *Let $T \in [1..\sigma]^{n-1}\#$ be a string. There are two representations of ST_T that support the operations in Table 1 and in Table 2 with the specified time and space complexities.*

Proof. Operation $\texttt{selectLeaf}(i)$ returns an identifier of the i-th leaf of ST_T in lexicographic order. Recall from Section 2.4 that we store in a variable $\gamma.\texttt{right}$ the number of characters of the right extension implied by arc γ of CDAWG_T. Thus, the length of the suffix associated with a leaf of ST_T (or equivalently, the position of that leaf in right-to-left string order) is the sum of all weights in the source-to-sink path of CDAWG_T that corresponds to the leaf. Since the sum of such weights is a telescoping function, we use the data structures in Lemma 6, built on these weights, to compute the value s of the sum in $O(\log n)$ time, and we return tuple (v, s, i, i), where v is the sink of CDAWG_T. Returning $|T| - s + 1$ instead is enough to implement $\mathsf{SA}_T[i]$. Since Lemma 6 supports also the extraction of all values of a telescoping function inside a depth-first range of leaves $[i..j]$, implementing $\mathsf{SA}_T[i..j]$ is straightforward.

Operation $\texttt{lca}(i, j)$ returns the identifier of the lowest common ancestor, in ST_T, of the i-th and the j-th leaf in lexicographic order. We use Lemma 6 to compute both the node v of CDAWG_T that corresponds to such common ancestor, and its string depth s, returning tuple (v, s, x, y), where the range $[x..y] \supseteq [i..j]$ of the lowest common ancestor is computed during the top-down traversal of CDAWG_T using the weighted ancestor data structure on $\tau(\mathsf{CDAWG}_T)$. A similar approach allows one to return $\mathsf{LCP}[i]$, and a slight variation of the approach used to compute $\mathsf{SA}_T[i..j]$ supports also $\mathsf{LCP}[i..j]$. Operation $\texttt{depth}(\texttt{id}(v))$ returns the depth of the node v of ST_T whose identifier is $\texttt{id}(v)$. Since $\texttt{id}(v)$ contains the range $[i..j]$ of v in BWT_T, we can proceed as in operation $\texttt{lca}(i, j)$, and return the length of the path that the search traversed from the source of CDAWG_T to the node of CDAWG_T that corresponds to v. Operation $\texttt{leftmostLeaf}(\texttt{id}(v))$ returns the identifier of the smallest leaf in lexicographic order in the subtree of ST_T rooted at node v. Let $\texttt{id}(v) = (v', \ell, i, j)$, and let W be the longest maximal repeat in the equivalence class of node v'. Then, $\texttt{leftmostLeaf}(\texttt{id}(v)) = (w', \ell + v'.\texttt{left}, i, i)$, where w' is the sink of CDAWG_T, and $v'.\texttt{left}$ is the string length of the path, in ST_T, that goes from the node of ST_T with string label W to its leftmost leaf. We store $v'.\texttt{left}$ at every node

v' of the CDAWG. Operation `rightmostLeaf` can be handled symmetrically. Operation $\texttt{stringAncestor}(\texttt{id}(v), d)$ (respectively, $\texttt{ancestor}(\texttt{id}(v), d)$) returns the identifier of the highest ancestor of v in ST_T whose string depth (respectively, depth) is at least d. This can be implemented with the weighted ancestor query provided by Lemma 6, where the weight of arc γ of CDAWG_T is $\gamma.\texttt{right}$ (respectively, one).

Finally, by Property 4, we support access to the value of the inverse suffix array at string position i by building the data structures of Lemma 6 on the compacted $\overline{\mathsf{CDAWG}}_T$, with arc weights corresponding to offsets between nested BWT intervals, and with a weighted ancestor data structure on $\tau(\overline{\mathsf{CDAWG}}_T)$ based on offsets between string positions. Note that all arcs that end at the same node of the compacted $\overline{\mathsf{CDAWG}}_T$ have distinct weights. Then, we evaluate the sum of edge weights from the root of $\mathcal{T}(\overline{\mathsf{CDAWG}}_T)$ to its i-th leaf in depth-first order. Implementing $\mathsf{ISA}_T[i..j]$ is also straightforward, and $\mathsf{PLCP}[i]$ can be supported using $\mathsf{ISA}_T[i]$. Assume that, while building $\overline{\mathsf{CDAWG}}_T$, we keep the first character of the label of every arc of CDAWG_T that starts from the root, we propagate it during compaction, and we store it at the nodes as described in Lemma 3. Then, since $\mathcal{T}(\overline{\mathsf{CDAWG}}_T)$ is a parse tree of T, we can also return $T[i]$ in $O(\log n)$ time and $T[i..j]$ in $O(\log n + j - i)$ time. Since the compacted reversed CDAWG is a grammar for T, the time for extracting $T[i..j]$ can be reduced to $O(\log n + (j-i)/\log_\sigma n)$ by using the `access` query described in [3]. ◀

▶ **Corollary 8.** *Given a string $T \in [1..\sigma]^{n-1}\#$, there is a representation of the suffix array of T, of the inverse suffix array of T, of the LCP array of T, of the permuted LCP array of T, and of T itself, that takes $O(e_T)$ words of space, and that supports random access to any position in $O(\log n)$ time.*

Note that Corollary 8 yields immediately a representation of the compressed suffix array of T [22] that takes $O(e_T)$ words of space.

4 Extensions and conclusion

Our data structures provide immediate support for a number of queries of common use in pattern matching, in addition to those listed in Tables 1 and 2. For example, recall that an *internal pattern matching query* (i, j) asks for all the `occ` starting positions of $T[i..j]$ inside a string T of length n. We can support such query in $O(\log n + \texttt{occ})$ time, by combining an inverse suffix array query, a string ancestor query, and the extraction strategy of Lemma 6. Similarly, combining an inverse suffix array query with a lowest common ancestor query and a string depth query, allows one to compute the longest common prefix between two given suffixes of T in $O(\log n)$ time. Along the same lines, operation $\texttt{letter}(\texttt{id}(v), i)$, which returns the i-th character of the label of node v of the suffix tree, can be supported in $O(\log n)$ time. We can also implement in constant time operation $\texttt{deepestNode}(\texttt{id}(v))$, which returns the identifier of the first node with largest depth (or string depth) in the subtree of the suffix tree rooted at v [19]. If we choose not to store the BWT intervals of the nodes of the CDAWG as in the second row of Tables 1 and 2, we can implement in $O(\log n)$ time operation $\texttt{suffixLink}(\texttt{id}(v), i)$, which returns the identifier of the node of the suffix tree that is reachable from v after taking i suffix links. This can be done by computing $\texttt{lca}(\texttt{id}(u), \texttt{id}(w))$, where $\texttt{id}(v) = (v', k, a, b)$, $\texttt{id}(u) = (z, e, x, x)$, $\texttt{id}(w) = (z, f, y, y)$, z is the sink of the CDAWG, $e = |T| - (\mathsf{SA}[a] + i) + 1$, $f = |T| - (\mathsf{SA}[b] + i) + 1$, $x = \mathsf{ISA}[\mathsf{SA}[a] + i]$ and $y = \mathsf{ISA}[\mathsf{SA}[b] + i]$. By using the representation described in [7], we can also support in $O(\log n)$ time operations like $\texttt{preorderSelect}(i)$, $\texttt{postorderSelect}(i)$, $\texttt{preorderRank}(v)$, $\texttt{postorderRank}(v)$, $\texttt{treeLevelSuccessor}(v)$ and `treeLevelPredecessor`(v). However, some operations on the topology of the suffix tree are not yet implemented by our data structures (see e.g. [19]): it would be interesting to know whether they can be supported efficiently within the same space budget.

Recall from Section 2.4 that our current representation of the suffix tree supports reading the label of an arc in $O(\log\log n)$ time per character, using the RLBWT of $\overline{T}$. It would be interesting to know whether this bound can be improved, and whether the RLBWT of $\overline{T}$ can be dropped. Another question for further research is whether the ubiquitous $O(\log n)$ term in Table 1 can be reduced while keeping the same asymptotic space budget, or whether a lower bound makes it impossible, along the lines of [25].

On the applied side, it is not yet clear whether there is a subset of our algorithms that is practically applicable, and whether it could achieve competitive tradeoffs with respect to state-of-the-art suffix tree representations for highly repetitive collections. It would also be interesting to try and use our data structures for tuning specific applications to repetitive strings in practice, like matching statistics and substring kernels. For example, it turns out that some weighting functions used in substring kernels are telescoping [24]. Since our data structures support matching statistics [4], and since the computation of some substring kernels can be mapped onto matching statistics [24], we can compute some substring kernels between a fixed T and a query string of length m in $O(m \log n)$ time, using a data structure that takes just $O(e_T)$ words of space.

References

1 Andrés Abeliuk, Rodrigo Cánovas, and Gonzalo Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013. doi:10.3390/a6020319.

2 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3(2):19, May 2007. doi:10.1145/1240233.1240242.

3 Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In Nikhil Bansal and Irene Finocchi, editors, *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA 2015)*, volume 9294 of *LNCS*, pages 142–154. Springer, 2015. doi:10.1007/978-3-662-48350-3_13.

4 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 26–39. Springer, Springer, 2015. doi:10.1007/978-3-319-19929-0_3.

5 Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.

6 Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994. doi:10.1016/S0022-0000(05)80002-9.

7 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015. doi:10.1137/130936889.

8 Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987. doi:10.1145/28869.28873.

9 Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In Alberto Apostolico and Jotun Hein, editors, *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM 1997)*, volume 1264 of *LNCS*, pages 116–129. Springer, 1997. doi:10.1007/3-540-63220-4_55.

10 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

11 Travis Gagie. Large alphabets and incompressibility. *Inf. Process. Lett.*, 99(6):246–251, 2006. doi:10.1016/j.ipl.2006.04.008.

12 Rodrigo González, Gonzalo Navarro, and Héctor Ferrada. Locally compressed suffix arrays. *ACM J. Exp. Algorithmics*, 19:1–1, 2015. doi:10.1145/2594408.

13 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

14 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

15 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM 2005)*, volume 3537 of *LNCS*, pages 45–56. Springer, Springer, 2005. doi:10.1007/11496656_5.

16 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010. doi:10.1089/cmb.2009.0169.

17 Gonzalo Navarro and Alberto Ordóñez Pereira. Faster compressed suffix trees for repetitive text collections. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA 2014)*, volume 8504 of *LNCS*, pages 424–435. Springer, 2014. doi:10.1007/978-3-319-07959-2_36.

18 Gonzalo Navarro and Luís M. S. Russo. Fast fully-compressed suffix trees. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2014 Data Compression Conference (DCC 2014)*, pages 283–291. IEEE, IEEE, 2014. doi:10.1109/DCC.2014.40.

19 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16, 2014. doi:10.1145/2601073.

20 Mathieu Raffinot. On maximal repeats in strings. *Inf. Process. Lett.*, 80(3):165–169, 2001. doi:10.1016/S0020-0190(01)00152-1.

21 Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Trans. Algorithms*, 7(4):53:1–53:34, 2011. doi:10.1145/2000807.2000821.

22 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

23 Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In Amihood Amir, Andrew Turpin, and Alistair Moffat, editors, *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008)*, volume 5280 of *LNCS*, pages 164–175. Springer, 2008. doi:10.1007/978-3-540-89097-3_17.

24 Alexander J. Smola and S. V. N. Vishwanathan. Fast kernels for string and tree matching. In Suzanna Becker, Sebastan Thrun, and Klaus Obermayer, editors, *Advances in Neural Information Processing Systems (NIPS 2002)*, volume 15, pages 585–592. MIT Press, 2002. URL: http://papers.nips.cc/paper/2272-fast-kernels-for-string-and-tree-matching.pdf.

25 Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In Johannes Fischer and Peter Sanders, editors, *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *LNCS*, pages 247–258. Springer, Springer, 2013. doi:10.1007/978-3-642-38905-4_24.

26 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

Position Heaps for Parameterized Strings*

Diptarama[1], Takashi Katsura[2], Yuhei Otomo[3], Kazuyuki Narisawa[4], and Ayumi Shinohara[5]

1 **Graduate School of Information Sciences, Tohoku University, Sendai, Japan**
`diptarama@shino.ecei.tohoku.ac.jp`
2 **Graduate School of Information Sciences, Tohoku University, Sendai, Japan**
`katsura@shino.ecei.tohoku.ac.jp`
3 **Graduate School of Information Sciences, Tohoku University, Sendai, Japan**
`otomo@shino.ecei.tohoku.ac.jp`
4 **Graduate School of Information Sciences, Tohoku University, Sendai, Japan**
`narisawa@ecei.tohoku.ac.jp`
5 **Graduate School of Information Sciences, Tohoku University, Sendai, Japan**
`ayumi@ecei.tohoku.ac.jp`

Abstract

We propose a new indexing structure for parameterized strings, called parameterized position heap. Parameterized position heap is applicable for parameterized pattern matching problem, where the pattern matches a substring of the text if there exists a bijective mapping from the symbols of the pattern to the symbols of the substring. We propose an online construction algorithm of parameterized position heap of a text and show that our algorithm runs in linear time with respect to the text size. We also show that by using parameterized position heap, we can find all occurrences of a pattern in the text in linear time with respect to the product of the pattern size and the alphabet size.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases string matching, indexing structure, parameterized pattern matching, position heap

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.8

1 Introduction

String matching problem is to find occurrences of a pattern string in a text string. Formally, given a text string t and a pattern string p over an alphabet Σ, output all positions at which p occurs in t. Suffix tree and suffix array are most widely used data structures and provide many applications for various string matchings (see e.g. [11, 6]).

Ehrenfeucht *et al.* [8] proposed an indexing structure for string matching, called a *position heap*. Position heap uses less memory than suffix tree does, and provides efficient search of patterns by preprocessing the text string, similarly to suffix tree and suffix array. A position heap for a string t is a *sequence hash tree* [4] for the ordered set of all suffixes of t. In [8], the suffixes are ordered in the ascending order of length, and the proposed construction algorithm processes the text from right to left. Later, Kucherov [13] considered the ordered set of suffixes in the descending order of length and proposed a linear-time

* This work is supported by Tohoku University Division for Interdisciplinary Advance Research and Education, ImPACT Program of Council for Science, Technology and Innovation (Cabinet Office, Government of Japan), and JSPS KAKENHI Grant Number JP15H05706.

online construction algorithm based on the Ukkonen's algorithm [16]. Nakashima *et al.* [14] proposed an algorithm to construct a position heap for a set of strings, where the input is given as a *trie* of the set. Gagie *et al.* [10] proposed a position heap with limited height and showed some relations between position heap and suffix array.

The parameterized pattern matching that focuses on a structure of strings is introduced by Baker [2]. Let Σ and Π be two disjoint sets of symbols. A string over $\Sigma \cup \Pi$ is called a *parameterized string* (p-string for short). In the parameterized pattern matching problem, given p-strings t and p, find positions of substrings of t that can be transformed into p by applying one-to-one function that renames symbols in Π. The parameterized pattern matching is motivated by applying to the software maintenance [1, 2, 3], the plagiarism detection [9], the analysis of gene structure [15], and so on. Similar to the basic string matching problem, some indexing structures that support the parameterized pattern matching are proposed, such as parameterized suffix tree [2], structural suffix tree [15], and parameterized suffix array [7, 12].

In this paper, we propose a new indexing structure called *parameterized position heap* for the parameterized pattern matching. The parameterized position heap is a sequence hash tree for the ordered set of prev-encoded [2] suffixes of a parameterized string. We give an online construction algorithm of a parameterized position heap based on Kucherov's algorithm [13] that runs in $O(n \log(|\Sigma| + |\Pi|))$ time and an algorithm that runs in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time to find the occurrences of a pattern in the text, where n is the length of the text, m is the length of the pattern, $|\Sigma|$ is the number of constant symbols, $|\Sigma|$ is the number of parameter symbols, and occ is the number of occurrences of the pattern in the text.

2 Notation

Let Σ and Π be two disjoint sets of symbols. Σ is a set of *constant* symbols and Π is a set of *parameter* symbols. An element of Σ^* is called a *string*, and an element of $(\Sigma \cup \Pi)^*$ is called a *parameterized string*, or *p-string* for short. For a p-string $w = xyz$, x, y, and z are called *prefix*, *substring*, and *suffix* of w, respectively. $|w|$ denotes the length of w, and $w[i]$ denotes the i-th symbol of w for $1 \leq i \leq |w|$. The substring of w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq |w|$. Moreover, let $w[:i] = w[1:i]$ and $w[i:] = w[i:|w|]$ for $1 \leq i \leq |w|$. The empty p-string is denoted by ε, that is $|\varepsilon| = 0$. For convenience, let $w[i:j] = \varepsilon$ if $i > j$. Let $\mathcal{N}$ denote the set of all non-negative integers.

Given two p-strings w_1 and w_2, w_1 and w_2 are a *parameterized match* or *p-match*, denoted by $w_1 \approx w_2$, if there exists a bijection f from the symbols of w_1 to the symbols of w_2, such that f is identity on the constant symbols [2]. We can determine whether $w_1 \approx w_2$ or not by using an encoding called *prev-encoding* defined as follows.

▶ **Definition 1** (Prev-encoding [2])**.** For a p-string w over $\Sigma \cup \Pi$, the *prev-encoding* for w, denoted by $prev(w)$, is a string x of length $|w|$ over $\Sigma \cup \mathcal{N}$ defined by

$$x[i] = \begin{cases} w[i] & \text{if } w[i] \in \Sigma, \\ 0 & \text{if } w[i] \in \Pi \text{ and } w[i] \neq w[j] \text{ for } 1 \leq j < i, \\ i - \max\{j \mid w[j] = w[i] \text{ and } 1 \leq j < i\} & \text{otherwise.} \end{cases}$$

For any p-strings w_1 and w_2, $w_1 \approx w_2$ if and only if $prev(w_1) = prev(w_2)$. For example, given $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\Pi = \{u, v, x, y\}$, $s_1 = uvuv\mathtt{a}uuv\mathtt{b}$ and $s_2 = xyxy\mathtt{a}xxy\mathtt{b}$ are p-matches where $prev(w_1) = prev(w_2) = 0022\mathtt{a}314\mathtt{b}$.

The parameterized pattern matching is a problem to find occurrences of a p-string pattern in a p-string text defined as follows.

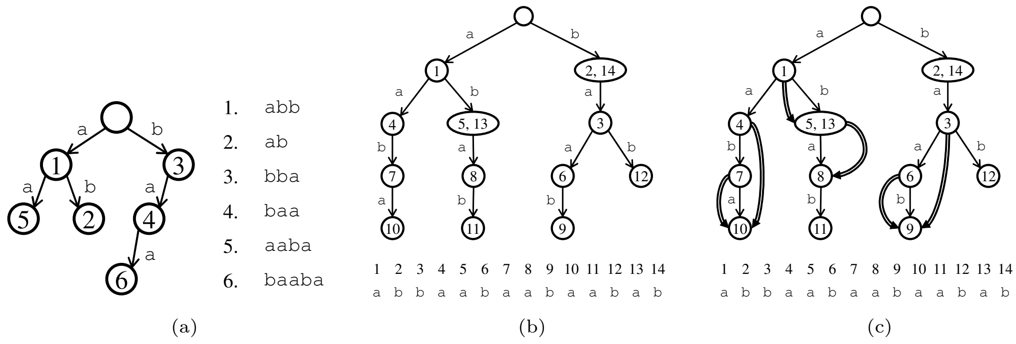


Figure 1 (a) A sequence hash tree for (aab, ab, bba, baa, aaba, baaba). (b) A position heap for a string abbaabaabaabab, (c) An augmented position heap for a string abbaabaabaabab. Maximal-reach pointers for $mrp(i) \neq i$ are illustrated by doublet arrows.

▶ **Definition 2** (Parameterized pattern matching [2]). Given two p-strings, text t and pattern p, find all positions i in t such that $t[i : i + |p| - 1] \approx p$.

For example, let us consider a text $t = uv\mathtt{a}u\mathtt{b}u\mathtt{a}v\mathtt{b}v$ and a pattern $p = x\mathtt{a}y\mathtt{b}y$ over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\Pi = \{u, v, x, y\}$. Because $p \approx t[2 : 6]$ and $p \approx t[6 : 10]$, we should output 2 and 6.

Throughout this paper, let t be a text of length n and p be a pattern of length m.

3 Position Heap

In this section, we briefly review the position heap for strings. First we introduce the *sequence hash tree* that is a trie for hashing proposed by Coffman and Eve [4]. Each edge of the trie is labeled by a symbol and each node can be identified with the string obtained by concatenating all labels found on the path from root to the node.

▶ **Definition 3** (Sequence Hash Tree). Let $\mathbf{W} = (w_1, \ldots, w_n)$ be an ordered set of strings over Σ and $\mathbf{W}_i = (w_1, \ldots, w_i)$ for $1 \leq i \leq n$. A *sequence hash tree* $SHT(\mathbf{W}) = (V_n, E_n)$ for $\mathbf{W}$ is a trie over Σ defined recursively as follows. Let $SHT(\mathbf{W}_i) = (V_i, E_i)$. Then,

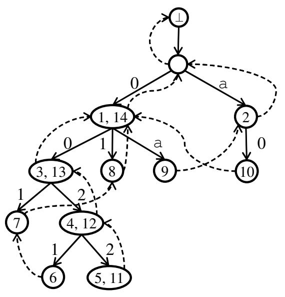
$$SHT(\mathbf{W}_i) = \begin{cases} (\{\varepsilon\}, \emptyset) & (\text{if } i = 0)\,, \\ (V_{i-1} \cup \{p_i\}, E_{i-1} \cup \{(q_i, c, p_i)\}) & (\text{if } 1 \leq i \leq n)\,. \end{cases}$$

where p_i is the shortest prefix of w_i such that $p_i \notin V_{i-1}$, and $q_i = w_i[1 : |p_i| - 1]$, $c = w_i[|p_i|]$. If no such p_i exists, then $V_i = V_{i-1}$ and $E_i = E_{i-1}$.

Each node in a sequence hash tree stores one or several indices of strings in the input set. An example of a sequence hash tree is shown in Figure 1 (a).

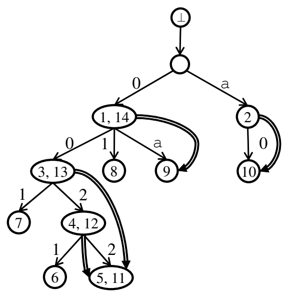
The *position heap* proposed by Ehrenfeucht *et al.* [8] is a sequence hash tree for the ordered set of all suffixes of a string. Two types of position heap are known. The first one is proposed by Ehrenfeucht *et al.* [8], that constructed by the ordered set of suffixes in ascending order of length and the second one is proposed by Kucherov [13], which constructed in descending order. We adopt the Kucherov [13] type and his online construction algorithm for constructing position heaps for parameterized strings in Section 4. Here we recall the definition of the position heap by Kucherov.

▶ **Definition 4** (Position Heap [13]). Given a string $t \in \Sigma^n$, let $\mathbf{S}_t = (t[1 :], t[2 :], \ldots, t[n :])$ be the ordered set of all suffixes of t except ε in descending order of length. The *position heap* $PH(t)$ for t is $SHT(\mathbf{S}_t)$.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	x	a	x	y	x	y	x	y	y	a	x	y	x	y
1	0	a	2	0	2	2	2	2	1	a	4	3	2	2
2		a	0	0	2	2	2	2	1	a	4	3	2	2
3			0	0	2	2	2	2	1	a	4	3	2	2
4				0	0	2	2	2	1	a	4	3	2	2
5					0	0	2	2	1	a	4	3	2	2
6						0	0	2	1	a	4	3	2	2
7							0	0	1	a	4	3	2	2
8								0	1	a	0	3	2	2
9									0	a	0	3	2	2
10										a	0	0	2	2
11											0	0	2	2
12												0	0	2
13													0	0
14														0

(a)



(b)

Figure 2 Let $\Sigma = \{\mathtt{a}\}$, $\Pi = \{x, y\}$ and $t = x\mathtt{a}xyxyxyy\mathtt{a}xyx$. (a) A parameterized position heap $PPH(t)$. Broken arrows denote suffix pointers. (b) An augmented parameterized position heap $APPH(t)$. Parameterized maximal-reach pointers for $pmrp(i) \neq i$ are illustrated by doublet arrows.

Each node except the *root* in a position heap stores either one or two integers those are beginning positions of corresponding suffixes. We call them *regular node* and *double node* respectively. Assume that i and j are positions stored by a double node v in $PH(t)$ where $i < j$, i and j are called the *primary position* and the *secondary position* respectively. Figure 1 (b) shows an example of a position heap.

In order to find occurrences of the pattern in $O(m + occ)$ time, Ehrenfeucht *et al.* [8] and Kucherov [13] added additional pointer called *maximal-reach pointer* to the position heap and called this extended data structure as *augmented position heap*. An example of an augmented position heap is showed in Figure 1 (c).

4 Parameterized Position Heap

In this section, we propose a new indexing structure called *parameterized position heap*. It is based on the position heap proposed by Kucherov [13].

4.1 Definition and Property of Parameterized Position Heap

The parameterized position heap is a sequence hash tree [4] for the ordered set of prev-encoded suffixes in the descending order of length.

▶ **Definition 5** (Parameterized Position Heap). Given a p-string $t \in (\Sigma \cup \Pi)^n$, let $\mathbf{S}_t = (prev(t[1:]), prev(t[2:]), \ldots, prev(t[n:]))$ be the ordered set of all prev-encoded suffixes of the p-string t except ε in descending order of length. The *parameterized position heap* $PPH(t)$ for t is $SHT(\mathbf{S}_t)$.

Figure 2 (a) shows an example of a parameterized position heap. A parameterized position heap $PPH(t)$ for a p-string t of length n consists of the root and nodes that corresponds to $prev(t[1:]), prev(t[2:]), \ldots, prev(t[n:])$, so $PPH(t)$ has at most $n + 1$ nodes. Each node in $PPH(t)$ holds either one or two of beginning positions of corresponding p-suffixes similar to the standard position heaps. We can specify each node in $PPH(t)$ by its primary position, its secondary position, or the string obtained by concatenating labels found on the path from the root to the node.

Different from standard position heap, $prev(t[i:]) = prev(t)[i:]$ does not necessarily hold for some cases. For example, for $t = \mathtt{xaxyxyxyyaxyxy}$, $prev(t[3:]) = 0022221\mathtt{a}4322$ while $prev(t)[3:] = 0222221\mathtt{a}4322$. Therefore, the construction and matching algorithms for the standard position heaps cannot be directly applied for the parameterized position heaps. However, we can similar properties to construct parameterized position heaps efficiently.

▶ **Lemma 6.** *For i and j, where $1 \le i \le j \le n$, if $prev(t[i:j])$ is represented in $PPH(t)$, then a prev-encoded string for any substring of $t[i:j]$ is also represented in $PPH(t)$.*

Proof. First we will show that prev-encoding of any prefix of $t[i:j]$ is represented in $PPH(t)$. From the definition of prev-encoding, $prev(t[i:j])[1:i-j] = prev(t[i:j-1])$. In other words, $prev(t[i:j-1])$ is a prefix of $prev(t[i:j])$. From the definition of $PPH(t)$, prefixes of $prev(t[i:j])$ are represented in $PPH(t)$. Therefore, $prev(t[i:j-1])$ is represented in $PPH(t)$. Similarly, $prev(t[i:j-2]), \cdots, prev(t[i:i])$ are represented in $PPH(t)$.

Next, we will show that prev-encoding of any suffix of $t[i:j]$ is represented in $PPH(t)$. From the above discussion, there are positions $b_0 < b_1 < \cdots < b_{j-i} = i$ in t such that $prev(t[b_k : b_k + k]) = prev(t[i : i+k])$. From the definition of parameterized position heap, $prev(t[b_1+1 : b_1+1])$ is represented in $PPH(t)$. Since $prev(t[b_k+1 : b_k+k])$ is a prefix of $prev(t[b_{k+1}+1 : b_{k+1}+k+1])$ for $0 < k < j-i$, if $prev(t[b_k+1 : b_k+k])$ is represented in $PPH(t)$ then $prev(t[b_{k+1}+1 : b_{k+1}+k+1])$ is also represented in $PPH(t)$ recursively. Therefore, $prev(t[b_{j-i}+1 : b_{j-i}+j-i]) = prev(t[i+1:j])$ is represented in $PPH(t)$. Similarly, $prev(t[i+2:j]), \cdots, prev(t[j:j])$ are represented in $PPH(t)$.

Since any prefix and suffix of $prev(t[i:j])$ is represented in $PPH(t)$, we can say that any substring of $prev(t[i:j])$ is represented in $PPH(t)$ by induction. ◀

4.2 Online Construction Algorithm of Parameterized Position Heap

In this section, we propose an online algorithm that constructs parameterized position heaps. Our algorithm is based on Kucherov's algorithm, although it cannot be applied easily. The algorithm updates $PH(t[1:k])$ to $PH(t[1:k+1])$ when $t[k+1]$ is read, where $1 \le k \le n-1$. Updating of the position heap begins from a special node, called the *active node*. A position specified by the active node is called the *active position*. At first, we show that there exists a position similar to the active position in the parameterized position heap.

▶ **Lemma 7.** *If j is a secondary position of a double node in a parameterized position heap, then $j+1$ is also a secondary position.*

Proof. Let i be the primary position and j be the secondary position of node v, where $i < j$. This means there is a position h such that $prev(t[i:h]) = prev(t[j:])$. By Lemma 6, there is a node that represents $prev(t[i+1:h])$. Since $prev(t[j+1:]) = prev(t[i+1:h])$, then $j+1$ will be the secondary positions of node $prev(t[i+1:h])$. ◀

Lemma 7 means that there exists a position s which splits all positions in $t[1:n]$ into two intervals, similar to the *active position* in [13]. Positions in $[1:s-1]$ and $[s:n]$ are called primary and secondary positions, respectively. We also call the position s as active position.

Assume we have constructed $PPH(t[1:k])$ and we want to construct $PPH(t[1:k+1])$ from $PPH(t[1:k])$. The primary positions $1, \ldots, s-1$ in $PPH(t[1:k])$ become primary positions also in $PPH(t[1:k+1])$, because $prev(t[i:k]) = prev(t[i:k+1])[1:k-1+1]$ holds for $1 \le i \le s-1$. Therefore, we do not need to update the primary positions.

On the other hand, the secondary positions $s, \ldots, k$ require some modifications. When inserting a new symbol, two cases can occur. The first case is that $prev(t[i:k+1])$ is not represented in $PPH(t[1:k])$. In this case, a new node $prev(t[i:k+1])$ is created as a child node of $prev(t[i:k])$ and position i becomes the primary position of the new node. The second case is that $prev(t[i:k+1])$ was already represented in $PPH(t[1:k])$. In this case,

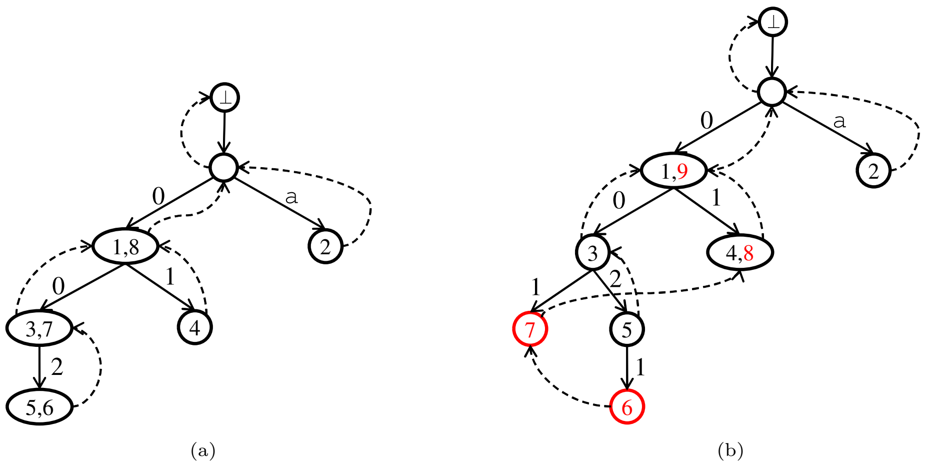


Figure 3 An example of updating a parameterized position heap, from (a) $PPH(x\mathtt{a}xyyxyx)$ to (b) $PPH(x\mathtt{a}xyyxyxx)$. The updated positions are colored red. The secondary positions 6 and 7 in $PPH(x\mathtt{a}xyyxyx)$ are become primary positions in $PPH(x\mathtt{a}xyyxyxx)$, while the secondary position 8 in $PPH(x\mathtt{a}xyyxyx)$ is become a secondary position of another node in $PPH(x\mathtt{a}xyyxyxx)$. The active position is updated from 6 to 8.

the secondary position i that is stored in $prev(t[i:k])$ currently should be moved to the child node $prev(t[i:k+1])$, and position i becomes the secondary position of this node.

From Lemma 6, if the node $prev(t[i:k])$ has an edge to the node $prev(t[i:k+1])$, $prev(t[i+1:k])$ also has an edge to $prev(t[i+1:k+1])$. Therefore, there exists r, with $1 \leq s \leq r \leq k$, that splits the interval $[s:k]$ into two subintervals $[s:r-1]$ and $[r:k]$, such that the node $prev(t[i:k])$ does not have an edge to $prev(t[i:k+1])$ for $s \leq i \leq r-1$, and does have such an edge for $r \leq i \leq k$.

The above analysis leads to the following lemma that specifies the modifications from $PPH(t[1:k])$ to $PPH(t[1:k+1])$.

▶ **Lemma 8.** *Given $t \in (\Sigma \cup \Pi)^n$, consider $PPH(t[1:k])$ for $k < n$. Let s be the active position, stored in the node $prev(t[s:k])$. Let $r \geq s$ be the smallest position such that node $prev(t[r:k])$ has an outgoing edge labeled with $prev(t[r:k+1])[k-r+2]$. $PPH(t[1:k+1])$ can be obtained by modifying $PPH(t[1:k])$ in the following way:*

1. *For each node $prev(t[i:k])$, $s \leq i < r$, create a new child $prev(t[i:k+1])$ linked by an edge labeled $prev(t[i:k+1])[k-i+2]$. Delete the secondary position i from the node $prev(t[i:k])$ and assign it as the primary position of the new node $prev(t[i:k+1])$,*
2. *For each node $prev(t[i:k])$, $r \leq i \leq k$, move the secondary position i from the node $prev(t[i:k])$ to the node $prev(t[i:k+1])$.*

Moreover, r will be the active position in $PPH(t[1:k+1])$.

Proof. Consider the first case that i be a secondary position in $PPH(t[1:k])$ and $s \leq i < r$. From the definition of r, there is no node $prev(t[i:k+1])$ in $PPH(t[i:k])$. Therefore, i will be a primary position of the node $prev(t[i:k+1])$ in $PPH(t[1:k+1])$. We can update the position heap from $PPH(t[1:k])$ to $PPH(t[1:k+1])$ by delete i from secondary position of the node $prev(t[i:k])$ and create a new node $prev(t[i:k+1])$ and assign i to its primary position for the case $s \leq i < r$.

Next case, i be a secondary position in $PPH(t[1:k])$ and $r \leq i \leq k$. In this case, there is a node $prev(t[i:k+1])$ in $PPH(t[i:k])$ and the node $prev(t[i:k+1])$ is also represented in $PPH(t[i:k+1])$. Therefore, i will be a secondary position of the node $prev(t[i:k+1])$ in

$PPH(t[1:k+1])$. We can update the position heap from $PPH(t[1:k])$ to $PPH(t[1:k+1])$ by delete i from secondary position of the node $prev(t[i:k])$ and assign i as secondary position of the node $prev(t[i:k+1])$ for the case $r \leq i \leq k$.

Since position i for $1 \leq i < r$ be a primary position in $PPH(t[1:k+1])$ and position i for $r \leq i \leq k+1$ be a secondary position in $PPH(t[1:k+1])$, r will be the active position in $PPH(t[1:k+1])$. ◀

Figure 3 show an example of updating a parameterized position heap. The modifications specified by Lemma 8 need to be applied to all secondary positions. In order to perform these modifications efficiently, we use parameterized suffix pointers.

▶ **Definition 9** (Parameterized Suffix Pointer)**.** For each node $prev(t[i:j])$ of $PPH(t)$, the *parameterized suffix pointer* of $prev(t[i:j])$ is defined by $psp(prev(t[i:j])) = prev(t[i+1:j])$.

By Lemma 6, whenever the node $prev(t[i:j])$ exists, the node $prev(t[i+1:j])$ exists too. This means that $psp(prev(t[i:j]))$ always exists. During the construction of the parameterized position heap, let $\perp$ be the auxiliary node that works as the parent of *root* and is connected to *root* with an edge labeled with any symbol $c \in \Sigma \cup 0$. We define $psp(root) = \perp$.

When s is the active position in $PPH(t[1:k])$, we call $prev(t[s:k])$ the *active node*. If no node holds a secondary position, *root* becomes the active node and the active position is set to $k+1$. The nodes for the secondary positions s, $s+1$, …, k can be visited by traversing with the suffix pointers from the active node. Thus, the algorithm only has to memorize the active position and the active node in order to visit any other secondary positions.

Updating $PPH(t[1:k])$ to $PPH(t[1:k+1])$ specified by Lemma 8 is processed as the following procedures. The algorithm traverses with the suffix pointers from the active node till the node that has the outgoing edge labeled with $prev(t[i:k+1])[k-i+2]$ is found, which is $i = r$. For each traversed node, a new node is created and linked by an edge labeled with $prev(t[i:k+1])[k-i+2]$ to each node. A suffix pointer to this new node is set from the previously created node. When the node that has the outgoing edge labeled with $prev(t[i:k+1])[k-i+2]$ is traversed, the algorithm moves to the node that is led to by this edge, and a suffix pointer to this node is set from the last created node, then the algorithm assigns this node to be the active node.

A pseudocode of our proposed construction algorithm is given as Algorithm 1. $prim(v)$ and $sec(v)$ denotes primary and secondary positions of v, respectively. From the property of prev-encoding, $prev(t[i+1:k+1])[k-i+1] = prev(t[i:k+1])[k-i+2]$ if $prev(t[i:k+1])[k-i+2] \in \Sigma$ or $prev(t[i:k+1])[k-i+2] \leq k-i$ and $prev(t[i+1:k])[k-i+1] = 0$ otherwise. Therefore, we use a function $normalize(c, j)$ that returns c if $c \in \Sigma$ or $c \leq j$ and returns 0 otherwise.

The construction algorithm consists of n iterations. In the i-th iteration, the algorithm read $t[i]$ and make $PPH(t[1:i])$. In the i-th iteration, the traversal of the suffix pointers as explained above is done. Since the depth of the current node decreases by traversing a suffix pointer, the number of the nodes that can be visited by traversal is $O(n)$. For each traversed node, all the operations such as creating a node, an edge and updating position can be done in $O(\log(|\Sigma| + |\Pi|))$. Therefore, the total time for the traversals is $O(n \log(|\Sigma| + |\Pi|))$.

From the above discussion, the following theorem is obtained.

▶ **Theorem 10.** *Given $t \in (\Sigma \cup \Pi)^n$, Algorithm 1 constructs $PPH(t)$ in $O(n \log(|\Sigma| + |\Pi|))$ time and space.*

Algorithm 1: Parameterized position heap online construction algorithm

```
Input: A p-string $t \in (\Sigma \cup \Pi)^n$
Output: A parameterized position heap $PPH(t)$
create $root$ and $\perp$ nodes;
$psp(root) = \perp$;
$child(\perp, c) = root$ for $c \in \Sigma \cup \{0\}$;
currentNode = $root$;
$s = 1$;
for $i = 1$ to $n$ do
    $c = normalize(prev(t)[i], depth($currentNode$))$;
    lastCreateNode = undefined;
    while $child($currentNode$, c) = null$ do
        create $newnode$;
        $prim(newnode) = s$;
        $child($currentNode$, c) = newnode$;
        if lastCreateNode $\neq$ undefined then $psp($lastCreateNode$) = newnode$;
        lastCreateNode = $newnode$;
        currentNode = $psp($currentNode$)$;
        $c = normalize(prev(t)[i], depth($currentNode$))$;
        $s = s + 1$;
    currentNode = $child($currentNode$, c)$;
    if lastCreateNode $\neq$ undefined then $psp($lastCreateNode$)$ = currentNode;
while $s \leq n$ do
    $sec($currentNode$) = s$;
    currentNode = $psp($currentNode$)$;
    $s = s + 1$;
```

4.3 Augmented Parameterized Position Heaps

We will describe *augmented parameterized position heaps*, the parameterized position heaps with an additional data structure called the *parameterized maximal-reach pointers* similar to the maximal-reach pointers for the position heap [8]. The augmented parameterized position heap gives an efficient algorithm for parameterized pattern matching.

▶ **Definition 11** (Parameterized Maximal-Reach Pointer)**.** For a position i on t, a *parameterized maximal-reach pointer of* $pmrp(i)$ is a pointer from node i to the deepest node whose path label is a prefix of $prev(t[i:])$.

Obviously, if i is a secondary position, then $pmrp(i)$ is node i itself. We assume that the parameterized maximal-reach pointer for a double node applies to the primary position of this node. Figure 2 (b) shows an example of an augmented parameterized position heap. Given a prev-encoded p-string $prev(w)$ represented in an augmented parameterized position heap $APPH(t)$ and a position $1 \leq i \leq n$, we can determine whether $prev(w)$ is a prefix of $prev(t[i:])$ or not in $O(1)$ time by checking whether $pmrp(i)$ is a descendant of $prev(w)$ or not. It can be done in $O(1)$ time by appropriately preprocessing $APPH(t)$ [5].

Parameterized maximal-reach pointers can be computed by using parameterized suffix pointers, similar to [13]. Algorithm 2 shows an algorithm to compute parameterized maximal-reach pointers. $pmrp(i)$ is computed iteratively for $i = 1, 2, \cdots, n$. Assume that we have computed $pmrp(i)$ for some i. Let $pmrp(i) = prev(t[i:l])$. Obviously, $prev(t[i+1:l])$ is a

Algorithm 2: Augmented parameterized position heap construction algorithm

```
Input: A p-string $t \in (\Sigma \cup \Pi)^n$ and $PPH(t)$
Output: An augmented parameterized position heap $APPH(t)$
1  let $t[n+1] = \$$ where $\$$ is a symbol that does not appear in $t$ elsewhere;
2  currentNode = $root$;
3  $l = 1$;
4  for $i = 1$ to $n$ do
5      $c = normalize(prev(t)[l], l - i)$;
6      while $child(\text{currentNode}, c) \neq null$ do
7          currentNode = $child(\text{currentNode}, c)$;
8          $l = l + 1$;
9          $c = normalize(prev(t)[l], l - i)$;
10     $pmrp(i)$ = currentNode;
11     currentNode = $psp(\text{currentNode})$;
```

prefix of the string represented by $pmrp(i+1)$. Thus, in order to compute $pmrp(i+1)$, we should extend the prefix $prev(t[i+1:l]) = psp(prev(t[i:l]))$ in $PPH(t)$ until we found l' such that node $prev(t[i+1:l'])$ does not have outgoing edge labeled with $prev(t[i+1:])[l'-i+1]$ and set $pmrp(i+1) = prev(t[i+1:l'])$. In this time, we need re-compute $prev(t[i+1:])$ by replacing $prev(t[i+1:])[j]$ with 0 if we found that $prev(t[i+1:])[j] \geq j$. The total number of extending $prev(t[i+1:l])$ in the algorithm is at most n because both i and l always increase in each iteration. In each iteration, operations such as traversing a child node can be done in $O(\log(|\Sigma| + |\Pi|))$. Therefore, we can get the following theorem.

▶ **Theorem 12.** *Parameterized maximal-reach pointers for $PPH(t)$ can be computed in $O(n \log(|\Sigma| + |\Pi|))$ time.*

4.4 Parameterized Pattern Matching with Augmented Parameterized Position Heaps

Ehrenfeucht *et al.* [8] and Kucherov [13] split a pattern p into segments $q_1, q_2, \cdots, q_k$, then compute occurrences of $q_1 q_2 \cdots q_j$ iteratively for $j = 1, \cdots, k$. The correctness depends on a simple fact that for strings $x = t[i : i + |x| - 1]$ and $y = t[i + |x| : i + |x| + |y| - 1]$ implies $xy = t[i : i + |xy| - 1]$. However, when x, y, and t are p-strings, $prev(x) = prev(t[i : i + |x| - 1])$ and $prev(y) = prev(t[i + |x| : i + |x| + |y| - 1])$ does not necessarily implies $prev(xy) = prev(t[i : i + |xy| - 1])$. Therefore, we need to modify the matching algorithm for parameterized strings.

Let x, y and w be p-strings such that $|w| = |xy|$, $prev(x) = prev(w[: |x|])$ and $prev(y) = prev(w[|x|+1 :])$. Let us consider the case that $prev(xy) \neq prev(w)$. From $prev(x) = prev(w[: |x|])$ and $prev(y) = prev(w[|x|+1 :])$, x and y have the same structure of $w[: |x|]$ and $w[|x|+1 :]$, respectively. However, the parameter symbols those are prev-encoded into 0 in $prev(y)$ and $prev(w[|x|+1 :])$, might be encoded differently in $prev(xy)$ and $prev(w)$, respectively. Therefore, we need to check whether $prev(xy)[|x| + i] = prev(w)[|x| + i]$ if $prev(y)[i] = 0$. Given $prev(xy)$ and the set of positions of 0 in $prev(y)$, $\mathbf{Z} = \{i \mid 1 \leq i \leq |y| \text{ such that } prev(y)[i] = 0\}$. We need to verify whether $prev(xy)[|x| + i] = prev(w)[|x| + i]$ or not for $i \in \mathbf{Z}$. Since the size of $\mathbf{Z}$ is at most $|\Pi|$, this computation can be done in $O(|\Pi|)$ time.

Algorithm 3: Parameterized pattern matching algorithm with APPH

```
Input: $t \in (\Sigma \cup \Pi)^n$ , $p \in (\Sigma \cup \Pi)^m$, and $APPH(t)$
Output: The list $ans$ of position $i$ such that $prev(p) = prev(t[i : i + m - 1])$
1  let $w$ be the longest prefix of $prev(p)$ represented in $APPH(t)$ and $u$ be the node
     represents $w$;
2  if $|w| = m$ then
3      $v = root$;
4      for $i = 1$ to $m$ do
5          $v = child(v, prev(p)[i])$;
6          if $pmrp(v) \in Des_{APPH(t)}(u)$ then add $prim(v)$ to $ans$;
7      add all primary and secondary position of descendants of $u$ to $ans$;
8  else
9      $v = root$;
10     $i = 1, j = 1$;
11     while $i \leq |w|$ do
12         $v = child(v, prev(p)[i])$;
13         $i = i + 1$;
14         if $pmrp(v) = u$ then add $prim(v)$ to $ans$;
15     while $i \neq m$ do
16         $j = i, v = root$;
17         $\mathbf{Z}$ = empty list;
18         while $i \neq m$ do
19             $c = normalize(prev(p)[i], i - j)$;
20             if $child(v, c) = null$ then break;
21             if $c = 0$ then add $i$ to $\mathbf{Z}$;
22             $v = child(v, c)$;
23             $i = i + 1$;
24         if $v = root$ then return empty list;
25         foreach $i' \in ans$ do
26             if $i = m$ then
27                 if $pmrp(i' + j - 1) \notin Des_{APPH(t)}(v)$ then remove $i'$ from $ans$;
28             else
29                 if $pmrp(i' + j - 1) \neq v$ then remove $i'$ from $ans$;
30             for $k = 1$ to $|\mathbf{Z}|$ do
31                 if $normalize(prev(t)[i' + \mathbf{Z}[k] - 1], \mathbf{Z}[k] - 1) \neq prev(p)[\mathbf{Z}[k]]$ then
32                     remove $i'$ from $ans$;
33 return $ans$;
```

A pseudocode of proposed matching algorithm for the parameterized pattern matching problem is shown in Algorithm 3. $Des_{APPH(t)}(u)$ denotes the set of all descendants of node u in $APPH(t)$ including node u itself. The occurrences of p in t have the following properties on $APPH(t)$.

▶ **Lemma 13.** *If $prev(p)$ is represented in $APPH(t)$ as a node u then p occurs at position i iff $pmrp(i)$ is u or its descendant.*

Proof. Let u be a node represents $prev(p)$. Assume p occurs at position i in t and represented in $APPH(t)$ as $prev(t[i : k])$. Since either $prev(t[i : k])$ is a prefix of $prev(p)$ or $prev(p)$ is a prefix of $prev(t[i : k])$, then i is either an ancestor or descendant of u. For both cases $pmrp(i)$ is a descendant of u, because p occurs at position i.

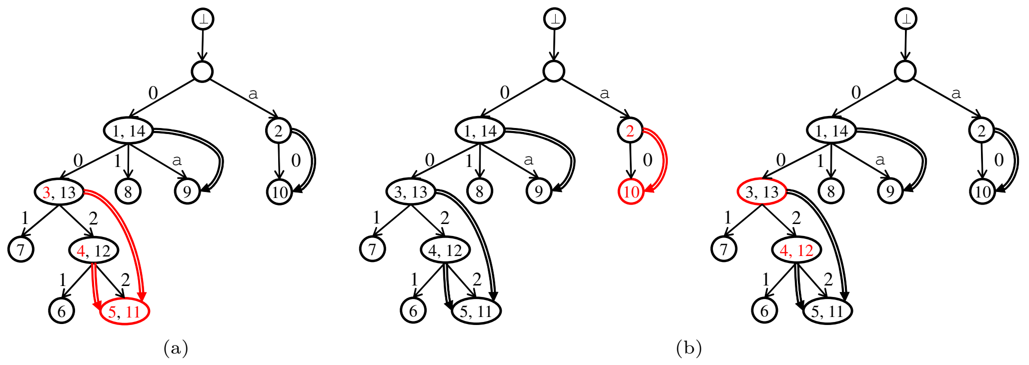


Figure 4 Examples of finding occurrence positions of a pattern using an augmented parameterized position heap $PPH(x\mathtt{a}xyxyxyy\mathtt{a}xyxy)$. (a) Finding $xyxy$ ($prev(xyxy) = 0022$). (b) Finding $\mathtt{a}xyx$ ($prev(\mathtt{a}xyx) = \mathtt{a}002$).

Next let i be a node such that $pmrp(i)$ is a descendant of u and represents $prev(t[i : k])$. In this case, $prev(p)$ is a prefix of $prev(t[i : k])$. Therefore p occurs at i. ◀

▶ **Lemma 14.** *Assume $prev(p)$ is not represented in $APPH(t)$. We can split p into $q_1, q_2, \cdots, q_k$ such that q_j is the longest prefix of $prev(p[|q_1 \cdots q_{j-1}| + 1 :])$ that is represented in $APPH(t)$. If p occurs at position i in t, then $pmrp(i + |q_1 \cdots q_{j-1}|)$ is the node $prev(q_j)$ for $1 \le j < k$ and $pmrp(i + |q_1 \cdots q_{k-1}|)$ is the node $prev(q_k)$ or its descendant.*

Proof. Let $p = q_1 q_2 \cdots q_k$ occurs at position i in t. Since $prev(q_1)$ is a prefix of $prev(p)$, then $pmrp(i)$ is the node that represents $prev(q_1)$ or its descendant. However, if $pmrp(i)$ is a descendant of node $prev(q_1)$, then we can extend q_1 which contradicts with the definition of q_1. Therefore, $pmrp(i)$ is the node represents $prev(q_1)$.

Similarly for $1 < j < k$, $prev(q_j)$ is a prefix of $prev(p[|q_1 \cdots q_{j-1}| + 1 :])$ and occurs at position $i + |q_1 \cdots q_{j-1}|$ in t. Therefore, $pmrp(i + |q_1 \cdots q_{j-1}|)$ is the node represents $prev(q_j)$. Last, since q_k is a suffix of p, then $pmrp(i + |q_1 \cdots q_{j-1}|)$ can be the node $prev(q_k)$ or its descendant. ◀

Algorithm 3 utilizes Lemmas 13 and 14 to find occurrences of p in t by using $APPH(t)$. First, if $prev(p)$ is represented in $APPH(t)$ then the algorithm will output all position i such that $pmrp(i)$ is a node $prev(p)$ or its descendant. Otherwise, it will split p into $q_1 q_2 \cdots q_k$ and find their occurrences as described in Lemma 14. The algorithm also checks whether $prev(q_1 \cdots q_j)$ occurs in t or not in each iteration as described the above.

Examples of parameterized pattern matching by using an augmented position heap are given in Figure 4. Let $t = x\mathtt{a}xyxyxyy\mathtt{a}xyxy$ be the text. In Figure 4 (a) we want to find the occurrence positions of a pattern $p_1 = xyxy$ in t. In this case, since $prev(p_1) = 0022$ is represented in $PPH(t)$, The algorithm outputs all positions i such that $pmrp(i)$ is the node 0022 or its descendants, those are 3, 4, 5, and 11. On the other hand, Figure 4 (b) shows how to find the occurrence positions of a pattern $p_2 = \mathtt{a}xyx$ in t. In this case, $prev(p_2) = \mathtt{a}002$ is not represented in $PPH(t)$. Therefore, The algorithm finds the longest prefix of $prev(p_2)$ that is represented in $PPH(i)$, which is $prev(p_2)[1 : 2] = \mathtt{a}0$. We can see that $prmp(2) = pmrp(10) = \mathtt{a}0$, then we save positions 2 and 10 as candidates to ans. Next, The algorithm finds the node that represents the longest prefix of $prev(p_2[3 :]) = 00$ which is $prev(p_2[3 :]) = 00$ itself. Since both of $pmrp(2 + |p_2[1 : 2]|) = pmrp(4)$ and $pmrp(10 + |p_2[1 : 2])| = pmrp(12)$ is descendants of the node 00, $prev(t[2 : 5][3]) = prev(t[10 : 13][3]) = prev(p_2)[[3]] = 0$, and $prev(t[2 : 5][4]) = prev(t[10 : 13][4]) = prev(p_2)[4] = 2$, then the algorithm outputs 2 and 10.

The time complexity of the matching algorithm is as follow.

▶ **Theorem 15.** *Algorithm 3 runs in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time.*

Proof. It is easily seen that we can compute line 4 to 7 in $O(m \log(|\Sigma| + |\Pi|) + occ)$ time. Assume that p can be decomposed into q_1, q_2, $\cdots$, q_k such that q_1 is the longest prefix of p and q_i is the longest prefix of $prev(p[|q_1 \cdots q_{j-1}| + 1 :])$ represented in $APPH(t)$. The loop for line 15 consists of $k-1$ iterations. In the loop line 18 in j-th iteration, q_{j+1} is extended up to reach $|q_{j+1}|$ length. This can be computed in $O(|q_{j+1}| \log(|\Sigma| + |\Pi|))$ time. After $k-1$ iterations, the total number of extending of q_{j+1} does not exceed m, because $\Sigma_{j=2}^{k} |q_j| < m$. In the loop for line 25, the algorithm verifies elements of ans. In j-th iteration, the size of ans is at most $|q_j|$. Thus, after $k-1$ iterations, the total number of elements verified in line 25 does not exceed m by the same reason for that of line 18. In each verification in line 25, the number of checks for line 27 and 29 is at most $|q_j|$. Therefore, it can be computed from line 25 to 32 in $O(m|\Pi|)$ time. ◀

5 Conclusion and Future Work

For the parameterized pattern matching problem, we proposed an indexing structure called a parameterized position heap. Given a p-string t of length n over a constant size alphabet, the parameterized position heap for t can be constructed in $O(n \log(|\Sigma| + |\Pi|))$ time by our construction algorithm. We also proposed an algorithm for the parameterized pattern matching problem. It can be computed in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time using parameterized position heaps with parameterized maximal-reach pointers. Gagie *et al.* [10] showed an interesting relationship between position heap and suffix array of a string. We will examine this relation for parameterized position heap and parameterized suffix array [7, 12] as a future work.

References

1 Brenda S. Baker. A program for identifying duplicated code. In H. Joseph Newton, editor, *Proceedings of the 24th Symposium on the Interface of Computing Science and Statistics: Graphics and Visualization*, volume 24, pages 49–57. Interface Foundation of North America, 1992. URL: `http://www.dtic.mil/dtic/tr/fulltext/u2/a266571.pdf`.

2 Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC 1993)*, pages 71–80. ACM, 1993. `doi:10.1145/167088.167115`.

3 Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996. `doi:10.1006/jcss.1996.0003`.

4 Edward G. Coffman Jr. and James Eve. File structures using hashing functions. *Commun. ACM*, 13(7):427–432, 1970. `doi:10.1145/362686.362693`.

5 Thomas H. Cormen, Charies E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 2009. URL: `https://mitpress.mit.edu/books/introduction-algorithms`.

6 Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology: Text Algorithms*. World Scientific, 2002. `doi:10.1142/9789812778222`.

7 Satoshi Deguchi, Fumihito Higashijima, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Parameterized suffix arrays for binary strings. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2008*, pages 84–94, Czech Technical

University in Prague, Czech Republic, 2008. URL: http://www.stringology.org/event/2008/p08.html.

8 Andrzej Ehrenfeucht, Ross M. McConnell, Nissa Osheim, and Sung-Whan Woo. Position heaps: A simple and dynamic text indexing data structure. *J. Discrete Algorithms*, 9(1):100–121, 2011. doi:10.1016/j.jda.2010.12.001.

9 Kimmo Fredriksson and Maxim Mozgovoy. Efficient parameterized string matching. *Inf. Process. Lett.*, 100(3):91–96, 2006. doi:10.1016/j.ipl.2006.06.009.

10 Travis Gagie, Wing-Kai Hon, and Tsung-Han Ku. New algorithms for position heaps. In Johannes Fischer and Peter Sanders, editors, *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *LNCS*, pages 95–106, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-38905-4_11.

11 Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

12 Tomohiro I, Satoshi Deguchi, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Lightweight parameterized suffix array construction. In Jirí Fiala, Jan Kratochvíl, and Mirka Miller, editors, *Proceedings of the 20th International Workshop on Combinatorial Algorithms (IWOCA 2009)*, volume 5874 of *LNCS*, pages 312–323, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-10217-2_31.

13 Gregory Kucherov. On-line construction of position heaps. *J. Discrete Algorithms*, 20:3–11, 2013. StringMasters 2011 Special Issue. doi:10.1016/j.jda.2012.08.002.

14 Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. The position heap of a trie. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE 2012)*, volume 7608 of *LNCS*, pages 360–371, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-34109-0_38.

15 Tetsuo Shibuya. Generalization of a suffix tree for RNA structural pattern matching. *Algorithmica*, 39(1):1–19, 2004. doi:10.1007/s00453-003-1067-9.

16 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

On-Line Pattern Matching on Similar Texts*

Roberto Grossi[1], Costas S. Iliopoulos[2], Chang Liu[3], Nadia Pisanti[4], Solon P. Pissis[5], Ahmad Retha[6], Giovanna Rosone[7], Fatima Vayani[8], and Luca Versari[9]

1 **Department of Computer Science, University of Pisa, Italy; and ERABLE Team, INRIA, France**
grossi@di.unipi.it

2 **Department of Informatics, King's College London, London, UK**
c.iliopoulos@kcl.ac.uk

3 **Department of Informatics, King's College London, London, UK**
chang.2.liu@kcl.ac.uk

4 **Department of Computer Science, University of Pisa, Italy; and ERABLE Team, INRIA, France**
pisanti@di.unipi.it

5 **Department of Informatics, King's College London, London, UK**
solon.pissis@kcl.ac.uk

6 **Department of Informatics, King's College London, London, UK**
ahmad.retha@kcl.ac.uk

7 **Department of Computer Science, University of Pisa, Pisa, Italy; and Department of Mathematical and Computer Science, University of Palermo, Palermo, Italy**
giovanna.rosone@unipi.it

8 **Department of Informatics, King's College London, London, UK**
fatima.vayani@kcl.ac.uk

9 **Scuola Normale Superiore, Pisa, Italy**
luca.versari@sns.it

Abstract

Pattern matching on a set of similar texts has received much attention, especially recently, mainly due to its application in cataloguing human genetic variation. In particular, many different algorithms have been proposed for the *off-line* version of this problem; that is, constructing a compressed index for a set of similar texts in order to answer pattern matching queries efficiently. However, the *on-line*, more fundamental, version of this problem is a rather undeveloped topic. Solutions to the on-line version can be beneficial for a number of reasons; for instance, efficient on-line solutions can be used in combination with partial indexes as practical trade-offs. We make here an attempt to close this gap via proposing two efficient algorithms for this problem. Notably, one of the algorithms requires time linear in the size of the texts' representation, for short patterns. Furthermore, experimental results confirm our theoretical findings in practical terms.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

* Roberto Grossi, Nadia Pisanti, and Giovanna Rosone are partially supported by the project UniPi PRA_2017_44 ("Advanced computational methodologies for the analysis of biomedical data"). Costas S. Iliopoulos is partially supported by the Onassis Foundation. Ahmad Retha is supported by the Graduate Teaching Scholarship scheme of the Department of Informatics at King's College London. Giovanna Rosone is partially supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L. Fatima Vayani is supported by an EPSRC Grant (Doctoral Training Grant #EP/M506357/1).

Keywords and phrases string algorithms, pattern matching, degenerate strings, elastic-degenerate strings, on-line algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.9

1 Introduction

It is possible to represent closely related sequences that have been aligned using a multiple sequence alignment (MSA) algorithm into one compacted form, that is able to represent the non-polymorphic sites (columns) of the MSA, as well as the polymorphic ones [10]. This representation compresses maximal sequences of non-polymorphic sites, while the polymorphic ones, containing substitutions, insertions, and deletions of letters, are represented as a set containing all possible variants observed at that location. Consider, for instance, the following:

```
ATGCAACGGGTA--TTTTA
ATGCAACGGGTATATTTTA
ATGCACCTGG----TTTTA
```

These sequences can be compacted into a single string $\tilde{T}$ containing some deterministic and some *non-deterministic* segments. Note that a non-deterministic segment is a finite set of deterministic strings and may contain an empty string ε corresponding to a deletion. The total number of segments is the *length* of $\tilde{T}$ and the total number of letters is the *size* of $\tilde{T}$.

$$\tilde{T} = \left\{ \texttt{ATGCA} \right\} \cdot \left\{ \begin{array}{c} \texttt{A} \\ \texttt{C} \end{array} \right\} \cdot \left\{ \texttt{C} \right\} \cdot \left\{ \begin{array}{c} \texttt{G} \\ \texttt{T} \end{array} \right\} \cdot \left\{ \texttt{GG} \right\} \cdot \left\{ \begin{array}{c} \texttt{TA} \\ \texttt{TATA} \\ \varepsilon \end{array} \right\} \cdot \left\{ \texttt{TTTTA} \right\}$$

This representation has been defined in [11] as an *elastic-degenerate* text. The natural problem that arises is finding all matches of a deterministic pattern P in text $\tilde{T}$. We call this the Elastic-Degenerate String Matching (*EDSM*) problem. The simplest version of this problem assumes that a degenerate segment can contain only single letters [9].

An elastic-degenerate text can represent, for example, a set of closely-related DNA sequences. For instance, a *pan-genome* [18, 24, 12, 21] is a reference sequence which is not just a single genome, but the result of an MSA of several of them that share large consensus regions and also exhibit differences at some positions. Recently, various data structures to store pan-genomes have been suggested [8, 4]. In particular, due to the application of cataloguing human genetic variation [23], there has been ample work in the literature on the *off-line* (indexing) version of the pattern matching problem [10, 14, 22, 15, 16]. In literature, there are also algorithms and applications for the problem of inferring motifs from degenerate input texts [20, 19]. However, to the best of our knowledge, the *on-line*, more fundamental, version of the *EDSM* problem has not been studied as much as indexing approaches. Solutions to the on-line version can be beneficial for a number of reasons: (a) efficient on-line solutions can be used in combination with partial indexes as practical trade-offs; (b) efficient on-line solutions for exact pattern matching can be applied for fast average-case approximate pattern matching, similar to standard strings [3]; (c) on-line solutions can be useful when one wants to search for a set of patterns in elastic-degenerate texts, similar to standard strings [1, 2].

Our Contributions. Let us denote by m the length of pattern P, by n the length of $\tilde{T}$, and by $N > m$ the size of $\tilde{T}$ (see Section 2 for definitions). In [11], an algorithm for solving the *EDSM* problem in time $\mathcal{O}(\alpha\gamma mn + N)$ and space $\mathcal{O}(N)$ was presented; where α and γ are parameters, respectively representing the maximum number of strings in any degenerate segment of the text and the maximum number of degenerate segments spanned by any occurrence of the pattern in the text. In this paper, we improve the state-of-the-art; we present two new algorithms to solve the same problem in an on-line manner. The first one requires time $\mathcal{O}(nm^2 + N)$ after a preprocessing stage with time and space $\mathcal{O}(m)$; the second algorithm requires time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ after a preprocessing stage with time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$, where w is the size of the computer word in the RAM model. Thus, the second algorithm requires time linear in the size of the texts' representation, for short patterns. Finally, we present experiments confirming our theoretical findings in practical terms.

2 Definitions

We begin with a few definitions, generally following [5]. An *alphabet* Σ is a non-empty finite set of letters of size $|\Sigma|$. A *(deterministic) string* on a given alphabet Σ is a finite sequence of letters of Σ. For this work, we assume that the alphabet is fixed, i.e. $|\Sigma| = \mathcal{O}(1)$. The *length* of a string x is denoted by $|x|$. For two positions i and j on x, we denote by $x[i\,..\,j] = x[i]\,..\,x[j]$ the *factor* (sometimes called *substring*) of x that starts at position i and ends at position j (it is empty if $j < i$), and by ε we denote the *empty string*. The set of all strings on an alphabet Σ (including the empty string ε) is denoted by Σ^*. For any string $y = uxv$, where u and v are strings, if $u = \varepsilon$ then x is a *prefix* of y. Similarly, if $v = \varepsilon$ then x is a *suffix* of y. We say that x is a *proper factor* (resp. prefix/suffix) of y if x is a factor (resp. prefix/suffix) of y distinct from y. By $\mathcal{B}_{u,v}$ we denote the set containing all indices i, such that the prefix $u[0\,..\,i]$ of string u is also a suffix of string v.

▶ **Example 1.** Suppose we have two strings $u =$ `ATATG` and $v =$ `CATAT`. Then $\mathcal{B}_{u,v} = \{1, 3\}$ because of prefix/suffix `AT` and prefix/suffix `ATAT`, respectively.

An *elastic-degenerate string* (ED string) $\tilde{X} = \tilde{X}[0]\tilde{X}[1] \ldots \tilde{X}[n-1]$, of *length* n, on an alphabet Σ, is a finite sequence of n *degenerate letters*. Every *degenerate letter* $\tilde{X}[i]$, for all $0 \le i < n$, is a non-empty set of strings $\tilde{X}[i][j]$, with $0 \le j < |\tilde{X}[i]|$, where each $\tilde{X}[i][j]$ is a deterministic string on Σ. The total *size* of $\tilde{X}$ is defined as

$$N = \sum_{i=0}^{n-1} \sum_{j=0}^{|\tilde{X}[i]|-1} |\tilde{X}[i][j]|.$$

Only for the purpose of computing N, $|\varepsilon| = 1$. We remark that, for an ED string $\tilde{X}$, the size and the length are two distinct concepts (see Example 2).

We say that a string Y *matches* an ED string $\tilde{X} = \tilde{X}[0] \ldots \tilde{X}[m'-1]$ of length $m' > 1$, denoted by $Y \approx \tilde{X}$, if and only if string Y can be decomposed into $y_0 \ldots y_{m'-1}$, $y_i \in \Sigma^*$, such that:

1. there exists a string $s \in \tilde{X}[0]$ such that a suffix of s is $y_0 \neq \varepsilon$;
2. if $m' > 2$, there exists $s \in \tilde{X}[i]$, for all $1 \le i \le m'-2$, such that $s = y_i$;
3. there exists a string $s \in \tilde{X}[m'-1]$ such that a prefix of s is $y_{m'-1} \neq \varepsilon$.

Note that, in the above definition, we require that both y_0 and $y_{m'-1}$ are non-empty to avoid spurious matches at the beginning or end of an occurrence. A string Y is said to have an *occurrence* ending at position j in an ED string $\tilde{T}$ if there exist $i < j$ such that $\tilde{T}[i] \ldots \tilde{T}[j] \approx Y$, or, if there exists $s \in \tilde{T}[j]$ such that Y occurs in s.

▶ **Example 2** (Running example). Suppose we have a pattern $P =$ `ACACA`, of length $m = 5$, and an ED string $\tilde{T}$, of length $n = 6$ and size $N = 18$; the first occurrence of P starts at position 1 and ends at position 2 of $\tilde{T}$; and the second one starts at position 2 and ends at position 4.

$$\tilde{T} = \left\{ \texttt{C} \right\} \cdot \left\{ \begin{array}{c} \texttt{A} \\ \texttt{C} \end{array} \right\} \cdot \left\{ \begin{array}{c} \texttt{AC} \\ \texttt{ACC} \\ \texttt{CACA} \end{array} \right\} \cdot \left\{ \begin{array}{c} \texttt{C} \\ \varepsilon \end{array} \right\} \cdot \left\{ \begin{array}{c} \texttt{A} \\ \texttt{AC} \end{array} \right\} \cdot \left\{ \texttt{C} \right\}$$

We are now in a position to formally define the main problem of this paper.

ELASTIC-DEGENERATE STRING MATCHING (*EDSM*)
Input: a string P, of length m, and an ED string $\tilde{T}$, of length n and size $N \geq m$
Output: all positions j in $\tilde{T}$ where at least one occurrence of P ends

3 Algorithmic Tools

The *suffix tree* $\mathcal{ST}_y$ of a string y, of length $n > 0$, is a compact trie representing all suffixes of y. The nodes of the trie which become nodes of the suffix tree are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. We let $\mathcal{P}(v)$ denote the *path-label* of a node v, that is, the concatenation of the edge labels along the path from the root to v. We say that v is path-labelled $\mathcal{P}(v)$. Node v is marked as a *terminal* node if its path-label is a suffix of y, that is, $\mathcal{P}(v) = y[i \mathinner{..} n-1]$ for some $0 \leq i < n$. Note that v is also labelled with index i. Thus, each factor of y is uniquely represented by an explicit or an implicit node of $\mathcal{ST}_y$. More details on suffix trees can be found in [7, 5].

▶ **Fact 3** ([6, 5]). *Given a string y of length n, $\mathcal{ST}_y$ can be constructed in time and space $\mathcal{O}(n)$. Finding all Occ_x occurrences of a string x, of length m, in y can be performed in time $\mathcal{O}(m + Occ_x)$ using $\mathcal{ST}_y$.*

A *border* of a non-empty string x is a proper factor of x that is both a prefix and a suffix of x. We introduce the function $\mathsf{border}(x)$ defined for every non-empty string x as the longest border of x. Let x be a string of length $m \geq 1$. We define the *border table* B: $\{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$ by $\mathsf{B}[k] = |\mathsf{border}(x[0 \mathinner{..} k])|$, for $k = 0, 1, \ldots, m-1$.

▶ **Fact 4** ([13, 5]). *Given a string x of length m, the border table of x can be computed on-line in time $\mathcal{O}(m)$. All borders of x can be specified within the same time complexity using the border table.*

We remark that the border table and the notion of border refer to a proper prefix and a proper suffix of the same string, whereas the indexes in set $\mathcal{B}_{x,y}$ refer to a string which is a prefix of a string (x) and a suffix of another (y), and that is not necessarily proper.

▶ **Lemma 5.** *Given a string x, of length m, and the suffix tree $\mathcal{ST}_y$ of a string y, of length n, $\mathcal{B}_{x,y}$ can be computed in time $\mathcal{O}(m)$.*

Proof. By applying Fact 3, we traverse $\mathcal{ST}_y$ to find the terminal node v corresponding to the longest prefix of x, which is path-labelled $\mathcal{P}(v)$. While traversing $\mathcal{ST}_y$ with x, we add index $n-i-1$ to $\mathcal{B}_{x,y}$ if we encounter a terminal node u, such that $\mathcal{P}(u) = y[i \mathinner{..} n-1]$. The longest such prefix of x is of length at most m. No longer prefix of x can be a suffix of y as it does not occur in y. ◀

4 Algorithm

An ED string can always represent an *exponential* number of strings (per ending position), where the exact number is the product of the number of deterministic strings at previous positions. Searching a pattern in all these strings separately is thus not acceptable.

Main idea. Our algorithm has a preprocessing phase where we build the suffix tree of the pattern P (Line 2 in pseudocode below). Then, in an on-line manner, we scan $\tilde{T}$ from left to right and, for each $\tilde{T}[i]$, we:

1. memorise the prefixes of the pattern that occur as suffixes of some $s \in \tilde{T}[i]$ (Lines 5 & 12 in pseudocode);
2. check whether at $\tilde{T}[i]$ it is possible to extend a partial occurrence of the pattern which has started earlier in the ED text (Lines $13 - 16$ in pseudocode);
3. in both previous cases we finally check whether a full occurrence of P actually also ends in $\tilde{T}[i]$ (Lines $6 - 8$ & $17 - 22$ in pseudocode).

We perform these steps by computing and storing, for each $0 \leq i < n$, the list $\mathcal{L}_i$ of the rightmost positions of prefixes of P that occur at the end of $\tilde{T}[i]$. Below, we formally present Algorithm EDSM that solves the *EDSM* in an on-line manner. Note that by INSERT$(A, \mathcal{L})$, we denote the operation that inserts the elements of a set A into a linked-list $\mathcal{L}$.

```
1  Algorithm EDSM($P, m, \tilde{T}, n$)
2      Construct $\mathcal{ST}_P$;
3      $\mathcal{L}_0 \leftarrow$ EMPTYLIST();
4      foreach $S \in \tilde{T}[0]$ do
5          Compute $\mathcal{B}_{P,S}$ using the border table; INSERT($\mathcal{B}_{P,S}, \mathcal{L}_0$);
6          if $|S| \geq m$ then
7              Search $P$ in $S$ using KMP and
8              report 0 if $P$ occurs in $S$ and CHECKDUPLICATE(0);
9      foreach $i \in [1, n-1]$ do
10         $\mathcal{L}_i \leftarrow$ EMPTYLIST();
11         foreach $S \in \tilde{T}[i]$ do
12             Compute $\mathcal{B}_{P,S}$ using the border table; INSERT($\mathcal{B}_{P,S}, \mathcal{L}_i$);
13             if $|S| < m$ then
14                 Search $S$ in $P$ using $\mathcal{ST}_P$; denote starting positions by $\mathcal{A}$;
15                 foreach ($p \in \mathcal{L}_{i-1}$, $j \in \mathcal{A}$) such that $p + 1 = j$ do
16                     INSERT($\{p + |S|\}, \mathcal{L}_i$);
17             if $|S| \geq m$ then
18                 Search $P$ in $S$ using KMP and
19                 report $i$ if $P$ occurs in $S$ and CHECKDUPLICATE($i$);
20             Compute $\mathcal{B}_{S,P}$ using $\mathcal{ST}_P$;
21             if there exists ($p \in \mathcal{L}_{i-1}$, $j \in \mathcal{B}_{S,P}$) such that $p + j + 2 = m$
                then
22                 Report $i$ if CHECKDUPLICATE($i$);
```

Example 6 shows Steps (1) and (2) on the running example. The border table shown in Example 6 has to be computed for all text positions, leading to the overall complexity stated in Lemma 7.

▶ **Example 6** (Running example). Let us consider again $P = \texttt{ACACA}$ and $\tilde{T}$ of Example 2. Assume we have already computed $\mathcal{L}_0$ and $\mathcal{L}_1$, and we move to position $i = 2$, where at $\tilde{T}[i]$ we have three strings $\{S_0, S_1, S_2\}$, with $S_0 = \texttt{AC}$, $S_1 = \texttt{ACC}$, and $S_2 = \texttt{CACA}$. We generate the string $X_i = X_2 = P\$_0S_0\$_1S_1\$_2S_2 = \texttt{ACACA}\$_0\texttt{AC}\$_1\texttt{ACC}\$_2\texttt{CACA}$ and build its border table B (Line 12 in pseudocode).

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$X_2[k]$	A	C	A	C	A	$\$_0$	A	C	$\$_1$	A	C	C	$\$_2$	C	A	C	A
$\mathsf{B}[k]$	0	0	1	2	3	0	1	2	0	1	2	0	0	0	1	2	3

In order to compute $\mathcal{B}_{P,S}$ (Line 12), we read $\mathsf{B}[7] = 2$, which gives the length of the longest string that is a prefix of P and a suffix of S_0. To check if there exist borders of length shorter than 2, we read $\mathsf{B}[2-1] = 0$, indicating that no shorter border exists. Therefore, we have $\mathcal{B}_{P,S_0} = \{1\}$. We then read $\mathsf{B}[11] = 0$, telling us that no prefix of P is a suffix of S_1, and hence $\mathcal{B}_{P,S_1} = \emptyset$. We read $\mathsf{B}[16] = 3$, which gives the length of the longest string that is a prefix of P and a suffix of S_2. To check if there exist shorter borders, we read $\mathsf{B}[3-1] = 1$, indicating that a shorter border of length 1 exists. Since $\mathsf{B}[1-1] = 0$, no shorter border exists. Therefore, we have $\mathcal{B}_{P,S_2} = \{0, 2\}$. This gives us a partial $\mathcal{L}_i = \{0, 1, 2\}$ for position $i = 2$ that concludes Step 1 for position $i = 2$ ($\textsc{Insert}(\mathcal{B}_{P,S}, \mathcal{L}_i)$, Line 12). Further on, at Step 2, we will add position 4 to $\mathcal{L}_2$ by extending the occurrence of P that had started at $\tilde{T}[1]$. Putting everything together, we get $\mathcal{L}_2 = \{0, 1, 2, 4\}$ ($\textsc{Insert}(\{p + |S|\}, \mathcal{L}_i)$, Lines $15-16$).

▶ **Lemma 7.** *Given P, of length m, and $\tilde{T}$, of length n and size N, the sets $\mathcal{B}_{P,S}$ with $S \in \tilde{T}[i]$, for all $i \in [0, n-1]$, can be computed in time $\mathcal{O}(N)$.*

Proof. For each position i, we generate a string $X_i = P\$_0S_0\$_1S_1\$_2S_2 \dots \$_{k-1}S_{k-1}$, where $S_j \in \tilde{T}[i]$, $0 \le j < k$, and $\$_j$'s are distinct letters not in Σ. We build the border table B of string X_i. By traversing B from left to right we can compute sets $\mathcal{B}_{P,S_j}$. Specifically, for any string S_j, all borders that are suffixes of S_j and prefixes of P can be computed in time $\mathcal{O}(|S_j|)$, since there exist at most $|S_j|$ such borders. By Fact 4, we can build all border tables, and hence compute all $\mathcal{B}_{P,S_j}$, for all $S_j \in \tilde{T}[i]$, in time $\mathcal{O}(|P| + \sum_{j=0}^{k-1} |S_j|)$. Since the length and the total size of $\tilde{T}$ are n and N, respectively, sets $\mathcal{B}_{P,S_j}$ can be computed in time $\mathcal{O}(nm + N)$. By noting that the border table for P can be computed only once and that the border table computation can be done on-line (Fact 4), the whole computation is bounded by $\mathcal{O}(N)$. ◀

▶ **Lemma 8.** *Given P, $\mathcal{ST}_P$, and $\tilde{T}$ of length n and size N, the sets $\mathcal{B}_{S,P}$, $S \in \tilde{T}[i]$, for all $i \in [1, n-1]$, can be computed in time $\mathcal{O}(N)$.*

Proof. By Lemma 5, for any $S \in \tilde{T}[i]$, $|S| \le |P|$, $\mathcal{B}_{S,P}$ can be computed in time $\mathcal{O}(|S|)$ using $\mathcal{ST}_P$. Since the total size of $\tilde{T}$ is N, sets $\mathcal{B}_{S,P}$ can be computed in time $\mathcal{O}(N)$. ◀

▶ **Lemma 9.** *Lists $\mathcal{L}_i$, for all $i \in [0, n-1]$, in Algorithm* EDSM *can be computed in time $\mathcal{O}(nm^2 + N)$.*

Proof. List $\mathcal{L}_0$ consists of the elements of $\mathcal{B}_{P,S}$ for position 0, which by Lemma 7 can be done within time $\mathcal{O}(N)$. For pattern P of length m, there exist at most $\frac{m(m+1)}{2}$ factors. For the strings $S_j \in \tilde{T}[i]$, $|S_j| \le m$, $0 \le j < k$, we can find at most $\frac{m(m+1)}{2} = \mathcal{O}(m^2)$ occurrences in pattern P. By Fact 3, finding all occurrences can be done in time $\sum_{j=0}^{k-1}(|S_j| + Occ_{S_j})$, and this is bounded by $\mathcal{O}(nm^2 + N)$ for all positions i. This is because, by definition, no $S_j, S_{j'} \in \tilde{T}[i]$ exist such that $S_j = S_{j'}$. Each occurrence can cause only one extension from

$\mathcal{L}_{i-1}$ to $\mathcal{L}_i$. To avoid duplicates in $\mathcal{L}_i$, we need to check if there exist more than one prefix extensions ending at the same position. Each check can be done in constant time using a bit vector of size m, which we set on only once per position i. Therefore, we can extend the prefixes in time $\mathcal{O}(m^2)$ for each position i, and in time $\mathcal{O}(nm^2)$ for the whole text $\tilde{T}$ of length n. By Lemma 7, sets $\mathcal{B}_{P,S}$ corresponding to new prefixes of pattern P which are suffixes of $\{S_0, S_1, \ldots, S_{k-1}\}$ at position $\tilde{T}[i]$ can be found in time $\mathcal{O}(N)$. Merging new prefixes with the prefixes extended from $\mathcal{L}_{i-1}$ can be done in time $\mathcal{O}(m)$, since both are at most m. Therefore, lists $\mathcal{L}_i$, for all $i \in [0, n-1]$, in EDSM can be computed in time $\mathcal{O}(nm^2 + N)$. ◀

Example 10 shows Step (3) on our running example.

▶ **Example 10** (Running example). Let us consider again $P = \texttt{ACACA}$ and $\tilde{T}$ of Example 2. For position $i = 4$, we have $\mathcal{L}_3 = \{1, 3\}$ and we have to compute $\mathcal{L}_4$. For $S_0 = \texttt{A}$, we have $\mathcal{B}_{\texttt{A},\texttt{ACACA}} = \{0\}$ (Line 20), so for $3 \in \mathcal{L}_3$, we have that $3 + 0 + 2 = 5 = m$ (Line 21). Hence, one occurrence of P has been found. Moreover, for $S_1 = \texttt{AC}$, we have $\mathcal{B}_{\texttt{AC},\texttt{ACACA}} = \{0, 1\}$ (Line 20), so for $3 \in \mathcal{L}_3$, we have that $3 + 0 + 2 = 5 = m$ (Line 21). Therefore, another occurrence of P has been found at the same position.

Since Algorithm EDSM reports all positions i in $\tilde{T}$ where at least one occurrence of P ends, and since more than one occurrence may end at the same position (as in Example 10), we need to avoid duplications. To this end, we can use a simple operation to check whether the current position i has already been reported (CheckDuplicate(i), Lines 8, 19, & 22).

▶ **Theorem 11.** *Algorithm EDSM solves the EDSM problem in an on-line manner in time $\mathcal{O}(nm^2 + N)$. Algorithm EDSM requires preprocessing time and space $\mathcal{O}(m)$.*

Proof. The correctness of the algorithm follows from the correctness of the KMP algorithm [13] if $|S| > m$, $S \in \tilde{T}[i]$, and from the combination of Lemmas 8 and 9, if $|S| \leq m$. By definition, we cannot have any other type of (ending) occurrence.

By Fact 3, the suffix tree $\mathcal{ST}_P$ can be computed in time and space $\mathcal{O}(m)$. By Lemma 9, lists $\mathcal{L}_i$, for all $i \in [0, n-1]$, can be computed in time $\mathcal{O}(nm^2 + N)$. By Lemma 8, sets $\mathcal{B}_{S,P}$ can be computed in time $\mathcal{O}(N)$. In case $|S| < m$, we use $\mathcal{L}_{i-1}$ and set $\mathcal{B}_{S,P}$ to find and report occurrence i in time $\mathcal{O}(m)$ using a bit vector of size m, which we initialise only once per position i. Finally, searching P in $S \in \tilde{T}[i]$, in case $|S| \geq m$, can be done in time $\mathcal{O}(|S|)$ using the KMP algorithm [13], which is bounded by $\mathcal{O}(N)$ for $\tilde{T}$ of total size N.

The algorithm reads a position i and reports whether i is an ending position of some occurrence of P, before reading position $i+1$. Therefore, Algorithm EDSM solves the *EDSM* problem in an on-line manner in time $\mathcal{O}(nm^2 + N)$, with preprocessing time and space $\mathcal{O}(m)$. ◀

5 Bit-Vector Algorithm

We introduce here Algorithm EDSM-BV, a *non-trivial* bit-vector version of Algorithm EDSM.

Main idea. The main idea of this algorithm is to simulate the previous algorithm using bit-level operations to maintain linked-lists $\mathcal{L}$ and do the matching. To this end, we also add a further preprocessing step to the suffix tree of the pattern. This augmented suffix tree allows us to retrieve a bit-vector representation of all occurrences of an $S \in \tilde{T}[i]$ in P in time linear in $|S|$. With this structure, we can use bit-level operations to compute $\mathcal{L}_i$ from $\mathcal{L}_{i-1}$.

We maintain a bit vector $\boldsymbol{B}$ of size m initialised with 0's, such that, for each position $0 \leq k < m$, $\boldsymbol{B}[k] = 1$ if and only if $P[0\,..\,k]$ has an occurrence ending at the current position of $\tilde{T}$. For each letter $c \in \Sigma$, we construct a bit vector $\boldsymbol{I}_c$ of size m initialised with 0's, such that for each position $0 < k < m-1$, $\boldsymbol{I}_c[k-1] = 1$, if and only if $P[k] = c$. We construct the suffix tree of P, denoted by $\mathcal{ST}_P$, and augment it with bit vectors of size m initialised with 0's for each explicit node as follows: for node u, we create bit vector $\boldsymbol{M}_u$ such that $\boldsymbol{M}_u[k-1] = 1$, if and only if the factor $\mathcal{P}(u)$ represented by node u occurs at position k in P, $0 < k < m-1$. The occurrences of $\mathcal{P}(u)$ can be found at terminal nodes in the subtree rooted at node u. We denote this augmented suffix tree of P by $\mathsf{Occ\text{-}Vector}_P$. We wish to answer the following type of on-line queries: given a string α, if α is a factor of P, then $\mathsf{Occ\text{-}Vector}_P(\alpha)$ finds the node w in $\mathcal{ST}_P$ which represents α, and returns a pointer to the bit vector $\boldsymbol{M}_u$, where u is the first explicit node in the subtree rooted at w. Otherwise (if α is not a factor of P), $\mathsf{Occ\text{-}Vector}_P(\alpha)$ returns a pointer to a bit vector consisting of m 0's. This operation can be trivially realised in time $\mathcal{O}(|\alpha|)$. Note that both $\boldsymbol{I}_c$ and $\boldsymbol{M}_u$ are shifted one bit to the left with respect to the pattern position they refer to; this is just an optimisation that will save us a shift in the algorithm.

Below, we formally present Algorithm EDSM-BV that solves the *EDSM* problem in an on-line manner.

```
Algorithm EDSM-BV($P, m, \tilde{T}, n, \Sigma$)
    Construct $\boldsymbol{I}_c$, for all $c \in \Sigma$, and Occ-Vector$_P$;
    $\boldsymbol{B}[0\,..\,m-1] \leftarrow 0$;
    foreach $S \in \tilde{T}[0]$ do
        Compute $\mathcal{B}_{P,S}$ using the border table;
        foreach $b \in \mathcal{B}_{P,S}$ do
            $\boldsymbol{B}[b] \leftarrow 1$;
        if $|S| \geq m$ then
            Search $P$ in $S$ using KMP and
            report 0 if $P$ occurs in $S$ and CheckDuplicate(0);
    foreach $i \in [1, n-1]$ do
        $\boldsymbol{B_1}[0\,..\,m-1] \leftarrow 0$;
        foreach $S \in \tilde{T}[i]$ do
            Compute $\mathcal{B}_{P,S}$ using the border table;
            foreach $b \in \mathcal{B}_{P,S}$ do
                $\boldsymbol{B_1}[b] \leftarrow 1$;
            if $|S| < m$ then
                $\boldsymbol{B_2} \leftarrow \boldsymbol{B}$ & Occ-Vector$_P(S)$;
                $\boldsymbol{B_1} \leftarrow \boldsymbol{B_1} \mid (\boldsymbol{B_2} \gg |S|)$;
            if $|S| \geq m$ then
                Search $P$ in $S$ using KMP and
                report $i$ if $P$ occurs in $S$ and CheckDuplicate($i$);
            $\boldsymbol{B_3} \leftarrow \boldsymbol{B}$;
            foreach $j \in [0, \min\{|S|, m-1\} - 1]$ do
                $\boldsymbol{B_3} \leftarrow \boldsymbol{B_3}$ & $\boldsymbol{I}_{S[j]}$;
                $\boldsymbol{B_3} \leftarrow \boldsymbol{B_3} \gg 1$;
                if $\boldsymbol{B_3}[m-1] = 1$ then
                    Report $i$ if CheckDuplicate($i$);
        $\boldsymbol{B} \leftarrow \boldsymbol{B_1}$;
```

In Algorithm EDSM-BV, at each iteration i, $\tilde{T}[i]$ is processed (Lines $11-29$) and, at the end, vector $\boldsymbol{B}$ stores indexes k such that $P[0\,..\,k]$ ends at position i.

▶ **Lemma 12.** *Bit vectors $\boldsymbol{I}_c$, for all $c \in \Sigma$, $\sigma = |\Sigma|$, can be constructed in time $\mathcal{O}(m+\sigma \cdot \lceil \frac{m}{w} \rceil)$ and space $\mathcal{O}(\sigma \cdot \lceil \frac{m}{w} \rceil)$. Occ-Vector$_P$ can be constructed in time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$.*

Proof. For the bit vectors $\boldsymbol{I}_c$, we first read the alphabet and construct σ bit vectors of size m initialised with 0's. Then we only need to read the pattern once, and for each position $0 < k < m-1$ in the pattern such that $P[k] = c$, we set $\boldsymbol{I}_c[k-1] = 1$. Reading the pattern once and setting $\boldsymbol{I}_c$ costs time $\mathcal{O}(m)$, so in total we need time $\mathcal{O}(m + \sigma \cdot \lceil \frac{m}{w} \rceil)$ for the bit vectors $\boldsymbol{I}_c$. The space for each bit vector of size m is $\mathcal{O}(\lceil \frac{m}{w} \rceil)$, so in total $\mathcal{O}(\sigma \cdot \lceil \frac{m}{w} \rceil)$ space is required.

By Fact 3, $\mathcal{ST}_P$ can be constructed in time and space $\mathcal{O}(m)$. We traverse $\mathcal{ST}_P$ and allocate a bit vector $\boldsymbol{M}_u$ of size m initialised with 0's for every explicit node u we visit. If u is a terminal node representing suffix $P[k \mathinner{.\,.} m-1]$, we set $\boldsymbol{M}_u[k-1] = 1$. If u is a non-terminal node, we set $\boldsymbol{M}_u[k-1] = 1$ for all terminal nodes representing suffixes $P[k \mathinner{.\,.} m-1]$ in the subtree rooted at u, $0 < k < m-1$. This can be realised by using an Or bitwise operation between the bit vectors of the children of node u. By applying this for all explicit nodes of $\mathcal{ST}_P$, we build Occ-Vector$_P$. We have exactly m terminal nodes, and no more than m non-terminal nodes in $\mathcal{ST}_P$, thus, the bit vectors $\boldsymbol{M}_u$ for $\mathcal{ST}_P$ can be constructed in time $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$. The space required for Occ-Vector$_P$ is $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$ since we have $\mathcal{O}(m)$ bit vectors and each bit vector requires space $\mathcal{O}(\lceil \frac{m}{w} \rceil)$. ◀

▶ **Theorem 13.** *Algorithm EDSM-BV solves the EDSM problem in an on-line manner in time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$. Algorithm EDSM-BV requires preprocessing time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$.*

Proof. The correctness of the algorithm follows from the correctness of the KMP algorithm [13] if $|S| \geq m, S \in \tilde{T}[i]$. By the definition of bit vectors $\boldsymbol{I}_c$, we read each $S \in \tilde{T}[i]$, letter by letter, and try to extend the prefixes of P, position by position, using Shift-And bitwise operations [17]. When we reach the end of the bit vector $\boldsymbol{B}_3$, we may find an occurrence. No other occurrences can be found since we extend position by position, which means if we cannot reach the end of $\boldsymbol{B}_3$, we must have had at least one mismatch which prevents the extension.

By Lemma 12, the time and space for the preprocessing of Algorithm EDSM-BV is bounded by $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$. For each $S \in \tilde{T}[i]$, $|S| \geq m$, searching P in S can be done in time $\mathcal{O}(|S|)$ using the KMP algorithm [13], which is bounded by $\mathcal{O}(N)$ for all S. The Shift-And bitwise operation can be done in time $\mathcal{O}(\lceil \frac{m}{w} \rceil)$ [17], and it is repeated $|S|$ or $m-1$ times for each S to find an occurrence. Since we choose the minimum of $|S|$ and $m-1$, this time is bounded by $\mathcal{O}(|S| \cdot \lceil \frac{m}{w} \rceil)$, which is bounded by $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ for $\tilde{T}$. By Lemma 7, sets $\mathcal{B}_{P,S}$ can be computed in time $\mathcal{O}(N)$. Updating $\boldsymbol{B}$ for position $i = 0$ and updating $\boldsymbol{B_1}$ for each position $i > 0$ using sets $\mathcal{B}_{P,S}$ can be done in time $\mathcal{O}(N)$ for $\tilde{T}$. For each $S \in \tilde{T}[i]$, $|S| < m$, Occ-Vector$_P(S)$ requires time $\mathcal{O}(|S|)$ to return the corresponding bit vector, and updating $\boldsymbol{B_1}$ requires time $\mathcal{O}(\lceil \frac{m}{w} \rceil)$ using bit-level operations. Note that $\boldsymbol{B_1}$ needs only to be updated if $\boldsymbol{B} \neq 0$. So for all $\tilde{T}[i]$, the total time of this step can be bounded by $\mathcal{O}(N + N' \cdot \lceil \frac{m}{w} \rceil)$, where N' is the number of strings S such that $|S| < |P|$ and $\boldsymbol{B} \neq 0$. Since $N' \leq N$, this time is bounded by $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$.

The algorithm reads a position i, and reports whether i is an ending position of some occurrence of P, before reading position $i+1$. Therefore, Algorithm EDSM-BV solves the *EDSM* problem in an on-line manner in time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$, with preprocessing time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$. ◀

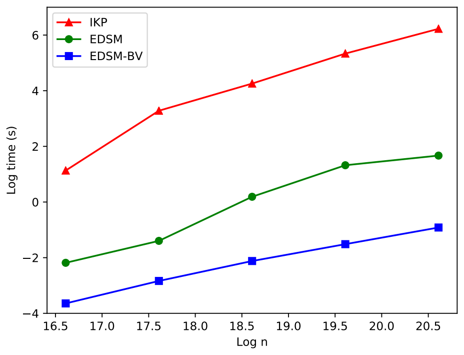
6 Experimental Results

We have implemented Algorithms EDSM and EDSM-BV in the C++ programming language. The implementation of the algorithm presented in [11], which we denote here by IKP, was taken from `https://github.com/Ritu-Kundu/ElDeS`. Recall that Algorithm IKP solves the *EDSM* problem in time $\mathcal{O}(\alpha\gamma mn + N)$ and space $\mathcal{O}(N)$; where α and γ are parameters, respectively representing the maximum number of strings in any degenerate position of the text and the maximum number of degenerate positions spanned by any occurrence of the pattern in the text. Note that Algorithm IKP outputs both the starting and ending positions of pattern occurrences, while the output of Algorithms EDSM and EDSM-BV is only the ending positions. All three programs were compiled with `g++` version 4.7.3 at optimisation level 3 (-O3). The following experiments were conducted on a desktop computer using one core of Intel® Core™ i7-2600S CPU at 2.8GHz and 8GB of RAM under 64-bit GNU/Linux. We compared the performance of EDSM, EDSM-BV, and IKP using synthetic data; as well as the performance of EDSM-BV—shown to be the fastest—using real data. The implementation of EDSM-BV is available at `https://github.com/webmasterar/edsm` under the terms of the GNU General Public License. The synthetic datasets referred to in this section are maintained at the same web-site.

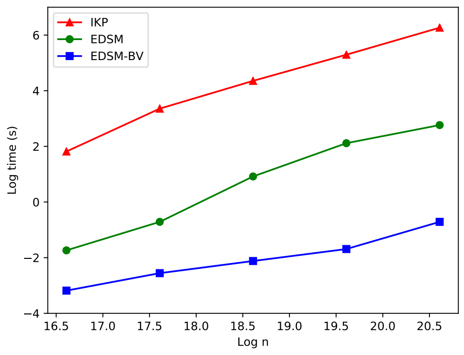
Synthetic data. Synthetic ED texts were created randomly (uniform distribution over the DNA alphabet) with n ranging from $100,000$ to $1,600,000$; and the percentage of degenerate positions was set to 10%. For each degenerate position within the synthetic ED texts, the number of strings was chosen randomly, with an upper bound set to 10. The length of each string of a degenerate position was chosen randomly, with an upper bound again set to 10. Every non-degenerate position within the synthetic ED texts contained a single letter. Four different patterns of length $m = 8, 16, 32$, or 64 were given as input to all three programs, along with the aforementioned synthetic ED texts, resulting in four sets of output.

Our theoretical findings showing that Algorithms EDSM and EDSM-BV are asymptotically faster than Algorithm IKP are validated in practice by the results illustrated in Figure 1. Note that the axes are in $\log_2$ scale. In particular, the results confirm that Algorithm EDSM-BV, which is asymptotically the fastest for short patterns, is also the fastest in practice by up to two orders of magnitude. As for Algorithm EDSM, not surprisingly, we observe that, as m grows, the m^2 factor in its time complexity becomes more and more significant overall. Note that searching for much longer patterns *exactly* is not relevant in applications of interest, where errors (substitutions, insertions, and deletions) must be accommodated as m grows.

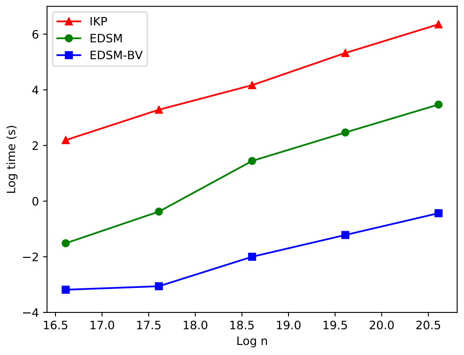
Real data. EDSM-BV was tested further using real-world datasets. Human genomic data was obtained from the $1,000$ Genomes Project [23]. Specifically, data was obtained from Phase 3 of the project, in which the genomes of $2,504$ individuals from 26 different populations were sequenced and aligned, producing a dataset which summarises the variation in the sample population. Files in Variant Call Format (VCF) include information about variations at each position in the reference genome, which makes the format ideal for our purposes. EDSM-BV was given a reference sequence (in FASTA format) and variation data (in VCF) for each of the ten smallest chromosomes as input, as well as synthetic patterns of length $m = 8, 16, 32$, or 64. The average percentage of degenerate positions across these chromosomes was approximately 3%; the average number of strings at degenerate positions was 2; and the average length of strings at degenerate positions was 1. The processing time of EDSM-BV was recorded; with *processing* we refer only to the actual CPU time used in executing the process—excluding the



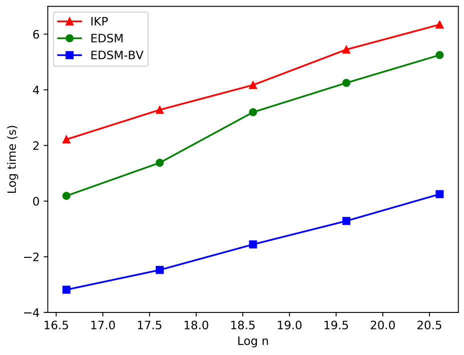
(a) Pattern of length $m = 8$



(b) Pattern of length $m = 16$



(c) Pattern of length $m = 32$



(d) Pattern of length $m = 64$

Figure 1 Elapsed time of EDSM, EDSM-BV, and IKP for synthetic ED texts of length n.

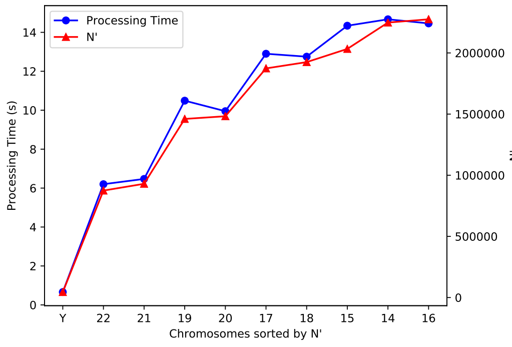
time to read the data in memory on-line. Chromosome 21, which is the smallest in length, has a VCF file of size 11.2GB. The results of this experiment are displayed in Figure 2.

The graphs in Figure 2 show, for the ten smallest chromosomes, a very clear *linear* relationship between the time taken for EDSM-BV to run and N', the total number of strings $S \in \tilde{T}[i]$ such that $|S| < |P|$ and $\boldsymbol{B} \neq 0$, per chromosome. Recall that the total time required by EDSM-BV for updating bit vector $\boldsymbol{B}_1$ from $\boldsymbol{B}$ is $\mathcal{O}(N + N' \cdot \lceil \frac{m}{w} \rceil)$. This is the most time-consuming operation in practice as it searches for S in the suffix tree of P and then updates $\boldsymbol{B}_1$ using bit-level operations. Note that, the total time to process strings $S \in \tilde{T}[i]$, with $|S| > |P|$, using KMP is $\mathcal{O}(N)$, which becomes insignificant overall in practice.

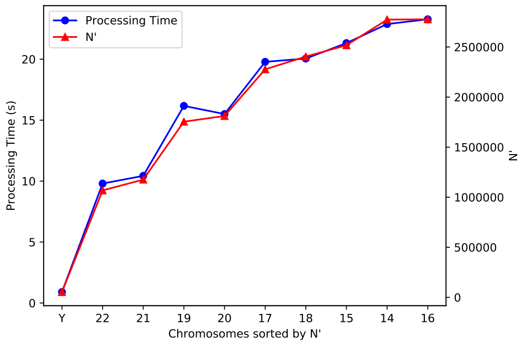
7 Final Remarks

We have presented two efficient algorithms for on-line pattern matching on a set of similar texts. Notably, one of the algorithms requires time linear in the size of the texts' representation, for short patterns, that is $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$. The presented experimental results confirm our theoretical findings in practical terms.

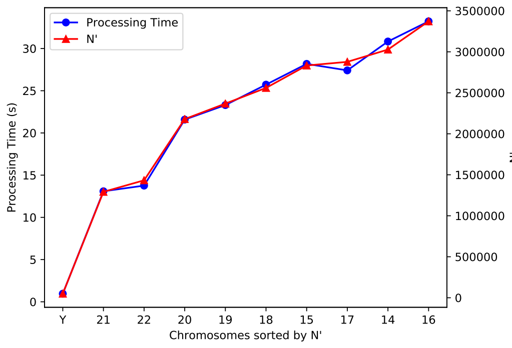
Our immediate target is to apply these on-line solutions for fast average-case approximate pattern matching or for multiple pattern matching on a set of similar texts. An open problem is to either improve on the $\mathcal{O}(nm^2 + N)$-time algorithm or show conditional lower bounds.



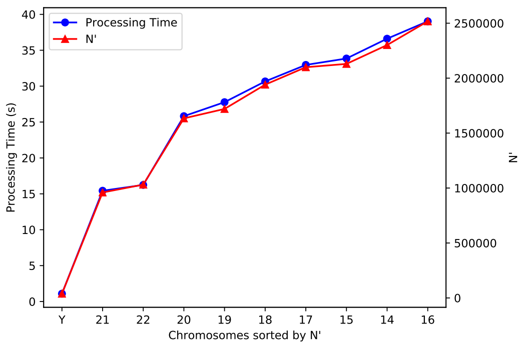
(a) Pattern of length $m = 8$



(b) Pattern of length $m = 16$



(c) Pattern of length $m = 32$



(d) Pattern of length $m = 64$

Figure 2 Processing time of EDSM-BV for real ED texts (Human chromosomes and variants).

Acknowledgements. We would like to thank Ritu Kundu (Department of Informatics, King's College London) for providing us with the implementation of Algorithm IKP [11].

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

2 Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, 1990. doi:10.1016/S0022-2836(05)80360-2.

3 Ricardo A. Baeza-Yates and Chris H. Perleberg. Fast and practical approximate string matching. *Inf. Process. Lett.*, 59(1):21–27, 1996. doi:10.1016/0020-0190(96)00083-X.

4 Uwe Baier, Timo Beller, and Enno Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. *Bioinformatics*, 32(4):497–504, 2016. doi:10.1093/bioinformatics/btv603.

5 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. doi:10.1017/cbo9780511546853.

6 Martin Farach. Optimal suffix tree construction with large alphabets. In Anna Karlin, editor, *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997)*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.

7 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

8 Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms Mol. Biol.*, 11:3, 2016. doi:10.1186/s13015-016-0066-8.

9 Jan Holub, William F. Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. *J. Discrete Algorithms*, 6(1):37–50, 2008. doi:10.1016/j.jda.2006.10.003.

10 Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):361–370, 2013. doi:10.1093/bioinformatics/btt215.

11 Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate texts. In Frank Drewes, Carlos Martín-Vide, and Bianca Truthe, editors, *Proceedings of the 11th International Conference on Language and Automata Theory and Applications (LATA 2017)*, volume 10168 of *LNCS*, pages 131–142. Springer International Publishing, 2017. doi:10.1007/978-3-319-53733-7_9.

12 Paul Julian Kersey, James E. Allen, Irina Armean, Sanjay Boddu, Bruce J. Bolt, Denise Carvalho-Silva, Mikkel Christensen, Paul Davis, Lee J. Falin, Christoph Grabmueller, Jay C. Humphrey, Arnaud Kerhornou, Julia Khobova, Naveen K. Aranganathan, Nicholas Langridge, Ernesto Lowy, Mark D. McDowall, Uma Maheswari, Michael Nuhn, Chuang Kee Ong, Bert Overduin, Michael Paulini, Helder Pedro, Emily Perry, Giulietta Spudich, Electra Tapanari, Brandon Walts, Gareth Williams, Marcela K. Tello-Ruiz, Joshua C. Stein, Sharon Wei, Doreen Ware, Daniel M. Bolser, Kevin L. Howe, Eugene Kulesha, Daniel Lawson, Gareth Maslen, and Daniel M. Staines. Ensembl genomes 2016: more genomes, more complexity. *Nucleic Acids Res.*, 44(Database-Issue):574–580, 2016. doi:10.1093/nar/gkv1209.

13 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

14 Sorina Maciuca, Carlos del Ojo Elias, Gil McVean, and Zamin Iqbal. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In Martin C. Frith and Christian Nørgaard Storm Pedersen, editors, *Proceedings of the 16th International Workshop on Algorithms in Bioinformatics (WABI 2016)*, volume 9838 of *LNCS*, pages 222–233. Springer, 2016. doi:10.1007/978-3-319-43681-4_18.

15 Joong Chae Na, Hyunjoon Kim, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment: A compressed index for similar strings. *Theor. Comput. Sci.*, 638:159–170, 2016. doi:10.1016/j.tcs.2015.08.008.

16 Gonzalo Navarro and Alberto Ordóñez Pereira. Faster compressed suffix trees for repetitive collections. *ACM J. Exp. Algorithmics*, 21(1):1.8:1–1.8:38, 2016. doi:10.1145/2851495.

17 Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002. doi:10.1017/cbo9781316135228.

18 Ngan Nguyen, Glenn Hickey, Daniel R. Zerbino, Brian J. Raney, Dent Earl, Joel Armstrong, W. James Kent, David Haussler, and Benedict Paten. Building a pan-genome reference for a population. *J. Comput. Biol.*, 22(5):387–401, 2015. doi:10.1089/cmb.2014.0146.

19 Nadia Pisanti, Henry Soldano, Mathilde Carpentier, and Joël Pothier. A relational extension of the notion of motifs: Application to the common 3D protein substructures searching problem. *J. Comput. Biol.*, 16(12):1635–1660, 2009. doi:10.1089/cmb.2008.0019.

20 Marie-France Sagot, Alain Viari, Joël Pothier, and Henry Soldano. Finding flexible patterns in a text: an application to three-dimensional molecular matching. *Comput. Appl. Biosci.*, 11(1):59–70, 1995. doi:10.1093/bioinformatics/11.1.59.

21 Siavash Sheikhizadeh, M. Eric Schranz, Mehmet Akdel, Dick de Ridder, and Sandra Smit. Pantools: representation, storage and exploration of pan-genomic data. *Bioinformatics*, 32(17):487–493, 2016. doi:10.1093/bioinformatics/btw455.

22 Jouni Sirén. Indexing variation graphs. In Sándor Fekete and Vijaya Ramachandran, editors, *Proceedings of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX 2017)*, pages 13–27. SIAM, 2017. `doi:10.1137/1.9781611974768.2`.

23 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015. `doi:10.1038/nature15393`.

24 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Brief. Bioinformatics*, pages 1–18, 2016. `doi:10.1093/bib/bbw089`.

A Family of Approximation Algorithms for the Maximum Duo-Preservation String Mapping Problem

Bartłomiej Dudek[1], Paweł Gawrychowski[2], and Piotr Ostropolski-Nalewaja[3]

1 **Institute of Computer Science, University of Wrocław, Wrocław, Poland**
2 **Institute of Computer Science, University of Wrocław, Wrocław, Poland; and University of Haifa, Haifa, Israel**
3 **Institute of Computer Science, University of Wrocław, Wrocław, Poland**

Abstract

In the Maximum Duo-Preservation String Mapping problem we are given two strings and wish to map the letters of the former to the letters of the latter as to maximise the number of duos. A duo is a pair of consecutive letters that is mapped to a pair of consecutive letters in the same order. This is complementary to the well-studied Minimum Common String Partition problem, where the goal is to partition the former string into blocks that can be permuted and concatenated to obtain the latter string.

Maximum Duo-Preservation String Mapping is APX-hard. After a series of improvements, Brubach [WABI 2016] showed a polynomial-time 3.25-approximation algorithm. Our main contribution is that, for any $\epsilon > 0$, there exists a polynomial-time $(2 + \epsilon)$-approximation algorithm. Similarly to a previous solution by Boria et al. [CPM 2016], our algorithm uses the local search technique. However, this is used only after a certain preliminary greedy procedure, which gives us more structure and makes a more general local search possible. We complement this with a specialised version of the algorithm that achieves 2.67-approximation in quadratic time.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases approximation scheme, minimum common string partition, local search

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.10

1 Introduction

A fundamental question in computational biology and, consequently, stringology, is comparing similarity of two strings. A textbook approach is to compute the edit distance, that is, the smallest number of operations necessary to transform one string into another, where every operation is inserting, removing, or replacing a character. While this can be efficiently computed in quadratic time, a major drawback from the point of view of biological applications is that every operation changes only a single character. Therefore, it makes sense to also allow moving arbitrary substrings as a single operation to obtain edit distance with moves. Such relaxation makes computing the smallest number of operations NP-hard [17], but Cormode and Muthukrishnan [9] showed an almost linear-time $O(\log n \cdot \log^* n)$-approximation algorithm. The problem is already interesting if the only allowed operation is moving a substring. This is usually called the Minimum Common String Partition (MCSP). Formally, we are given two strings X and Y, where Y is a permutation of X. The goal is to cut X into the least number of pieces that can be rearranged (without reversing) and concatenated to obtain Y.

MCSP is known to be APX-hard [12]. Chrobak et al. [8] analysed performance of the simple greedy approximation algorithm, that in every step extracts the longest common substring from the input strings, and Kaplan and Shafrir [16] further improved their bounds. This simple greedy algorithm can be implemented in linear time [13], and further tweaked to obtain better practical results [14]. Also, an exact exponential time algorithm [11] and different parameterizations were considered [15, 5, 6, 10].

There was also some interest in the complementary problem called the Maximum Duo-Preservation String Mapping (MPSM), introduced by Chen et al. [7]. The goal there is to map the letters of X to the letters of Y as to maximise the number of preserved duos. A duo is a pair of consecutive letters, and a duo of X is said to be preserved if its pair of consecutive letters is mapped to a pair of consecutive letters of Y (in the same order). MCSP and MPSM are indeed complementary, as one can think of preserving a duo as not splitting its two letters apart to see that the number of preserved duos and the number of pieces add up to $|X|$. Of course, this does not say anything about the relationship between the approximation guarantees for both problems. Chen et al. [7] designed a k^2-approximation algorithm based on linear programming for the restricted version of the problem, called k-MPSM, where each letter occurs at most k times. This was soon followed by an APX-hardness proof of 2-MPSM and a general 4-approximation algorithm provided by Boria et al. [3]. The approximation ratio was then improved to 3.5 [2] using a particularly clean argument based on local search. Finally, Brubach [4] obtained a 3.25-approximation, and Beretta et al. [1] considered parameterized tractability.

Our main contribution is a family of polynomial-time approximation algorithms for MPSM: for any $\varepsilon > 0$, we show a polynomial-time $(2 + \varepsilon)$-approximation algorithm. We complement this with a specialised (and simplified) version of the algorithm that achieves 2.67-approximation in quadratic time, which already improves on the approximation guarantee and the running time of the previous solutions, as the running time of the 3.5-approximation was $O(n^4)$. At a high level, we also apply local search, that is, we iteratively try to slightly change the current solution as long as such a change leads to an improvement. The intuition is that not being able to find such local improvement should imply a $(2 + \varepsilon)$-approximation guarantee. This requires considering larger and larger neighbourhoods of the current solution for smaller and smaller ε and seems problematic already for $\varepsilon = 1$. To overcome this, we apply local search only after a certain preliminary greedy procedure, which gives us more structure and makes a more general local search possible.

2 Preliminaries

In the Maximum Duo-Preservation String Mapping (MPSM) we are given two strings X and Y, where Y is a permutation of X. The goal is to map the letters of X to the letters of Y as to maximise the number of preserved duos. A duo is a pair of consecutive letters, and a duo of X is said to be preserved if its pair of consecutive letters is mapped to a pair of consecutive letters of Y (in the same order). This can be restated by creating a bipartite graph $G = (A \dot\cup B, E)$, where $n = |X| - 1 = |A| = |B|$ and $A = \{a_1, a_2, \dots, a_n\}$ and $B = \{b_1, b_2, \dots, b_n\}$. Node a_i corresponds to duo $(X[i], X[i+1])$ and similarly b_i corresponds to $(Y[i], Y[i+1])$. Two nodes are connected with an edge if their corresponding duos are the same, that is, $E = \{(a_i, b_j) : X[i] = Y[j] \text{ and } X[i+1] = Y[j+1]\}$. See Figure 1.

Now, we want to find a maximum matching in G that corresponds to a proper mapping of letters between the strings, that is, such that every two consecutive mapped duos (consisting of three consecutive letters) are mapped to two consecutive duos (in the same order). It is

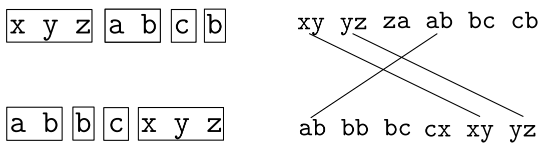


Figure 1 An optimal solution of MCSP for strings `xyzabcb` and `abbcxyz` (left). It corresponds to a solution of MPSM, where the mapping preserves duos $(x, y), (y, z)$, and (a, b) (right).

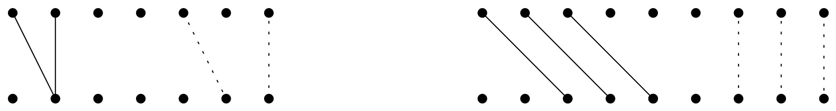


Figure 2 Two pairs of overlapping edges (left) and decomposition of a consecutive matching into streaks (right).

not necessary that all duos are mapped. Formally, a matching M is called consecutive if every two neighbouring nodes are either matched to two neighbouring nodes (preserving the order) or at least one of them is unmatched:

$$\forall_{i,j,j' \in \{1..n\}} \big(\langle a_i, b_j \rangle \in M \wedge \langle a_{i+1}, b_{j'} \rangle \in M\big) \Rightarrow \big(j' = j+1\big)$$

and a symmetric condition for the other side of the graph. Even though the graph G obtained as described above from an instance of MPSM has some additional structure, we focus only on the more general problem where the given bipartite graph $G = (A \dot\cup B, E)$ is arbitrary and we are looking for a consecutive matching of maximum cardinality. This was called the Maximum Consecutive Bipartite Matching (MCBM) by Boria et al. [3].

Definitions. We say that two edges $\langle a_i, b_j \rangle$ and $\langle a_{i'}, b_{j'} \rangle$ are overlapping if $|i - i'| \leq 1$ or $|j - j'| \leq 1$. Given a consecutive matching M, we define a streak to be a maximal (under inclusion) set of *consecutive* edges $e_1, e_2, \ldots, e_k$, such that for some p, q we have that $e_i = \langle a_{p+i}, b_{q+i} \rangle$ for all $i = 1, 2, \ldots, k$. See Figure 2. Note that from the definition, e_i overlaps with itself, e_{i-1} and e_{i+1} (assuming that these edges exist). This notion is extended to sets of edges: S_1 overlaps with S_2 if there exist $e_1 \in S_1, e_2 \in S_2$ such that e_1 overlaps with e_2. Similarly, we define overlaps between an edge and a set of edges. Note that every consecutive matching M can be uniquely decomposed into a set of streaks such that no two of them are overlapping with each other.

3 Greedy Algorithm

Consider a simple greedy procedure, that in every step takes the longest possible streak from G and, if the streak consists of at least k edges, adds it to the solution. See Algorithm 1.

To analyse quality of the returned solution, we fix an optimal solution OPT and would like to compare $|ALG|$ with $|OPT|$. Let s_i be the streak that was removed in the i-th step of the algorithm and o_i be the set of edges from OPT that are overlapping with s_i, but were not overlapping with $s_1, s_2, \ldots, s_{i-1}$. In other words, o_i consists of those edges from OPT that after $i-1$ steps of the algorithm still could have been added to the solution, but are no longer available after the i-th step. Note that o_i contains all the edges of $OPT \cap s_i$, because every edge overlaps with itself. Observe that $|o_i| \leq 2|s_i| + 4$ as there can be at most $|s_i| + 2$ edges from o_i overlapping with s_i at each side of G. Moreover, even a stronger property holds:

Algorithm 1 Choosing the largest possible streak greedily.

```
1: function GREEDY($k$)
2:     $ALG := \emptyset$
3:     while true do
4:         $s :=$ the largest streak in $G$
5:         if $|s| < k$ then
6:             break
7:         remove $s$ and all edges overlapping with $s$ from $G$
8:         $ALG := ALG \cup s$
9:     return $ALG$
```

▶ **Lemma 1.** $|o_i| \leq 2|s_i| + 2$.

Proof. Suppose that the endpoints of s_i at one side of the graph (say A) form a sequence of nodes $a_j, a_{j+1}, \ldots, a_{j+|s_i|-1}$. Define $\mathcal{E} = \{a_{j-1}, a_j, \ldots, a_{j+|s_i|-1}, a_{j+|s_i|}\}$ (assuming that a_{j-1} and $a_{j+|s_i|}$ exist). We will show that at most $|s_i| + 1$ edges from o_i can end in $\mathcal{E}$. Then, applying the same reasoning to the other side of the graph will finish the proof. If $|\mathcal{E}| < |s_i| + 2$ then the claim holds. Otherwise, if $|\mathcal{E}| = |s_i| + 2$ there are three cases to consider:

1. There are two or more streaks from o_i ending in $\mathcal{E}$. Then they cannot end in all nodes from $\mathcal{E}$, because at least two of them would be overlapping with each other. Thus there is at least one node from $\mathcal{E}$ that is not an endpoint of edge from o_i, so there are at most $|s_i| + 1$ of them.
2. There is one streak from o_i ending in $\mathcal{E}$. Then the streak cannot be larger than $|s_i|$, because then the greedy algorithm would have taken the larger streak (recall that o_i consists of edges that could have been added to the solution in the i-th step). Thus there are at most $|s_i|$ edges of o_i ending in $\mathcal{E}$.
3. There is no streak from o_i ending in $\mathcal{E}$. Then the statement holds trivially. ◀

We still need to specify the algorithm for smaller streaks (consisting of less than k edges), but before doing so in the next section we bound the quality of the solution found by the greedy algorithm.

Let m be the number of steps performed by the greedy algorithm. The algorithm returns $ALG = \bigcup_{i=1}^m s_i$ which should be compared with the set of edges of OPT that can no longer be taken due to the decisions made by the greedy algorithm, that is, $\bigcup_{i=1}^m o_i \subseteq OPT$. Using Lemma 1 we can compute the desired ratio as follows:

$$\frac{|\bigcup_{i=1}^m o_i|}{|\bigcup_{i=1}^m s_i|} = \frac{\sum_{i=1}^m |o_i|}{\sum_{i=1}^m |s_i|} \leq \frac{\sum_{i=1}^m (2|s_i| + 2)}{\sum_{i=1}^m |s_i|} = 2 + \frac{m \cdot 2}{\sum_{i=1}^m |s_i|} \leq 2 + \frac{m \cdot 2}{m \cdot k} = 2 + \frac{2}{k}$$

where the last inequality holds because all taken streaks consist of at least k edges.

To conclude, the solution ALG found by the greedy algorithm is at most $2 + \frac{2}{k}$ times smaller than the set of edges from OPT that is overlapping with ALG. Informally, on average we discard only a few edges of OPT for every edge from ALG. After running the algorithm for $k = 1$, there will be no edges left and thus we have a simple 4-approximation algorithm. To obtain a better approximation ratio, we will increase k and focus on the subgraph G' of G consisting of all edges that are not overlapping with any streak s_i already taken by the algorithm (and hence still available). The crucial insight is that we can analyse the performance of the greedy algorithm on $G \setminus G'$ and the performance of the algorithm for small k on G' separately. We know that the approximation ratio of the greedy algorithm on

$G \setminus G'$ is $2 + \frac{2}{k}$ and size of the optimal solution for G' is at least $|OPT - \bigcup_{i=1}^{m} o_i|$. Then, due to the definition of G', any solution found for G' can be combined with ALG to obtain a solution for the original instance, so the final approximation ratio is the maximum of $2 + \frac{2}{k}$ and the ratio of the algorithm used for G'.

4 Algorithm for Small k

As stated above, applying the greedy algorithm with $k = 1$ immediately implies a 4-approximation algorithm. For larger values of k we need another phase to find a solution for the remaining part of the graph. For $k = 2$, we present a simple algorithm based on maximum bipartite matching (not consecutive) that can be used to obtain a 3-approximation. For larger values of k, we first consider $k = 3$ and design a quadratic-time algorithm based on the local search technique. Then, we move to a general k and develop a more involved polynomial-time algorithm that achieves $(2 + \varepsilon)$-approximation.

4.1 3-approximation Based on Maximum Matching for $k = 2$

After running Greedy(2) there are no streaks of size 2. Recall that $G' = (A\dot{\cup}B, E')$ is the subgraph of the original graph G consisting of all edges that are not overlapping with the already taken edges. Consider the following algorithm:

1. Create a bipartite graph $H = (A'\dot{\cup}B', F)$ where:
   - $A' = \{a_{(1,2)}, a_{(3,4)}, \ldots, a_{(n-1,n)}\}$ and similarly for B'. In other words, nodes of A' correspond to merged pairs of neighbouring nodes of A (if n is odd, the last node of A' corresponds to a single node of A).
   - $F = \Big\{\{a_{(2i-1,2i)}, b_{(2j-1,2j)}\} : \{a_{2i-1}, a_{2i}\} \times \{b_{2j-1}, b_{2j}\} \cap E' \neq \emptyset\Big\}$. In other words, there is an edge between two merged pairs of nodes if there was an edge between a node from the first pair and a node from the second pair.
2. Find the maximum matching M' in H.
3. For every edge of M', choose an edge of G' connecting nodes from the corresponding pairs (if there are multiple possibilities, choose any of them). Let M be the set of chosen edges.
4. Let $ALG \leftarrow \emptyset$. Process all edges of M in arbitrary order. For an edge $(a_i, b_j) \in M$:
   - remove from M all edges ending in nodes $a_{i-1}, a_{i+1}, b_{j-1}$ and b_{j+1},
   - add (a_i, b_j) to ALG.
5. Return ALG.

Consider the optimal solution OPT. As G' contains no streaks consisting of 2 or more edges, the endpoints of any two of its edges cannot be neighbouring. Therefore, OPT can be translated into a matching in H with the same cardinality, so $|OPT| \leq |M'|$.

We claim that after including an edge $(a_i, b_j) \in M$ in ALG at most 2 other edges are removed from M. Assume otherwise, that is, there are 3 such edges. Without loss of generality, one of them ends in a_{i-1} and one in a_{i+1}. Depending on the parity of i, edge (a_i, b_j) and the edge ending in either a_{i-1} or a_{i+1} correspond in H to edges ending in the same node of A'. This is a contradiction, because all edges in M' have distinct endpoints. Because initially $|M'| = |M|$, we conclude that $|ALG| \geq |M'|/3$.

Combining the inequalities gives us $3 \cdot |ALG| \geq |M'| \geq |OPT|$, so the above algorithm is a 3-approximation for graphs with no streaks of size at least 2. Combining it with Greedy(2), that guarantees approximation ratio of $2 + \frac{2}{k} = 2 + \frac{2}{2} = 3$, gives us a 3-approximation algorithm for the whole problem.

Algorithm 2 Local improvements in $O(m^2n^2)$ time.

```
1: function LocalImprovements
2:     $ALG := \emptyset$
3:     while true do
4:         if $\exists e \notin ALG$ s.t. $ALG \cup \{e\}$ is a valid solution then
5:             $ALG := ALG \cup \{e\}$
6:         if $\exists e_1, e_2 \notin ALG, e' \in ALG$ s.t. $ALG \setminus \{e'\} \cup \{e_1, e_2\}$ is a valid solution then
7:             $ALG := ALG \setminus \{e'\} \cup \{e_1, e_2\}$
8:         if $|ALG|$ was not increased then
9:             break
10:    return $ALG$
```

4.2 2.67-approximation for $k = 3$

For $k = 3$ we use procedure LocalImprovements based on the local search technique. See Algorithm 2. Essentially the same method was used to obtain the 3.5-approximation [2]. The algorithm consists of a number of steps in which it tries to either add a single edge or remove one edge so that two other edges can be added. However, the crucial difference is that in our case there are no streaks of size greater than 2 in G'. This allows for a better bound on the approximation ratio.

Fix an optimal solution OPT. We want to bound the total number C of overlaps between the edges from ALG and OPT. First, observe that an edge from ALG can overlap with at most 4 edges from OPT, because there are no streaks of size 3 in the graph. Thus:

$$4 \cdot |ALG| \geq C. \tag{1}$$

Second, let k_1 be the number of edges from OPT that overlap with exactly one edge from ALG. Then all other edges from OPT overlap with at least two edges from ALG (because otherwise the algorithm would have taken an edge not overlapping with any already taken edge), so:

$$C \geq k_1 + 2 \cdot (|OPT| - k_1) = 2 \cdot |OPT| - k_1. \tag{2}$$

▶ **Lemma 2.** $k_1 \leq |ALG|$.

Proof. Suppose that $k_1 > |ALG|$. Then there are two edges $e_1, e_2 \in OPT$ that overlap with only one and the very same edge $e_{del} \in ALG$. But then the algorithm would have increased size of the solution by removing e_{del} and adding e_1 and e_2, so we obtain a contradiction. ◀

Applying Lemma 2 to (2) and combining with (1) we get $4 \cdot |ALG| \geq C \geq 2 \cdot |OPT| - |ALG|$ and thus $2.5 \cdot |ALG| \geq |OPT|$. Recall that the approximation ratio of the first greedy part of the algorithm is $2 + \frac{2}{3} < 2.67$, so the overall ratio of the combined algorithm is also 2.67. The algorithm clearly runs in polynomial time as in every iteration of the main loop the size of ALG increases by one and is bounded by n. In [2] the running time was further optimised to $O(n^4)$, but in the remaining part of this section we will describe how to decrease the time to $O(n^2)$. We will also show how to implement the greedy algorithm in the same $O(n^2)$ complexity, thus obtaining an 2.67-approximation algorithm in $O(n^2)$ time.

Greedy part in $O(n^2)$ time. We show how to implement Greedy(k) in $O(n^2)$ time. Recall that in every iteration the algorithm chooses the longest streak in the remaining part of the graph, includes it in the solution, and removes all edges that overlap with it from the graph. The procedure terminates if the streak contains less than k edges.

We start with creating a list L of edges $\langle x, y\rangle$ sorted lexicographically first by x and then by y. This can be done in $O(n^2)$ time using bucket sort and while sorting we can also retrieve for every edge the edge that would be its predecessor in a streak. Then we iterate over the edges in L and split them into streaks. The edges of every streak are stored in a doubly linked list and every edge stores a pointer to its streak. We also keep streaks grouped by size, that is, D_s contains all streaks of size s. To allow insertions and deletions in $O(1)$ time, D_s is internally also implemented as a doubly linked list, but in order not to confuse it with the lists storing edges inside a streak, later on we will refer to lists D_s as groups.

Having split all edges into streaks and grouped streaks by their sizes, we iterate over the groups $D_n, D_{n-1}, \ldots, D_k$ and retrieve a streak s from the non-empty group with the largest index. We add s to the solution and remove all edges overlapping with s from the graph. Every removed edge either decreases the size of its streak by one or splits it into two smaller streaks. In both cases, the smaller streak(s) is moved between the appropriate groups. Removing an edge takes constant time and every edge is removed at most once from the graph. Similarly, moving or splitting of a streak due to a removed edge takes constant time as the size of the smaller streak can be computed in constant time by looking at its first and last edge. Thus, the overall time of the procedure is $O(n^2)$.

▶ Remark. Recall that we have generalised the MPSM problem and now are working with an arbitrary bipartite graph G. However, if G was constructed from an instance of MPSM, then finding the longest available streak corresponds to finding the longest string that occurs in both X and Y without overlapping with any of the previously chosen substrings. Goldstein and Lewenstein [13] showed how to implement such a procedure in $O(n)$ total time.

Local improvements in $O(n^2)$ time. Recall that to analyse the approximation ratio (in Lemma 2), we only need that after termination of the algorithm there are no three edges $e_1, e_2 \notin ALG, e_{del} \in ALG$ such that $ALG \setminus \{e_{del}\} \cup \{e_1, e_2\}$ is a valid solution. At a high level, FastLocalImprovements keeps track of edges that can potentially increase size of the solution in a queue Q. As long as Q is not empty, we retrieve a candidate edge e from Q. First, we verify that $e \notin ALG$ and e overlaps with at most one edge from ALG. If e can be added to ALG, we do so and continue after adding to Q all edges overlapping with e. Otherwise, we check if some other edge e' can be added while removing another edge e_{del} at the same time using procedure TryAddingPairWith(e), and if so, we add to Q all edges overlapping with one of the modified edges (e, e' and e_{del}). See Algorithm 3 and Algorithm 4.

The algorithm uses the following data structures and functions:

- For every node $v \in G'$, we keep a list of all edges from E ending in v and separately edges of ALG ending in v.
- Close(e) is the set of nodes of G' at distance at most 1 from the endpoints of edge e. In other words, Close(e) is the set of up to 6 nodes where edges overlapping with e can end.
- Overlap(e) is the set of edges overlapping with edge e. It is computed on the fly, by iterating through edges ending in $v \in$ Close(e).
- Queue Q of candidate edges. For every edge in E we remember if it is currently in Q in order not to store any duplicates and keep the space usage $O(m)$.
- For every node $v \in G'$ we keep a list L_v of edges from $E \setminus ALG$ that overlap with exactly one edge from ALG and end in v. To keep these lists updated, every time an edge $e = \langle x, y\rangle$ is enqueued or added or removed from ALG, we count the edges from ALG it

Algorithm 3 Local improvements in $O(n^2)$ time.

```
1: function FastLocalImprovements
2:     $Q$.enqueue($E$)
3:     while $Q$ is not empty do
4:         $e := Q$.dequeue()
5:         if $e \in ALG$ or $e$ overlaps with more than one edge from $ALG$ then
6:             continue
7:         if $ALG \cup \{e\}$ is a valid solution then
8:             $ALG := ALG \cup \{e\}$
9:             $Q$.enqueue($\mathsf{Overlap}(e)$)
10:            continue
11:        TryAddingPairWith($e$)
```

Algorithm 4 Adding a pair with edge e.

```
1: function TryAddingPairWith($e$)
2:     $e_{del}$ := the only edge from $ALG$ overlapping with $e$
3:     for each $e'$ that can be a neighbour of $e$ in a streak do          ▷ $O(1)$
4:         if $ALG \setminus \{e_{del}\} \cup \{e, e'\}$ is a valid solution then
5:             $ALG := ALG \setminus \{e_{del}\} \cup \{e, e'\}$
6:             $Q$.enqueue($\mathsf{Overlap}(e) \cup \mathsf{Overlap}(e') \cup \mathsf{Overlap}(e_{del})$)
7:             return
8:     for each node $v \in \mathsf{Close}(e_{del}) \setminus \mathsf{Close}(e)$ do          ▷ $O(1)$
9:         for each edge $e' \in L_v$ do          ▷ see Lemma 3
10:            if $ALG \setminus \{e_{del}\} \cup \{e, e'\}$ is a valid solution then
11:                $ALG := ALG \setminus \{e_{del}\} \cup \{e, e'\}$
12:                $Q$.enqueue($\mathsf{Overlap}(e) \cup \mathsf{Overlap}(e') \cup \mathsf{Overlap}(e_{del})$)
13:                return
```

overlaps with. If there is only one of them, we make sure that e is in L_x and L_y, otherwise we remove e from L_x and L_y.

Clearly, after termination of the algorithm there is no triple of edges e_1, e_2 and e_{del} that can be used to increase the solution, because every time an edge is added to or removed from the solution, all of its overlapping edges are enqueued. It remains to prove that Algorithm 3 indeed runs in $O(n^2)$ time. First, observe that $|\mathsf{Close}(e)| \leq 6$, so from the definition of overlapping edges $|\mathsf{Overlap}(e)| \leq |\mathsf{Close}(e)| \cdot n \in O(n)$, as there are at most n edges ending in a node. So, every time the algorithm enqueues a set of edges, there are at most $O(n)$ of them. As this happens only after increasing the size of ALG, which can happen at most n times, in total there are $O(n^2)$ enqueued edges. So it suffices to prove that every time an edge e is dequeued, it takes $O(1)$ time to check if it can be used to increase the solution. Here we disregard the time for enqueuing edges due to increasing the size of ALG, as it adds up to $O(n^2)$ as mentioned before. Note that both counting the edges overlapping with e and finding the unique edge from ALG overlapping with e takes $O(1)$ time, as we just need to check edges from ALG ending in $\mathsf{Close}(e)$. Similarly, as ALG is always a valid solution, each validity check takes $O(1)$ time, as we always try to modify a constant number of edges. By the same argument, loops in lines 3 and 8 take constant number of iterations, and also:

▶ **Lemma 3.** *There are $O(1)$ iterations of the loop in line 9 of* TryAddingPairWith*(e).*

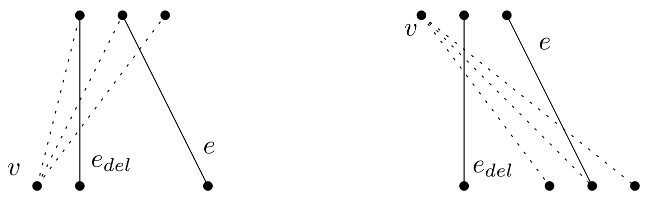


Figure 3 Dotted lines show the only 3 possible edges $e' \in L_v$ that overlap with e. Among any 4 edges in L_v, at least one can be used to increase $|ALG|$ and break the loop.

Algorithm 5 Improvements of bounded size.

```
 1: function BoundedSizeImprovements($t$)
 2:     $ALG := \emptyset$
 3:     while true do
 4:         for each $E_{\text{remove}}, E_{\text{add}} \subseteq E$ such that $|E_{\text{remove}}| < |E_{\text{add}}| \leq t$ do
 5:             $ALG' := ALG \setminus E_{\text{remove}} \cup E_{\text{add}}$
 6:             if $ALG'$ is a valid solution then
 7:                 $ALG := ALG'$
 8:                 break
 9:         if $ALG$ was not improved then
10:             break
11:     return $ALG$
```

Proof. Consider an edge $e' \in L_v$ such that $ALG' := ALG \setminus \{e_{del}\} \cup \{e, e'\}$ is not a valid solution. From the definition of L_v, e' overlaps only with $e_{del} \in ALG$, so both $ALG \setminus \{e_{del}\} \cup \{e\}$ and $ALG \setminus \{e_{del}\} \cup \{e'\}$ are valid solutions. Thus, the only reason for ALG' not being valid is that e' overlaps with e. But v is at distance 2 or more from the endpoint of e, so e and e' can be overlapping only at the other side of the graph. There are at most 3 possible endpoints of such e' at the other side, see Figure 3. Consequently, after checking 4 edges from L_v we will surely find one that can be used to increase $|ALG|$. ◀

To conclude, Greedy(3) with FastLocalImprovements yield 2.67-approximation in $O(n^2)$ time.

5 $(2+\varepsilon)$-approximation

Given $\varepsilon > 0$ we would like to create a polynomial time $(2+\varepsilon)$-approximation algorithm. We set $k = \lceil \frac{2}{\varepsilon} \rceil$ and run Greedy(k) to remove all streaks of size at least k from the graph G. From now we focus on the subgraph G' remaining after the first greedy phase and let OPT denote the optimal solution in G'.

Let $t = \lceil \frac{4}{\varepsilon} \rceil + 1$ and ALG be the solution found by BoundedSizeImprovements(t), see Algorithm 5. Similarly to the case $k = 3$, the algorithm tries to improve the current solution using local optimisations, however now the number of edges that we try to add or remove in every step is bounded by t (that depends on ε). We want to prove that $(2+\varepsilon) \cdot |ALG| \geq |OPT|$. To this end, we assign $(2+\varepsilon)$ units of credit to every edge of ALG. Then the goal is to distribute the credits from the edges of ALG to the edges of OPT, so that every edge of OPT receives at least one credit. Alternatively, we can think of transferring credits to the streaks from OPT, in such a way that a streak consisting of s edges receives at least s credits. This will clearly demonstrate the required inequality.

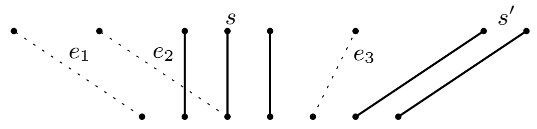


Figure 4 Dotted lines denote edges from ALG. According to the scheme, e_1 and e_2 transfer a credit to an edge from s, but e_3 does not because its endpoint is between s and s'.

Credit distribution scheme. Every edge from ALG distributes $(1 + \frac{\varepsilon}{2})$ credits from each of its two endpoints independently. Consider an endpoint v_i of an edge from ALG. Let $\ldots, v_{i-1}, v_i, v_{i+1}, \ldots$ be all nodes at the corresponding side of the graph G. If there is an edge $e \in OPT$ ending in v_i, then e receives 1 credit. Now consider the case when no edge of OPT ends in v_i. If exactly one edge from OPT ends in v_{i+1} or v_{i-1} then the credit is transferred to that edge. If there are no edges ending there then the credit is not transferred at all. Finally, if there is an edge $e \in OPT$ ending at v_{i-1} and another edge $e' \in OPT$ ending at v_{i+1}, then for the time being neither e nor e' receives the credit. In such a situation we say that the node v_i is between the streak containing e and the streak containing e', call the credit *uncertain* and defer deciding whether it should be transferred to e or e'. Observe that the only case when an edge $e \in ALG$ overlapping with a streak s does not transfer the credit to s is when the endpoint of e is between two streaks s and s', see Figure 4. Note that two credits can be transferred from e to s if both endpoints of e transfer its credits to s. The remaining $\frac{\varepsilon}{2}$ credits are not transferred to any specific edge yet. We will aggregate and redistribute them using a more global argument, but first need some definitions.

Gaps and balance. Define the balance of a streak s from OPT as the number of credits obtained in the described scheme (ignoring the uncertain credits) minus the number of edges in s. A gap is an edge of OPT that has not received any credits yet and $\mathsf{gaps}(s)$ is the number of gaps in s. Observe that the balance of a streak s is at least $-\mathsf{gaps}(s)$. After running the greedy algorithm and BoundedSizeImprovements(t), even a stronger property holds:

▶ **Lemma 4.** *The balance of every streak is at least* -2.

Proof. Consider a streak s. If there are less than 2 gaps in s then the claim holds. Otherwise, let g_1 and g_2 be the first and the last gap in s, so that we can write $s = Ag_1Mg_2B$, see Figure 5. Note that the balance of both A and B is non-negative, as from the definition there are no gaps inside, so every edge there receives at least one credit. However, there might be multiple gaps in M. Suppose that the balance of M is negative. But the size of M is smaller than $k < t$, so BoundedSizeImprovements(t) would have replaced a subset of edges from ALG with M to increase size of the solution. Therefore, the balance of M is nonnegative. Finally, observe that the balance of s is equal to the sum of balances of A, M and B minus 2 (for the gaps g_1 and g_2), so it is at least -2 in total. ◀

The following corollary that follows from the above proof will be useful later:

▶ **Corollary 5.** *Every streak s with balance -2 can be represented as $s = Ag_1Mg_2B$ where g_1 and g_2 are the first and last gap of s, respectively. The balance of Ag_1 and g_2B is -1 while the balance of M is 0.*

Analysis of the scheme. We construct an auxiliary multi-graph H, where the vertices are streaks of OPT with balance at least -1. Streaks with balance -2 are split into two smaller

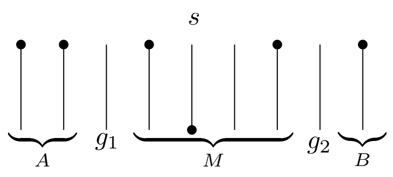


Figure 5 Black dots denote endpoints of edges from ALG, g_1 and g_2 is the first and the last gap, respectively.

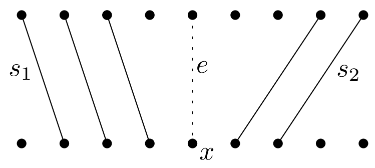


Figure 6 If there is an endpoint x of edge $e \in ALG$ that is between two streaks s_1, s_2 of OPT then we add an edge between s_1 and s_2 in H.

streaks (called substreaks) with balance -1 as explained in Corollary 5. We create an edge between two streaks in H when they both overlap with an endpoint of an edge from ALG. In other words, when edge e from ALG has an endpoint x overlapping with two streaks of OPT, then there is an edge in H between the vertices corresponding to these streaks, see Figure 6. Observe that then there is no edge of OPT ending in x and there can be at most two edges between any pair of streaks.

Now we will show that for every connected component of H there are enough credits to distribute at least one credit to every edge from OPT in the component. The intuition behind considering the connected components of H is that we have deferred distribution of the uncertain credits, and now a connected component is a set of streaks that needs to decide together how to spend those uncertain credits. At a high level, for every connected component $\mathcal{C}$ of H there will be two cases two consider. First, if the balance of C is non-negative, then we are done. Otherwise, we will show that the balance of $\mathcal{C}$ is equal to -1. We also know that the component is so big that BoundedSizeImprovements was not able to increase the solution. From this we will conclude that, by gathering the remaining $\frac{\varepsilon}{2}$ credits together, it is possible to cover the deficit.

Consider one connected component $\mathcal{C}$ on w vertices. We want to prove that there are at least w credits transferred to all edges of $\mathcal{C}$ in total. From the construction we have that every vertex of $\mathcal{C}$ has balance at least -1. Moreover, as the component is connected, there are at least $w-1$ edges, each adding one uncertain credit. Thus, the total balance of the whole component (including the uncertain credits) is at least -1. Observe that the only case when the total balance of the component is -1 is a tree (with exactly $w-1$ edges) where every node has balance -1. In all other cases the balance is non-negative already.

We denote by $K_{\mathcal{C}}$ the set of edges of OPT from all vertices of $\mathcal{C}$ (recall that they correspond to original streaks with balance -1 and substreaks). We also define an auxiliary set $M_{\mathcal{C}}$ that consists of the middle parts M of the original streaks. More precisely, for every streak s of balance -2, if it was a part of $\mathcal{C}$ (due to the substreak Ag_1 or g_2B, where $s = Ag_1Mg_2B$), we add to $M_{\mathcal{C}}$ all edges from M. From Corollary 5, the balance of every such M is 0. Now consider the following set of edges $X_{\mathcal{C}} = K_{\mathcal{C}} \cup M_{\mathcal{C}}$. There are two cases to consider depending on how many credits have been transferred to $X_{\mathcal{C}}$:

1. If there are at least $c \geq \frac{4}{\varepsilon}$ credits transferred to the edges of $X_{\mathcal{C}}$ (each credit from an endpoint of an edge from ALG), then we can use half of the remaining $\frac{\varepsilon}{2}$ credit of each

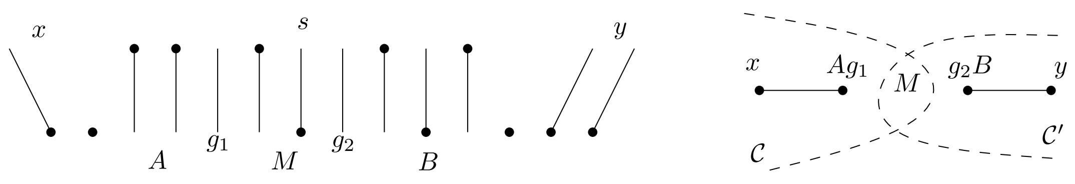


Figure 7 As there is an *uncertain* credit between streaks x and Ag_1, there will be an edge between them in H, so they will be in a connected component $\mathcal{C}$ of H. Similarly for g_2B and y in $\mathcal{C}'$. Observe that the middle part M of the split streak s is accounted for in both $M_{\mathcal{C}}$ and $M_{\mathcal{C}'}$.

endpoint and transfer it to the component. Note that for each credit from those c already assigned to $X_{\mathcal{C}}$ there is one endpoint still having additional $\frac{\varepsilon}{4}$ credit that can be spent on $X_{\mathcal{C}}$. We can use only half of the remaining $\frac{\varepsilon}{2}$ credit because some edges (from the middle parts of original streaks) can belong to both $X_{\mathcal{C}}$ and $X_{\mathcal{C}'}$ for two different components $\mathcal{C}$ and $\mathcal{C}'$, see Figure 7, and they might need to transfer additional credit to both of them. Thus, for each of the c credits we transfer additional $\frac{\varepsilon}{4}$ credit, so in total we transfer at least one full credit, which is enough to cover the deficit of the component.

2. In the second case, the edges from $X_{\mathcal{C}}$ received less than $\frac{4}{\varepsilon}$ credits, so there are less than $\frac{4}{\varepsilon}+1$ edges from OPT (recall that the overall balance of the component is -1). Note that if we add all edges from $X_{\mathcal{C}}$ and remove all edges from ALG that have transferred credits to the edges from $X_{\mathcal{C}}$, the size of the solution will increase as earlier the overall balance was negative. The solution will still be valid, because we have removed all edges from ALG overlapping with the edges of $X_{\mathcal{C}}$. Also for the split streaks, we took edges up to (but not including) a gap which from the definition does not share an endpoint with an edge from ALG. Furthermore, as the size of $X_{\mathcal{C}}$ is at most $\frac{4}{\varepsilon}+1 \leq t$, it would have been considered as the set E_{add} of edges to be checked by our algorithm. Thus, this case is impossible, as we would have been able to improve the current solution.

To conclude, every connected component containing w edges receives at least w credits, so $(2+\varepsilon)\cdot|ALG| \geq |OPT|$. As the approximation ratio of the first greedy part is also $(2+\varepsilon)$, as explained before the overall algorithm is an $(2+\varepsilon)$-approximation for MPSM. It remains to analyse its time complexity. Let m denote the number of edges of G'. There are at most n steps of the algorithm, as in each of them size of the solution increases by at least one and is bounded by n. There are $\binom{m}{t} \in O(m^t)$ candidates for E_{add} and E_{remove} and we can check in $O(m)$ time if a given solution is valid. In total, substituting $t = \lceil\frac{4}{\varepsilon}\rceil + 1$ the total time complexity is $O(m^{2t+1}) = O(n^{4t+2}) = O(n^{\frac{16}{\varepsilon}+6}) = n^{O(1/\varepsilon)}$.

▶ **Theorem 6.** *Combining the greedy algorithm with local improvements yields a* $(2+\varepsilon)$*-approximation for MCBM in* $n^{O(1/\varepsilon)}$ *time, for any* $\varepsilon > 0$.

▶ **Corollary 7.** *There exists a* $(2+\varepsilon)$*-approximation algorithm for MPSM running in* $n^{O(1/\varepsilon)}$ *time, for any* $\varepsilon > 0$.

References

1 Stefano Beretta, Mauro Castelli, and Riccardo Dondi. Parameterized tractability of the maximum-duo preservation string mapping problem. *Theor. Comput. Sci.*, 646:16–25, 2016. doi:10.1016/j.tcs.2016.07.011.

2 Nicolas Boria, Gianpiero Cabodi, Paolo Camurati, Marco Palena, Paolo Pasini, and Stefano Quer. A 7/2-approximation algorithm for the maximum duo-preservation string mapping problem. In Roberto Grossi and Moshe Lewenstein, editors, *Proceedings of the 27th Annual*

Symposium on Combinatorial Pattern Matching (CPM 2016), volume 54 of *LIPIcs*, pages 11:1–11:8. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.CPM.2016.11`.

3 Nicolas Boria, Adam Kurpisz, Samuli Leppänen, and Monaldo Mastrolilli. Improved approximation for the maximum duo-preservation string mapping problem. In Daniel G. Brown and Burkhard Morgenstern, editors, *Proceedings of the 14th International Workshop on Algorithms in Bioinformatics (WABI 2014)*, volume 8701 of *LNCS*, pages 14–25. Springer, 2014. `doi:10.1007/978-3-662-44753-6_2`.

4 Brian Brubach. Further improvement in approximating the maximum duo-preservation string mapping problem. In Martin C. Frith and Christian Nørgaard Storm Pedersen, editors, *Proceedings of the 16th International Workshop on Algorithms in Bioinformatics (WABI 2016)*, volume 9838 of *LNCS*, pages 52–64. Springer, 2016. `doi:10.1007/978-3-319-43681-4_5`.

5 Laurent Bulteau, Guillaume Fertin, Christian Komusiewicz, and Irena Rusu. A fixed-parameter algorithm for minimum common string partition with few duplications. In Aaron E. Darling and Jens Stoye, editors, *Proceedings of the 13th International Workshop on Algorithms in Bioinformatics (WABI 2013)*, volume 8126 of *LNCS*, pages 244–258. Springer, 2013. `doi:10.1007/978-3-642-40453-5_19`.

6 Laurent Bulteau and Christian Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In Chandra Chekuri, editor, *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, pages 102–121. SIAM, 2014. `doi:10.1137/1.9781611973402.8`.

7 Wenbin Chen, Zhengzhang Chen, Nagiza F. Samatova, Lingxi Peng, Jianxiong Wang, and Maobin Tang. Solving the maximum duo-preservation string mapping problem with linear programming. *Theor. Comput. Sci.*, 530(Complete):1–11, 2014. `doi:10.1016/j.tcs.2014.02.017`.

8 Marek Chrobak, Petr Kolman, and Jiří Sgall. The greedy algorithm for the minimum common string partition problem. *ACM Trans. Algorithms*, 1(2):350–366, October 2005. `doi:10.1145/1103963.1103971`.

9 Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007. `doi:10.1145/1219944.1219947`.

10 Peter Damaschke. Minimum common string partition parameterized. In Keith A. Crandall and Jens Lagergren, editors, *Proceedings of the 8th International Workshop on Algorithms in Bioinformatics (WABI 2008)*, volume 5251 of *LNCS*, pages 87–98. Springer, 2008. `doi:10.1007/978-3-540-87361-7_8`.

11 Bin Fu, Haitao Jiang, Boting Yang, and Binhai Zhu. Exponential and polynomial time algorithms for the minimum common string partition problem. In Weifan Wang, Xuding Zhu, and Ding-Zhu Du, editors, *Proceedings of the 5th International Conference on Combinatorial Optimization and Applications (COCOA 2011)*, volume 6831 of *LNCS*, pages 299–310. Springer, 2011. `doi:10.1007/978-3-642-22616-8_24`.

12 Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. *Electron. J. Comb.*, 12, 2005. URL: `http://www.combinatorics.org/Volume_12/Abstracts/v12i1r50.html`.

13 Isaac Goldstein and Moshe Lewenstein. Quick greedy computation for minimum common string partition. *Theor. Comput. Sci.*, 542:98–107, July 2014. `doi:10.1016/j.tcs.2014.05.006`.

14 Dan He. A novel greedy algorithm for the minimum common string partition problem. In Ion I. Mandoiu and Alexander Zelikovsky, editors, *Proceedings of the 3rd International Symposium on Bioinformatics Research and Applications (ISBRA 2007)*, volume 4463 of *LNCS*, pages 441–452. Springer, 2007. `doi:10.1007/978-3-540-72031-7_40`.

15 Haitao Jiang, Binhai Zhu, Daming Zhu, and Hong Zhu. Minimum common string partition revisited. *J. Comb. Optim.*, 23(4):519–527, 2012. `doi:10.1007/s10878-010-9370-2`.

16 Haim Kaplan and Nira Shafrir. The greedy algorithm for edit distance with moves. *Inf. Process. Lett.*, 97(1):23–27, 2006. `doi:10.1016/j.ipl.2005.08.010`.

17 Dana Shapira and James A. Storer. Edit distance with move operations. *J. Discrete Algorithms*, 5(2):380–392, 2007. `doi:10.1016/j.jda.2005.01.010`.

Revisiting the Parameterized Complexity of Maximum-Duo Preservation String Mapping[*]

Christian Komusiewicz[1], Mateus de Oliveira Oliveira[2], and Meirav Zehavi[3]

1 **Friedrich-Schiller-Universität Jena, Jena, Germany**
`christian.komusiewicz@uni-jena.de`
2 **University of Bergen, Bergen, Norway**
`mateus.oliveira@ii.uib.no`
3 **University of Bergen, Bergen, Norway**
`meirav.zehavi@ii.uib.no`

Abstract

In the Maximum-Duo Preservation String Mapping (Max-Duo PSM) problem, the input consists of two related strings A and B of length n and a nonnegative integer k. The objective is to determine whether there exists a mapping m from the set of positions of A to the set of positions of B that maps only to positions with the same character and preserves at least k *duos*, which are pairs of adjacent positions. We develop a randomized algorithm that solves Max-Duo PSM in time $4^k \cdot n^{O(1)}$, and a deterministic algorithm that solves this problem in time $6.855^k \cdot n^{O(1)}$. The previous best known (deterministic) algorithm for this problem has running time $(8e)^{2k+o(k)} \cdot n^{O(1)}$ [Beretta et al., Theor. Comput. Sci. 2016]. We also show that Max-Duo PSM admits a problem kernel of size $O(k^3)$, improving upon the previous best known problem kernel of size $O(k^6)$.

1998 ACM Subject Classification G.2.1 Combinatorics, F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases comparative genomics, parameterized complexity, kernelization

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.11

1 Introduction

Computing distances between strings is a fundamental task in computer science. For many distance measures, the distance between two strings A and B is defined as the minimum number of local operations that are needed to transform A into B, for example the deletion or insertion of a character. For these measures, the distance between two strings A and B can be usually computed in polynomial time [13, 22]. In some applications, however, it is necessary to consider nonlocal operations that transform one string into the other. In comparative genomics, for example, genomes are modeled as strings with one character corresponding to a complete gene and one is interested in determining the evolutionary distance between two genomes. During biological evolution, genomes may be altered by large-scale mutations such as the reversal or the transposition of larger parts of the genome [19].

One approach to approximate the distance between two strings A and B with respect to many of these operations is to compute a smallest *common string partition* [11, 26].

[*] Christian Komusiewicz gratefully acknowledges support from the DFG, project MAGZ (KO 3669/4-1). Mateus de Oliveira Oliveira gratefully acknowledges support from the Bergen Research Foundation.

Informally, a size-ℓ common string partition of two strings A and B is a partition of A and B, each into ℓ nonoverlapping substrings, such that the resulting two multisets of substrings of A and B are the same. The problem to compute a smallest common string partition, known as Minimum Common String Partition, is NP-hard [11, 21].

An alternative way of defining such a partition is to ask for a partition of A into ℓ nonoverlapping substrings such that permuting the order of these substrings and concatenating them subsequently gives the string B. This second view implies a mapping m that (bijectively) maps each position i of A to a position $m(i)$ of B such that $A[i] = B[m(i)]$. The size of the common string partition is then exactly the number of pairs of consecutive positions i and $i+1$ such that $m(i)+1 \neq m(i+1)$; these positions are called *duos*. Therefore, computing a mapping m that maps only positions with the same characters to each other and maximizes the number k of consecutive positions for which $m(i)+1 = m(i+1)$ directly yields a minimum common string partition of A and B. The problem of computing such a mapping is known as Maximum-Duo Preservation String Mapping (Max-Duo PSM). Since Max-Duo PSM is simply a dual of the Minimum Common String Partition problem, it is NP-hard as well. Motivated by this hardness, we study Max-Duo PSM from the viewpoint of parameterized algorithmics. More precisely, our aim is to obtain efficient algorithms when the parameter is k, the number of preserved duos. Before describing previous and our results, we give a formal problem definition.

Formal Problem Definition. Let A and B be two strings over a finite set of symbols Σ. Throughout this work, we assume that $|A| = |B| = n$ and that A and B are *related*, that is, B is a permutation of A. A *mapping of A into B* is a (bijective) function $m : [n] \to [n]$ where for each $i \in [n]$,[1] $A[i] = B[m(i)]$. A *duo* in A is a pair of consecutive positions $(i, i+1)$ of A. We say that a mapping m *preserves a duo* $(i, i+1)$ if $m(i)+1 = m(i+1)$. Accordingly, the Max-Duo PSM problem is defined as follows.

Maximum-Duo Preservation String Mapping (Max-Duo PSM)
Input: Two related strings, A and B, and a nonnegative integer k.
Question: Does there exist a (bijective) mapping m of A into B such that the number of preserved duos is at least k?

Previous Work. Initially, Max-Duo PSM has been proposed as an alternative possibility of achieving approximation algorithms for Minimum Common String Partition (MCSP) [10], because the best known polynomial-time approximation algorithm has an approximation factor of $O(\log n \log^* n)$ [12]. Consequently, most work on Max-Duo PSM focuses on approximation algorithms with the first constant-factor approximation algorithm achieving an approximation factor of 4 [6]. This was subsequently improved to a factor of 3.5 [5] and then to a factor of 3.25 [7]. Recently further progress concerning the approximation factor has been reported [18, 27].

Bretta et al. [2, 1] initiated the study of Max-Duo PSM from the viewpoint of parameterized algorithmics. They studied both the fixed-parameter tractability and the kernelization complexity of Max-Duo PSM, showing that this problem can be solved in time $(8e)^{2k+o(k)} \cdot n^{O(1)}$, and that it admits a kernel of size $O(k^6)$. Thus, Bretta et al. [2, 1] were the first to show that Max-Duo PSM is FPT and that it admits a polynomial kernel.

[1] We use $[n]$ as shorthand for $\{1, 2, \ldots, n\}$.

The fixed-parameter algorithm of Bretta et al. [2, 1] is based on a combination of color coding and dynamic programming.

In comparison with Max-Duo PSM, MCSP has been investigated more thoroughly from the viewpoint of parameterized algorithms. Damaschke [15] presented the first fixed-parameter algorithms for MCSP, for combined parameters such as "partition size ℓ plus repetition number of the input strings".[2] Subsequently, MCSP was shown to be fixed-parameter tractable with the single parameter partition size ℓ [9]. Jiang et al. [23] considered the combined parameter "partition size ℓ plus maximum occurrence d of any character" and showed that MCSP can be solved in time $(d!)^k \cdot n^{O(1)}$. Subsequently, this running time was improved to $O(d^{2k} \cdot kn)$ [8].

Our Contribution. We make two main contributions. First, we develop two algorithms for the Max-Duo PSM problem that are substantially faster than the (deterministic) algorithm by Bretta et al. [2, 1], which runs in time $(8e)^{2k+o(k)} \cdot n^{O(1)}$. Specifically, we develop a randomized algorithm that solves Max-Duo PSM in time $4^k \cdot n^{O(1)}$, as well as a deterministic algorithm that solves this problem in time $6.855^k \cdot n^{O(1)}$. Here, in the context of our randomized algorithm, we mean that if we determine that the input is a yes-instance, then this answer is necessarily correct, and if we determine that the input is a no-instance, then this answer is correct with probability at least $9/10$.[3] For the purpose of developing our algorithms, we present a reduction from Max-Duo PSM to a problem of finding paths in an edge-colored graph, which might be of independent interest. This reduction lies at the heart of our algorithms, since by employing advanced tools from the field of parameterized algorithmics, namely, the methods of narrow sieves [4, 3] and representative sets [20], it is possible to quickly solve the resulting graph problem.

Second, we prove that Max-Duo PSM admits a kernel of size $O(k^3)$, improving upon the kernel of size $O(k^6)$ by Bretta et al. [2].

Preliminaries. We use $[i, j]$ to denote the set $\{i, i+1, \ldots, j\}$ of natural numbers between i and j. Moreover, given a string A, we denote the substring starting at position i and ending at position j by $A[i, j]$. For a (directed) graph G, let $V(G)$ denote the vertex set of G and $E(G)$ the edge set of G.

The field of parameterized algorithmics studies *parameterized problems*, where each problem instance is associated with a *parameter* k, usually a nonnegative integer. Given a parameterized problem, the first question is whether the problem is *fixed-parameter tractable (FPT)*, that is, whether it can be solved in time $f(k) \cdot |X|^{O(1)}$, where f is an arbitrary function that depends *only* on k and $|X|$ is the size of the input instance. In other words, the notion of FPT signifies that the combinatorial explosion can be confined to the parameter k. A second question is whether the problem also admits a *polynomial kernelization*. Here, a problem Π is said to admit a polynomial kernelization if there exists a polynomial-time algorithm that, given an instance (X, k) of Π, outputs an equivalent instance $(\widehat{X}, \widehat{k})$ of Π, called a *kernel*, where $|\widehat{X}| = \widehat{k}^{O(1)}$ and $\widehat{k} \leq k$; kernelization is a mathematical concept that aims to analyze preprocessing procedures in a formal, rigorous manner. For further details, refer to [17, 14].

Due to lack of space, several proofs are deferred to an appendix.

[2] The repetition number of a nonempty string x is defined as the largest i such that $x = uv^iw$ where v is nonempty.

[3] Clearly, the probability of success can be improved by running the algorithm multiple times and determining that the input is a yes-instance if and only if at least one of the calls determined so.

2 Reduction to a Path Finding Problem

In this section, we present a reduction from Max-Duo PSM to the following graph problem.

Long Blue Path
Input: A directed acyclic graph (DAG) G, an edge-coloring $c : E(G) \to \{R, B\}$, a vertex-labeling $\ell : V(G) \to \mathbb{N}$, and nonnegative integers k and r.
Question: Does G contain a directed path P such that

- $|V(P)| \leq r$,
- for all $u, v \in V(P)$, $\ell(u) \neq \ell(v)$, and
- $|\{e \in E(P) : c(e) = B\}| \geq k$.

Construction. Let (A, B, k) be an instance of Max-Duo PSM. We construct an instance (G, c, ℓ, k, r) of Long Blue Path as follows (here, the parameter k is the same). First, we initialize G to be an empty graph. Now, for every pair of substrings $A[i, j]$ of A and $B[p, q]$ of B such that $j - i \leq k$ and $A[i, j] = B[p, q]$, we insert a directed path $P_{i,j,p,q}$ on $j - i + 1$ new vertices into G whose edges are colored blue and such that the label of the dth vertex on this path is $(p + d - 1)$. The purpose of this path is to represent the possibility to preserve all duos in $A[i, j]$ by mapping this substring to $B[p, q]$. The labels of the vertices are meant to ensure that every position in B is mapped only once. Now, a complete mapping of A to B can be seen as a combination of mappings of substrings that are represented by the paths. Thus, we next turn to connect the paths we have just constructed by adding new edges.

For every two paths $P_{i,j,p,q}$ and $P_{i',j',p',q'}$ such that $j < i'$, we add a red edge from the last vertex of the path $P_{i,j,q,p}$ to the first vertex of the path $P_{i',j',q',p'}$. Informally, the manner in which we direct these edges is meant to ensure that every position in A is mapped only once. Clearly, the resulting graph G is a DAG. Finally, we set $r = 2k$.

Correctness. We first note that the construction can be done in time $O(|V(G)| + |E(G)|)$. Now, observe that the number of paths $P_{i,j,p,q}$ that G contains is bounded by $n^2(k + 1)$ (as the index q equals $p + (j - i)$), and that each path $P_{i,j,p,q}$ consists of at most $(k + 1)$ vertices. Hence, it holds that $|V(G)| \leq n^2(k + 1)^2$ which directly implies $|E(G)| < n^4(k + 1)^2$. Thus, we have the following observation.

▶ **Observation 1.** *The instance (G, c, ℓ, k, r) can be constructed in time $O(n^4k^2)$.*

We prove the correctness by proving two lemmata that together imply that the instances (A, B, k) and (G, c, ℓ, k, r) are equivalent.

▶ **Lemma 1.** *If (A, B, k) is a yes-instance of* Max-Duo PSM, *then (G, c, ℓ, k, r) is a yes-instance of* Long Blue Path.

Proof. Let m be a mapping from A into B preserving at least k duos. Consider the set $\{A_1, \ldots, A_r\}$ of substrings of A containing exactly the first k preserved duos, where we assume that A_i precedes A_{i+1} in A. Consider any A_z and let $[i_z, j_z]$ be the set of positions of A_z in A. Since the mapping preserves the duos in A_z, there is a substring $B[q_z, p_z]$ such that $m(i_z + s) = q_z + s$, $0 \leq s \leq j - i$. This implies that $A[i_z, j_z] = B[q_z, p_z]$. Thus, G contains the path $P_z := P_{i_z,j_z,q_z,p_z}$.

By the above, for each A_z, G contains a path P_z containing $|A_z| - 1$ blue edges. Moreover, for A_i and A_j, the vertices in P_i and P_j have different labels since the mapping m is injective. Finally, there is a red edge from the last vertex of P_i to the first vertex of P_{i+1} since the last position of A_i is strictly smaller than the first position of A_{i+1}. Thus, the concatenation of P_1, P_2 until P_r gives a path in G. The number of blue edges in this path is exactly k, and the number of vertices in this path is at most $2k$, since every P_i contains at least one blue edge. ◀

▶ **Lemma 2.** *If (G, c, ℓ, k) is a yes-instance of* Long Blue Path*, then (A, B, k) is a yes-instance of* Max-Duo PSM*.*

Proof. Let P be a solution of the Long Blue Pathinstance. That is, P is a path in G on at most $2k$ vertices, all with different labels, containing at least k blue edges. Let $\{P_1, \dots, P_r\}$ be the set of disjoint paths obtained from P by removing all red edges where we assume that there is a red edge from P_i to P_{i+1} for all $i \in [r-1]$. Consider some P_z. By the construction of G, $P_z = P_{i,j,q,p}$ for some $i < j$ and $q < p$. Hence, there is a substring $A[i, j]$ of A and a substring $B[q, p]$ of B such that $A[i, j] = B[q, p]$. Call these two substrings the substrings of A and B, respectively, that *correspond* to P_z. Observe that for P_i and P_j, $i < j$, the substrings corresponding to P_i and P_j are disjoint: For the substrings in A this is due to the fact that the indices of the corresponding substring for P_i are lower than those of the substring of A corresponding to P_j. For the substrings in B this is due to the fact that the vertices in P_i and P_j have different labels. Thus, there is a mapping from A into B that maps the corresponding strings for each path P_i and maps all other positions arbitrarily. The number of duos preserved by this mapping is at least k. ◀

Altogether, we arrive at the following.

▶ **Lemma 3.** *Given an instance (A, B, k) of* Max-Duo PSM*, an equivalent instance (G, c, ℓ, k, r) of* Long Blue Path *where $r = 2k$ can be constructed in time $O(n^4k^2)$.*

3 A Randomized Algorithm based on Narrow Sieves

In this section, we adapt the method of *narrow sieves* that was applied to solve the k-Path problem [4] to solve Long Blue Path. More precisely, our objective is to provide a constructive proof for the following result.

▶ **Lemma 4.** *There exists a randomized algorithm that solves* Long Blue Path *in time $2^r \cdot r^{O(1)} \cdot |E(G)|$ and polynomial space.*

In light of Lemma 3, once we have Lemma 4 at hand, we immediately obtain the following theorem.

▶ **Theorem 5.** *There exists a randomized algorithm that solves* Max-Duo PSM *in time $4^k \cdot k^{O(1)} \cdot n^4$ and polynomial space.*

In the following, we focus on the proof of Lemma 4. To this end, let (G, c, ℓ, k, r) be an instance of Long Blue Path. Clearly, we can assume that $|V(G)| \leq |E(G)|$. To be able to rely on dynamic programming later, we need to define a notion of a partial solution:

▶ **Definition 6.** Let P be a directed path in G. Given a vertex $v \in V(G)$, $s \in [r]$ and $b \in [r] \cup \{0\}$, we say that P is a (v, s, b)*-path* if the last vertex of P is v, $|V(P)| = s$ and $|\{e \in E(P) : c(e) = B\}| = b$. If for all $u, w \in V(P)$, it holds that $\ell(u) \neq \ell(v)$, then we say that P is a *good path.*

To employ the method of narrow sieves, we need to associate labels with entities whose uniqueness should be preserved. For this purpose, we have the following definition:

▶ **Definition 7.** Let P be a (v,s,b)-path. Given $f : V(P) \to [r]$, we say that (P,f) is a *(v,s,b)-pair*. If P is good, then we say that (P,f) is a *good pair*, and if f is an injective function, then we say that (P,f) is an *injective pair*. Given $L \subseteq [r]$ such that the image of f is a subset of L, we say that (P,f) is an *L-labeled pair*.

Now, we define two central sets of labeled partial solutions. The first one, $\mathcal{P}$, consists of every pair (P,f) that is an injective (v,s,b)-pair for some $v \in V(G)$ and $s,b \in [r]$ such that $b \geq k$. The second one, $\mathcal{Q}$, consists of every good pair (P,f) in $\mathcal{P}$. Note that for every pair $(P,f) \in \mathcal{Q}$, it holds that P is a solution for Long Blue Path, and for every solution P for Long Blue Path, by letting f be a function that assigns i to the ith vertex on P, we obtain a pair $(P,f) \in \mathcal{Q}$. Thus, we have the following observation.

▶ **Observation 2.** *The instance (G,c,ℓ,k,r) is a yes-instance if and only if $\mathcal{Q} \neq \emptyset$.*

With these definitions at hand, we may describe the rough idea of the approach. We represent all labeled partial solutions of $\mathcal{P}$ by a polynomial in such a way that each labeled partial solution corresponds to one monomial. We will ensure that the partial solutions of $\mathcal{P} \setminus \mathcal{Q}$ cancel each other out which will imply that the polynomial is not identically 0 if and only if $\mathcal{Q} \neq \emptyset$. To this end, we now describe how we represent labeled partial solutions by monomials. For every label $i \in \text{image}(\ell)$ and integer $j \in [r]$, we introduce the variable $x_{i,j}$, and for every edge $e \in E(G)$, we introduce the variable y_e. This gives the following representation:

▶ **Definition 8.** Let (P,f) be a (v,s,b)-pair. Then, the monomial associated with (P,f) is defined as follows.

$$\mathsf{mon}(P,f) = \prod_{v \in V(P)} x_{\ell(v),f(v)} \cdot \prod_{e \in E(P)} y_e.$$

Accordingly, we define the following polynomial (which would be evaluated over a field of characteristic 2).

▶ **Definition 9.** $\mathsf{POL} = \sum_{(P,f) \in \mathcal{P}} \mathsf{mon}(P,f)$.

To analyze this polynomial, we first observe that given a monomial associated with a pair $(P,f) \in \mathcal{Q}$, we can uniquely recover the pair (P,f). To see this, consider some monomial M that is associated with a pair $(P,f) \in \mathcal{Q}$. Then, the variables y_e of M specify exactly which edges are used by P, and therefore the path P is recovered. Now, since the pair (P,f) belongs to $\mathcal{Q}$, we have that P is a good path. Hence, the variables $x_{i,j}$ of M specify exactly how f labels the vertices of P. In other words, we have the following observation.

▶ **Observation 3.** *For all $(P,f) \in \mathcal{Q}$, there does not exist $(P',f') \in \mathcal{P} \setminus \{(P,f)\}$ such that* $\mathsf{mon}(P,f) = \mathsf{mon}(P',f')$.

The following lemma will be used to show that the partial solutions of $\mathcal{P} \setminus \mathcal{Q}$ cancel each other out.

▶ **Lemma 10.** *There exists a function $g : \mathcal{P} \setminus \mathcal{Q} \to \mathcal{P} \setminus \mathcal{Q}$ such that for all $(P,f) \in \mathcal{P} \setminus \mathcal{Q}$, it holds that* $\mathsf{mon}(P,f) = \mathsf{mon}(g(P,f))$*, $g(P,f) \neq (P,f)$, and $g(g(P,f)) = (P,f)$.*

Proof. Let $<$ be some order on $\{\{u,v\} : u,v \in V(P)\}$. Given $(P,f) \in \mathcal{P} \setminus \mathcal{Q}$, define $\mathsf{rep}(P,f) = \{\{u,v\} : u,v \in V(P), u \neq v, \ell(u) = \ell(v)\}$. Since P is not a good path, it holds that $\mathsf{rep}(P,f) \neq \emptyset$. Hence, it is well defined to let $\{u,v\}$ be the smallest set in $\mathsf{rep}(P,f)$ according to $<$. We let h be defined as f except that $h(u) = f(v)$ and $h(v) = f(u)$. Now, we set $g(P,f) = (P, h_{P,f})$. Clearly, $g(P,f) \in \mathcal{P}$. Note that $\mathsf{rep}(P,f) = \mathsf{rep}(P, h_{P,f})$, and hence $g(P,f) \notin \mathcal{Q}$ and $g(g(P,f)) = (P,f)$. Since $(P,f) \in \mathcal{P}$, it holds that f is an injective function; therefore $f(v) \neq f(u)$, which implies that $g(P,f) \neq (P,f)$. Finally, since $\ell(u) = \ell(v)$, it holds that $\mathsf{mon}(P,f) = \mathsf{mon}(g(P,f))$. ◀

Let $\mathbb{F}$ be a field of characteristic 2 (to be determined). From now on, we suppose that POL is evaluated over $\mathbb{F}$. Notice that

$$\mathsf{POL} = \sum_{(P,f)\in\mathcal{Q}} \mathsf{mon}(P,f) + \sum_{(P,f)\in\mathcal{P}\setminus\mathcal{Q}} \mathsf{mon}(P,f).$$

Suppose that POL is evaluated over $\mathbb{F}$. By Lemma 10, we have that $\mathsf{POL} = \sum_{(P,f)\in\mathcal{Q}} \mathsf{mon}(P,f)$. Then, by Observation 3, we have that POL is not identically 0 if and only if $\mathcal{Q}$ is not empty. Hence, by Observation 2, we have the following lemma.

▶ **Lemma 11.** *The instance (G,c,ℓ,k,r) is a yes-instance if and only if* POL *is not identically 0.*

In light of Lemma 11, our task is to determine whether POL is identically 0. For this purpose, we need the following notation. Given $v \in V(G)$, $s \in [r]$, $b \in [r] \cup \{0\}$ and $L \subseteq [r]$, let $\mathcal{P}_{v,s,b,L}$ denote the set of every L-labeled (v,s,b)-pair (P,f), and $\mathsf{POL}_{v,s,b,L} = \sum_{(P,f)\in\mathcal{P}_{v,s,b,L}} \mathsf{mon}(P,f)$. Moreover, denote

$$\mathcal{P}_L = \bigcup_{v\in V(G),\, s,b\in[r],\, b\geq k} \mathcal{P}_{v,s,b,L},$$

and $\mathsf{POL}_L = \sum_{(P,f)\in\mathcal{P}_L} \mathsf{mon}(P,f)$. By the principle of inclusion-exclusion, we have that $\mathsf{POL} = \sum_{L\subseteq[r]} (-1)^{r-|L|}\mathsf{POL}_L$. Then, since $\mathbb{F}$ is a field of characteristic 2 (refer to [4] for further details) we obtain the following.

▶ **Observation 4.** $\mathsf{POL} = \sum_{L\subseteq[r]} \mathsf{POL}_L$.

Hence, to determine whether POL is identically 0, it is sufficient to determine whether $\sum_{L\subseteq[r]} \mathsf{POL}_L$ is identically 0. To proceed, we need to recall the following well-known lemma.

▶ **Lemma 12** ([24, 28, 16])**.** *Let $p(x_1, x_2, \dots, x_n)$ be a nonzero polynomial of total degree at most d over a finite field $\mathbb{K}$. Then, for $a_1, a_2, \dots, a_n \in \mathbb{K}$ selected independently and uniformly at random, $Pr(p(a_1, a_2, \dots, a_n) \neq 0) \geq 1 - d/|\mathbb{K}|$.*

Notice that POL is a polynomial of total degree at most $2r$. Therefore, by setting $|\mathbb{F}| = 2^{\lceil \log(20r)\rceil}$, from Lemma 11, Observation 4, and Lemma 12, we have that

▶ **Lemma 13.** *For a random assignment to all variables $x_{i,j}$ and y_e, if (G,c,ℓ,k,r) is a no-instance, then $\sum_{L\subseteq[r]} \mathsf{POL}_L$ evaluates to 0, and otherwise it does* not *evaluate to a 0 with probability at least* 9/10.

In light of Lemma 13, to conclude that Lemma 4 is correct, it is sufficient to prove the following result.

▶ **Lemma 14.** *Given $L \subseteq [r]$ and an assignment to all variables $x_{i,j}$ and y_e, the polynomial* POL_L *can be evaluated in time $r^{O(1)} \cdot |E(G)|$.*

Finally, we would like to remark that if one is interested in finding a mapping that is a solution rather than just determining whether such a mapping exists, this goal can be achieved by standard means of self-reduction. Briefly, if k is not positive, then we are done. Else, if the algorithm determines that there exists a solution, then we may "guess" (i.e., perform exhaustive search) a longest substring A' of A that is mapped by some solution while preserving all duos in A' as well as the substring B' of B to which it is mapped. If our guess is correct, then the symbol preceding A' in A is not equal to the symbol preceding B' in B and the symbol after A' in A is also not equal to the symbol after B' in B (if such symbols exist). Then, we may replace A' and B' in A and B, respectively, by some new symbol, decrease k by $|A'| - 1$, and call the algorithm recursively. Notice that the length of A' should be at least 2, and hence the size of the input has decreased.

4 Deterministic Algorithm: Representative Sets

In this section, we adapt the approach in which the method of *representative sets* is applied to solve the k-Path problem [20]. More precisely, our objective is to provide a constructive proof for the following result.

▶ **Lemma 15.** *There exists a deterministic algorithm that solves* Long Blue Path *in time $O((\frac{1+\sqrt{5}}{2})^{r+o(r)} \cdot |E(G)| \cdot \log |E(G)|)$.*

Combining Lemma 3 and 15 gives us the following.

▶ **Theorem 16.** *There exists a deterministic algorithm that solves* Max-Duo PSM *in time $O((\frac{1+\sqrt{5}}{2})^{2k+o(k)} \cdot n^4 \log n) = O(6.855^k \cdot n^4 \log n)$.*

5 A Cubic Problem Kernel

In this section we will show that Max-Duo PSM admits a kernel of size $O(k^3)$. Let (A, B, k) be an instance of Max-Duo PSM, and let $S \in \{A, B\}$. If $S = A$, then we let $\overline{S} = B$. Analogously, if $S = B$, then we let $\overline{S} = A$.

Let m be a map of S into $\overline{S}$, and let D be a set of duos. We denote by $m(D) = \{(m(i), m(i+1)) \mid (i, i+1) \in D\}$ the image of D under m. We say that m *preserves* D if m preserves each duo in D. Let C_A and C_B be sets of duos. We say that the pair (C_A, C_B) is *complete* for (A, B, k) if whenever there is a map m of A into B that preserves k duos, then there is a subset $D \subseteq C_A$ with $|D| = k$ and a map m' such that m' preserves D and $m'(D) \subseteq C_B$. The size of (C_A, C_B) is defined as $|C_A| + |C_B|$. Let $f : \mathbb{N} \to \mathbb{N}$ be a function. A complete pair (C_A, C_B) of size $f(k)$ for (A, B, k) can be used to construct a kernel (A', B', k) of size $O(f(k))$ for (A, B, k).

▶ **Theorem 17** ([2, Section 4.2])**.** *Let (C_A, C_B) be a complete pair of size $f(k)$ for (A, B, k). Then one can construct in time $O(f(k))$ related strings A' and B', each of size $O(f(k))$ such that (A, B, k) is a yes-instance of* Max-Duo PSM *if and only if (A', B', k) is a yes-instance of* Max-Duo PSM.

Using Theorem 17, it is sufficient to show that one can obtain in polynomial time a complete pair (C_A, C_B) for (A, B, k) of size $O(k^3)$.

A *block* of size s is a set $X = \{(i, i+1), (i+1, i+2), ..., (i+s-1, i+s)\}$ consisting of s consecutive duos. We say that $(i, i+1)$ is the *root* of X. If S is a string of length at least $i+s$, then we let $str(S, X) = S[i, i+s]$ be the substring of S corresponding to the positions that occur in X. The following observation is immediate.

▶ **Observation 5.** *Let* (A, B, k) *be an instance of* Max-Duo PSM *and let* m *be a map of* A *into* B *that preserves a block* X *of size* k*. Then* (A, B, k) *is a yes-instance of* Max-Duo PSM*. Additionally, the instance* (A', B', k) *where* $A' = str(A, X)$ *and* $B' = str(B, m(X))$ *is also yes-instance of* Max-Duo PSM*.*

In the remainder of this section we assume that no map m of A into B preserves a block of size k. Our algorithm is based on the notion of *rare* duo, which we define next. For each two symbols $a, b \in \Sigma$, and each string $S \in \{A, B\}$, we let

$$n(S, a, b) := |\{i : 1 \leq i \leq |S| - 1,\ S[i, i+1] = ab\}|$$

be the number of occurrences of the length-two string ab as a substring of S. We say that a length-two string ab is *rare for* S if ab occurs as a sub-string of both S and $\overline{S}$ and $n(S, a, b) \leq n(\overline{S}, a, b)$. Observe that if ab occurs as many times in S as it occurs in $\overline{S}$, then ab is rare for both S and $\overline{S}$. We say that a duo $(i, i+1)$ is rare for S if $S[i, i+1]$ is rare for S. We let $rare(S)$ be the set of duos that are rare for S.

▶ **Lemma 18.** *If either* $|rare(A)| \geq 4k$ *or* $|rare(B)| \geq 4k$*, then* (A, B, k) *is a yes-instance.*

Proof. The duo graph associated with A and B is the bipartite graph $G(A, B) = (V_A \dot{\cup} V_B, E)$ defined as follows:

$$\begin{aligned} V_A &= \{(i, i+1) \mid 1 \leq i \leq n-1\}, \\ V_B &= \{(j, j+1) \mid 1 \leq j \leq n-1\}, \\ E &= \{[(i, i+1), (j, j+1)] \mid A[i] = B[j],\ A[i+1] = B[j+1]\}. \end{aligned}$$

Intuitively, each of the sets V_A and V_B contains all pairs of consecutive positions from $[n]$. A duo $(i, i+1)$ in V_A is connected to a duo $(j, j+1)$ in V_B if and only if the length-two string $A[i]A[i+1]$ is equal to $B[j]B[j+1]$.

If $e = [(i, i+1), (j, j+1)]$ is an edge of $G(A, B)$, then we say that $(i, i+1)$ is the left endpoint of e and $(j, j+1)$ is the right endpoint of e. If M is a matching in $G(A, B)$, then we let M_A be the set of duos in V_A that are left endpoints of edges in M, and M_B be the set of duos in V_B that are right endpoints of edges in M.

Assume that either $|rare(A)| \geq 4k$ or $|rare(B)| \geq 4k$. Then G contains a matching of size at least $4k$. Let $\mathbf{M}$ be a maximum matching in $G(A, B)$.

It has been shown in [6] that given a matching $\mathbf{M}$ of size at least $4k$ for the graph $G(A, B)$, one can construct a sub-matching M of $\mathbf{M}$ of size at least k such that M directly gives a map preserving at least k duos. Therefore, the instance is a yes-instance in this case. ◀

In the remainder of this section we thus assume that there are less than $4k$ duos that are rare for A, and less than $4k$ duos that are rare for B. This implies that we may add all rare duos to the sets C_A and C_B without surpassing the desired size bound of $O(k^3)$.

Let S be a string in $\{A, B\}$. We say that a duo $(j, j+1)$ is a *match for a duo* $(i, i+1)$ in $\overline{S}$ if there exists a map m of S into $\overline{S}$ that preserves $(i, i+1)$, and $(m(i), m(i+1)) = (j, j+1)$. If X and Y are blocks, then we say that Y is a *match* for X in $\overline{S}$ if there exists a map m of S into $\overline{S}$ such that m preserves X, and $m(X) = Y$.

Algorithm 1

```
procedure ROOTS($S, i, i+1$)
    $R = \emptyset$
    $k' \leftarrow$ size of the maximal block which is rooted at $(i, i+1)$, rare for $S$, and has a
            match in $\overline{S}$. Note that $k' \leq k-1$.
    for $\ell = k'$ to 1 do
        $X \leftarrow$ unique block of size $\ell$ rooted at $(i, i+1)$
        for $j = 1$ to $n-1$ do
            if $|R| < 2k-1$ and $|j' - j| > k \; \forall j' \in R$ and
              $(j, j+1)$ is a root for a match of $X$ in $\overline{S}$ then
                $R \leftarrow R \cup \{(j, j+1)\}$
    output $R$
```

▶ **Observation 6.** *Let $S \in \{A, B\}$ and let $(j, j+1)$ be a match for $(i, i+1)$ in $\overline{S}$. Then if $(i, i+1)$ is not rare for S, $(j, j+1)$ is rare for $\overline{S}$.*

Proof. Since $(j, j+1)$ is a match for $(i, i+1)$ in $\overline{S}$, there is some length-two string ab such that $S[i]S[i+1] = \overline{S}[j]\overline{S}[j+1] = ab$. Since $(i, i+1)$ is not rare for S, the string ab occurs strictly more often in S than it occurs in $\overline{S}$. In other words, $n(S, a, b) > n(\overline{S}, a, b)$. This implies that $(j, j+1)$ is rare for $\overline{S}$. ◀

This observation is useful because it tells us that for each match in a map, one of the two duos is rare, so by adding all the rare duos to C_A and C_B, we essentially pick up one half of each match. We now consider two types of matched blocks that may occur in the solution. First, there may be pairs of matched blocks X and Y that both contain nonrare duos. We can add all duos of these blocks by considering a sufficiently large neighborhood of all rare duos. To this end, for each $i \in \{1, ..., n-1\}$, let

$$\mathcal{B}_k(i) = \{(i', i'+1) \mid i' \in \{1, ..., n-1\},\ i-k \leq i' \leq i+k\}$$

denote the *ball of radius* k around the duo $(i, i+1)$

The following lemma essentially implies that by adding the ball of radius k around each rare duo, we add all pairs of matched blocks that both contain at least one nonrare duo.

▶ **Lemma 19.** *Let $S \in \{A, B\}$, X be a block of size at most $k-1$ containing a duo $(i, i+1)$ that is not rare for S, and let m be a map of S into $\overline{S}$ such that X is preserved by m. Then $(m(i), m(i+1))$ is rare for $\overline{S}$ and $m(X) \subseteq \mathcal{B}_k(m(i))$.*

Proof. Since $(i, i+1)$ is preserved by m, $(m(i), m(i+1))$ is a match for $(i, i+1)$ in $\overline{S}$. Since $(i, i+1)$ is not rare for S, by Observation 6, $(m(i), m(i+1))$ is rare for $\overline{S}$. Since m preserves X and since $|X| \leq k-1$, $m(X)$ is a block of size at most $k-1$. Therefore, all duos in $m(X)$ must be in the ball of radius k around $(m(i), m(i+1))$, that is, $m(X) \subseteq \mathcal{B}_k(m(i))$. ◀

We now turn to the second type of matched pairs of blocks, those where one block X of S has only rare duos for S; we call such a block X *rare*. Since X is rare, it is rooted at some rare duo $(i, i+1)$. To obtain the complete set, we need to add duos in $\overline{S}$. This is done by the procedure ROOTS which receives as input a string S and a duo in S and returns a set of duos ROOTS$(S, i, i+1)$.

Intuitively, for each block X that is rare for S and rooted at $(i, i+1)$, the set ROOTS$(S, i, i+1)$ contains a selection of roots of matches for X in the string $\overline{S}$. This selection is made according to two criteria. First, roots of matches for larger blocks are added first. Second, the roots in $\textsc{Roots}(S, i, i+1)$ are sufficiently far apart from each other. Now consider the set

$$F(S, i, i+1) = \bigcup_{(j,j+1) \in \textsc{Roots}(S,i,i+1)} \mathcal{B}_k(j).$$

Intuitively, $F(S, i, i+1)$ consists of all duos that are sufficiently close to duos in $\textsc{Roots}(S, i, i+1)$. The next lemma states that if some map m of S into $\overline{S}$ preserves some block X that is rooted at $(i, i+1)$ and rare for S, then this map can be transformed into a map m' that preserves X, that sends X to $F(S, i, i+1)$, and that is equal to m on every duo outside X.

▶ **Lemma 20.** *Let m be a map of S into $\overline{S}$, D be a set of duos such that $|D| = k$, and $X \subseteq D$ be a block that is rooted at $(i, i+1)$, that is rare for S and that is preserved by m. Then there is a map m' of S into $\overline{S}$ such that X is preserved by m', such that $m'(X) \subseteq F(S, i, i+1)$ and such that $(m'(i'), m'(i'+1)) = (m(i'), m(i'+1))$ for each $(i', i'+1) \in D \backslash X$.*

Proof. Let $(j, j+1)$ be the root of $m(X)$ in $\overline{S}$. Let $n(S, X)$ be the number of duos in $\textsc{Roots}(S, i, i+1)$ that are roots of matches for X in $\overline{S}$. Suppose that $n(S, X) < 2k - 1$. Then either $(j, j+1) \in \textsc{Roots}(S, i, i+1)$ or $(j, j+1)$ does not belong to $\textsc{Roots}(S, i, i+1)$ and there exists some duo $(j', j'+1) \in \textsc{Roots}(S, i, i+1)$ with $|j' - j| < k$. Note that if this were not the case, the duo $(j, j+1)$ would have been added to $\textsc{Roots}(S, i, i+1)$, since all three conditions of the 'If' instruction of Algorithm 1 would have been satisfied. In any case, $m(X) \subseteq \mathcal{B}_k(j') \subseteq F(S, i, i+1)$. Therefore, if $n(S, X) < 2k - 1$, we may simply set $m' = m$.

Now assume that $n(S, X) = 2k - 1$. Note that for each duo $(j, j+1)$ there are no three distinct j_1, j_2 and j_3 such that $(j_l, j_l + 1) \in \textsc{Roots}(S, i, i+1)$ and $(j, j+1) \in \mathcal{B}(j_l)$ for $l \in \{1, 2, 3\}$. In other words $(j, j+1)$ can intersect at most two balls of radius k rooted at duos in $\textsc{Roots}(S, i, i+1)$. Therefore, since $|D \backslash X| \leq k - 1$, the set $D \backslash X$ intersects at most $2k - 2$ balls of radius r rooted at duos in $\textsc{Roots}(S, i, i+1)$. In other words, there is at least one $(j', j'+1) \in \textsc{Roots}(S, i, i+1)$ that is the root of a match for X in S and such that $\mathcal{B}_k(j') \cap (D \backslash X) = \emptyset$. Therefore, we may set m' as the map of S into $\overline{S}$ that preserves X, that sends the root of X to $(j', j'+1)$, and that is equal to m on every duo $(i', i'+1) \in D \backslash X$. ◀

Now, for each $S \in \{A, B\}$, consider the following set C_S of duos.

$$C_S = \left[\bigcup_{(i,i+1) \in rare(S)} \mathcal{B}_k(i) \right] \cup \left[\bigcup_{(i,i+1) \in rare(\overline{S})} F(\overline{S}, i, i+1) \right]. \quad (1)$$

In other words, for each duo $(i, i+1)$ that is rare for S, C_S contains all duos in the ball of radius k around $(i, i+1)$. Moreover, for each duo $(i, i+1)$ that is rare for $\overline{S}$, C_S contains all duos in the set $F(\overline{S}, i, i+1)$. The following lemma states that if a map m of S into $\overline{S}$ preserves a set D containing k duos, then there exists a map m' that also preserves D in such a way that $m'(D) \subseteq C_{\overline{S}}$.

▶ **Lemma 21.** *Let D be a set of duos such that $|D| = k$. Let m be a map of S into $\overline{S}$ that preserves all duos in D. Then there is a map m' of S into $\overline{S}$ that preserves all duos in D, and such that $m'(D) \subseteq C_{\overline{S}}$.*

Proof. Let $X_1, ..., X_r$ be the set of rare blocks that are contained in D and that are maximal with respect to set inclusion. In other words, for each $j \in \{1, ..., r\}$ and each Y such that $X_j \subseteq Y \subseteq D$, we have that Y is not a rare block. Note that since these blocks are rare and maximal, they are pairwise disjoint, i.e., $X_j \cap X_{j'} = \emptyset$ for $j \neq j'$. For each $j \in \{1, ..., r\}$ let $(i_j, i_j + 1)$ be the root of X_j and $D_j = D \backslash X_j$. Additionally, let $D' = D \backslash \bigcup_{j=1}^{r} X_j$. Note that $D' \subseteq D_j$ for each $j \in \{1, ..., r\}$.

Let $m_0, m_1, ..., m_r$ be maps of S into $\overline{S}$ defined inductively as follows. First, we set $m_0 = m$. Now, for each $j \in \{1, ..., r\}$, we let m_j be a map of S into $\overline{S}$ constructed according to Lemma 20. More precisely, m_j preserves X_j, $m_j(X_j) \subseteq F(S, i_j, i_j + 1)$, and $(m_j(i), m_j(i+1)) = (m_{j-1}(i), m_{j-1}(i+1))$ for each duo $(i, i+1) \in D_j = D \backslash X_j$.

Using the maps $m_0, m_1, ..., m_r$ defined above, it follows by induction on j that for each $l \in \{1, ..., j\}$, $m_j(X_l) \subseteq F(S, i_l, i_l + 1) \subseteq C_{\overline{S}}$ and $(m_j(i), m_j(i+1)) = (m(i), m(i+1))$ for each $(i, i+1) \in D'$. In particular, for each $l \in \{1, ..., r\}$, $m_r(X_l) \subseteq F(S, i_l, i_l + 1) \subseteq C_{\overline{S}}$ and $(m_r(i), m_r(i+1)) = (m(i), m(i+1))$ for each $(i, i+1) \in D' \subseteq D_j$. This shows, that m_r preserves D, and sends $\bigcup_{j=1}^{r} X_j$ to a subset of $C_{\overline{S}}$ and agrees with m in every duo in $D' = D \backslash \bigcup_{j=1}^{r} X_j$.

Let $m' = m_r$. It remains to show that m' also sends blocks that are not rare for S to subsets of $C_{\overline{S}}$. Let $X'_1, ..., X'_s$ be the maximal blocks that are contained in D and that are not rare for S. Note that these blocks are indeed contained in D' and form a partition of D'. Since for each $j \in \{1, ..., s\}$, X'_j has at least one duo $(i, i+1)$ that is not rare for S, Lemma 19 implies that $(m'(i), m'(i))$ is rare for $\overline{S}$ and that $m'(X_j) \subseteq \mathcal{B}_k(m'(i)) \subseteq C_{\overline{S}}$. Since $X'_1, ..., X'_s$ forms a partition of D', $m'(D') \subseteq C_{\overline{S}}$. Since by the discussion above, $m'(\bigcup_{j=1}^{r} X_j)$ is also a subset of $C_{\overline{S}}$, we have that $m'(D) \subseteq C_{\overline{S}}$. ◀

Let C_A and C_B be sets of duos constructed according to Equation 1. We can show that (C_A, C_B) is complete for (A, B, k) by applying Lemma 21 twice. More precisely, once with respect to maps of A into B, and once with respect to maps of B into A.

▶ **Lemma 22.** *The pair (C_A, C_B) is complete for (A, B, k).*

Proof. Let D_1 be a set of duos of size k. Let m_1 be a map of A into B which preserves all duos in D_1. Then by Lemma 21 there is a map m_2 of A into B which also preserves all duos in D_1, but with the property that $m_2(D_1) \subseteq C_B$. Now let $D_2 = m_2(D_1)$, and $m_3 = m_2^{-1}$ be the inverse of m_2. In other words, m_3 is a map of B into A such that for each $i \in [n]$, $m_2(i) = j$ if and only if $m_3(j) = i$. Then m_3 preserves all duos in D_2. By Lemma 21 there is a map m_4 of B into A that also preserves all duos in D_2 but with the additional property that $m_4(D_2) \subseteq C_A$.

Let $D_3 = m_4(D_2)$, and let $m_5 = m_4^{-1}$ be the inverse of m_4. Then m_5 is a map of A into B that preserves $D_3 \subseteq C_A$ and such that $m_5(D_3) = D_2 \subseteq C_B$. Since $|D_3| = |D_2| = k$, the pair (C_A, C_B) is complete for (A, B, k). ◀

Now, we can upper-bound the size of C_S and the time needed to construct C_S, thus arriving at our main theorem.

▶ **Theorem 23.** *Given an instance $I = (A, B, k)$ of* Max-Duo PSM*, one can construct in time $O(|\Sigma|^2 \cdot n + k^3 \cdot n)$ an instance $I' = (A', B', k)$ of* Max-Duo PSM *with $|A'|$ and $|B'|$ bounded by $O(k^3)$ such that I is a yes-instance if and only if I' is a yes-instance.*

References

1 Stefano Beretta, Mauro Castelli, and Riccardo Dondi. Corrigendum to "Parameterized tractability of the maximum-duo preservation string mapping problem" [Theoret. Comput. Sci. 646 (2016) 16–25]. *Theor. Comput. Sci.*, 653:108–110, 2016. doi:10.1016/j.tcs.2016.09.015.

2 Stefano Beretta, Mauro Castelli, and Riccardo Dondi. Parameterized tractability of the maximum-duo preservation string mapping problem. *Theor. Comput. Sci.*, 646:16–25, 2016. doi:10.1016/j.tcs.2016.07.011.

3 Andreas Björklund. Determinant sums for undirected Hamiltonicity. *SIAM J. Comput.*, 43(1):280–299, 2014. doi:10.1137/110839229.

4 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Narrow sieves for parameterized paths and packings. *J. Comput. Syst. Sci.*, 87:119–139, 2017. arXiv:1007.1161, doi:10.1016/J.JCSS.2017.03.003.

5 Nicolas Boria, Gianpiero Cabodi, Paolo Camurati, Marco Palena, Paolo Pasini, and Stefano Quer. A 7/2-approximation algorithm for the maximum duo-preservation string mapping problem. In Roberto Grossi and Moshe Lewenstein, editors, *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *LIPIcs*, pages 11:1–11:8. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.11.

6 Nicolas Boria, Adam Kurpisz, Samuli Leppänen, and Monaldo Mastrolilli. Improved approximation for the maximum duo-preservation string mapping problem. In Daniel G. Brown and Burkhard Morgenstern, editors, *Proceedings of the 14th International Workshop on Algorithms in Bioinformatics (WABI 2014)*, volume 8701 of *LNCS*, pages 14–25. Springer, 2014. doi:10.1007/978-3-662-44753-6_2.

7 Brian Brubach. Further improvement in approximating the maximum duo-preservation string mapping problem. In Martin C. Frith and Christian Nørgaard Storm Pedersen, editors, *Proceedings of the 16th International Workshop on Algorithms in Bioinformatics (WABI 2016)*, volume 9838 of *LNCS*, pages 52–64. Springer, 2016. doi:10.1007/978-3-319-43681-4_5.

8 Laurent Bulteau, Guillaume Fertin, Christian Komusiewicz, and Irena Rusu. A fixed-parameter algorithm for minimum common string partition with few duplications. In Aaron E. Darling and Jens Stoye, editors, *Proceedings of the 13th International Workshop on Algorithms in Bioinformatics (WABI 2013)*, volume 8126 of *LNCS*, pages 244–258. Springer, 2013. doi:10.1007/978-3-642-40453-5_19.

9 Laurent Bulteau and Christian Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In Chandra Chekuri, editor, *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, pages 102–121. SIAM, 2014. doi:10.1137/1.9781611973402.8.

10 Wenbin Chen, Zhengzhang Chen, Nagiza F. Samatova, Lingxi Peng, Jianxiong Wang, and Maobin Tang. Solving the maximum duo-preservation string mapping problem with linear programming. *Theor. Comput. Sci.*, 530:1–11, 2014. doi:10.1016/j.tcs.2014.02.017.

11 Xin Chen, Jie Zheng, Zheng Fu, Peng Nan, Yang Zhong, Stefano Lonardi, and Tao Jiang. Assignment of orthologous genes via genome rearrangement. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 2(4):302–315, 2005. doi:10.1109/TCBB.2005.48.

12 Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007. doi:10.1145/1219944.1219947.

13 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007. doi:10.1017/CBO9780511546853.

14 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.

15 Peter Damaschke. Minimum common string partition parameterized. In Keith A. Crandall and Jens Lagergren, editors, *Proceedings of the 8th International Workshop on Algorithms in Bioinformatics (WABI 2008)*, volume 5251 of *LNCS*, pages 87–98. Springer, 2008. doi:10.1007/978-3-540-87361-7_8.

16 Richard A. DeMillo and Richard J. Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7(4):193–195, 1978. doi:10.1016/0020-0190(78)90067-4.

17 Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013. doi:10.1007/978-1-4471-5559-1.

18 Bartłomiej Dudek, Paweł Gawrychowski, and Piotr Ostropolski-Nalewaja. A family of approximation algorithms for the maximum duo-preservation string mapping problem. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *LIPIcs*, pages 10:1–10:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. arXiv:1702.02405, doi:10.4230/LIPIcs.CPM.2017.10.

19 Guillaume Fertin, Anthony Labarre, Irena Rusu, Eric Tannier, and Stéphane Vialette. *Combinatorics of Genome Rearrangements*. Computational molecular biology. MIT Press, 2009. doi:10.7551/mitpress/9780262062824.001.0001.

20 Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, and Saket Saurabh. Efficient computation of representative families with applications in parameterized and exact algorithms. *J. ACM*, 63(4):29:1–29:60, 2016. doi:10.1145/2886094.

21 Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. *Electron. J. Comb.*, 12, 2005. URL: http://www.combinatorics.org/Volume_12/Abstracts/v12i1r50.html.

22 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

23 Haitao Jiang, Binhai Zhu, Daming Zhu, and Hong Zhu. Minimum common string partition revisited. *J. Comb. Optim.*, 23(4):519–527, 2012. doi:10.1007/s10878-010-9370-2.

24 Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980. doi:10.1145/322217.322225.

25 Hadas Shachnai and Meirav Zehavi. Representative families: A unified tradeoff-based approach. *J. Comput. Syst. Sci.*, 82(3):488–502, 2016. doi:10.1016/j.jcss.2015.11.008.

26 Krister M. Swenson, Mark Marron, Joel V. Earnest-DeYoung, and Bernard M. E. Moret. Approximating the true evolutionary distance between two genomes. *ACM J. Exp. Algorithmics*, 12, 2008. doi:10.1145/1227161.1402297.

27 Yao Xu, Yong Chen, Taibo Luo, and Guohui Lin. A local search 2.917-approximation algorithm for duo-preservation string mapping, 2017. arXiv:1702.01877.

28 Richard Zippel. Probabilistic algorithms for sparse polynomials. In Edward W. Ng, editor, *Proceedings of an International Symposiumon on Symbolic and Algebraic Manipulation (EUROSAM 1979)*, volume 72 of *LNCS*, pages 216–226. Springer, 1979. doi:10.1007/3-540-09519-5_73.

A Proofs of Section 3

▶ **Lemma 14.** *Given $L \subseteq [r]$ and an assignment to all variables $x_{i,j}$ and y_e, the polynomial* POL_L *can be evaluated in time $O(r^{O(1)} \cdot |E(G)|)$.*

Proof Sketch. The evaluation can be performed by a simple procedure based on dynamic programming. For the sake of completeness, we present the base cases and recursive formula below. For simplicity, we abuse notation by using the symbols $x_{i,j}$ and y_e to refer to the values assigned to the variables $x_{i,j}$ and y_e, respectively.

The procedure uses a table M, which has an entry $M[v, s, b]$ for all $v \in V(G)$, $s \in [r]$ and $b \in [r] \cup \{0\}$. The purpose of this entry is to store the evaluation of $\mathsf{POL}_{v,s,b,L}$. Then, the evaluation of POL_L is given by $\displaystyle\sum_{\substack{v \in V(G),\\ s,b \in [r], b \geq k}} M[v,s,b]$.

The basis consists of the following cases:

- If $b \geq s$, then $M[v,s,b] = 0$.
- Else if $s = 1$, then $M[v,s,b] = \displaystyle\sum_{i \in L} x_{\ell(v),i}$.

Now, consider an entry $M[v,s,b]$ not computed in the basis. We assume that a reference to an undefined entry returns 0. Then,

$$M[v,s,b] = \sum_{\substack{(u,v) \in E(G),\\ c(u,v) = R}} \left((\sum_{i \in L} x_{\ell(v),i}) \cdot y_{(u,v)} \cdot M[u,s-1,b] \right) + \sum_{\substack{(u,v) \in E(G),\\ c(u,v) = B}} \left((\sum_{i \in L} x_{\ell(v),i}) \cdot y_{(u,v)} \cdot M[u,s-1,b-1] \right). \blacktriangleleft$$

B Proofs of Section 4

In this section, we adapt the approach in which the method of *representative sets* is applied to solve the k-Path problem [20]. More precisely, our objective is to provide a constructive proof for the following result.

▶ **Lemma 15.** *There exists a deterministic algorithm that solves* Long Blue Path *in time* $O((\frac{1+\sqrt{5}}{2})^{r+o(r)} \cdot |E(G)| \log |E(G)|)$.

In light of Lemma 3, once we have Lemma 15 at hand, we directly obtain the following theorem.

▶ **Theorem 16.** *There exists a deterministic algorithm that solves* Max-Duo PSM *in time* $O((\frac{1+\sqrt{5}}{2})^{2k+o(k)} \cdot n^4 \log n) = O(6.855^k \cdot n^4 \log n)$.

Next, we focus on the proof of Lemma 15. To this end, let (G,c,ℓ,k,r) be an instance of Long Blue Path. Without loss of generality, we can assume that the image of ℓ is a subset of $[|V(G)|]$ and that $|V(G)| \leq |E(G)|$. Here, a *p-set* is a set of size p. To describe our algorithm, we need to present the definition of a representative family.

▶ **Definition 27** ([20]). Given a universe U and a family $\mathcal{S}$ of p-subsets of U, we say that a subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ *t-represents* $\mathcal{S}$ if for every pair of sets $X \in \mathcal{S}$, and $Y \subseteq U \setminus X$ of size $t-p$, there exists a set $\widehat{X} \in \widehat{\mathcal{S}}$ such that $\widehat{X} \cap Y = \emptyset$.

The papers [20] and [25] present an algorithm, to which we refer as RepAlg, that given a universe U and a family $\mathcal{S}$ of p-subsets of U, computes a subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ of size $S(|U|,t,p)$ that t-represents $\mathcal{S}$ in time $|\mathcal{S}| \cdot T(|U|,t,p)$, such that the following condition is satisfied:

$$\sum_{p=1}^{t} |U| \cdot S(|U|,t,p-1) \cdot T(|U|,t,p) = (\frac{1+\sqrt{5}}{2})^{t+o(t)} \cdot |U| \log |U|.$$

We proceed by presenting a procedure that is based on a combination of dynamic programming and calls to RepAlg. For this purpose, we use a table M that has an entry

$M[v,s,b]$ for all $v \in V(G)$, $s \in [r]$ and $b \in [r] \cup \{0\}$. Let $\mathcal{P}_{v,s,b}$ denote the set of all *good* (v,s,b)-paths (see Definition 6). Give a (v,s,b)-path, define $\ell(P) = \{\ell(v) : v \in V(P)\}$. Moreover, define $\mathcal{S}_{v,s,b} = \{\ell(P) : P \in \mathcal{P}_{v,s,b}\}$. The purpose of the entry $M[v,s,b]$ would be to store a subfamily of $\mathcal{S}_{v,s,b}$ that r-represents it. Next, we show how to compute the entries of M. Here, the calls to RepAlg correspond to the universe $[|E(G)|]$ and with $t = r$.

The basis consists of the following cases:

- If $s = 1$ but $b \neq 0$, then $M[v,s,b] = \emptyset$.
- Else if $s = 1$, then $M[v,s,b] = \{\{\ell(v)\}\}$.

Now, consider an entry $M[v,s,b]$ not computed in the basis. We assume that a reference to an undefined entry returns an empty set. Then, we first compute the two following families.

- $\mathcal{A}_{v,s,b} = \{X \cup \{\ell(v)\} : (u,v) \in E(G), c(u,v) = R, X \in M[u,s-1,b], \ell(v) \notin X\}$.
- $\mathcal{B}_{v,s,b} = \{X \cup \{\ell(v)\} : (u,v) \in E(G), c(u,v) = B, X \in M[u,s-1,b-1], \ell(v) \notin X\}$.

Accordingly, we compute $M[v,s,b]$ as follows.

$M[v,s,b] = \mathsf{RepAlg}(\mathcal{A}_{v,s,b} \cup \mathcal{B}_{v,s,b})$.

First, note that the entire computation can be performed in time

$$\begin{aligned} & O(\sum_{v \in V(G)} \sum_{s=1}^{r} \sum_{b=0}^{r} \sum_{(u,v) \in E(G)} S(|E(G)|, r, s) \cdot T(|E(G)|, r, s)) \\ = \quad & O(\sum_{s=1}^{r} r|E(G)| \cdot S(|E(G)|, r, s) \cdot T(|E(G)|, r, s)). \end{aligned}$$

Thus, we have the following observation.

▶ **Observation 7.** *The table M is computed in time $O((\frac{1+\sqrt{5}}{2})^{r+o(r)} \cdot |E(G)| \log |E(G)|)$.*

Next, we prove that the computation of M is correct.

▶ **Lemma 28.** *The computation of M ensures that for all $v \in V(G)$, $s \in [r]$ and $b \in [r] \cup \{0\}$, $M[v,s,b]$ r-represents $\mathcal{S}_{v,s,b}$.*

Proof. We prove the statement by induction on s. In the basis, where $s = 1$, it is clear that $M[v,s,b]$ is simply assigned $\mathcal{S}_{v,s,b}$, and therefore it also 1-represents $\mathcal{S}_{v,s,b}$. Now, fix some $s \geq 2$, and suppose that the statement is correct for $s-1$. To prove that the statement is correct for s, choose some $v \in V(G)$, $b \in [r] \cup \{0\}$, $X \in \mathcal{S}_{v,s,b}$ and $Y \subseteq [|E(G)|] \setminus X$ such that $|Y| = r - s$. We need to show that there exists $\widehat{X} \in M[v,s,b]$ such that $\widehat{X} \cap Y = \emptyset$. Note that $M[v,s,b]$ r-represents $\mathcal{A}_{v,s,b} \cup \mathcal{B}_{v,s,b}$, and therefore is $\mathcal{A}_{v,s,b} \cup \mathcal{B}_{v,s,b}$ contains a set that is disjoint from Y, so does $M[v,s,b]$. Thus, it is sufficient that we show that there exists $\widehat{X} \in \mathcal{A}_{v,s,b} \cup \mathcal{B}_{v,s,b}$ such that $\widehat{X} \cap Y = \emptyset$.

Since $X \in \mathcal{S}_{v,s,b}$, there exists a good (v,s,b)-path P such that $\ell(P) = X$. Let u be the vertex on P that precedes v, and let Q be the path obtained by removing v from P. Note that $\ell(Q) = X \setminus \{\ell(v)\}$. Thus, if $c(u,v) = R$, then Q is a good $(u,s-1,b)$-path and therefore $X \setminus \{\ell(v)\} \in \mathcal{S}_{u,s-1,b}$, and otherwise Q is a good $(u,s-1,b-1)$-path and therefore $X \setminus \{\ell(v)\} \in \mathcal{S}_{u,s-1,b-1}$. First, let us assume that $X \setminus \{\ell(v)\} \in \mathcal{S}_{u,s-1,b}$. By the inductive hypothesis, $M[u,s-1,b]$ r-represents $\mathcal{S}_{u,s-1,b}$, and therefore $M[u,s-1,b]$ contains a set Z such that $Z \cap (Y \cup \{\ell(v)\}) = \emptyset$. Thus, $Z \cup \{\ell(v)\} \in \mathcal{A}_{v,s,b}$, and we conclude that the statement is correct. Now, let us assume that $X \setminus \{\ell(v)\} \in \mathcal{S}_{u,s-1,b-1}$. By the inductive hypothesis, $M[u,s-1,b-1]$ r-represents $\mathcal{S}_{u,s-1,b-1}$, and therefore $M[u,s-1,b-1]$ contains a set Z such that $Z \cap (Y \cup \{\ell(v)\}) = \emptyset$. Thus, $Z \cup \{\ell(v)\} \in \mathcal{B}_{v,s,b}$, and again we conclude that the statement is correct. ◀

With these lemmas at hand, we are ready to prove Lemma 15.

Proof. By Observation 7 and Lemma 28, we first compute M, ensuring that the condition in Lemma 28 is satisfied, in time $O((\frac{1+\sqrt{5}}{2})^{r+o(r)} \cdot |E(G)| \log |E(G)|)$. Then, we determine that the input instance is a yes-instance if and only if there exist $v \in V(G)$, $s \in [r]$ and $b \in [r]$ such that $b \geq k$ and $M[v, s, b] \neq \emptyset$. On the one hand, since for all $v \in V(G)$, $s \in [r]$ and $b \in [r]$, $M[v, s, b] \subseteq \mathcal{S}_{v,s,b}$, it is clear that if we accept, the input instance is indeed a yes-instance. On the other hand, if the input instance is a yes-instance, then there exist $v \in V(G)$, $s \in [r]$ and $b \in [r]$ such that $b \geq k$ and $\mathcal{S}_{v,s,b} \neq \emptyset$. Then, since $M[v, s, b]$ 0-represents $\mathcal{S}_{v,s,b}$, it holds that $M[v, s, b] \neq \emptyset$, and therefore we accept. ◀

C Proofs of Section 5

Proof of Theorem 23. We first show the running time to construct the kernel.

▶ **Proposition 29.** *For each $S \in \{A, B\}$, $|C_S| = O(k^3)$ and C_S can be constructed in time $O(|\Sigma|^2 \cdot n) + O(k^3 n)$.*

Proof. By assumption $|rare(S)| \leq 4k$. Additionally, for each i, the ball $\mathcal{B}_k(i)$ has size at most $2k + 1$. Finally, for each duo $(i, i + 1)$ that is rare for S, the set $F(\overline{S}, i, i + 1)$ has at most $(2k - 1)(2k + 1)$ duos. Therefore, $|C_S| \leq 4k(2k + 1) + 4k(2k - 1)(2k + 1) = O(k^3)$.

Now let us analyze the time to construct C_S. First, the construction of the sets $rare(S)$ and $rare(\overline{S})$ takes time $O(|\Sigma|^2 \cdot n)$, since we just need to count for each length-two string $ab \in \Sigma \times \Sigma$, the number of times $n(S, a, b)$ that ab occurs in S and the number of times $n(\overline{S}, a, b)$ that ab occurs in $\overline{S}$. Now, for each position $i \in \{1, ..., n - 1\}$, we add $(i, i + 1)$ to $rare(S)$ if $S[i]S[i + 1] = ab$ and $n(S, a, b) \leq n(\overline{S}, a, b)$. Analogously, we add $(i, i + 1)$ to $rare(\overline{S})$ if $\overline{S}[i]\overline{S}[i + 1] = ab$ and $n(\overline{S}, a, b) \leq n(S, a, b)$.

Now, the construction of the set $\text{Roots}(S, i, i + 1)$ according to Algorithm 1 takes time $O(k^2 \cdot n)$. Since $\text{Roots}(S, i, i + 1) \leq 2k - 1$, and by assumption $|rare(S)| \leq 4k$, the construction of $F(S, i, i + 1)$ also takes time $O(k^2 \cdot n)$. Analogously, the construction of $F(\overline{S}, i, i + 1)$ takes time $O(k^2 \cdot n)$. Therefore, the construction of C_S takes time at most $O(|\Sigma|^2 \cdot n) + O(k^3 \cdot n)$. ◀

▶ **Theorem 23.** *Given an instance $I = (A, B, k)$ of* Max-Duo PSM*, one can construct in time $O(|\Sigma|^2 \cdot n + k^3 \cdot n)$ an instance $I'(A', B', k)$ of* Max-Duo PSM *with $|A'|$ and $|B'|$ bounded by $O(k^3)$ such that I is a yes-instance if and only if I' is a yes-instance.*

Proof. First, if some map m of A into B preserves a block X of size k, then (A, B, k) is a yes-instance for Max-Duo PSM and we can output in $O(1)$ time an equivalent instance of constant size. Note that this condition can be verified in time $O(n)$ by solving the Longest Common Substring problem for A and B.

Second, if $rare(A) \geq 4k$ or $rare(B) \geq 4k$, then (A, B, k) is a yes-instance for Max-Duo PSM, and we can output in $O(1)$ time an equivalent instance of constant size.

Now since no map preserves a block of size k, and if both $rare(A) < 4k$ and $rare(B) < 4k$, then by Lemma 22, the pair (C_A, C_B) constructed according to Equation 1 is complete for (A, B, k). Additionally, by Proposition 29, $|C_A| = |C_B| = O(k^3)$, and both C_A and C_B can be constructed in time $O(|\Sigma| \cdot n + k^3 \cdot n)$.

Since the complete pair (C_A, C_B) constructed has size at most $O(k^3)$, we can apply Theorem 17 to construct in time $O(k^3)$ an instance (A', B', k) for Max-Duo PSM of size $O(k^3)$ such that (A', B', k) is a yes-instance if and only if (A, B, k) is a yes-instance. Therefore, the overall time to construct (A', B', k) is upper-bounded by $O(|\Sigma|^2 \cdot n + k^3 \cdot n)$. ◀

Clique-Based Lower Bounds for Parsing Tree-Adjoining Grammars

Karl Bringmann[1] **and Philip Wellnitz**[2]

1 **Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
`kbringma@mpi-inf.mpg.de`

2 **Saarland University, Saarland Informatics Campus, Saarbrücken, Germany**
`s8phwell@stud.uni-saarland.de`

Abstract

Tree-adjoining grammars are a generalization of context-free grammars that are well suited to model human languages and are thus popular in computational linguistics. In the tree-adjoining grammar recognition problem, given a grammar Γ and a string s of length n, the task is to decide whether s can be obtained from Γ. Rajasekaran and Yooseph's parser (JCSS'98) solves this problem in time $O(n^{2\omega})$, where $\omega < 2.373$ is the matrix multiplication exponent. The best algorithms avoiding fast matrix multiplication take time $O(n^6)$.

The first evidence for hardness was given by Satta (J. Comp. Linguist.'94): For a more general parsing problem, any algorithm that avoids fast matrix multiplication and is significantly faster than $O(|\Gamma|\, n^6)$ in the case of $|\Gamma| = \Theta(n^{12})$ would imply a breakthrough for Boolean matrix multiplication.

Following an approach by Abboud et al. (FOCS'15) for context-free grammar recognition, in this paper we resolve many of the disadvantages of the previous lower bound. We show that, even on constant-size grammars, any improvement on Rajasekaran and Yooseph's parser would imply a breakthrough for the k-Clique problem. This establishes tree-adjoining grammar parsing as a practically relevant problem with the unusual running time of $n^{2\omega}$, up to lower order factors.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems, F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases conditional lower bounds, k-Clique, parsing, tree-adjoining grammars

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.12

1 Introduction

Introduced in [14, 15], tree-adjoining grammars (TAGs) are a system to manipulate certain trees to arrive at strings, see Section 2 for a definition. TAGs are more powerful than context-free grammars, capturing various phenomena of human languages which require more formal power; in particular TAGs have an "extended domain of locality" as they allow "long-distance dependencies" [16]. These properties, and the fact that TAGs are efficiently parsable [29], make them highly desirable in the field of computer linguistics. This is illustrated by the large literature on variants of TAGs (see, e.g., [30, 21, 24, 9]), their formal language properties (see, e.g., [29, 16]), as well as practical applications (see, e.g., [25, 13, 26, 2]), including the XTAG project which developed a tree-adjoining grammar for the English language [10]. In fact, TAGs are so fundamental to computer linguistics that there is a biannual meeting called "International Workshop on Tree-Adjoining Grammars and Related Formalisms" [7], and they are part of the undergraduate curriculum (at least at Saarland University).

The prime algorithmic problem on TAGs is *parsing* (sometimes called recognition): Given a TAG Γ and a string s of length n, decide whether Γ can generate s. The first TAG parsers ran in time[1] $O(n^6)$ [29, 23], which was improved by Rajasekaran and Yooseph [20] to $O(n^{2\omega})$, where $\omega < 2.373$ is the exponent of (Boolean) matrix multiplication.

A limited explanation for the complexity of TAG parsing was given by Satta [22], who designed a reduction from Boolean matrix multiplication to TAG parsing, showing that any TAG parser running faster than $O(|\Gamma|\, n^6)$ on grammars of size $|\Gamma| = \Theta(n^{12})$ yields a Boolean matrix multiplication algorithm running faster than $O(n^3)$. This result has several shortcomings: (1) It holds only for a more general parsing problem, where we need to determine for each substring of the given string s whether it can be generated from Γ. (2) It gives a matching lower bound only in the unusual case of $|\Gamma| = \Theta(n^{12})$, so that it cannot exclude time, e.g., $O(|\Gamma|^2 n^4)$. (3) It gives matching bounds only restricted to *combinatorial* algorithms, i.e., algorithms that avoid fast matrix multiplication[2]. Thus, so far there is no satisfying explanation of the complexity of TAG parsing.

1.1 Context-free grammars

The classic problem of parsing context-free grammars, with important applications in programming languages, was in a very similar situation as tree-adjoining grammar parsing until very recently. Parsers in time $O(n^3)$ were known since the 60s [8, 31, 17, 11]. In a breakthrough, Valiant [27] improved this to $O(n^\omega)$. Finally, a reduction from Boolean matrix multiplication due to Lee [18] showed a matching lower bound for combinatorial algorithms for a more general parsing problem in the case that the grammar size is $\Theta(n^6)$.

Abboud et al. [1] gave the first satisfying explanation for the complexity of context-free parsing, by designing a reduction from the classic k-Clique problem, which asks whether there are k pairwise adjacent vertices in a given graph G. For this problem, for any fixed k the trivial running time of $O(n^k)$ can be improved to $O(n^{\omega k/3})$ for any k divisible by 3 [19] (see [12] for the case of k not divisible by 3). The fastest combinatorial algorithm runs in time $O(n^k/\log^k n)$ [28]. The *k-Clique hypothesis* states that both running times are essentially optimal, specifically that k-Clique has no $O(n^{(\omega/3-\varepsilon)k})$ algorithm and no combinatorial $O(n^{(1-\varepsilon)k})$ algorithm for any $k \geq 3, \varepsilon > 0$. The main result of Abboud et al. [1] is a reduction from the k-Clique problem to context-free grammar recognition on a specific, constant-size grammar Γ, showing that any $O(n^{\omega-\varepsilon})$ algorithm or any combinatorial $O(n^{3-\varepsilon})$ algorithm for context-free grammar recognition would break the k-Clique hypothesis, and thus improve decades-old algorithms. This matching conditional lower bound removes all disadvantages of Lee's lower bound at the cost of introducing a hypothesis, see [1] for further discussions.

1.2 Our contribution

We extend the approach by Abboud et al. to the more complex setting of TAGs. Specifically, we design a reduction from the $6k$-Clique problem to TAG recognition:

▶ **Theorem.** *There is a tree-adjoining grammar Γ of constant size such that if we can decide in time $T(n)$ whether a given string of length n can be generated from Γ, then $6k$-Clique can be solved in time $O\big(T(n^{k+1}\log n)\big)$, for any fixed $k \geq 1$. This reduction is combinatorial.*

[1] In most running time bounds we ignore the dependence on the grammar size, as we are mostly interested in constant-size grammars in this paper.
[2] There is no agreed upon formal definition of combinatorial algorithms.

Via this reduction, any $O(n^{2\omega-\varepsilon})$ algorithm for TAG recognition would prove that $6k$-Clique is in time $\tilde{O}(n^{(2\omega-\varepsilon)(k+1)}) = O(n^{(\omega/3-\varepsilon/9)6k})$, for sufficiently large[3] k. Furthermore, any combinatorial $O(n^{6-\varepsilon})$ algorithm for TAG recognition would yield a combinatorial algorithm for $6k$-Clique in time $\tilde{O}(n^{(6-\varepsilon)(k+1)}) = O(n^{(1-\varepsilon/9)6k})$, for sufficiently large k. As both implications would violate the $6k$-Clique conjecture, we obtain tight conditional lower bounds for TAG recognition. As our result (1) works directly for TAG recognition instead of a more general parsing problem, (2) holds for constant size grammars, and (3) does not need the restriction to combinatorial algorithms, it overcomes all shortcomings of the previous lower bound based on Boolean matrix multiplication, at the cost of using the well-established k-Clique hypothesis, which has also been used in [1, 5, 6, 3, 4].

We thus establish TAG parsing as a practically relevant problem with the quite unusual running time of $n^{2\omega}$, up to lower order factors. This is surprising, as the authors are aware of only one other problem with a (conjectured or conditional) optimal running time of $n^{2\omega\pm o(1)}$, namely 6-Clique.

1.3 Techniques

The essential difference of tree-adjoining and context-free grammars is that the former can grow strings at four positions, see Figure 3a. Writing a vertex v_1 in one position of the string, and writing the neighborhoods of vertices v_2, v_3, v_4 at other positions in the string, a simple tree-adjoining grammar can test whether v_1 is adjacent to v_2, v_3, and v_4. Extending this construction, for k-cliques C_1, C_2, C_3, C_4 we can test whether $C_1 \cup C_2, C_1 \cup C_3$, and $C_1 \cup C_4$ form $2k$-cliques. Using two permutations of this test, we ensure that $C_1 \cup C_2 \cup C_3 \cup C_4$ forms an almost-$4k$-clique, i.e., only the edges $C_3 \times C_4$ might be missing (in Figure 2b below this situation is depicted for cliques C_2, C_5, C_1, C_6 instead of C_1, C_2, C_3, C_4). Finally, we use that a $6k$-clique can be decomposed into 3 almost-$4k$-cliques, see Figure 2a.

In the constructed string we essentially just enumerate 6 times all k-cliques of the given graph G, as well as their neighborhoods, with appropriate padding symbols (see Section 3). We try to make the constructed tree-adjoining grammar as easily accessible as possible by defining a certain programming language realized by these grammars, and phrasing our grammar in this language, which yields subroutines with an intuitive meaning (see Section 4).

2 Preliminaries on tree-adjoining grammars

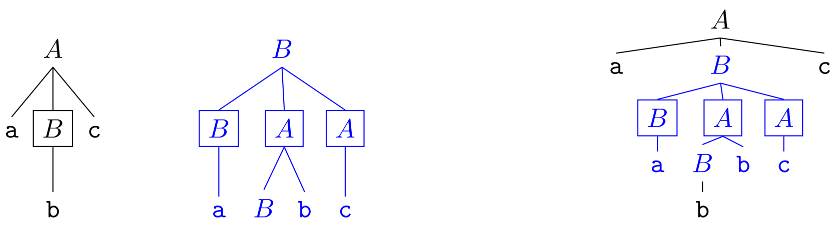
In this section we define tree-adjoining grammars and give examples. Fix a set T of terminals and a set N of non-terminals. In the following, conceptually we partition the nodes of any tree into its *leaves*, the *root*, and the remaining *inner nodes*. An *initial tree* is a rooted tree where

- the root and each inner node is labeled with a non-terminal,
- each leaf is labeled with a terminal, and
- each inner node can be marked for adjunction.

See Figure 1a for an example; nodes marked for adjunction are annotated by a rectangle. An *auxiliary tree* is a rooted tree where

- the root and each inner node is labeled with a non-terminal,
- *exactly one leaf, called the* foot node*, is labeled with the same non-terminal as the root,*
- each remaining leaf is labeled with a terminal, and
- each inner node can be marked for adjunction.

[3] For this and the next statement it suffices to set $k > 18/\varepsilon$.



(a) An initial tree (left) and an auxiliary tree (right); the internal nodes labeled A and B are marked for adjunction.

(b) Resulting tree after adjoining the auxiliary tree into the initial tree.

Figure 1 The basic building blocks and operation of tree-adjoining grammars.

Initial trees are the starting points for derivations of the tree-adjoining grammar. These trees are then extended by repeatedly replacing nodes marked for adjunction by auxiliary trees. Formally, given an initial or auxiliary tree t that contains at least one inner node v marked for adjunction and given an auxiliary tree a whose root r has the same label as v, we can combine these trees with the following operation called *adjunction*, see Figure 1 for an example.

1. Replace a's foot node by the subtree rooted at v.
2. Replace the node v with the tree obtained from the last step, which is rooted at r.

Note that these steps make sense, since r and v have the same label. Note that adjunction does not change the number leaves labeled with a non-terminal symbol, i.e., an initial tree will stay an initial tree and an auxiliary tree will stay an auxiliary tree.

A *tree-adjoining grammar* is now defined as a tuple $\Gamma = (I, A, T, N)$ where

- I is a finite set of initial trees and
- A is a finite set of auxiliary trees,

using the same terminals T and non-terminals N as labels. The set D of *derived trees* of Γ consists of all trees that can be generated by starting with an initial tree in I and repeatedly adjoining auxiliary trees in A. (Note that each derived tree is also an initial tree, but not necessarily in I.) Finally, a string s over alphabet T can be *generated* by Γ, if there is a derived tree t in D such that

- t contains no nodes marked for adjunction and
- s is obtained by concatenating the labels of the leaves of t from left to right.

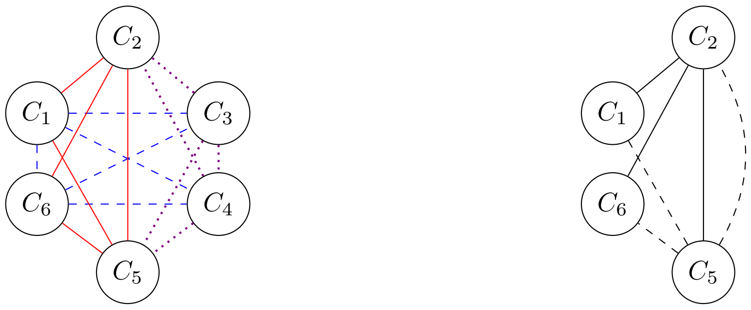
The language $L(\Gamma)$ is then the set of all strings that can be generated by Γ.

3 Encoding graphs

Given a graph $G = (V, E)$, we construct a string $\mathrm{GG}_k(G)$ that encodes its k-cliques, over the terminal alphabet $T = \{0, 1, \$, \#, |, \S, e, l_1, \dots, l_6, r_1, \dots, r_6\}$ of size 19. In the next section we then design a tree-adjoining grammar Γ that generates $\mathrm{GG}_k(G)$ if and only if G contains a $6k$-clique. We assume that $V = [|V|]$, and we denote the binary representation of any $v \in V$ by $\overline{v}$ and the neighborhood of v by $N(v)$. For two strings a and b, we use $a \circ b$ to denote their concatenation and a^R to denote the reverse of a.

We start with *node* and *list gadgets*, encoding a vertex and its neighborhood, respectively:

$$\mathrm{NG}(v) := \$\,\overline{v}\,\$ \quad \text{and} \quad \mathrm{LG}(v) := \bigcirc_{u \in N(v)} \mathrm{NG}(u) \;=\; \bigcirc_{u \in N(v)} \$\,\overline{u}\,\$$$



(a) Each C_i is a k-clique and there is an edge between two k-cliques of some highlighting style if the clique gadgets of that style ensure that these two cliques together form a $2k$-clique.

(b) We will generate an almost-$4k$-clique as in (a) by generating two claws. (This tests the edges (C_1, C_6), (C_2, C_5), and (C_3, C_4) in (a) twice.)

Figure 2 Structure of our test for $6k$-cliques.

Note that u and v are adjacent iff $\mathrm{NG}(u)$ is a substring of $\mathrm{LG}(v)$.

Next, we build clique versions of these gadgets, that encode a k*-clique* C and its neighborhood, respectively:

$$\mathrm{CNG}(C) := \bigcirc_{v \in C} (\#\ \mathrm{NG}(v)\ \#)^k \quad \text{and} \quad \mathrm{CLG}(C) := \Big(\bigcirc_{v \in C} \#\ \mathrm{LG}(v)\ \#\Big)^k$$

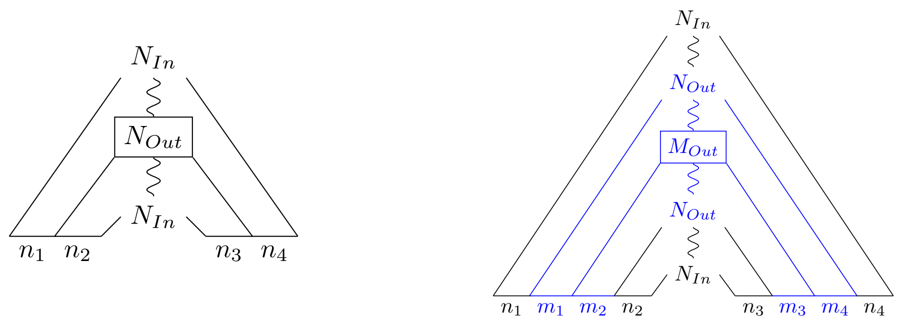
Note that two k-cliques C and C' form a $2k$-clique if and only if $\mathrm{CNG}(C)$ is a subsequence of $\mathrm{CLG}(C')$, since every pair of a vertex in C and a vertex in C' is tested for adjacency. We will later show how to implement this test for forming a $2k$-clique with a tree-adjoining grammar.

Conceptually, we split any $6k$-clique into six k-cliques. Thus, let $\mathcal{C}_k$ be the set of all k-cliques in G. Our final encoding of the graph is:

$$\begin{aligned}
\mathrm{GG}_k(G) := & \bigcirc_{C \in \mathcal{C}_k} \mid \underline{\mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R}\ l_1\ r_1\ \underline{\mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R} \mid \\
& \circ \bigcirc_{C \in \mathcal{C}_k} \mid \underline{\mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R}\ l_2\ r_2\ \underline{\mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R} \mid \\
& \circ \bigcirc_{C \in \mathcal{C}_k} \mid \underline{\mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R}\ l_3\ r_3\ \underline{\mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R} \mid \\
& \circ\ e \\
& \circ \bigcirc_{C \in \mathcal{C}_k} \mid \underline{\mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R}\ l_4\ r_4\ \underline{\mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R} \mid \\
& \circ \bigcirc_{C \in \mathcal{C}_k} \mid \underline{\mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R}\ l_5\ r_5\ \underline{\mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R} \mid \\
& \circ \bigcirc_{C \in \mathcal{C}_k} \mid \underline{\mathrm{CLG}(C)\ \S\ \mathrm{CLG}(C)^R}\ l_6\ r_6\ \underline{\mathrm{CNG}(C)\ \S\ \mathrm{CLG}(C)^R} \mid
\end{aligned}$$

As we will show, there is a tree-adjoining grammar of constant size that generates the string $\mathrm{GG}_k(G)$ iff G contains a $6k$-clique. The structure of this test is depicted in Figure 2. The clique-gadgets of the same highlighting style together allow us to test for an almost-$4k$-clique, as it is depicted in Figure 2a. The two gadgets of the same highlighting style then test for two claws of cliques, as depicted in Figure 2b.

As the graph has n nodes, for any node u the node and list gadgets $\mathrm{NG}(u), \mathrm{LG}(u)$ have a length of $O(n \log n)$, and for a k-clique C the clique neighborhood gadgets $\mathrm{CNG}(C), \mathrm{CLG}(C)$ thus have a length of $O(k^2 n \log n)$. As our encoding of the graph consists of $O(n^k)$ clique neighborhood gadgets, the resulting string length is $O(k^2 n^{k+1} \log n) = O(n^{k+1} \log n)$. It is



(a) A normal tree N. **(b)** The tree resulting after adjoining M into N.

Figure 3 Adjoining normal trees.

easy to see that it is also possible to construct all gadgets and in particular the encoding of a graph in linear time with respect to their length.

4 Programming with trees

It remains to design a clique-detecting tree-adjoining grammar. To make our reduction more accessible, we will think of tree-adjoining grammars as a certain programming language. In the end, we will then present a "program" that generates (a suitable superset of) the set all strings that represent a graph containing a $6k$-clique. We start by defining programs.

A *normal tree* N with *input* N_{In} and *output* N_{Out} is an auxiliary tree where:

- the root is labeled with N_{In},
- exactly one node is marked for adjunction, and
- this node lies on the path from the root to the foot node and is labeled N_{Out}.

See Figure 3a for an illustration. The special structure of a normal tree N allows us to split its nodes into four categories (excluding the path from N's root to its foot node): subtrees of left children of the path from N's root to N_{Out}, subtrees of left children of the path from N_{Out} to N's foot node, subtrees of right children of the path from N_{Out} to N's foot node, and the remaining nodes (i.e., subtrees of right children of the path from N's root to N_{Out}). The concatenation of all terminal symbols in N's leaves from left to right can then be split into four parts n_1, n_2, n_3, n_4 where each part contains symbols from exactly one category. We say that the normal tree N *generates* the tuple (n_1, n_2, n_3, n_4).

▶ **Lemma 4.1.** *Given normal trees N with input N_{In}, output N_{Out} and M with input $M_{In} = N_{Out}$, output M_{Out}, the derived tree $N \cdot M$ obtained by adjoining M into N is a normal tree with input N_{In} and output M_{Out}. Further, if N and M generate the tuples (n_1, n_2, n_3, n_4) and (m_1, m_2, m_3, m_4), then $N \cdot M$ generates the tuple $(n_1 \circ m_1, m_2 \circ n_2, n_3 \circ m_3, m_4 \circ n_4)$.*

Proof. See Figure 3. ◀

We now define a *program* P *with input* P_{In} *and output* P_{Out} as a set of normal trees that contains a tree with input P_{In} and a tree with output P_{Out}. Note that all trees derived by starting with a tree in P and repeatedly adjoining trees from P are normal, by Lemma 4.1. An *execution* of the program P is a derived tree of P with input P_{In} and output P_{Out}. Further, the *set computed by P*, denoted by $L(\mathsf{P})$, is the set of all tuples generated by P's executions.

We will later use programs as subroutines of tree-adjoining grammars. Let $N(\mathsf{P})$ be the set of non-terminals of P. Formally, we say that P is a *subroutine* of a grammar Γ if

- the set of trees P is a subset of the auxiliary trees of Γ, and
- no remaining auxiliary tree of Γ has a root label in $N(\mathsf{P}) \setminus \{\mathsf{P}_{Out}\}$.

These restrictions ensure that any "call" to the program P terminates at P_{Out}. Indeed, consider any sequence of adjunctions in Γ ending in a tree without nodes marked for adjunction. If this sequence contains an adjunction of a node labeled P_{In}, meaning that program P is called, then this adjunction must be followed by an execution of P, i.e., it must generate a derived tree of P with output P_{Out}. Indeed, any derived tree of P is normal and thus contains exactly one node marked for adjunction. To get rid of this node, we have to adjoin some auxiliary tree, but the remaining auxiliary trees can only adjoin to P_{Out}. We will frequently make use of this observation that ensures coherence of programs.

We now show how to perform two programs sequentially one after another. To avoid interference, we ensure that the two programs have disjoint non-terminals, except for their input and output. In particular, we will model two sequential calls to the same program by creating two copies of the program.

▶ **Lemma 4.2** (Combining programs). *For programs P and Q, let Q' denote the program obtained from Q by replacing each non-terminal by a fresh copy, ensuring that P and Q' have disjoint non-terminals. Further, let Q'' denote the program obtained from Q' by replacing Q'_{In} by P_{Out}. Then $\mathsf{P} \cdot \mathsf{Q} := \mathsf{P} \cup \mathsf{Q}''$ is a program computing the set*

$$L(\mathsf{P} \cdot \mathsf{Q}) := \{(a \circ a', b' \circ b, c \circ c', d' \circ d) \mid (a, b, c, d) \in L(\mathsf{P}), (a', b', c', d') \in L(\mathsf{Q})\}.$$

Proof. As every execution of P and Q'' is a normal tree, the claim follows from Lemma 4.1. ◀

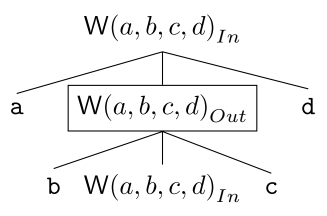
We can think of $\cdot$ as an operator on programs; the above lemma shows that it is associative.

4.1 Basic programs

We now present some easy programs that will later be used as subroutines.

4.1.1 Writing characters

We start by demonstrating a program that writes exactly one character to each of the four positions. Formally, given a 4-tuple of characters (a, b, c, d), let the program $\mathsf{W}(a, b, c, d)$ be defined by the following auxiliary tree:

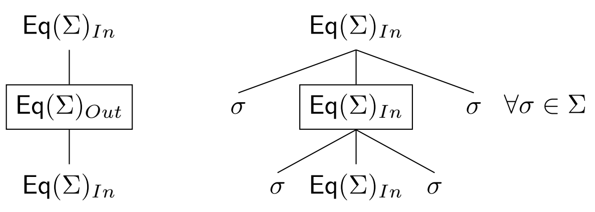


$\mathsf{W}(a,b,c,d)_{In}$
a
$\mathsf{W}(a,b,c,d)_{Out}$
d
b
$\mathsf{W}(a,b,c,d)_{In}$
c

Clearly, this tree is normal with input $\mathsf{W}(a, b, c, d)_{In}$ and output $\mathsf{W}(a, b, c, d)_{Out}$, so that $\mathsf{W}(a, b, c, d)$ is a program. The tree itself is an execution of the program, and it is the only execution. Thus, this program computes the set $L(\mathsf{W}(a, b, c, d)) = \{(a, b, c, d)\}$. We write $\mathsf{W}(a)$ to denote the program $\mathsf{W}(a, a, a, a)$.

4.1.2 Testing equality

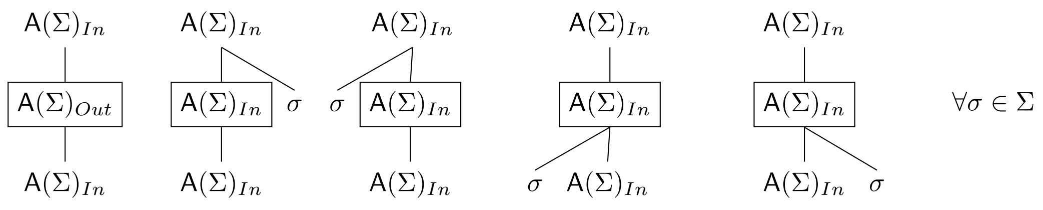
We give a program that tests equality of four strings, by writing the same arbitrary string to all four positions. Formally, for any terminal alphabet Σ, let the program $\mathsf{Eq}(\Sigma)$ be defined by the following set of $|\Sigma| + 1$ auxiliary trees:


$\mathsf{Eq}(\Sigma)_{In}$
$\mathsf{Eq}(\Sigma)_{Out}$
$\mathsf{Eq}(\Sigma)_{In}$
$\mathsf{Eq}(\Sigma)_{In}$
σ
$\mathsf{Eq}(\Sigma)_{In}$
σ
$\forall\sigma \in \Sigma$
σ
$\mathsf{Eq}(\Sigma)_{In}$
σ

A simple induction shows that $L(\mathsf{Eq}(\Sigma)) = \{(v, v^R, v, v^R) \mid v \in \Sigma^*\}$.

4.1.3 Writing anything

We will need to write appropriate strings surrounding some carefully constructed substrings. As it turns out, being able to write anything will be sufficient; this is achieved by the following program. Given an alphabet Σ, let the program $\mathsf{A}(\Sigma)$ be defined by the following set of $4|\Sigma| + 1$ trees:


$\mathsf{A}(\Sigma)_{In}$
$\mathsf{A}(\Sigma)_{Out}$
$\mathsf{A}(\Sigma)_{In}$
$\mathsf{A}(\Sigma)_{In}$
$\mathsf{A}(\Sigma)_{In}$ σ
$\mathsf{A}(\Sigma)_{In}$
$\mathsf{A}(\Sigma)_{In}$
σ $\mathsf{A}(\Sigma)_{In}$
$\mathsf{A}(\Sigma)_{In}$
$\mathsf{A}(\Sigma)_{In}$
$\mathsf{A}(\Sigma)_{In}$
σ $\mathsf{A}(\Sigma)_{In}$
$\mathsf{A}(\Sigma)_{In}$
$\mathsf{A}(\Sigma)_{In}$
$\mathsf{A}(\Sigma)_{In}$ σ
$\forall\sigma \in \Sigma$

As this program allows writing anything, it is easy to see that $\mathsf{A}(\Sigma)$ computes the set $(\Sigma^*)^4$.

4.2 Detecting Cliques

With the help of the above programs, we now design programs that detect a $6k$-clique.

4.2.1 Detecting claws

Our next program can detect whether four nodes form a claw graph.

$$\mathsf{NC} := \mathsf{W}(\#) \cdot \mathsf{A}(\{0,1,\$\}) \cdot \mathsf{W}(\$) \cdot \mathsf{Eq}(\{0,1\}) \cdot \mathsf{W}(\$) \cdot \mathsf{A}(\{0,1,\$\}) \cdot \mathsf{W}(\#)$$

▶ **Lemma 4.3.** *For any nodes v_1, v_2, v_3, v_4, the program* NC *generates the tuple*

$$(a, b, c, d) := (\#\,\mathrm{NG}(v_1)\,\#,\ \#\,\mathrm{LG}(v_2)^R\,\#,\ \#\,\mathrm{LG}(v_3)\,\#,\ \#\,\mathrm{LG}(v_4)^R\,\#)$$

and any of its cyclic rotations (i.e., (b, c, d, a), (c, d, a, b), and (d, a, b, c)) if and only if v_1 is adjacent to each one of v_2, v_3, and v_4.

Proof. By Lemma 4.2 and the properties of basic programs, we see that NC computes all tuples of the form

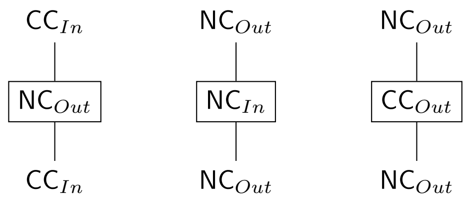
$$(\#\ \alpha_1\ \$\ v\ \$\ \alpha_2\ \#, \#\ \alpha_3\ \$\ v^R\ \$\ \alpha_4\ \#, \#\ \alpha_5\ \$\ v\ \$\ \alpha_6\ \#, \#\ \alpha_7\ \$\ v^R\ \$\ \alpha_8\ \#)$$

where $v \in \{0,1\}^*$ and $\alpha_1, \ldots, \alpha_8 \in \{0,1,\$\}^*$. From the construction of node and list gadgets we see that all tuples $(\#\mathrm{NG}(v_1)\#, \#\mathrm{LG}(v_2)^R\#, \#\mathrm{LG}(v_3)\#, \#\mathrm{LG}(v_4)^R\#)$ are of this form.

For the other direction, for any generated tuple (a,b,c,d), where a is $\#\,\$\,v\,\$\,\#$, it holds that $\$\,v\,\$$ or its reverse is a substring of b, c, and d. Hence, $\mathrm{NG}(v_1)$ is a substring of $\mathrm{LG}(v_2), \mathrm{LG}(v_3)$, and $\mathrm{LG}(v_4)$. This implies that v_1 is adjacent to v_2, v_3, and v_4. ◀

4.2.2 Detecting claws of cliques

We now extend NC to a program that can detect claws of k-cliques, see Figure 2b. We define the program CC by the following set of 3 trees (additional to the trees of NC):



Each execution of CC starts with the first tree, then repeatedly adjoins the second tree followed by some execution of NC, and finally adjoins the last tree. As the number of repetitions is arbitrary, the program CC can perform any number of sequential calls to NC.[4]

▶ **Lemma 4.4.** *For any k-cliques C_1, C_2, C_3, C_4 in G, the program* CC *generates the tuple $(a,b,c,d) := (\mathrm{CNG}(C_1), \mathrm{CLG}(C_2)^R, \mathrm{CLG}(C_3), \mathrm{CLG}(C_4)^R)$ and all of its cyclic rotations (i.e., (b,c,d,a), (c,d,a,b), and (d,a,b,c)) if and only if $C_1 \cup C_2$, $C_1 \cup C_3$, and $C_1 \cup C_4$ each form a $2k$-clique in G.*

Proof. For any nodes v_i^j, with $i \in [4], j \in [m], m \geq 1$, set

$$n_i := \bigcirc_{j \in [m]} \# NG(v_i^j)\# \qquad \text{and} \qquad \ell_i := \bigcirc_{j \in [m]} \# LG(v_i^j)\#.$$

As program CC can perform any number of calls to NC, and by Lemma 4.3, program CC generates the tuple $(n_1, \ell_2, \ell_3, \ell_4)$ if and only if v_1^j is adjacent to v_2^j, v_3^j, and v_4^j for all j.

Observe that for any k-cliques $C_1 = \{v_1, \ldots, v_k\}, C_2 = \{u_1, \ldots, u_k\}$, both $\mathrm{CNG}(C)$ and $\mathrm{CLG}(C)$ can be split into k^2 blocks by splitting between two consecutive #-characters:

$$\mathrm{CNG}(C_1) = \#\,\mathrm{NG}(v_1)\,\#\#\,\mathrm{NG}(v_1)\,\#\cdots\#\,\mathrm{NG}(v_1)\,\#\#\,\mathrm{NG}(v_2)\,\#\cdots$$
$$\mathrm{CLG}(C_2) = \#\,\mathrm{LG}(u_1)\,\#\#\,\mathrm{LG}(u_2)\,\#\cdots\#\,\mathrm{LG}(u_k)\,\#\#\,\mathrm{LG}(u_1)\,\#\cdots$$

This layout is chosen so that each node v_i in C_1 is paired up with each node u_j in C_2 exactly once. The claim follows from these two insights. ◀

4.2.3 Detecting almost-4k-cliques

We now use CC twice to test for two claws, thus detecting "almost-$4k$-cliques", as depicted in Figure 2b:

$$\mathsf{C} := \mathsf{CC} \cdot \mathsf{W}(\S) \cdot \mathsf{CC}.$$

Lemmas 4.4 and 4.2 directly imply the following, see Figure 2b.

[4] Actually, we already know how many calls to NC we want to perform, namely k^2. However, encoding this number into the grammar would result in a grammar size depending on k, which we want to avoid.

▶ **Lemma 4.5.** *For any k-cliques C_a, C_b, C_c, C_d the program* C *generates the tuple*

$$(\mathrm{CNG}(C_a)\ \S\ \mathrm{CLG}(C_a)^R, \mathrm{CLG}(C_b)\ \S\ \mathrm{CLG}(C_b)^R, \mathrm{CLG}(C_c)\ \S\ \mathrm{CLG}(C_c)^R, \mathrm{CNG}(C_d)\ \S\ \mathrm{CLG}(C_d)^R)$$

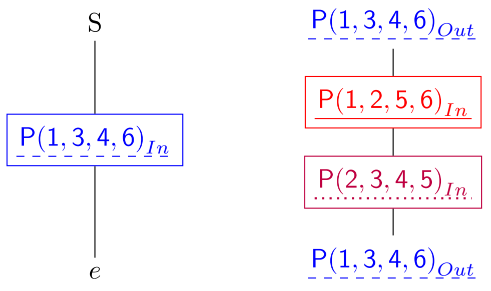
if and only if $C_a \cup C_b \cup C_d$ and $C_a \cup C_c \cup C_d$ both form a $3k$-clique. A similar statement holds if we pick any two other positions in the tuple for the $\mathrm{CNG}(\cdot)$ *gadgets.*

4.2.4 Detecting 6k-cliques

As in Figure 2a, we now want to test for three almost-$4k$-cliques to detect a $6k$-clique. Recall that $T = \{0, 1, \$, \#, |, \S, e, l_1, \ldots, l_6, r_1, \ldots, r_6\}$ is the terminal alphabet that we constructed our strings over. The following programs will generate the highlighted groups in Figure 2a:

$$\begin{aligned}
&\mathsf{P}(1,3,4,6) := \mathsf{A}(T) \cdot \mathsf{W}(|) \cdot \mathsf{C} \cdot \mathsf{W}(l_1, r_3, l_4, r_6)\\
&\mathsf{P}(1,2,5,6) := \mathsf{W}(r_1, l_2, r_5, l_6) \cdot \mathsf{C} \cdot \mathsf{W}(|) \cdot \mathsf{A}(T),\\
&\mathsf{P}(2,3,4,5) := \mathsf{W}(r_2, l_3, r_4, l_5) \cdot \mathsf{C} \cdot \mathsf{W}(|) \cdot \mathsf{A}(T),
\end{aligned}$$

We now deviate from our notion of normal trees by explicitly *not* marking $\mathsf{P}(1,2,5,6)_{Out}$ and $\mathsf{P}(2,3,4,5)_{Out}$ for adjunction. Our final tree-adjoining grammar Γ consists of the following initial and auxiliary trees (as well as all auxiliary trees used by its subroutines):



Note that the latter tree is the only one in Γ that has more than one node marked for adjunction, so it needs special treatment.

▶ **Lemma 4.6.** *For any graph G, the grammar Γ generates the encoding $\mathrm{GG}_k(G)$ if and only if G contains a $6k$-clique. Moreover, Γ has constant size (independent of k).*

Proof. First, assume that Γ can generate $\mathrm{GG}_k(G)$. Then there is a derived tree whose leaves, if read from left to right, yield $\mathrm{GG}_k(G)$. All derivations of Γ start with the single initial tree, and then adjoin an execution of the program $\mathsf{P}(1{,}3{,}4{,}6)$ into it. (As $\mathsf{P}(1{,}3{,}4{,}6)$ is a subroutine, only a full execution can be adjoined.) This execution generates some tuple of strings (x_1, x_2, x_3, x_4) and leaves exactly the node labeled $\mathsf{P}(1,3,4,6)_{Out}$ as the sole node marked for adjunction. Therefore, in the next step the auxiliary tree rooted with that node will be adjoined, which in turn leaves exactly the nodes $\mathsf{P}(1,2,5,6)_{In}$ and $\mathsf{P}(2,3,4,5)_{In}$ as nodes marked for adjunction. Again, these are input nodes of subroutines, therefore at both nodes one (complete) execution of the corresponding programs must be adjoined. The program execution of program $\mathsf{P}(1,2,5,6)$ generates a tuple of strings (y_1, y_2, y_3, y_4), and the execution of $\mathsf{P}(2,3,4,5)$ generates (z_1, z_2, z_3, z_4). The grammar Γ ensures that these tuples will be placed in the order $(x_1, y_1, y_2, z_1, z_2, x_2, x_3, z_3, z_4, y_3, y_4, x_4)$, see Figure 4 for a visualization. At this point, no more adjunctions are possible, since we explicitly forced $\mathsf{P}(1,2,5,6)_{Out}$ and $\mathsf{P}(2,3,4,5)_{Out}$ not to be marked for adjunction. (Also

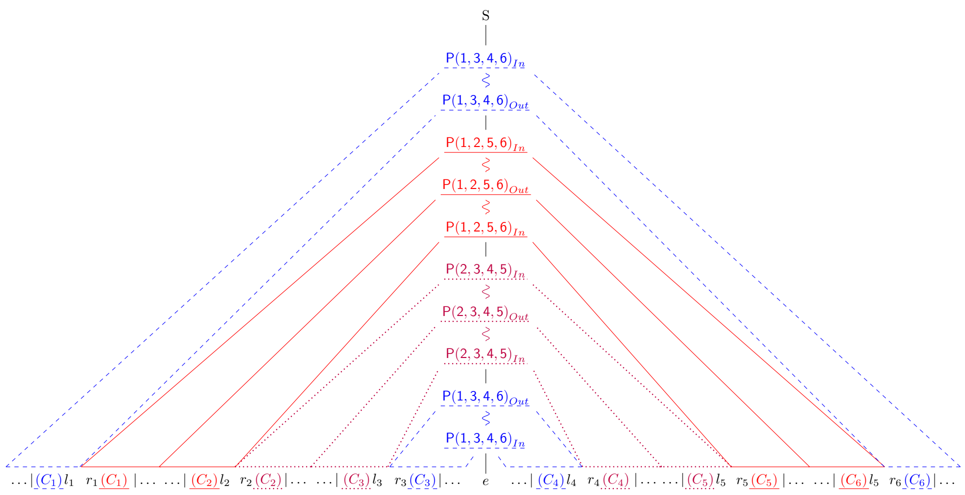

S
P(1,3,4,6)$_{In}$
P(1,3,4,6)$_{Out}$
P(1,2,5,6)$_{In}$
P(1,2,5,6)$_{Out}$
P(1,2,5,6)$_{In}$
P(2,3,4,5)$_{In}$
P(2,3,4,5)$_{Out}$
P(2,3,4,5)$_{In}$
P(1,3,4,6)$_{Out}$
P(1,3,4,6)$_{In}$
$\ldots|(C_1)\,l_1$ $r_1\,(C_1)\,|\ldots$ $\ldots|(C_2)\,l_2$ $r_2\,(C_2)\,|\ldots$ $\ldots|(C_3)\,l_3$ $r_3\,(C_3)\,|\ldots$ e $\ldots|(C_4)\,l_4$ $r_4\,(C_4)\,|\ldots$ $\ldots|(C_5)\,l_5$ $r_5\,(C_5)\,|\ldots$ $\ldots|(C_6)\,l_5$ $r_6\,(C_6)\,|\ldots$

Figure 4 Global structure of a parsing of $\mathrm{GG}_k(G)$ by Γ. (Clique gadgets are abbreviated.)

note that this structure is the only possibility to obtain a tree containing no more nodes marked for adjunction.) Hence, $\mathrm{GG}_k(G)$ can be partitioned as:

$$\mathrm{GG}_k(G) = \underline{x_1} \circ \underline{y_1} \circ \underline{y_2} \circ \underline{z_1} \circ \underline{z_2} \circ \underline{x_2} \circ \underline{x_3} \circ \underline{z_3} \circ \underline{z_4} \circ \underline{y_3} \circ \underline{y_4} \circ \underline{x_4}.$$

Consider the strings $\underline{x_1}$ and $\underline{y_1}$. By the definitions of $\mathsf{P}(1{,}3{,}4{,}6)$ and $\mathsf{P}(1{,}2{,}5{,}6)$, and Lemma 4.2, we know that $\underline{x_1}$ must end with the terminal symbol l_1 and that $\underline{y_1}$ must start with the symbol r_1. Whenever $l_1\, r_1$ occurs in $\mathrm{GG}_k(G)$, it does so in the string

$$|\ \underline{\mathrm{CNG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}\ l_1\ r_1\ \underline{\mathrm{CLG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}\ |,$$

for some k-clique C_1. Since $\underline{x_1} \circ \underline{y_1}$ is a substring of $\mathrm{GG}_k(G)$, and the program C cannot produce a $|$-terminal, but the $\mathsf{W}(|)$ part of $\mathsf{P}(\cdot,\cdot,\cdot,\cdot)$ will always write such a $|$-character, $\underline{x_1}$ must have $|\ \underline{\mathrm{CNG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}\ l_1$ as a suffix and $\underline{y_1}$ must have $r_1\ \underline{\mathrm{CLG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}\ |$ as a prefix. This also means that the program C must generate the string between $|$ and l_1 in $\underline{x_1}$ and between $|$ and r_1 in $\underline{y_1}$.

Similar statements hold for the other ten strings. In total we obtain that the program C generates the following tuples for some k-cliques $C_1, \ldots, C_6$:

- $t_1 := (\underline{\mathrm{CNG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}, \underline{\mathrm{CLG}(C_3)\ \S\ \mathrm{CLG}(C_3)^R},$
  $\underline{\mathrm{CLG}(C_4)\ \S\ \mathrm{CLG}(C_4)^R}, \underline{\mathrm{CNG}(C_6)\ \S\ \mathrm{CLG}(C_6)^R})$ in $\mathsf{P}(1{,}3{,}4{,}6)$,
- $t_2 := (\underline{\mathrm{CLG}(C_1)\ \S\ \mathrm{CLG}(C_1)^R}, \underline{\mathrm{CNG}(C_2)\ \S\ \mathrm{CLG}(C_2)^R},$
  $\underline{\mathrm{CNG}(C_5)\ \S\ \mathrm{CLG}(C_5)^R}, \underline{\mathrm{CLG}(C_6)\ \S\ \mathrm{CLG}(C_6)^R})$ in $\mathsf{P}(1{,}2{,}5{,}6)$, and
- $t_3 := (\underline{\mathrm{CLG}(C_2)\ \S\ \mathrm{CLG}(C_2)^R}, \underline{\mathrm{CNG}(C_3)\ \S\ \mathrm{CLG}(C_3)^R},$
  $\underline{\mathrm{CNG}(C_4)\ \S\ \mathrm{CLG}(C_4)^R}, \underline{\mathrm{CLG}(C_5)\ \S\ \mathrm{CLG}(C_5)^R})$ in $\mathsf{P}(2{,}3{,}4{,}5)$.

By Lemma 4.5, this implies that all $C_i \cup C_j$ form a $2k$-clique and thus $C_1 \cup \ldots \cup C_6$ forms a $6k$-clique (see Figure 2a to check that all pairs are covered).

For the other direction, consider a graph G that contains a $6k$-clique C^*. Then we can split C^* into 6 vertex-disjoint k-cliques $C_1, \ldots, C_6$. Further we know that every three of these six k-cliques together form a $3k$-clique. Thus, the program C generates the tuples t_1, t_2, t_3 as above. We can then use the three programs $\mathsf{P}(\cdot,\cdot,\cdot,\cdot)$ to generate such tuples surrounded with symbols $|$, l_i, and r_i at appropriate positions. Adding the surrounding strings by $\mathsf{A}(T)$ and following the global structure of Γ generates the encoding $\mathrm{GG}_k(G)$.

To see that Γ is of constant size, note that we only use constantly many programs. Thus using a new set of terminal symbols for every instance of a program will still yield a constant total number of non-terminal symbols. Further, we only use 19 terminal symbols. ◀

The above lemma and the bound $|\mathrm{GG}_k(G)| = O(n^{k+1} \log n)$ imply the main theorem.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is Valiant's parser. In Venkatesan Guruswami, editor, *Proc. of the 56th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2015)*, pages 98–117. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.16.

2 Anne Abeillé. Parsing French with tree adjoining grammar: some linguistic accounts. In D. Vargha, editor, *Proc. of the 12th Conf. on Computational Linguistics (COLING 1988)*, pages 7–12. Assoc. for Computational Linguistics, 1988. doi:10.3115/991635.991637.

3 Arturs Backurs, Nishanth Dikkala, and Christos Tzamos. Tight hardness results for maximum weight rectangles. In I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi, editors, *Proc. of the 43rd Int'l Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *LIPIcs*, pages 81:1–81:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.81.

4 Arturs Backurs and Christos Tzamos. Improving Viterbi is hard: Better runtimes imply faster clique algorithms, 2016. arXiv:1607.04229.

5 Karl Bringmann, Allan Grønlund, and Kasper Green Larsen. A dichotomy for regular expression membership testing, 2016. arXiv:1611.00918.

6 Yi-Jun Chang. Hardness of RNA folding problem with four symbols. In Roberto Grossi and Moshe Lewenstein, editors, *Proc. of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *LIPIcs*, pages 13:1–13:12. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.13.

7 David Chiang and Alexander Koller, editors. *Proc. of the 12th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+12)*. ACL, 2016. URL: http://www.aclweb.org/anthology/W16-33.

8 John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970. URL: http://www.softwarepreservation.org/projects/FORTRAN/CockeSchwartz_ProgLangCompilers.pdf.

9 Vera Demberg, Frank Keller, and Alexander Koller. Incremental, predictive parsing with psycholinguistically motivated tree-adjoining grammar. *Comput. Ling.*, 39(4):1025–1066, 2013. doi:10.1162/COLI_a_00160.

10 Christy Doran, Dania Egedi, Beth Ann Hockey, Bangalore Srinivas, and Martin Zaidel. XTAG system: a wide coverage grammar for English. In Yorick Wilks, editor, *Proc. of the 15th Conf. on Computational Linguistics (COLING 1994)*, pages 922–928. ACL, 1994. doi:10.3115/991250.991297.

11 Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970. doi:10.1145/362007.362035.

12 Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.*, 326(1-3):57–67, 2004. doi:10.1016/j.tcs.2004.05.009.

13 Katherine Forbes, Eleni Miltsakaki, Rashmi Prasad, Anoop Sarkar, Aravind Joshi, and Bonnie Webber. D-LTAG system: Discourse parsing with a lexicalized tree-adjoining grammar. *J. Log. Lang. Inf.*, 12(3):261–279, 2003. doi:10.1023/A:1024137719751.

14 Aravind K. Joshi. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? In D. R. Dowty, L. Karttunen, and A. M.

Zwicky, editors, *Natural Language Processing: Psychological, Computational, and Theoretical Perspectives*. CUP, 1985. doi:10.1017/cbo9780511597855.007.

15 Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1):136–163, 1975. doi:10.1016/S0022-0000(75)80019-5.

16 Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Beyond Words*, volume 3 of *Handbook of Formal Languages*, pages 69–123. Springer, 1997. doi:10.1007/978-3-642-59126-6_2.

17 Tadao Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical Report R-257, Coordinated Science Laboratory, University of Illinois, 1966. URL: http://hdl.handle.net/2142/74304.

18 Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002. doi:10.1145/505241.505242.

19 Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Comment. Math. Univ. Carolin.*, 26(2):415–419, 1985. URL: http://dml.cz/dmlcz/106381.

20 Sanguthevar Rajasekaran and Shibu Yooseph. TAL recognition in $O(M(N^2))$ time. *J. Comput. Syst. Sci.*, 56(1):83–89, 1998. doi:10.1006/jcss.1997.1537.

21 Philip Resnik. Probabilistic tree-adjoining grammar as a framework for statistical natural language processing. In Antonio Zampolli, editor, *Proc, of the 14th Conf. on Computational Linguistics (COLING 1992)*, pages 418–424. ACL, 1992. doi:10.3115/992133.992135.

22 Giorgio Satta. Tree-adjoining grammar parsing and boolean matrix multiplication. *Comput. Ling.*, 20(2):173–191, June 1994. URL: http://dl.acm.org/citation.cfm?id=972525.972527.

23 Yves Schabes and Aravind K. Joshi. An Earley-type parsing algorithm for tree adjoining grammars. In J. Hobbs, editor, *Proc. of the 26th Annual Meeting of the Association for Computational Linguistics (ACL 1988)*, pages 258–269. ACL, 1988. doi:10.3115/982023.982055.

24 Stuart M. Shieber and Yves Schabes. Synchronous tree-adjoining grammars. In H. Karlgren, editor, *Proc. of the 13th Conf. on Computational Linguistics (COLING 1990)*, pages 253–258. ACL, 1990. doi:10.3115/991146.991191.

25 Matthew Stone and Christine Doran. Sentence planning as description using tree adjoining grammar. In Philip R. Cohen and Wolfgang Wahlster, editors, *Proc. of the 35th Annual Meeting of the Association for Computational Linguistics (ACL 1997)*, pages 198–205. ACL, 1997. doi:10.3115/976909.979643.

26 Yasuo Uemura, Aki Hasegawa, Satoshi Kobayashi, and Takashi Yokomori. Tree adjoining grammars for RNA structure prediction. *Theor. Comput. Sci.*, 210(2):277–303, 1999. doi:10.1016/S0304-3975(98)00090-5.

27 Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975. doi:10.1016/s0022-0000(75)80046-8.

28 Virginia Vassilevska. Efficient algorithms for clique problems. *Inf. Process. Lett.*, 109(4):254–257, 2009. doi:10.1016/j.ipl.2008.10.014.

29 K. Vijay-Shankar and Aravind K. Joshi. Some computational properties of tree adjoining grammars. In W. C. Mann, editor, *Proc. of the 23rd Annual Meeting of the Association for Computational Linguistics (ACL 1985)*, pages 82–93. ACL, 1985. doi:10.3115/981210.981221.

30 K. Vijay-Shanker and Aravind K. Joshi. Feature structures based tree adjoining grammars. In Dénes Vargha, editor, *Proc. of the 12th Conf. on Computational Linguistics (COLING 1988)*, pages 714–719. ACL, 1988. doi:10.3115/991719.991783.

31 Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Inf. Control*, 10(2):189–208, 1967. doi:10.1016/S0019-9958(67)80007-X.

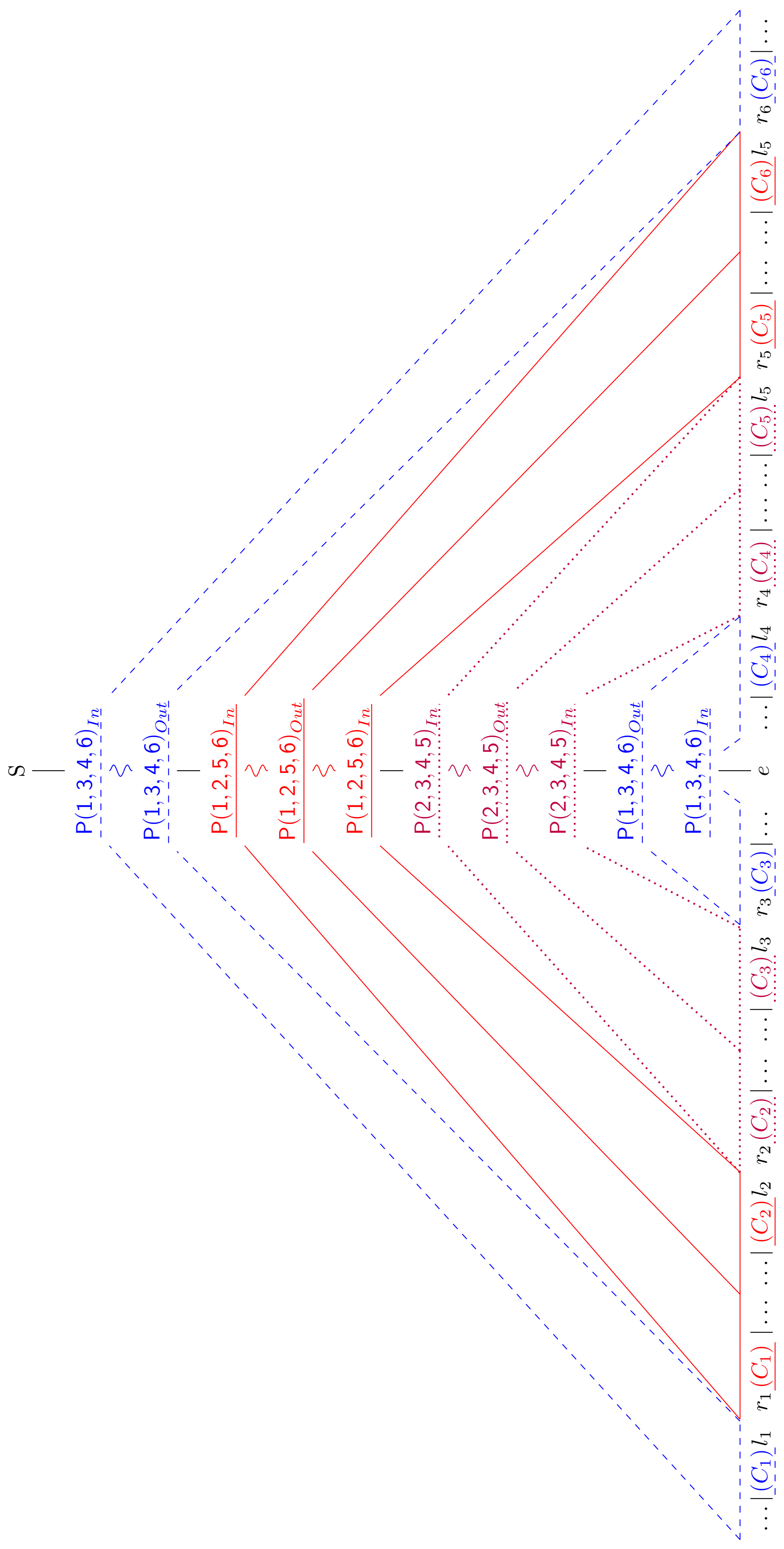

S
P(1,3,4,6)_In
P(1,3,4,6)_Out
P(1,2,5,6)_In
P(1,2,5,6)_Out
P(1,2,5,6)_In
P(2,3,4,5)_In
P(2,3,4,5)_Out
P(2,3,4,5)_In
P(1,3,4,6)_Out
P(1,3,4,6)_In
e
… | (C1) l1 r1 (C1) | … … | (C2) l2 r2 (C2) | … … | (C3) l3 r3 (C3) | … … | (C4) l4 r4 (C4) | … … | (C5) l5 r5 (C5) | … … | (C6) l6 r6 (C6) | …

Figure 5 *Enlarged version of Figure 4.* Global structure of a parsing of $\mathrm{GG}_k(G)$ by Γ. (Clique gadgets are abbreviated.)

Communication and Streaming Complexity of Approximate Pattern Matching

Tatiana Starikovskaya

Université Paris-Diderot – Paris 7, Paris, France
tat.starikovskaya@gmail.com

Abstract

We consider the approximate pattern matching problem. Given a text T of length $2n$ and a pattern P of length n, the task is to decide for each prefix $T[1, j]$ of T if it ends with a string that is at the edit distance at most k from P. If this is the case, we must output the edit distance and the corresponding edit operations. We first look at the communication complexity of the problem. We show the following:

- If Alice and Bob both share the pattern and Alice holds the first half of the text and Bob the second half, then the deterministic one-way communication complexity of the problem is $\Theta(k \log n)$.
- If Alice holds the first half of the text, Bob the second half of the text, and Charlie the pattern, then there is a deterministic one-way communication protocol that uses $\mathcal{O}(k\sqrt{n} \log n)$ bits.

We then develop the first sublinear-space streaming algorithm for the problem.

- There exists a streaming algorithm that solves the problem in $\mathcal{O}(k^8\sqrt{n} \log^6 n)$ space. The worst-case time complexity of the algorithm $\mathcal{O}((k^2\sqrt{n}+k^{13}) \cdot \log^4 n)$ per arrival. The algorithm is randomised with error probability at most $1/\text{poly}(n)$.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases approximate pattern matching, edit distance, randomised algorithms, streaming algorithms, communication complexity

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.13

1 Introduction

In this work we study the famous approximate pattern matching problem. Recall that the edit distance between two strings S_1, S_2 is the minimum number of insertions, deletions, and substitutions required to transform S_1 to S_2. Assume we are given a pattern P and a text T. We say that a substring S of T is a k-mismatch occurrence of P if the edit distance between S and P is at most k. In the approximate pattern matching problem we must find all prefixes $T[1, j]$ of T that end with a k-mismatch occurrence of P. The problem has numerous applications in bioinformatics, signal processing, text retrieval, and has received a lot of attention in the literature.

1.1 Our results

We first study the communication complexity of the problem, namely, we consider the following setting. Let T be a text of length $2n$ and P be a pattern of length n. Let Alice hold the information about the first half of the text, and let Bob hold the information about the second half of the text. Alice sends Bob a message, and Bob's task is to find all prefixes $T[1, j]$ that end with a k-mismatch occurrence of P and the edit operations that transform the occurrence into P using only Alice's message and his half of the text. The minimal size of

Alice's message that allows Bob to complete the task is called the communication complexity of the problem.

It is not hard to see that if both Bob and Alice have access to the pattern, the communication complexity is $\Theta(k \log n)$. Indeed, from the information theoretic lower bound it follows that Alice has to send at least $k \log n$ bits. On the other hand we can consider the following (deterministic) protocol. Alice first finds the smallest i such that the edit distance between $T[i, n]$ and some prefix $P[1, j]$ of the pattern is at most k, and then sends the edit operations and j to Bob. Bob uses the message from Alice to restore $T[i, n]$. He then knows both $T[i, 2n]$ and P and therefore can compute all outputs. (Note that the edit distance between P and any substring of T that starts in $[1, i]$ and ends in $[n + 1, 2n]$ is at least k, and therefore Bob does not need any information about $T[1, i]$). However, the situation is different when only the third party, Charlie, knows the pattern, as in this case Alice can no longer use the pattern to encode her half of the text. We show the following theorem.

▶ **Theorem 1.** *When both P and T are binary, the one-way deterministic communication complexity of the approximate pattern matching problem for three parties is $\mathcal{O}(k\sqrt{n} \log n)$.*

The main idea of the proof is that if k-mismatch occurrences of the pattern in the text are rare, Alice can send them all to Bob. If on the other hand there are many k-mismatch occurrences of the pattern, two of them will be located close to each other and therefore the underlying text will be weakly periodic, which will allow to encode it in small space.

Our motivation to study the communication complexity of the problem is twofold. First, it can be viewed as a generalisation of the document exchange problem, where we have two parties Alice and Bob, Alice holds one string and Bob holds the other string, and Bob's task is to decide the edit distance between their strings using the message Alice sends to him and his half of the text. If the distance is at most k, Bob must output the edit operations that transform his string into Alice's string. Otherwise, he may simply say that the distance is too large. In the paper we will refer to Alice's message as document exchange sketch. The problem has been studied both in deterministic and randomised settings [1, 4, 3, 8, 11, 15]. The protocol shown by Orlitsky in 1991 [15] has optimal complexity $\Theta(k \log n)$ and is deterministic. However, Bob needs $n^{\mathcal{O}(k)}$ time to compute the distance. Recently, Belazzougui showed a new deterministic protocol [1]. It has complexity $\mathcal{O}(k^2 + k \log^2 n)$ and much lower computation time of $n \cdot \text{poly}(\log n)$. The best randomised protocol is due to [2] and has $\mathcal{O}(k \cdot (\log^2 k + \log n))$ complexity and $n \cdot \text{poly}(\log n)$ computation time.

The second reason to study the communication complexity of the problem is its relation to streaming algorithms. Let us first remind the setting. Consider a pattern P of length n and a text T of length $2n$ arriving as a stream, one symbol at a time. When a new symbol arrives we must decide if the current text ends with a k-mismatch occurrence of P and if so output the edit operations that transform the occurrence into P. We assume the standard RAM model of computation. The time complexity is defined in the usual way, and the space complexity is defined as all the space used by the algorithm. In particular, if we store a copy of the pattern or of the text we must account for it. It is well-known that a communication complexity lower bound implies a similar space lower bound for a streaming algorithm. However, upper bounds provide some insight as well. Imagine that the algorithm processes the stream in non-overlapping blocks, then it needs an efficient way to encode the edit distances in each of the blocks, and one possible approach is to use the message that Alice sends to Bob in the communication complexity protocol. We will use this idea to show the first sublinear-space streaming algorithm for the problem.

▶ **Theorem 2.** *Assume that both P and T are binary and that $k < n^{1/c}$ for a sufficiently large constant $c > 0$. There is a streaming algorithm that solves the approximate pattern matching problem in $\mathcal{O}(k^8\sqrt{n}\log^6 n)$ space and $\mathcal{O}((k^2\sqrt{n}+k^{13})\cdot\log^4 n)$ worst-case time per symbol. The algorithm is correct with probability $1-1/\mathrm{poly}(n)$.*

The main advance has become possible due to the result of Belazzougui and Zhang [2], who showed a sketch that can be used to compute the exact value of the edit distance between two strings if it is at most k. Our algorithm maintains such sketches for $\mathcal{O}(\sqrt{n})$ suffixes of the text. To compute the edit distance between the pattern and the text the algorithm divides the pattern into two parts, a short prefix and a suffix aligned with one of the sketched suffixes of the text. The edit distance between the short prefix and the text is computed beforehand using dynamic programming and stored very compactly using the communication complexity approach. The edit distance between the suffix and the text is computed with the help of the sketches. We note that the requirement on the text length is not restrictive. Indeed, if the text's length is larger than $2n$, then one can split it into blocks of length $2(n+k)$ which overlap by $n+k$ symbols (the last block can be shorter) and run the algorithm of Theorem 2 independently for each of the blocks. For each k-mismatch occurrence of P there is a block containing it and therefore the algorithm is correct. The complexity of the algorithm and the error probability do not change.

As we have already mentioned the problem has been extensively studied in the literature. For a survey of previous solutions see [14]. The solutions can be roughly classified into four main types: dynamic programming algorithms, automata-based algorithms, filtering algorithms, and bit-parallelism. To the best of author's knowledge, all previously known solutions require at least $\Omega(n/\log n)$ space, and thus our result exhibits a remarkable improvement in space complexity. On the other hand, the running time of our algorithm is rather large. This is because the nature of the sketches is very complex and we have to maintain them independently. We give further details in Sections 3 and 4.

1.2 Related work

Lower bounds. In this work we focus on computing small edit distances between a pattern and a stream. If however we were interested in computing all edit distances, we would have to spend at least $n^{1-\varepsilon}$ amortised time per output for any constant $\varepsilon > 0$ unless the strong Exponential Time Hypothesis is false. (The original bound was given for computing the edit distance between two strings, and our problem is harder.) The best unconditional time lower bound was shown by Clifford et al. [6] who considered the problem in the cell-probe model, where the time complexity of algorithm is measured as the number of cells that must be accessed to compute the output. This model is particularly strong and any lower bounds that hold in it hold in the RAM model as well. Clifford et al. showed that the expected amortised time of any randomised algorithm that solves the edit distance problem is $\Omega(\sqrt{\log n}/(\log\log n)^{3/2})$ per output.

Approximate pattern matching in a stream. Another formalisation of approximate pattern matching is the k-mismatch problem, where one must find all substrings of the text such that the Hamming distance between them and the pattern is at most k. The first streaming algorithm for this problem was given in [16]. It used $\mathcal{O}(k^3\log^7 n/\log\log n)$ space and $\mathcal{O}(k^2\log^5 n/\log\log n)$ time per arriving symbol. In [5] this result was improved in terms of the dependency on k to $\mathcal{O}(k^2\log^{11} n/\log\log n)$ space and $\mathcal{O}(\sqrt{k}\log k+\log^5 n)$ time per arriving symbol. Finally, in [7] the authors studied communication and streaming complexities of computing approximate values of all Hamming distances between the pattern and the text.

2 Communication complexity

In this section we show Theorem 1. Recall that Alice holds the first half of the text, Bob the second half of the text, and only Charlie holds the pattern. Bob must find all prefixes $T[1, j]$ of T that end with a k-mismatch occurrence of P and output the edit operations that convert the occurrence into P.

2.1 Periodicity under edit distance

We start by introducing a notion of approximate period for the edit distance. The idea is that two close k-mismatch occurrences of the pattern imply weak periodicity of the text. We will use this property of the text to encode it in small space.

▶ **Definition 3.** The α-period of a string S is a minimal integer $\ell > \alpha$ such that the edit distance between some prefix of S and $S[\ell + 1, n]$ is at most α.

▶ **Example 4.** The 1-period of a string $S = bbaabb$ is 3. This is because $S[3, 6] = aabb$ cannot be transformed into a prefix of S using just one edit operation, while the edit distance between $S[4, 6] = abb$ and $S[1, 2]$ is exactly one.

The condition $\ell > \alpha$ is essential as any suffix $S[\ell + 1, n]$ can be transformed into S by ℓ insertions. We now show that the α-period can be used to encode the pattern in an efficient way similar to the way the period of a string can be used to encode it.

▶ **Lemma 5.** *If the $4k$-period of a string S of length n is $\rho > 4k$, then S can be encoded in $\mathcal{O}(\rho + k \log n)$ bits.*

Proof. The encoding will occupy $\mathcal{O}(\rho + k \log n)$ bits and contain the prefix and the suffix of S of length ρ (both taking $\mathcal{O}(\rho)$ bits to store), and the at most $4k$ edit operations that transform a prefix S' of S into $S[\rho + 1, n]$. The information about the edit operations will include the type of the operation (insertion, deletion, substitution), the position, and the symbol itself.

We now show that the encoding is lossless. Consider the first ρ symbols of S'. Let $i_1 \le 4k$ be the number of these symbols that must be deleted. It follows that the remaining $\rho - i_1$ symbols of S' must be aligned against the symbols of $S[\rho + 1, n]$. Therefore, using the encoding, we can restore (at least) the first $\rho - i_1$ symbols of $S[\rho + 1, n]$ and consequently $S'[1, 2\rho - i_1]$. (Recall that insertions and replacements are stored in the encoding explicitly.) We then consider $S'[\rho, 2\rho - i_1]$. Let i_2, $i_1 + i_2 \le 4k$, be the number of symbols in $S'[\rho, 2\rho - i_1]$ that must be deleted. We can then use the remaining symbols to restore the first $2\rho - i_1 - i_2$ symbols of $S[\rho + 1, n]$ and consequently $S'[1, 3\rho - i_1 - i_2]$. We continue in a similar way until we reach the end of S'. At this point, we will restore all symbols of S except for maybe the last ρ symbols which we already know from the encoding. ◀

2.2 Communication complexity protocol

We first explain what Charlie sends to Alice, and what Alice sends to Bob. Let $B = k\sqrt{n} \log n$ and $n_B = \lceil n/B \rceil$. Charlie sends to Alice document exchange sketches for each prefix $P[1, (n_B - i) \cdot B]$ and for each suffix $P[(n_B - i) \cdot B + 1, n]$. We use deterministic document exchange sketches of size $\mathcal{O}(k^2 + k \log^2 n)$ bits [1]. (We note that using $\mathcal{O}(k \log n)$-space sketches [15] would not improve the complexity but would drastically increase the computation time for Alice and Bob. For this reason, even though time is not the focus of this work, we

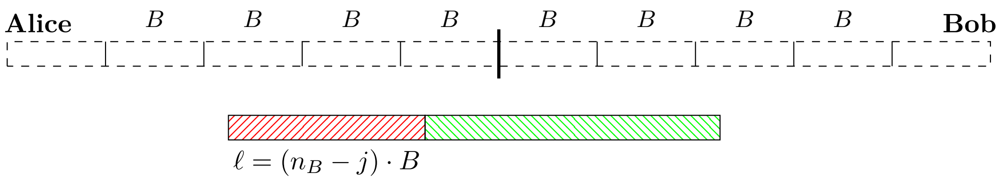


Figure 1 Let i be the first block containing two k-mismatch occurrences of $P[1,(n_B - i) \cdot B]$ that start at least $2k$ positions apart. To compute the edit distances in a block $j < i$ Bob divides the pattern into two parts, a prefix $P[1,\ell]$ and the suffix $P[\ell+1,n]$, and computes the distance for each of the two parts separately.

prefer the sketches [1].) Alice starts by dividing her half of the text into non-overlapping blocks of length B except for the last one which may be shorter, that is in total there are n_B blocks.

▶ **Definition 6.** A position p of a block i is k-good if it is the left endpoint of a k-mismatch occurrence of $P[1,(n_B - i) \cdot B]$.

Alice considers each block i in turn and finds all k-good positions in the block using the pattern sketches. Suppose first that all k-good positions in the block are at distance $< 4k$. In this case all k-mismatch occurrences of $P[1,(n_B - i) \cdot B]$ that start in these positions end in an interval of length at most $6k$. For each position in this interval Alice finds the substring that ends in it and has the smallest edit distance from $P[1,(n_B - i) \cdot B]$ (using the pattern sketches again) and sends the distance and the corresponding edit operations to Bob. In total this information occupies $\mathcal{O}(k^2 \log n)$ bits per block. Suppose now that block i contains two k-good positions p_1, p_2, where $p_2 - p_1 > 4k$, and let i be the first such block. Let $\ell = (n_B - i) \cdot B$ and let ED be the edit distance between two strings.

▶ **Lemma 7.** *The $4k$-period of $T[p_1, p_2 + \ell - 1]$ is at most B.*

Proof. By the definition both p_1 and p_2 are starting positions of k-mismatch occurrences of $P[1,\ell]$. Therefore, $ED(T[p_1, p_1 + \ell - 1], P[1,\ell]) \leq 2k$ and $ED(T[p_2, p_2 + \ell - 1], P[1,\ell]) \leq 2k$. From the triangle inequality it follows that $ED(T[p_1, p_1 + \ell], T[p_2, p_2 + \ell - 1]) \leq 4k$ and from the definition of approximate periods it follows that the $4k$-period of $T[p_1, p_2 + \ell - 1]$ is at most B. ◀

By Lemma 5 the substring $T[p_1, p_2 + \ell - 1]$ and therefore $T[p_1, n - B]$ can be encoded in $\mathcal{O}(B + k \log n)$ bits. Alice sends the encoding to Bob (note that she only does it for the first block containing distant k-good positions). Finally, she sends Bob the last $(B + k)$ symbols of her half of the text and also forwards the sketches received from Charlie. The total size of Alice's message is $\mathcal{O}((n/B) \cdot k^2 \log^2 n + B) = \mathcal{O}(k\sqrt{n} \log n)$ bits.

We now explain how Bob computes the distances. Suppose that he wants to compute the edit distance between the pattern a substring starting to the left of position p_1. Using the encoding of $T[p_1, n - B]$, the last B symbols of Alice's half of the text, and his half of the text he can restore all symbols of $T[p_1, 2n]$. He can then use the pattern sketch to compute the edit distance and operations. Consider now the case when the substring starts in a block $j < i$ (see Fig. 1). Let S be the substring for which Bob wants to compute the edit distance and $\ell = (n_B - j) \cdot B$. Bob starts by dividing the pattern into two parts, a prefix $P[1,\ell]$ and the suffix $P[\ell + 1, n]$. The following observation is a corollary of the definition of the edit distance.

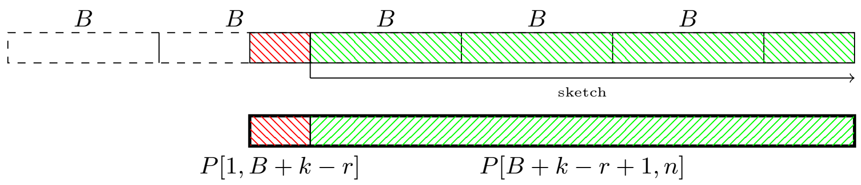


Figure 2 The algorithm processes the text in blocks of size B. To decide whether the current stream ends with a k-mismatch occurrence of P, the algorithm divides the pattern into two parts, a prefix of length at most $B+k$ and the remaining suffix and computes the edit distance for each of the parts separately.

▶ **Observation 8.** *Let* $\Delta = \min_{\ell' \in [\ell-k, \ell+k]} \{ED(P[1,\ell], S[1,\ell']) + ED(P[\ell+1,n], S[\ell'+1,n])\}$. *If* $\Delta > k$*, then the edit distance between* S *and* P *is larger than* k*, and otherwise it is equal to* Δ.

Since $j < i$, Bob knows all positions ℓ' of S for which there exists a k-mismatch occurrence of $P[1,\ell]$ ending at this position (and also the edit operations that convert the occurrence into $P[1,\ell]$). On the other hand, since Bob knows the last $B+k$ symbols of Alice's half of the text, he knows $S[\ell'+1,n]$ and can use the sketch of $P[\ell+1,n]$ to compute the edit distance and the edit operations between the two. He can therefore decide if S is a k-mismatch occurrence of P and the edit operations that transform S into P.

3 Streaming

We now show a streaming algorithm for approximate pattern matching. As soon as a new symbol arrives we must decide if the current stream ends with a k-mismatch occurrence of P and output the edit operations between P and the occurrence. The algorithm processes the text by blocks of size $B = \sqrt{n}$ (see Fig. 2). Suppose that the text ends with a k-mismatch occurrence of the pattern P. This occurrence can be divided into two parts, a prefix of length at most B, and a suffix that starts at a block border. From Observation 8 it follows that there exists a position $i \in [1, B+k]$ such that the prefix of the occurrence must be aligned with $P[1,i]$, and the suffix of the occurrence must be aligned with $P[i+1,n]$. The algorithm will therefore need to be able to compute the edit distances between each block and prefixes $P[1,i]$, and the edit distances between suffixes of the text starting at block borders and suffixes $P[i+1,n]$.

3.1 Prefixes

Consider a block of the text T. For each i such that the block ends with a k-mismatch occurrence of $P[1,i]$ we define S_i to be the suffix of the block with the smallest edit distance from $P[1,i]$. Below we will show a hybrid dynamic programming algorithm that computes all suffixes S_i, the corresponding edit distances and edit operations in $\mathcal{O}((B+k)\cdot k)$ space and in $\mathcal{O}(k)$ time per symbol of the block. But first, let us explain how we apply it. Note that the suffixes S_i, the distances and the operations will be used only n/B blocks later. A naive approach would be to compute all this information and to store it explicitly until that time. However, the total space requirement of this approach is too large. Instead, we develop a different approach which runs the algorithm twice. Upon having received a new text block, we run the algorithm for the first time and compute suffixes S_i for all $i \in [1, B+k]$. Let

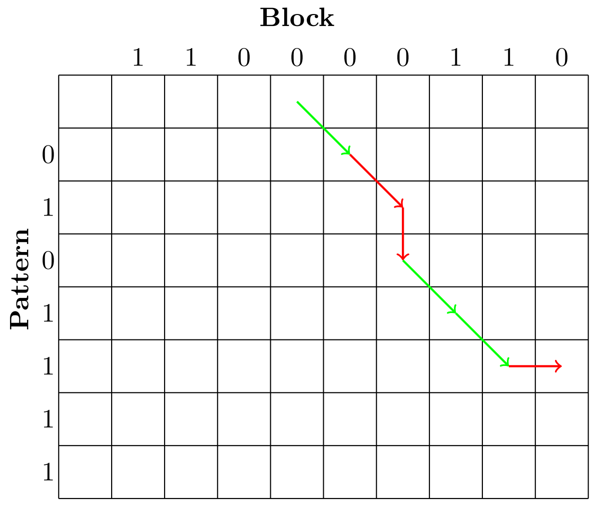


Figure 3 The graph shows a 3-path that encodes the edit operations between $P[1,5] = 01011$ and a suffix 00110 of the block. The three red arrows show the edit operations: a replacement, an insertion, and a deletion.

$S^\star = S_j$ be the longest of the retrieved suffixes. We encode the block as a tuple consisting of the position j, and the at most k edit operations that transform $P[1,j]$ into $S^\star$ (see also Introduction). After having read $n/B-2$ more blocks we use the encoding and $P[1,B+k]$ to restore $S^\star$ and then run the algorithm on $S^\star$ to compute the suffixes S_i and the corresponding edit operations.

We now describe our algorithm. The algorithm uses the same approach as the hybrid dynamic programming algorithms for the approximate pattern matching problem [12, 13] (see also [10, Chapter 12.2.4]). We assume that $P[1,B+k]$ is stored explicitly. The algorithm receives as an input a text block of length $\leq B$. The algorithm starts by preprocessing the $P[1,B+k]$ and the block for longest common extension queries. For a pair of positions (p_1,p_2), a longest common extension query finds the longest substring starting at position p_1 of the block that matches a substring starting at position p_2 of $P[1,B+k]$. The preprocessing phase takes $\mathcal{O}(B+k)$ time and space [9]. The algorithm then considers a table of size $(B+k+1)\times(B+1)$ and builds a set of paths from the first row to the last column of the table. Each such path will correspond to a suffix of the block that is a k-mismatch occurrence of $P[1,i]$ and encode the edit operations that transform the suffix into $P[1,i]$.

The algorithm runs in k rounds. In round m, $1 \leq m \leq k$, it processes each of the diagonals of the table in turn and finds a path that corresponds to at most m edit operations (m-path) and ends in the lowest cell in the current diagonal. Each of the paths starts in one of the cells in the first row of the table. From a cell (p_1,p_2) a path can go either to a cell (p_1+1,p_2), or to (p_1,p_2+1), or to (p_1+1,p_2+1). Let a be the (p_1+1)-th symbol of the block and b be the (p_2+1)-th symbol of the pattern. A move to (p_1+1,p_2) corresponds to deletion of a, a move to (p_1,p_2+1) to insertion of b, and a move to (p_1+1,p_2+1) to a replacement of a by b if $a \neq b$. If symbols a,b are not edited, the path makes a diagonal step as well. Suppose that in round m, $m \leq k$, a path reaches a cell (B,i) of the last column of the table for the first time. From construction it follows that this path corresponds to the suffix S_i.

It remains to explain how the algorithm finds the m-paths. Consider a diagonal i. To find the m-path that ends in the lowest cell in the diagonal, the algorithm tries to extend the $(m-1)$-paths for diagonals $i-1$, i, and $i+1$. Consider first the $(m-1)$-path for diagonal i. Suppose that it ends in a cell $(j, j+i)$. The algorithm makes a step from $(j, j+i)$ to $(j+1, j+i+1)$ that corresponds to a replacement of a symbol and then tries to extend the path further down along the diagonal until it meets the next pair of mismatching symbols. Note that this step can be performed in $\mathcal{O}(1)$ time using a longest common extension query. The $(m-1)$-paths in diagonals $i-1$ and $i+1$ are extended in a similar fashion, except that from the end of the $(m-1)$-path in diagonal $i+1$ the algorithm makes a horizontal step (corresponds to a deletion of a symbol of the block) and from the end of the $(m-1)$-path in diagonal $i+1$ the algorithms makes a vertical step (corresponds to an insertion of a symbol). It is not hard to see that in this way the algorithm finds the end of the m-path for a fixed diagonal in $\mathcal{O}(1)$ time, meaning that overall the algorithm uses $\mathcal{O}((B+k)\cdot k)$ time and $\mathcal{O}((B+k)\cdot k)$ space per block.

▶ Remark. Note that the running time of the algorithm can be de-amortised to spend $\mathcal{O}(k)$ time per arrival in the worst case: When we apply the algorithm to a block i for the first time, we de-amortise its running timer over block $i+1$ by running $\Omega(k)$ steps of the algorithm each time a new block symbol arrives, and when we run the algorithm for the second time we de-amortise its running time over block $i+n/B-2$.

3.2 Suffixes

To compute the distances from suffixes of the pattern to the text the algorithm uses sketches by Belazzougui and Zhang [2, Theorem 13].

▶ **Theorem 9** ([2]). *Assume $k < n^{1/c}$ for some sufficiently large constant $c > 0$. There is a sketch of size $\mathcal{O}(k^8 \log^5 n)$ that can be used to compute the edit distance between two binary strings of length at most n in $\mathcal{O}(k^{12} \log^3 n)$ time correctly with probability 0.9. Given a string arriving as a stream its sketch can be constructed in $\mathcal{O}(k^2 \log^4 n)$ amortised time per symbol.*

The space and time bounds are not given in [2, Theorem 13] but can be derived from its proof. We will show the following corollary.

▶ **Corollary 10.** *Assume $k < n^{1/c}$ for some sufficiently large constant $c > 0$. There is a sketch of size $\mathcal{O}(k^8 \log^6 n)$ that can be used to compute the edit distance between two binary strings of length at most n in $\mathcal{O}(k^{12} \log^4 n)$ time correctly with probability $1 - \mathrm{poly}(n)$. Given a string arriving as a stream its sketch can be constructed in $\mathcal{O}(k^2 \log^4 n)$ worst-case time per symbol.*

We boost the probability of Theorem 13 [2] from 0.9 to $1 - \mathrm{poly}(n)$ in a standard way, by repeating the computation independently $\mathcal{O}(\log n)$ times and taking the smallest edit distance as an answer, which yields the extra $\log n$ factors in the complexities.

For completeness and to explain how to de-amortise the time bound, we give the definition of the sketches. The sketches are constructed using a random walk embedding from edit to Hamming distance [4]. The embedding maps strings of length n onto strings of length $3n$. Consider a string S and set a pointer to $S[1]$. At each step, the embedding copies the symbol at which the pointer is currently at to the resulting string $E(S)$ and either moves the pointer to the right or stays in place. After having reached the end of S it stops, and if the length of $E(S)$ is $\ell < 3n$, it appends $3n - \ell$ zeros to it. The moves of the pointer are defined by a random string $R \in \{0,1\}^{6n}$. If i is the current position of the pointer in S, and j is the length of $E(S)$, then the pointer moves to the right if $R[S[i] + 2j] = 1$ and otherwise it stays in place.

▶ **Theorem 11** ([4]). *For every constant $c > 0$ and every pair of binary strings S_1, S_2 of length at most n, the Hamming distance between $E(S_1), E(S_2)$ is at most $c \cdot \big(ED(S_1, S_2)\big)^2$ with probability at least $1 - 12/\sqrt{c}$.*

The intuition behind the proof is that the difference between the pointers' positions as they move along two strings S_1, S_2 behaves as a one-dimensional random walk. In more details, since R is a random binary string, at each time moment when the difference is not zero and there is a mismatch between $E(S_1)$ and $E(S_2)$ the difference does not change with probability $1/2$, increases by one with probability $1/4$, and decreases by one with probability $1/4$.

The mismatched symbols of $E(S_1)$ and $E(S_2)$ and their respective positions in S_1 and S_2 can be used to construct a set of edit operations that transform S_1 to S_2. The set might be not optimal, but it gives some evidence of which positions in S_1 and S_2 must be edited. Belazzougui and Zhang first developed sketches of the embeddings $E(S_1), E(S_2)$ that allow to retrieve both the mismatched symbols and their positions in S_1 and S_2. Their sketches are based on the Hamming distance sketches of Porat and Lipsky [17] and can be constructed in $\mathcal{O}(\log^2 n)$ worst-case time per symbol of an embedding. They further suggested to consider $\mathcal{O}(k^2 \log^2 n)$ independent random walk embeddings and showed that they give enough information to derive the optimal set of edit operations.

To de-amortise the time bound of Theorem 13 [2] we notice that in the random walk embedding a pointer advances by at least one position of the initial string each $3 \log n$ steps with probability at least $1 - 1/n^3$. Therefore if the sketch construction algorithm gets stuck at some position for more than $3 \log n$ steps, we can simply abandon it. This incomplete sketch might result in erroneous outputs, but the probability of this event is small.

3.3 Algorithm

We are now ready to give a full description of the algorithm. We assume that the algorithm first receives the pattern and preprocesses it in a streaming fashion. Namely, it remembers the first $B + k$ symbols of the pattern and also computes sketches of each suffix $P[i, n]$, $i \in [1, B + k]$. The sketches occupy $\mathcal{O}((B + k) \cdot k^8 \log^6 n)$ space in total.

After a new block of the text has arrived, the algorithm computes its encoding defined in Section 3.1. In total all block encodings occupy $\mathcal{O}((n/B) \cdot k \log n)$ space. Also, while reading block i, the algorithm decodes block $i + 2 - n/B$ and runs the algorithm of Section 3.1 to compute the edit distances for the prefixes of P. Recall that this step can be de-amortised to take $\mathcal{O}(k)$ worst-case time per arrival. Finally, the algorithm considers each of the suffixes of the current text that starts at a block border as a separate stream and computes its sketch in a streaming manner. That is, when a new symbol $T[j]$ arrives the algorithm updates each of the $\mathcal{O}(n/B)$ suffix streams and each of its sketches in $\mathcal{O}((n/B) \cdot k^2 \log^4 n)$ time. The suffix sketches occupy $\mathcal{O}((n/B) \cdot k^8 \log^6 n)$ space in total.

We finally explain how the algorithm computes an output for a new arrival $T[j]$ in a block i. Recall that the task is to decide if $T[1, j]$ ends with a k-mismatch occurrence of P and if so to output the edit operations between the pattern and the occurrence. The length of the occurrence must be in $[n - k, n + k]$. It therefore starts either in block $i - n/B$ or in block $i + 1 - n/B$. The two cases are analogous and we consider only the case when the occurrence starts in block $i - n/B$. Let S be the suffix of $T[1, j]$ starting at the right border of block $i - n/B$ (in Fig. 2 the suffix is shown in green). S must be aligned with one of the $2k$ suffixes of the pattern of length in $[|S| - k, |S| + k]$. Using the sketches, we compute the edit distances (and the edit operations) from each of these suffixes to S. Consider a suffix

$P[i+1,n]$. If it is aligned with S, the prefix $P[1,i]$ must be aligned with some suffix of block $i-n/B$ and we have computed the minimal edit distance from $P[1,i]$ to the block or we know that it is larger than k. For each i, we sum the edit distances for the prefix and for the suffix and take the minimum. If the minimum is smaller than k, then by Observation 8 $T[1,j]$ ends with a k-mismatch occurrence of the pattern P and we can output the edit distance and the edit operations. In total, this step takes $\mathcal{O}(k^{13}\log^4 n)$ time.

We choose $B=\sqrt{n}$. The space complexity of the algorithm is then $\mathcal{O}(k^8\sqrt{n}\log^6 n)$. The time for updating the sketches is $\mathcal{O}(k^2\sqrt{n}\log^4 n)$ per arrival, and the time for computing the edit distance is $\mathcal{O}(k^{13}\log^4 n)$, meaning that the total time complexity is $\mathcal{O}((k^2\sqrt{n}+k^{13})\cdot\log^4 n)$ per arrival.

4 Conclusion

In this work we studied the approximate pattern matching problem. In particular we showed the first sublinear-space streaming algorithm for the problem. The space complexity of our algorithm is $\mathcal{O}(k^8\sqrt{n}\log^6 n)$, which is significantly better than that of the previously known solutions. We note that on the other hand the time complexity of our algorithm is quite large as we have to update sketches of $\sqrt{n}$ text suffixes each time a new symbol arrives. One possibility to improve the time complexity is to maintain sketches of the blocks of the text rather than sketches of the suffixes (this way, the algorithm will need to update only one sketch per arrival). However, it is not clear whether the block sketches can be used to compute suffix sketches and therefore the edit distance. This is because the moves of a pointer in a suffix' blocks are not independent, in other words the image of a block under the random walk embedding depends on all preceding blocks. We leave this challenging question for further research.

References

1. Djamal Belazzougui. Efficient deterministic single round document exchange for edit distance, 2015. arXiv:1511.09229.
2. Djamal Belazzougui and Qin Zhang. Edit distance: Sketching, streaming, and document exchange. In Irit Dinur, editor, *Proceedings of the 57th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2016)*, pages 51–60. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.15.
3. Joshua Brakensiek, Venkatesan Guruswami, and Samuel Zbarsky. Efficient low-redundancy codes for correcting multiple deletions. In Robert Krauthgamer, editor, *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2016)*, pages 1884–1892. SIAM, 2016. doi:10.1137/1.9781611974331.ch132.
4. Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2016)*, pages 712–725. ACM, 2016. doi:10.1145/2897518.2897577.
5. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. The k-mismatch problem revisited. In Robert Krauthgamer, editor, *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2016)*, pages 2039–2052. SIAM, 2016. doi:10.1137/1.9781611974331.ch142.
6. Raphaël Clifford, Markus Jalsenius, and Benjamin Sach. Cell-probe bounds for online edit distance and other pattern matching problems. In Piotr Indyk, editor, *Proceedings of the*

26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2015), pages 552–561. SIAM, 2015. doi:10.1137/1.9781611973730.37.

7 Raphaël Clifford and Tatiana Starikovskaya. Approximate Hamming distance in a stream. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *Proceedings of the 43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *LIPIcs*, pages 20:1–20:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.20.

8 Graham Cormode, Mike Paterson, Süleyman Cenk Sahinalp, and Uzi Vishkin. Communication complexity of document exchange. In David B. Shmoys, editor, *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 197–206. ACM/SIAM, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338252.

9 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In Moshe Lewenstein and Gabriel Valiente, editors, *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, volume 4009 of *LNCS*, pages 36–48. Springer, 2006. doi:10.1007/11780441_5.

10 Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

11 Hossein Jowhari. Efficient communication protocols for deciding edit distance. In Leah Epstein and Paolo Ferragina, editors, *Proceedings of the 20th Annual European Symposium on Algorithms (ESA 2012)*, volume 7501 of *LNCS*, pages 648–658. Springer, 2012. doi:10.1007/978-3-642-33090-2_56.

12 Gad M. Landau and Uzi Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing (STOC 1986)*, pages 220–230. ACM, 1986. doi:10.1145/12130.12152.

13 Eugene W. Myers. An $O(nd)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986. doi:10.1007/BF01840446.

14 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001. doi:10.1145/375360.375365.

15 Alon Orlitsky. Interactive communication: Balanced distributions, correlated files, and average-case complexity. In Michael Sipser, editor, *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS 1991)*, pages 228–238. IEEE Computer Society, 1991. doi:10.1109/SFCS.1991.185373.

16 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In Daniel A. Spielman, editor, *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2009)*, pages 315–323. IEEE Computer Society, 2009. doi:10.1109/FOCS.2009.11.

17 Ely Porat and Ohad Lipsky. Improved sketching of Hamming distance with error correcting. In Bin Ma and Kaizhong Zhang, editors, *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM 2007)*, volume 4580 of *LNCS*, pages 173–182. Springer, 2007. doi:10.1007/978-3-540-73437-6_19.

The Longest Filled Common Subsequence Problem

Mauro Castelli[1], Riccardo Dondi[2], Giancarlo Mauri[3], and Italo Zoppis[4]

1 NOVA IMS, Universidade Nova de Lisboa, Lisbon, Portugal
mcastelli@isegi.unl.pt
2 Dipartimento di Lettere, Filosofia, Comunicazione, Università degli Studi di Bergamo, Bergamo, Italy
riccardo.dondi@unibg.it
3 Dipartimento di Informatica, Sistemistica e Comunicazione, Università degli Studi di Milano-Bicocca, Milano, Italy
mauri@disco.unimib.it
4 Dipartimento di Informatica, Sistemistica e Comunicazione, Università degli Studi di Milano-Bicocca, Milano, Italy
zoppis@disco.unimib.it

Abstract

Inspired by a recent approach for genome reconstruction from incomplete data, we consider a variant of the longest common subsequence problem for the comparison of two sequences, one of which is incomplete, i.e. it has some missing elements. The new combinatorial problem, called Longest Filled Common Subsequence, given two sequences A and B, and a multiset $\mathcal{M}$ of symbols missing in B, asks for a sequence B^* obtained by inserting the symbols of $\mathcal{M}$ into B so that B^* induces a common subsequence with A of maximum length.

First, we investigate the computational and approximation complexity of the problem and we show that it is NP-hard and APX-hard when A contains at most two occurrences of each symbol. Then, we give a $\frac{3}{5}$−approximation algorithm for the problem. Finally, we present a fixed-parameter algorithm, when the problem is parameterized by the number of symbols inserted in B that "match" symbols of A.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics

Keywords and phrases longest common subsequence, approximation algorithms, computational complexity, fixed-parameter algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.14

1 Introduction

The comparison of sequences via Longest Common Subsequence (LCS) has been applied in several contexts where we want to retrieve the maximum number of elements that appear in the same order in two or more sequences. There are well-known fields of application of LCS like scheduling and data compression, a notable example is the DIFF utility to compute the differences between two files.

The extraction of common subsequences has been widely applied to compare molecular sequences in bioinformatics [17, 14]. For example, the comparison of biological sequences provides a measure of their similarities and differences, aiming at understanding whether they encode similar/different functionalities. Different approaches for the comparison of two genomes based on LCS have been considered in the last years, leading to variants of the longest common subsequence problem, like the constrained longest common subsequence [13, 8, 18, 11, 4] or the repetition-free longest common subsequence and variants thereof [7, 1, 6, 12].

The approaches based on LCS for genome comparison assume that the input sequences are complete, that is there are no missing data. However, while Next Generation Sequencing technologies are able to produce a huge amount of DNA/RNA fragments, the cost of reconstructing a complete genome is still high [10]. Hence, released genomes often contain errors or are incomplete. These incomplete genomes are called scaffolds. One approach to the reconstruction of genome is to fill scaffolds with missing genes, based on the comparison of an incomplete genome with a reference genome [16, 15, 9, 19]. Given an incomplete genome B, a set of missing genes (symbols) $\mathcal{M}$ and a reference genome A, the goal is to insert the missing symbols in B so that the number of common adjacencies between the resulting genome B^* and A is maximized. We have a common adjacency when two genes a, b are consecutive both in A and B^*, independently from the order. We mention briefly that there is also a variant of the scaffold filling approach that compares two incomplete genomes [15, 9].

Inspired by methods for genome comparison based on LCS and by the scaffold filling approach, we introduce a new variant of the LCS problem, called the *Longest Filled Common Subsequence* problem, for the comparison of a complete genome A and an incomplete genome B. The goal of the problem is to find the maximum number of genes that appear in the same order in both genomes. However, since some of the genes in B are missing (a multiset $\mathcal{M}$ of symbols), we have to compute a longest common subsequence of A and of a filling B^* of B, that is of a sequence obtained from B by inserting the symbols of $\mathcal{M}$ into B. Notice that while the scaffold filling problem aims to reconstruct a complete genome from an incomplete one by maximizing the number of common adjacencies, here we aim to infer only those elements (genes) that appear in the same order in the complete genome A and in the completed genome B^*.

In this paper, we investigate different algorithmic and complexity aspects of the Longest Filled Common Subsequence problem. First, in Section 3 we prove that it is NP-hard and APX-hard, even when genome A contains at most two occurrences of each symbol. Notice that bounding the maximum number of occurrences of symbols in a sequence is relevant in this case, as usually the number of copies of a gene inside a genome is bounded. Then, in Section 4 we present a polynomial-time approximation algorithm of factor $\frac{3}{5}$. In Section 5, we give a fixed-parameter algorithm, where the parameter is the number of inserted symbols that lead to a "match" with symbols of sequence A. Such a parameter can be of interest when the number of missing elements, and in particular those that lead to a "match" with symbols of A, is moderate, as the complexity of the algorithm depends exponentially only on this parameter.

Some of the proofs are omitted due to page limit.

2 Preliminaries

In this section we introduce some basic definitions that will be useful in the rest of the paper and we give the formal definition of the Longest Filled Common Subsequence problem. Let S be a sequence over an alphabet Σ, we denote by $|S|$ the length of S. Given a position i, with $1 \leq i \leq |S|$, we denote by $S[i]$ the symbol in position i of S. Given two positions i, j in S, with $1 \leq i \leq j \leq |S|$, we denote by $S[i, j]$ the substring of S that starts at position i and ends at position j. Given two sequences S and T, we denote by $S \cdot T$ the sequence that results by concatenating S and T.

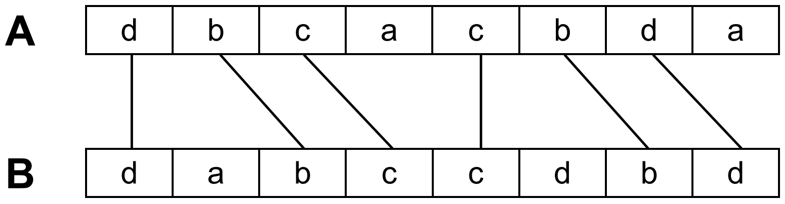


Figure 1 The threading schema of two sequences A and B: lines connect matched positions of A and B.

A subsequence of S is a sequence S' that is obtained from S by deleting some symbols (possibly none). A common subsequence S of two sequences A and B is a subsequence of both A and B. A longest common subsequence of A and B is a common subsequence of A and B having maximum length.

Given two sequences A and B, a common subsequence can be defined by aligning A and B and by connecting two positions of A and B containing an identical symbol with a line, such that there is no pair of crossing lines. This is called a *threading schema* (see Fig. 1). Given a threading schema for sequences A, B, a connection between two symbols in A and B, respectively, is called a *match* and the two positions incident in a line are said to be *matched*.

Given a sequence S and a multiset of symbols $\mathcal{M}$, we define a *filling* of S with $\mathcal{M}$ as a sequence S' obtained by inserting a subset $\mathcal{M}'$ of symbols of $\mathcal{M}$ into S. Notice that in a filling of S with $\mathcal{M}$ not all the symbols of $\mathcal{M}$ have to be inserted in S. Informally, we may not insert those symbols that do not induce matches, to simplify the algorithms we describe in Section 4 and in Section 5. Now, we are ready to present the formal definition of Longest Filled Common Subsequence.

▶ **Problem 1.** *Longest Filled Common Subsequence ($\mathcal{LFCS}$)*
Instance: *two sequences A and B over an alphabet Σ, and a multiset $\mathcal{M}$ over Σ.*
Solution: *a filling B^* of B with $\mathcal{M}$.*
Measure: *the length of a longest common subsequence of A and B^* (to be maximized).*

Given two sequences A, B and a multiset $\mathcal{M}$ over Σ, let B^* be a filling of B with $\mathcal{M}$. Consider a common subsequence of A and B^*, and their corresponding threading schema. A position of A can have two possible kinds of matches (see Fig. 2): a match with a position of B^* that contains a symbol of $\mathcal{M}$ inserted in B, called *match by insertion*, or a match with a position of B^* not involved in an insertion, called *match by alignment*. We can easily compute in polynomial-time two upper bounds on the number of positions of A that can be matched by alignment and by insertion, that will be useful in Section 4. The first upper bound is related to a longest common subsequence L of A and B, which can be computed in polynomial time. In fact, the maximum number of positions of A (and of a filling B^* of B with $\mathcal{M}$) that are matched by alignment is at most the length of L.

Next we show how to compute in polynomial-time an upper bound on the number of positions of a sequence A that can be matched by insertion. First, given a multiset $\mathcal{M}$ of symbols, we define an *ordering of* $\mathcal{M}$ as a sequence obtained by defining an order among each element of $\mathcal{M}$, that is each occurrence of a symbol of $\mathcal{M}$.

Consider the positions of A and of a filling B^* of B with $\mathcal{M}$ that are matched by insertion; the positions of A induce a subsequence A' of A, while the positions of B^* induce an ordering M of a subset $\mathcal{M}' \subseteq \mathcal{M}$. An upper bound on the length of M can be computed in polynomial time with the following greedy algorithm.

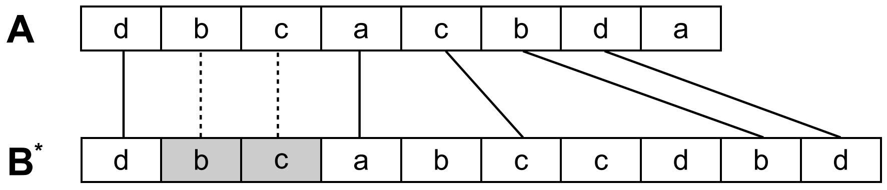


Figure 2 A filling B^* of sequence B in Fig. 1, computed by inserting a symbol in position 2 (symbol b) and a symbol in position 3 (symbol c), both in grey. A subsequence of A and B^* is induced by the threading schema of A and B^*, where straight lines represent matches by alignment, dashed lines represent matches by insertion.

Algorithm 1:

```
Data: $A$, $\mathcal{M}$
Result: a subsequence $A'$ of $A$ that matches the maximum number of symbols of a
        sequence $M$ obtained by ordering $\mathcal{M}$
1 $i := 1$;
2 $A'$ is an empty sequence;
3 while $i \leq |A|$ do
4     if $\alpha \in \mathcal{M}$ with $A[i] = \alpha$ then
5         $A' := A' \cdot \alpha$;
6         $\mathcal{M} := \mathcal{M} \setminus \{\alpha\}$;
7     $i := i++$;
```

Next, we prove the correctness of Algorithm 1.

▶ **Lemma 1.** *Given a sequence A, a multiset $\mathcal{M}$ on Σ, and a substring $A[1,i]$ of A, Algorithm 1 computes a subsequence of $A[1,i]$ that matches the maximum number of symbols of an ordering M of $\mathcal{M}$.*

3 Complexity of $\mathcal{LFCS}$

In this section, we investigate the computational (and approximation) complexity of the $\mathcal{LFCS}$ problem, and we prove that it is APX-hard when A contains at most two occurrences of each symbol in Σ (we denote this restriction of $\mathcal{LFCS}$ by 2-$\mathcal{LFCS}$). We prove the result by an L-reduction from the Maximum Independent Set problem on Cubic Graphs (Max-ISC), which is known to be APX-hard [2](see [5] for details on L-reduction). Max-ISC, given a cubic graph $G = (V, E)$[1], asks for a maximum cardinality subset $V' \subseteq V$ such that given $v_i, v_j \in V'$ it holds $\{v_i, v_j\} \notin E$.

Given a cubic graph $G = (V, E)$, with $V = \{v_1, v_2, \dots, v_n\}$ and $|E| = m$, in the following we show how to construct an instance $(A, B, \mathcal{M})$ of 2-$\mathcal{LFCS}$. Define an order on the edges incident on a vertex $v_i \in V$ assuming $\{v_i, v_j\} < \{v_i, v_h\}$ if $j < h$. Given a vertex v_i, and the edges $\{v_i, v_j\}, \{v_i, v_h\}, \{v_i, v_z\} \in E$, with $j < h < z$, we say that $\{v_i, v_j\}$ ($\{v_i, v_h\}$, $\{v_i, v_z\}$, respectively) is the first (second, third, respectively) edge incident on v_i.

First, we define the alphabet Σ:

$$\Sigma = \{x_{i,j} : v_i \in V, 1 \leq j \leq 3\} \cup \{y_{i,j} : v_i \in V, 1 \leq j \leq 2\} \cup \{z_{i,j} : 1 \leq i \leq n+m-1, 1 \leq j \leq 4\}.$$

[1] We recall that a cubic graph is an undirected graph where each vertex has degree exactly three.

The input sequences A and B are built by concatenating several substrings.

For each $v_i \in V$, we define the following substrings of the input sequences A, B:

$$A(v_i) = y_{i,1}y_{i,2}x_{i,1}x_{i,2}x_{i,3} \qquad B(v_i) = x_{i,1}x_{i,2}x_{i,3}y_{i,1}y_{i,2}\,.$$

For each $\{v_i, v_j\} \in E$, with $i < j$ (which is the p-th edge, $1 \leq p \leq 3$, incident on v_i and the q-th edge, $1 \leq q \leq 3$, incident on v_j), define the following substrings of A, B:

$$A(\{v_i, v_j\}) = x_{i,p}x_{j,q} \qquad B(\{v_i, v_j\}) = x_{j,q}x_{i,p}\,.$$

Finally, define $2(n + m - 1)$ additional substrings $S_{A,1}, S_{A,2}, \dots, S_{A,m+n-1}$, $S_{B,1}, S_{B,2}, \dots,$ $S_{B,m+n-1}$ where $S_{A,i}$, $S_{B,i}$, with $1 \leq i \leq m + n - 1$, are defined as follows:

$$S_{A,i} = S_{B,i} = z_{i,1}z_{i,2}z_{i,3}z_{i,4}.$$

Now, we are able to define the input sequences A and B, by concatenating the substrings previously defined, where substrings associated with edges of G are concatenated assuming some edge ordering (we assume that $\{v_1, v_w\}$ is the first edge, while $\{v_r, v_t\}$ is the last edge according to the ordering):

$$A = A(v_1) \cdot S_{A,1} \cdot A(v_2) \cdot \dots \cdot S_{A,n-1} \cdot A(v_n) \cdot S_{A,n} \cdot A(\{v_1, v_w\}) \cdot \dots \cdot S_{A,n+m-1} \cdot A(\{v_r, v_t\})\,,$$

$$B = B(v_1) \cdot S_{B,1} \cdot B(v_2) \cdot \dots \cdot S_{B,n-1} \cdot B(v_n) \cdot S_{B,n} B(\{v_1, v_w\}) \cdot \dots \cdot S_{B,n+m-1} \cdot B(\{v_r, v_t\})\,.$$

Notice that each substring associated with an edge $\{v_i, v_j\}$ appears exactly once in both A and B.

$\mathcal{M}$ (in this case is a set) is defined as follows: $\mathcal{M} = \{x_{i,t} : v_i \in V, 1 \leq t \leq 3\}$.

First, we prove that $(A, B, \mathcal{M})$ is an instance of 2-$\mathcal{LFCS}$, that is we prove that each symbol has at most two occurrences in A.

▶ **Lemma 2.** *Each symbol of Σ occurs at most twice in A.*

Proof. Notice that each symbol appearing in a substring $S_{A,i}$, $1 \leq i \leq m + n - 1$, does not appear in any other subsequence of A. Now, consider a symbol $y_{i,t}$, $1 \leq i \leq n$ and $1 \leq t \leq 2$, appearing in substring $A(v_i)$; $y_{i,t}$ does not appear in any other substring of A. Finally, consider a symbol $x_{i,t}$, $1 \leq i \leq n$ and $1 \leq t \leq 3$; $x_{i,t}$ has one occurrence in exactly two subsequences of A: subsequence $A(v_i)$ and subsequence $A(\{v_i, v_j\})$ (where $\{v_i, v_j\}$ is the t-th edges incident on v_i). ◀

Let B^* be a solution of 2-$\mathcal{LFCS}$ over instance $(A, B, \mathcal{M})$. We denote by $S_{B^*,i}$ ($B^*(v_i)$, $B^*(\{v_i, v_j\})$, respectively), the substring of a solution B^* corresponding (after some insertion) to the substring $S_{B,i}$ ($B(v_i)$,$B(\{v_i, v_j\})$, respectively), of B.

Next, we show that we can assume that in a solution B^* of 2-$\mathcal{LFCS}$ over instance $(A, B, \mathcal{M})$, a longest common subsequence of A and B^* matches by alignment a position of a subsequence $S_{A,i}$, $1 \leq i \leq m + n - 1$, only with a position of $S_{B^*,i}$, $1 \leq i \leq m + n - 1$.

▶ **Lemma 3.** *Given a cubic graph G, let $(A, B, \mathcal{M})$ be the corresponding instance of 2-$\mathcal{LFCS}$, and B^* a solution of 2-$\mathcal{LFCS}$ over $(A, B, \mathcal{M})$. Then a longest common subsequence of A and B^* contains each symbol $z_{t,q}$, with $1 \leq t \leq m + n - 1$ and $1 \leq q \leq 4$.*

Proof. Consider a solution B^* of 2-$\mathcal{LFCS}$ over instance $(A, B, \mathcal{M})$ and assume that it does not contain a symbol $z_{t,q}$, with $1 \leq t \leq m + n - 1$ and $1 \leq q \leq 4$. By construction a longest common subsequence of B^* and A matches by alignment a position of $A(v_i)$ either with a position of $B(v_i)$ or with a position of $B(\{v_i, v_j\})$.

First, we prove that a longest common subsequence between A and B^* matches by alignment a position of $A(v_i)$ only with a position of $B(v_i)$. Assume that i is the minimum value such that a longest common subsequence S of A and B^* matches by alignment a position of $A(v_i)$ and a position of $B^*(\{v_i, v_j\})$. Notice that, by construction of $(A, B, \mathcal{M})$, no position of $S_{A,i}$ can be matched. Now, starting from S we can compute a common subsequence S' of A and B^*, with $|S'| > |S|$, by modifying the alignment of S as follows: (i) match by alignment the positions of $A(v_i)$ and the positions of $B^*(v_i)$ containing symbols $y_{i,1}, y_{i,2}$; (ii) match by alignment the positions of subsequences $S_{A,i}$ containing symbol $z_{i,q}$ with position of subsequences $S_{B,i}$ containing symbol $z_{i,q}$; (iii) any other match is not modified. It follows that the number of positions in $A(v_i)$ matched by S' with respect to S is decreased by at most three, since eventually positions of $A(V_i)$ containing symbols $x_{i,1}$, $x_{i,2}$, $x_{i,3}$ will not be matched. The number of positions in $S_{A,i}$ matched by S' with respect to S is increased by at least 4, since each position of $S_{A,i}$ is not matched by S and it is matched by S'. By iterating this procedure, we eventually find a longest common subsequence S' of A and B^* where if each position of $A(V_i)$ is matched by alignment, then it is matched with a position of $B(v_i)$. By the maximality of S', this implies that each position of A containing a symbol $z_{t,q}$, with $1 \leq t \leq m + n - 1$ and $1 \leq q \leq 4$, matches a position of B^* containing symbol $z_{t,q}$. ◀

Consider a vertex $v_i \in V$ and the corresponding substrings $A(v_i)$, $B(v_i)$ of A and B. Moreover, let $\{v_i, v_j\}, \{v_i, v_h\}, \{v_i, v_z\} \in E$ be the three edges of G incident on v_i and consider the corresponding substrings $A(\{v_i, v_j\})$, $A(\{v_i, v_h\})$, $A(\{v_i, v_z\})$ ($B(\{v_i, v_j\})$, $B(\{v_i, v_h\})$, $B(\{v_i, v_z\})$, respectively), of A (of B, respectively). Informally, the reduction shows that there are essentially two possible configurations (called *I-configuration* and *C-configuration*) of the substring $B^*(v_i)$ (and possibly $B^*(\{v_i, v_j\})$, $B^*(\{v_i, v_h\})$ and $B^*(\{v_i, v_z\})$) of a filling B^* of B. A substring $B^*(v_i)$ having an I-configuration is related to the vertex v_i in an independent set of G, while a substring $B^*(v_i)$ having a C-configuration is related to the vertex v_i in a vertex cover of G.

We define now the two possible configurations, called *I-configuration* and *C-configuration*, for $B^*(v_i)$ and, possibly, for the substrings $B^*(\{v_i, v_j\})$, $B^*(\{v_i, v_h\})$ and $B^*(\{v_i, v_z\})$ of a filling B^* of B. An *I-configuration* for the substrings $B^*(v_i)$, $B^*(\{v_i, v_j\})$, $B^*(\{v_i, v_h\})$ and $B^*(\{v_i, v_z\})$ is defined as follows:

- $B^*(v_i) = B(v_i)$ (hence there is no insertion in $B(v_i)$).
- For each $\{v_i, v_t\}$, with $t \in \{j, h, z\}$, where $\{v_i, v_t\}$ is the p-th edge incident on v_i, $1 \leq p \leq 3$, and the q-th edge incident on v_t, $1 \leq q \leq 3$, $B^*(\{v_i, v_t\}) = x_{i,p}x_{j,q}x_{i,p}$ (hence $x_{i,p}$ is inserted in $B(\{v_i, v_t\})$).

If $B^*(v_i)$, $B^*(\{v_i, v_j\})$, $B^*(\{v_i, v_h\})$, $B^*(\{v_i, v_z\})$ have an *I-configuration*, a longest common subsequence of $B^*(v_i)$ and $A(v_i)$ has length three (it matches the positions containing $x_{i,1}$, $x_{i,2}$, $x_{i,3}$), and a longest common subsequence of $A(\{v_i, v_t\})$ and $B^*(\{v_i, v_t\})$, with $t \in \{j, h, z\}$, has length two (it matches the positions containing $x_{i,p}$, $x_{j,q}$).

A *C-configuration* for the substring $B^*(v_i)$ is defined as follows:

- $B^*(v_i) = x_{i,1}x_{i,2}x_{i,3}y_{i,1}y_{i,2}x_{i,1}x_{i,2}x_{i,3}$ (hence $B^*(v_i) = B(v_i) \cdot x_{i,1}x_{i,2}x_{i,3}$).

If $B^*(v_i)$ has a *C-configuration*, a longest common subsequence of $B^*(v_i)$ and $A(v_i)$ has length five, it matches the positions containing $y_{i,1}$, $y_{i,2}$, $x_{i,1}$, $x_{i,2}$, $x_{i,3}$.

Next, we present the main lemmata of this section.

▶ **Lemma 4.** *Let G be a cubic graph, instance of Max-ISC, and let $(A, B, \mathcal{M})$ be the corresponding instance of 2-$\mathcal{LFCS}$. Then, given an independent set I of G, we can compute* *in polynomial time a solution B^* of 2-$\mathcal{LFCS}$ over instance $(A, B, \mathcal{M})$ inducing a longest common subsequence with A of length $4(m+n-1)+6|I|+5(n-|I|)+m$.*

Proof. Consider an independent set I and define a solution B^* of 2-$\mathcal{LFCS}$ over instance $(A, B, \mathcal{M})$ as follows. For each $v_i \in I$, with $\{v_i, v_j\}, \{v_i, v_h\}, \{v_i, v_z\} \in E$ the three edges of G incident on v_i, define an *I-configuration* for $B^*(v_i)$, $B^*(\{v_i, v_j\})$, $B^*(\{v_i, v_h\})$, $B^*(\{v_i, v_z\})$. For each $v_i \in V \setminus I$, define a *C-configuration* for $B^*(v_i)$. For each edge $\{v_i, v_j\} \in E$ if $v_i, v_j \in V \setminus I$, then $B^*(\{v_i, v_j\}) = B(\{v_i, v_j\})$; notice that in this case a longest common subsequence of $A(\{v_i, v_j\})$ and $B^*(\{v_i, v_j\})$ has length one, as it matches exactly one position containing either $x_{i,p}$ or $x_{j,q}$. Finally, each position of A in the substring $S_{A,i}$, with $1 \le i \le m+n-1$, is matched by alignment with the corresponding position of $S_{B^*,i}$.

Notice that the solution B^* is well-defined, as each $B^*(\{v_i, v_j\})$, with $\{v_i, v_j\} \in E$, can belong to an *I-configuration* of at most one of $B^*(v_i)$ and $B^*(v_j)$, since at most one of v_i, v_j belongs to I.

Now, consider a longest common subsequence S of A and B^*. S matches $4(m+n-1)$ positions in substrings $S_{A,1}, \ldots, S_{A,m+n-1}$, since all the positions of these substrings are matched and, by construction, the overall length of $S_{A,1}, \ldots, S_{A,m+n-1}$ is $4(m+n-1)$. Moreover, by definition of *I-configuration* and *C-configuration*, for each $v_i \in I$, S matches 3 positions of $A(v_i)$ and 2 positions of each $A(\{v_i, v_j\})$, with $\{v_i, v_j\} \in E$; for each $v_i \in V \setminus I$, S matches 5 positions of $A(v_i)$; for each $\{v_i, v_j\} \in E$, with $v_i, v_j \in V \setminus I$, S matches one position of $A(\{v_i, v_t\})$. Hence, S matches $4(m+n-1)+6|I|+5(n-|I|)+m$ positions of A and B^*. ◀

Based on Lemma 3, we can prove the following result.

▶ **Lemma 5.** *Let G be a cubic graph, instance of Max-ISC, and let $(A, B, \mathcal{M})$ be the corresponding instance of 2-$\mathcal{LFCS}$. Then, given a solution B^* of 2-$\mathcal{LFCS}$ over instance $(A, B, \mathcal{M})$ of length $4(m+n-1)+6p+5(n-p)+m$, we can compute in polynomial time an independent set of G of size at least p.*

By Lemmata 4 and 5, and by the APX-hardness of Max-ISC [2] we can conclude that the 2-$\mathcal{LFCS}$ problem is APX-hard.

▶ **Theorem 6.** *2-$\mathcal{LFCS}$ is APX-hard.*

4 Approximating $\mathcal{LFCS}$

In this section we give a polynomial-time approximation algorithm for $\mathcal{LFCS}$ of factor $\frac{3}{5}$. The approximation algorithm picks the largest number of matched positions returned by two polynomial-time algorithms, Approx-Algorithm-1 and Approx-Algorithm-2. Notice that each algorithm does not return a filling of B with $\mathcal{M}$, but two disjoint subsets of positions of A that have to be matched by alignment and by insertion, respectively, by a subsequence of A and of a filling of B with $\mathcal{M}$. We can easily compute in polynomial time a filling B^* of B with $\mathcal{M}$ so that there exists a common subsequence of A and B^* that matches these two subsets of positions.

Both algorithms consist of two phases.

Approx-Algorithm-1. In the first phase, Approx-Algorithm-1 computes in polynomial time a longest common subsequence of A and B. Denote by $R_{1,a}$ the positions of A matched by alignment in the first phase and by A' the subsequence of A obtained by removing the positions of $R_{1,a}$. The second phase greedily computes in polynomial time a set $R_{1,i}$ of

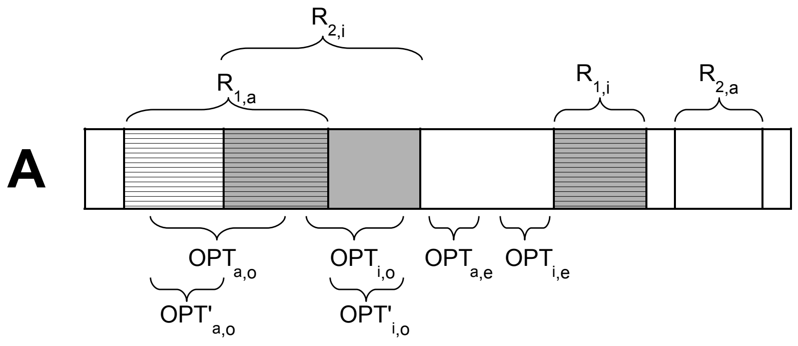


Figure 3 The input sequence A and the positions matched by solution R_1 (dashed) and by solution R_2 (in grey). In the upper part, brackets represent the subsets $R_{1,a}$ and $R_{1,i}$ of R_1, and $R_{2,a}$ and $R_{2,i}$ of R_2. In the lower part, the brackets represent the positions matched by OPT.

positions of A' of maximum size that matches $\mathcal{M}$ by insertion, applying Algorithm 1 on $(A', \mathcal{M})$. Denote by $R_1 = R_{1,a} \cup R_{1,i}$ the set of positions returned by Approx-Algorithm-1.

Approx-Algorithm-2. In the first phase, Approx-Algorithm-2 computes a subset $R_{2,i}$ of positions of A of maximum size that matches $\mathcal{M}$ by insertion applying Algorithm 1 on $(A, \mathcal{M})$. Denote by A'' the subsequence of A obtained by removing the positions of $R_{2,i}$. The second phase computes a longest common subsequence of B and A''; denote by $R_{2,a}$ the set of positions of A'' (and A) matched by this phase. Denote by $R_2 = R_{2,a} \cup R_{2,i}$ the set of positions returned by Approx-Algorithm-2.

Next, we show that the maximum number of positions matched by one of Approx-Algorithm-1 and Approx-Algorithm-2 gives a $\frac{3}{5}$-approximated solution. First, we introduce some notations (see Fig. 3). Let B^{opt} be an optimal solution of $\mathcal{LFCS}$ on instance $(A, B, \mathcal{M})$, and let OPT be a longest common subsequence of A and B^{opt}. We consider the following sets of positions of OPT. Denote by OPT_a the set of positions of A matched by alignment in OPT and by OPT_i the set of positions of A matched by insertion in OPT. Notice that by construction it holds $OPT_a \cap OPT_i = \emptyset$.

Define $OPT_{a,o} = OPT_a \cap (R_{1,a} \cup R_{2,i})$ and $OPT_{i,o} = OPT_i \cap (R_{1,a} \cup R_{2,i})$. Moreover, define $OPT_{a,e} = OPT_a \setminus OPT_{a,o}$ and $OPT_{i,e} = OPT_i \setminus OPT_{i,o}$. Informally, $OPT_{a,e}$ ($OPT_{i,e}$, respectively) is the set of positions of A matched by alignment (by insertion, respectively) in OPT that is not matched in the first phase by Approx-Algorithm-1 (in the second phase by Approx-Algorithm-2, respectively). Finally, define $OPT'_{i,o} = OPT_{i,o} \setminus R_{1,a}$ and $OPT'_{a,o} = OPT_{a,o} \setminus R_{2,i}$.

By definition of OPT, $OPT_{a,o}$, $OPT_{i,o}$, $OPT_{a,e}$ and $OPT_{i,e}$, it holds $|OPT| = |OPT_{a,o}| + |OPT_{a,e}| + |OPT_{i,o}| + |OPT_{i,e}|$.

We will show that the largest set between R_1 and R_2 gives a $\frac{3}{5}$-approximate solution, that is $\max(|R_1|, |R_2|) \geq \frac{3}{5}|OPT|$. We start by showing two bounds on OPT_i and OPT_a.

▶ **Lemma 7.** *$|R_{1,a}| \geq |OPT_a|$ and $|R_{2,i}| \geq |OPT_i|$.*

Proof. First, we prove that $|R_{1,a}| \geq |OPT_a|$. Consider the set of positions in OPT_a. Since each position in OPT_a is a position of A matched by alignment, it follows that the set OPT_a induces a common subsequence of A and B. Since the set $R_{1,a}$ of positions of A induces a longest common subsequence of A and B, it follows that $|R_{1,a}| \geq |OPT_a|$.

Now, we prove that $|R_{2,i}| \geq |OPT_i|$. Consider the set of positions in OPT_i. Each position in OPT_i is matched by insertion, hence it is matched with an inserted symbol of $\mathcal{M}$. By

Lemma 1, $R_{2,i}$ is a set of positions of A of maximum cardinality that can be matched by insertion with symbols of $\mathcal{M}$, hence $|R_{2,i}| \geq |OPT_i|$. ◀

As a consequence of Lemma 7, it follows that $|R_{1,a}| + |R_{2,i}| \geq |OPT_i| + |OPT_a| \geq |OPT|$. Hence the maximum of R_1, R_2 is (at least) $\frac{1}{2}|OPT|$. In the following, we show with a more refined analysis that the maximum of $|R_1|$, $|R_2|$ is at least $\frac{3}{5}|OPT|$.

We prove some bounds on $R_{1,i}$ and $R_{2,a}$, then we consider three cases depending on the values of $OPT_{a,o}$, $OPT_{i,o}$, $OPT_{a,e}$, $OPT_{i,e}$, $OPT'_{i,o}$ and $OPT'_{a,o}$. First, the following result holds.

▶ **Lemma 8.** $|R_{1,i}| \geq |OPT'_{i,o}| + |OPT_{i,e}|$ *and* $|R_{2,a}| \geq |OPT'_{a,o}| + |OPT_{a,e}|$.

Now, in the analysis of the approximation factor of Approx-Algorithm-1 and Approx-Algorithm-2, we consider three cases, depending on the values of $OPT_{i,e}$, $OPT_{i,o}$, $OPT'_{i,o}$.

Case 1

Assume that $|OPT_{i,e}| + |OPT'_{i,o}| \geq \frac{1}{2}|OPT_{i,o}|$, we show the following result.

▶ **Lemma 9.** *Assume that* $|OPT_{i,e}| + |OPT'_{i,o}| \geq \frac{1}{2}|OPT_{i,o}|$*, then* $|R_1| \geq \frac{3}{5}|OPT|$.

Proof. Since $|R_{1,i}| \geq |OPT'_{i,o}| + |OPT_{i,e}|$ by Lemma 8, it follows that

$$|R_{1,a}| + |R_{1,i}| \geq |R_{1,a}| + |OPT'_{i,o}| + |OPT_{i,e}| \geq$$

$$\frac{3}{5}(|R_{1,a}| + |OPT_{i,e}|) + \frac{2}{5}(|R_{1,a}| + |OPT_{i,e}|) + |OPT'_{i,o}|.$$

By Lemma 7 it follows that $|R_{1,a}| \geq |OPT_a|$ and, since $|OPT_a| = |OPT_{a,o}| + |OPT_{a,e}|$, it follows that $|R_{1,a}| \geq |OPT_{a,o}| + |OPT_{a,e}|$, hence

$$\frac{3}{5}(|R_{1,a}| + |OPT_{i,e}|) + \frac{2}{5}(|R_{1,a}| + |OPT_{i,e}|) + |OPT'_{i,o}| \geq$$

$$\frac{3}{5}(|OPT_{a,o}| + |OPT_{a,e}| + |OPT_{i,e}|) + \frac{2}{5}(|R_{1,a}| + |OPT_{i,e}|) + |OPT'_{i,o}|.$$

Hence, it holds

$$|R_{1,a}| + |R_{1,i}| \geq \frac{3}{5}(|OPT_{a,o}| + |OPT_{a,e}| + |OPT_{i,e}|) + \frac{2}{5}(|R_{1,a}| + |OPT_{i,e}|) + |OPT'_{i,o}|. \quad (1)$$

Notice that $|R_{1,a}| + |OPT'_{i,o}| \geq |OPT_{i,o}|$, since, by construction, each position in $OPT_{i,o}$ is either in $OPT'_{i,o}$ or in $R_{1,a}$. Then,

$$\frac{2}{5}(|R_{1,a}| + |OPT'_{i,o}|) \geq \frac{2}{5}|OPT_{i,o}|. \quad (2)$$

Since we are assuming that $|OPT_{i,e}| + |OPT'_{i,o}| \geq \frac{1}{2}|OPT_{i,o}|$, it holds

$$\frac{2}{5}(|OPT_{i,e}| + |OPT'_{i,o}|) \geq \frac{1}{5}|OPT_{i,o}|. \quad (3)$$

Combining Inequalities 2 and 3 with Inequality 1, we can conclude that, under the hypothesis $|OPT_{i,e}| + |OPT'_{i,o}| \geq \frac{1}{2}|OPT_{i,o}|$, it holds

$$|R_{1,a}| + |R_{1,i}| \geq \frac{3}{5}(|OPT_{a,o}| + |OPT_{a,e}| + |OPT_{i,e}|) + \frac{2}{5}(|R_{1,a}| + |OPT_{i,e}|) + |OPT'_{i,o}| \geq$$

$$\frac{3}{5}(|OPT_{a,o}| + |OPT_{a,e}| + |OPT_{i,e}|) + \frac{2}{5}(|R_{1,a}| + |OPT'_{i,o}|) + \frac{2}{5}(|OPT_{i,e}| + |OPT'_{i,o}|) \geq$$

$$\frac{3}{5}(|OPT_{a,o}| + |OPT_{a,e}| + |OPT_{i,o}| + |OPT_{i,e}|).$$

It follows that, under the hypothesis $|OPT_{i,e}| + |OPT'_{i,o}| \geq \frac{1}{2}|OPT_{i,o}|$, it holds $|R_1| \geq \frac{3}{5}|OPT|$. ◀

Case 2

Assume that $|OPT_{a,e}| + |OPT'_{a,o}| \geq \frac{1}{2}|OPT_{a,o}|$. Similarly to Case 1, we can prove the following result.

▶ **Lemma 10.** *Assume that* $|OPT_{a,e}| + |OPT'_{a,o}| \geq \frac{1}{2}|OPT_{a,o}|$, *then* $|R_2| \geq \frac{3}{5}|OPT|$.

Case 3

Assume that both Case 1 and Case 2 do not hold. Then,

$$|OPT_{i,e}| + |OPT'_{i,o}| < \frac{1}{2}|OPT_{i,o}| \text{ and } |OPT_{a,e}| + |OPT'_{a,o}| < \frac{1}{2}|OPT_{a,o}|.$$

Since $|OPT_{i,e}| + |OPT'_{i,o}| < \frac{1}{2}|OPT_{i,o}|$, it follows that $|OPT_{i,e}| < \frac{1}{2}|OPT_{i,o}|$ and, since $|OPT_{a,e}| + |OPT'_{a,o}| < \frac{1}{2}|OPT_{a,o}|$, it follows that $|OPT_{a,e}| < \frac{1}{2}|OPT_{a,o}|$. But then, since $|OPT| = |OPT_{a,o}| + |OPT_{i,o}| + |OPT_{a,e}| + |OPT_{i,e}|$, it follows that

$$|OPT| \leq \frac{3}{2}(|OPT_{a,o}| + |OPT_{i,o}|)$$

We show that $|R_1| \geq |OPT_{a,o}| + |OPT_{i,o}|$, thus implying that $|R_1| \geq \frac{3}{5}|OPT|$.

▶ **Lemma 11.** $|R_{1,a} \cup R_{1,i}| \geq |OPT_{a,o}| + |OPT_{i,o}|$.

By Lemma 11, $|R_{1,a} \cup R_{1,i}| \geq |OPT_{a,o}| + |OPT_{i,o}|$. Since in this case we have shown that $|OPT| \leq \frac{3}{2}(|OPT_{a,o}| + |OPT_{i,o}|)$, it follows that $|R_1| = |R_{1,a} \cup R_{1,i}| \geq \frac{2}{3}|OPT| \geq \frac{3}{5}|OPT|$. From Lemma 9, Lemma 10 and Lemma 11, it follows the main result of this section.

▶ **Theorem 12.** *Given an instance* $(A, B, \mathcal{M})$ *of* $\mathcal{LFCS}$, *the largest solution returned by Approx-Algorithm-1 and Approx-Algorithm-2 is an approximate solution of factor* $\frac{3}{5}$.

Proof. From Lemma 9, Lemma 10 and Lemma 11, it follows that $\max(|R_1|, R_2|) \geq \frac{3}{5}|OPT|$.

We can compute a filling B_1 of B with $\mathcal{M}$ that matches at least $|R_1|$ positions with A as follows: we consider the positions in $R_{1,a}$ as matched by alignment, we insert symbols of $\mathcal{M}$ in B in order to match by insertion the positions in $R_{1,i}$. It follows that a longest common subsequence of A and B_1 matches at least $|R_1|$ positions.

Similarly, we can compute a filling B_2 of B with $\mathcal{M}$ that matches at least $|R_2|$ positions of A. We insert symbols of $\mathcal{M}$ in B so that the positions in $R_{1,i}$ are matched by insertion. Consider the subsequence A'' obtained after the removal of positions in $R_{1,i}$; a longest common subsequence of A'' and B matches at least $|R_{2,a}|$ positions. It follows that a longest common subsequence of A and B_2 matches at least $|R_2|$ positions. ◀

5 An FPT Algorithm

In this section, we present an FPT algorithm for $\mathcal{LFCS}$ parameterized by the number k of positions of A matched by insertions. Notice that $k < |\mathcal{M}|$. Here we assume that the input sequences A and B have been extended by adding two symbols $\$_A, \$_B \notin \Sigma$, respectively, in position 0 of A and B, respectively. Hence we assume that position 0 of A and of a filling B^* of B with $\mathcal{M}$ is not matched by alignment or by insertion by any solution of $\mathcal{LFCS}$ of length greater than zero.

The algorithm we present is based on the color-coding technique [3]. Next, we present the definition of perfect families of hash functions for a multiset of symbols, on which our color-coding approach is based.

▶ **Definition 13.** Let $\mathcal{M}$ be a multiset of positions and let F be a family of hash functions from $\mathcal{M}$ to a set $\{c_1, \dots, c_k\}$ of colors. F is called *perfect* if for any subset $W \subseteq \mathcal{M}$, such that $|W| = k$, there exists a function $f \in F$ which is injective on W.

A perfect family F of hash functions from $\mathcal{M}$ to $\{c_1, \dots, c_k\}$, having size $O(\log |\mathcal{M}| 2^{O(k)})$, can be constructed in time $O(2^{O(k)} |\mathcal{M}| \log |\mathcal{M}|)$ (see [3]).

Consider a perfect family of hash functions $F : \mathcal{M} \to \{c_1, \dots, c_k\}$. Let $f \in F$ be an injective function, and define $L[i, j, C, l]$, with $C \subseteq \{c_1, \dots, c_k\}$, $0 \leq i, l \leq |A|$ and $0 \leq j \leq |B|$, as follows:

- $L[i, j, C, l] = 1$ if and only if there exists a common subsequence of $A[0, i]$ and of a filling B^* of $B[0, j]$ with $\mathcal{M}$ having length l, such that there exist $|C|$ symbols of $\mathcal{M}$ inserted in $B[0, j]$, each one associated with a distinct color of C and matched by insertion with a position of A
- else $L[i, j, C, l] = 0$.

Next, we define the recurrence to compute $L[i, j, C, l]$, where $i \geq 1$ and $j \geq 0$.

$$L[i,j,C,l] = \max \begin{cases} L[i-1, j, C, l] & \\ L[i, j-1, C, l] & \text{if } j \geq 1 \\ L[i-1, j-1, C, l-1] & \text{if } A[i] = B[j] \text{ and } j \geq 1 \\ L[i-1, j, C \setminus \{c\}, l-1] & \text{if } A[i] = \alpha \text{ and there exists} \\ & \alpha \in \mathcal{M} \text{ with } f(\alpha) = c \in C \end{cases} \tag{4}$$

For the base case, since we have extended A and B so that position 0 in A and in the filling of B cannot be matched by insertions or by alignment, it holds $L[0, 0, C, l] = 1$, if $C = \emptyset$ and $l = 0$, else $L[0, 0, C, l] = 0$. Next, we prove the correctness of the recurrence.

▶ **Lemma 14.** *Let $F : \mathcal{M} \to \{c_1, \dots, c_k\}$ be a perfect family of hash functions, let $f \in F$ be an injective function and let C be a subset of $\{c_1, \dots, c_k\}$. Then there exists a common subsequence of length l, $l \geq 0$, of $A[0, i]$, $0 \leq i \leq |A|$, and of a filling of $B[0, j]$, $0 \leq j \leq |B|$, with $\mathcal{M}' \subseteq \mathcal{M}$ such that each symbol of $\mathcal{M}'$ matched by insertion is associated with a distinct color in C if and only if $L[i, j, C, l] = 1$.*

Now, we are able to prove the main result of this section.

▶ **Theorem 15.** *Let A, B be two sequences and $\mathcal{M}$ a mutliset of symbols. Then it is possible to compute in time $2^{O(k)} poly(|A| + |B| + |\mathcal{M}|)$ if there exists a solution B^* of $\mathcal{LFCS}$ over instance $(A, B, \mathcal{M})$ such that a longest common subsequence S of A and B^* has length l and it matches by insertion k positions of A.*

Proof. The correctness of the algorithm follows from Lemma 14 and from the fact that entry $L[|A|, |B|, C, l] = 1$ if and only if there exists a solution of $\mathcal{LFCS}$ over instance $(A, B, \mathcal{M})$ having length l that matches by insertion k positions of A.

Next, we consider the time complexity of the algorithm. A perfect family of hash functions that color-codes the symbols of $\mathcal{M}$ can be computed in time $2^{O(k)} poly(|\mathcal{M}|)$. Then, the algorithm iterates through $2^{O(k)} poly(|\mathcal{M}|)$ color-codings. For each color-coding, the table $L[i, j, C, l]$ is computed in time $O(2^k |A|^2 |B| k)$ (where $l \leq |A|$), since for each of the at most $O(2^k |A|^2 |B|)$ entries we need to look for at most k possible entries. The overall complexity is then $2^{O(k)} poly(|A| + |B| + |\mathcal{M}|)$. ◀

6 Conclusion

We have introduced a variant of the LCS problem, called Longest Filled Common Subsequence ($\mathcal{LFCS}$), to compare a sequence A with an incomplete sequence B to be filled with a multiset $\mathcal{M}$ of symbols. We have shown that the problem is APX-hard (hence NP-hard), even when each symbol occurs at most twice in the input sequence A. Then, we have given an approximation algorithm of factor $\frac{3}{5}$ and a fixed-parameter algorithm, where the parameter is the number of symbols in $\mathcal{M}$ matched by insertion.

There are some interesting open problems related to $\mathcal{LFCS}$. It would be interesting to extend $\mathcal{LFCS}$ to the comparison of two incomplete sequences, similar to what has been done for Scaffold Filling [15]. Moreover, it would be interesting to design more efficient parameterized algorithms for $\mathcal{LFCS}$, for example by considering the algebraic technique used for the repetition-free longest common subsequence [6]. Another open problem is whether $\mathcal{LFCS}$ is NP-hard on a constant size alphabet.

References

1 Said Sadique Adi, Marília D. V. Braga, Cristina G. Fernandes, Carlos Eduardo Ferreira, Fábio Viduani Martinez, Marie-France Sagot, Marco A. Stefanes, Christian Tjandraatmadja, and Yoshiko Wakabayashi. Repetition-free longest common subsequence. *Discrete Appl. Math.*, 158(12):1315–1324, 2010. doi:10.1016/j.dam.2009.04.023.

2 Paola Alimonti and Viggo Kann. Some apx-completeness results for cubic graphs. *Theor. Comput. Sci.*, 237(1-2):123–134, 2000. doi:10.1016/S0304-3975(98)00158-3.

3 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995. doi:10.1145/210332.210337.

4 Abdullah N. Arslan and Ömer Egecioglu. Algorithms for the constrained longest common subsequence problems. *Int. J. Found. Comput. Sci.*, 16(6):1099–1109, 2005. doi:10.1142/S0129054105003674.

5 Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties.* Springer-Verlag, Heidelberg, 1999. doi:10.1007/978-3-642-58412-1.

6 Guillaume Blin, Paola Bonizzoni, Riccardo Dondi, and Florian Sikora. On the parameterized complexity of the repetition free longest common subsequence problem. *Inf. Process. Lett.*, 112(7):272–276, 2012. doi:10.1016/j.ipl.2011.12.009.

7 Paola Bonizzoni, Gianluca Della Vedova, Riccardo Dondi, Guillaume Fertin, Raffaella Rizzi, and Stéphane Vialette. Exemplar longest common subsequence. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 4(4):535–543, 2007. doi:10.1145/1322075.1322078.

8 Paola Bonizzoni, Gianluca Della Vedova, Riccardo Dondi, and Yuri Pirola. Variants of constrained longest common subsequence. *Inf. Process. Lett.*, 110(20):877–881, 2010. `doi:10.1016/j.ipl.2010.07.015`.

9 Laurent Bulteau, Anna Paola Carrieri, and Riccardo Dondi. Fixed-parameter algorithms for scaffold filling. *Theor. Comput. Sci.*, 568:72–83, 2015. `doi:10.1016/j.tcs.2014.12.005`.

10 P. G. S. Chain and et al. Genomics. Genome project standards in a new era of sequencing. *Science*, 326:236–237, 2009. `doi:10.1126/SCIENCE.1180614`.

11 Francis Y. L. Chin, Alfredo De Santis, Anna Lisa Ferrara, N. L. Ho, and S. K. Kim. A simple algorithm for the constrained sequence problems. *Inf. Process. Lett.*, 90(4):175–179, 2004. `doi:10.1016/j.ipl.2004.02.008`.

12 Carlos Eduardo Ferreira and Christian Tjandraatmadja. A branch-and-cut approach to the repetition-free longest common subsequence problem. *Electron. Notes Discrete Math.*, 36:527–534, 2010. `doi:10.1016/j.endm.2010.05.067`.

13 Zvi Gotthilf, Danny Hermelin, and Moshe Lewenstein. Constrained LCS: hardness and approximation. In Paolo Ferragina and Gad M. Landau, editors, *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM 2008)*, volume 5029 of *LNCS*, pages 255–262. Springer, 2008. `doi:10.1007/978-3-540-69068-9_24`.

14 Tao Jiang and Ming Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM J. Comput.*, 24(5):1122–1139, 1995. `doi:10.1137/S009753979223842X`.

15 Nan Liu, Haitao Jiang, Daming Zhu, and Binhai Zhu. An improved approximation algorithm for scaffold filling to maximize the common adjacencies. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 10(4):905–913, 2013. `doi:10.1109/TCBB.2013.100`.

16 Adriana Muñoz, Chunfang Zheng, Qian Zhu, Victor A. Albert, Steve Rounsley, and David Sankoff. Scaffold filling, contig fusion and comparative gene order inference. *BMC Bioinformatics*, 11:304, 2010. `doi:10.1186/1471-2105-11-304`.

17 Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, 1981. `doi:10.1016/0022-2836(81)90087-5`.

18 Yin-Te Tsai. The constrained longest common subsequence problem. *Inf. Process. Lett.*, 88(4):173–176, 2003. `doi:10.1016/j.ipl.2003.07.001`.

19 Binhai Zhu. Genomic scaffold filling: A progress report. In Daming Zhu and Sergey Bereg, editors, *Proceedings of the 10th International Workshop on Frontiers in Algorithmics (FAW 2016)*, volume 9711 of *LNCS*, pages 8–16. Springer, 2016. `doi:10.1007/978-3-319-39817-4_2`.

Lempel-Ziv Compression in a Sliding Window

Philip Bille*[1], Patrick Hagge Cording†[2], Johannes Fischer[3], and Inge Li Gørtz‡[4]

1 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
`phbi@dtu.dk`

2 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
`phaco@dtu.dk`

3 **Technische Universität Dortmund, Department of Computer Science, Dortmund, Germany**
`johannes.fischer@cs.tu-dortmund.de`

4 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
`inge@dtu.dk`

Abstract

We present new algorithms for the sliding window Lempel-Ziv (LZ77) problem and the approximate rightmost LZ77 parsing problem.

Our main result is a new and surprisingly simple algorithm that computes the sliding window LZ77 parse in $O(w)$ space and either $O(n)$ expected time or $O(n \log\log w + z \log\log \sigma)$ deterministic time. Here, w is the window size, n is the size of the input string, z is the number of phrases in the parse, and σ is the size of the alphabet. This matches the space and time bounds of previous results while removing constant size restrictions on the alphabet size.

To achieve our result, we combine a simple modification and augmentation of the suffix tree with periodicity properties of sliding windows. We also apply this new technique to obtain an algorithm for the approximate rightmost LZ77 problem that uses $O(n(\log z + \log\log n))$ time and $O(n)$ space and produces a $(1+\epsilon)$-approximation of the rightmost parsing (any constant $\epsilon > 0$). While this does not improve the best known time-space trade-offs for exact rightmost parsing, our algorithm is significantly simpler and exposes a direct connection between sliding window parsing and the approximate rightmost matching problem.

1998 ACM Subject Classification E.4 Coding and Information Theory, E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Lempel-Ziv parsing, sliding window, rightmost matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.15

1 Introduction

The Lempel-Ziv parsing (LZ77) [36] of a string is a key component in data compression, detecting regularities in strings, pattern matching, and string indexing. LZ77 is the basis for several popular compression tools such as `gzip` and `7zip`, and is shown to compress well under certain measures of compressibility [21].

In general terms, given an input string S of length n, the LZ77 parsing divides S into z substrings $f_1 f_2 \dots f_z$, called *phrases*, in a greedy left-to-right order. The ith phrase f_i

* Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).
† Supported by the Danish Research Council (DFF – 4005-00267).
‡ Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).

starting at position p_i is either (a) the first occurrence of a character in S or (b) the longest substring that has at least one occurrence starting to the left of p_i. To compress S, we can then replace each phrase f_i of type (b) with a pair (r_i, l_i) such that r_i is the distance from p_i to the start of the previous occurrence, and l_i is the length of the previous occurrence. (This is actually the LZ77-variant of Storer and Szymanski [33]; the original one [36] adds a character to each phrase so that it outputs *triples* instead of tuples.)

Computing the LZ77 parse is a very well-studied problem. The simplest way to compute the parse is to build an index for the input string, and scan the string left-to-right looking for the longest prefix of the current suffix occurring to the left of the current position. Using a suffix tree to index the string this yields $O(n)$ time and space algorithm. Research on LZ77 parsing algorithms has since branched into practical and space-efficient computation [4, 12, 14, 16, 17, 19, 20, 22, 27, 30], parallel [31] and external computation [18], online parsing [28, 29, 32, 35], approximation of the parse [10], and algorithms that find the rightmost occurrence of a phrase [1, 8].

Almost all of the existing algorithms maintain an index of the entire input string, whereas almost all practical solutions only maintain a short *window* of length w, for some parameter w, of the input string near the current position in the string. This produces a *sliding window LZ77 parse* [2, 33] that has the property that a previous occurrence of a phrase starts no longer than w characters away from the current position. This limits the number of potential longest matches of a phrase to at most w and also reduces the number of bits needed to encode the reference to f_i. Our main result is a new technique for indexing a sliding window. Using the technique we obtain an algorithm for LZ77 sliding window parsing that improves the previous best known time and space bounds for integer alphabets (and matches the known bounds for constant alphabets). The algorithm is surprisingly simple.

We then turn our heads to rightmost LZ77 parsing. The greedy LZ77 parse is optimal in terms of the number of phrases [6]. However, if we use variable length encoding of the phrases we may reduce the number of bits needed to encode each phrase. By choosing to reference the rightmost occurrence for each phrase we minimize the number of bits needed to encode the greedy LZ77 parse. Though several efficient algorithms for computing the rightmost parse are known, most require highly non-trivial algorithmic techniques. As an interesting application of our technique for sliding window LZ77 parsing, we obtain a very simple efficient *approximate* rightmost parsing algorithm. Interestingly, this algorithm exposes a direct connection between sliding window parsing and the approximate rightmost matching problem.

In the remainder of this section we will formally state the problems, our results, and discuss previous work.

1.1 Sliding Window Parsing

Given a parameter w, the *sliding window LZ77 parse* (SWLZ77) of a string S is the LZ77 parse with the added requirement that the previous occurrence of a phrase f_i starts within distance at most w from the start of f_i. To limit memory consumption, the SWLZ77 parse is used in most compression tools based on LZ77 in practice.

Fiala and Greene [9] and Larsson [24] show how to efficiently maintain the suffix tree of a sliding window of size w. This immediately leads to an algorithm for computing the SWLZ77 parse in $O(n)$ time and $O(w)$ space. However, the algorithms are based on McCreight's [25] and Ukkonen's [34] suffix algorithms, respectively, and thus assume that the size of the alphabet is constant. (The same restriction on the alphabet size holds for the *w-truncated suffix tree* by Na et al.[26].) Furthermore, the algorithms require non-trivial modifications of the classic suffix tree algorithms and are thus quite complicated. In practice, a hash table is used for strings in the dictionary, often sacrificing optimality for speed (see e.g. [23] for a survey on this).

In this paper we show the following result.

▶ **Theorem 1.** *Let S be a string of length n over an alphabet of size σ. Given a parameter w, we can compute the sliding window Lempel-Ziv parse in*
(i) *$O(w)$ space and $O(n)$ expected time, or*
(ii) *$O(w)$ space and $O(\frac{n}{w}\mathrm{sort}(w,\sigma) + z\log\log\sigma)$ deterministic worst-case time.*
Here, z is number of phrases in the parsing, and $\mathrm{sort}(w,\sigma)$ *is the time for sorting w characters from an alphabet of size σ.*

Hence, compared to the previous bounds, Theorem 1(i) matches the previous space bounds while achieving linear expected time and with no restrictions on the alphabet size. If we require a deterministic bound, Theorem 1(ii) incurs a small overhead. Plugging in the currently fastest deterministic sorting algorithm [15], which uses $O(w\log\log w)$ time to sort w characters from an arbitrary alphabet, the bound becomes $O(n\log\log w + z\log\log\sigma)$. We note that the additive overhead of $O(z\,\mathrm{loglog}\,\sigma)$ is $O(n)$ for most combinations of σ, n, and z.

The main technical challenges in the result are restricting the search for a previous occurrence to a dynamic window and supporting searches for self-referential phrases of length $> w$ in $O(w)$ space. To achieve this, we present a simple modification and augmentation of suffix trees, which we call w-sliding window trees, that supports linear time searching within a window and show how to exploit periodicity properties of windows to compactly search for long phrases in $O(w)$ space. However, any text indexing data structure that supports basic suffix tree navigation operation can replace the w-sliding window tree in our solution if different time-space trade-offs are required.

1.2 Approximate Rightmost Parsing

Let $\hat{r}_i$ denote the smallest possible choice of reference r_i, i.e., $\hat{r}_i$ is the distance to rightmost substring matching p_i that begins before p_i in S. If $r_i = \hat{r}_i$, $i = 1,\ldots,z$ the parsing is *rightmost* and if $\hat{r}_i \le r_i \le c\cdot\hat{r}_i$ for some $c > 1$ the parsing is *c-rightmost*. The *rightmost parsing problem* is to compute the rightmost parsing, and the *approximate rightmost parsing problem* is to compute a c-rightmost parsing for some $c > 1$.

Ferragina et al. [8] present an algorithm for the rightmost parsing problem with running time $O(n(1+\frac{\log\sigma}{\log\log n}))$ and using $O(n)$ space. Recently, this was improved to $O(n(1+\frac{\log\sigma}{\sqrt{\log n}})$ time and $O(n\log\sigma)$ *bits* of space by Belazzougui and Puglisi [1]. Prior to these results, Crochemore et al. [5] presented, to the best of our knowledge, the only approximate rightmost parsing algorithm. Their algorithm runs in $O(n\log n)$ time and $O(n)$ space and it finds the *rightmost equal-cost position* (REP) for each phrase in the greedy LZ77 parse. The REP for a phrase f_i is some occurrence for which r_i requires the same number of bits to encode as $\hat{r}_i$. If $\hat{r}_i$ is a power of 2 the algorithm finds an occurrence where $r_i \le 2\hat{r}_i - 1$, i.e., roughly speaking their algorithm is producing a 2-rightmost parsing.

All of the above solutions require several highly non-trivial components to achieve their bounds. We show how our solution to the Lempel-Ziv sliding window problem immediately leads to an efficient approximate rightmost parsing algorithm summarized in the following theorem.

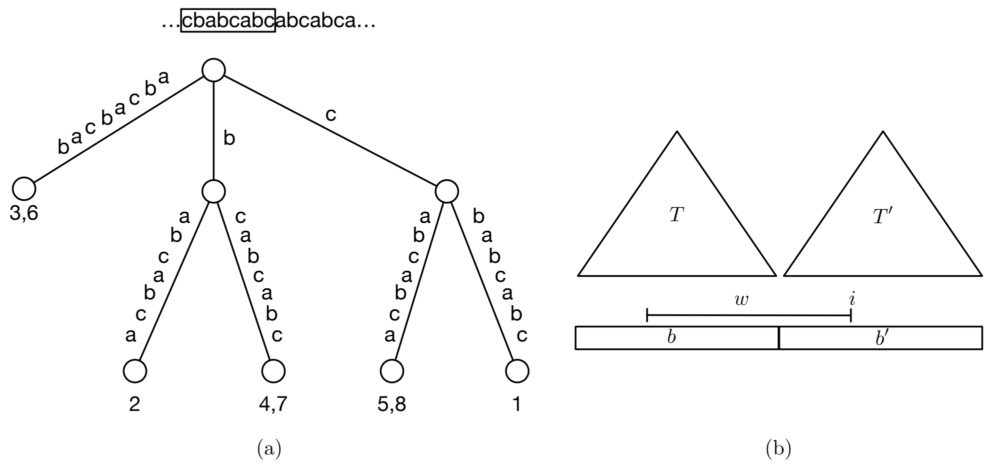


Figure 1 (a) The w-sliding window tree for window size $w = 8$. Text positions at the leaves are relative to the end e of the previous sliding window, implying they must be incremented by e to get absolute positions. (b) Parsing with w-sliding window trees T and T' for blocks b and b'.

▶ **Theorem 2.** *Let S be a string of length n. For any $\varepsilon > 0$ we can compute a $(1+\varepsilon)$-rightmost Lempel-Ziv parsing in $O(n \log z + n \log\log n)$ time and $O(n)$ space, where z is the number of phrases in the parse.*

While our result does not improve the best known trade-offs for rightmost parsing, the algorithm is significantly simpler. It applies our new technique of combining w-sliding window trees and periodicity properties and thereby exposes a direct connection between sliding windows and approximate rightmost matching problems.

2 Lempel-Ziv in a Sliding Window

We now show Theorem 1. Throughout the paper, let S be a string of length n over an alphabet of size σ. We partition S into blocks of size w and parse S from left to right. For these blocks we store a special suffix tree, which we call a *w-sliding window tree.*

▶ **Definition 3** (The w-sliding window tree)**.** The w-sliding window tree of a block is the compact trie of all length w strings starting in the block. Each leaf stores all starting positions of the substring it represents. For each edge e in the w-sliding window trees we store the minimum starting position, $\min(e)$, and the maximum starting position, $\max(e)$, stored in all leaves below it. The w-sliding window tree at position i is the w-sliding window tree for the block starting at position i.

See Figure 1(a) for an example of a w-sliding window tree. We have that the w-sliding window uses $O(w)$ space. Also, given the suffix tree of the string of length $2w$ starting at position i, we can easily build w-sliding window tree starting at position i in $O(w)$ time by truncating all suffixes to length w.

While showing Theorem 1 in the following sections, we will also show the following Lemma that we will need for our approximate rightmost matching algorithm. Given two indices i and j in S, let $\operatorname{lcp}(i,j)$ denote the length of the longest common prefix of $S[i,n]$ and $S[j,n]$.

▶ **Lemma 4.** *Given two w-sliding window trees at position x and $x+w$, respectively, we can find $\ell = \max_{i-w \le j < i} \operatorname{lcp}(i,j)$ for any $i \in [x+w, x+2w)$ in $O(\ell)$ time (assuming the suffix-trees support constant-time top-down-traversals).*

For simplicity, we first consider the case where the length of each phrase is at most w, and then extend the result to handle arbitrary length phrases.

2.1 Bounded Phrase Length

We first show how to find longest matches if the length of each phrase is at most w. We partition S into blocks of size w and parse S from left-to-right. We only maintain the last two blocks in memory.

2.1.1 Parsing

We implement the sliding window parsing using the w-sliding window trees as follows. See also Figure 1(b). Suppose we have parsed the first $i-1$ characters of S and currently have the w-sliding window trees T and T' for the last two blocks b and b' stored. To compute the phrase starting at position i, we traverse T and T' top-down according to the substring starting at position i. In T, we compare i to $\max(e)$ each time we follow edge e. If $\max(e)$ is within the window (if $\max(e) \geq i - w$) we continue the search and otherwise we stop the search. If we reach a leaf we also stop. When the search stops, we output $\max(e)$ of the previous edge e as the starting position of the longest match in T. In T' we compare with $\min(e)$ in the same fashion. That is, we only continue the search if $\min(e)$ is smaller than i. We output $\min(e)$ of the previous edge e as the starting position of the longest match in T'. We return the maximum of the longest matching path found in T and T' as the longest matching substring within the window.

2.1.2 Correctness

We argue that the algorithm correctly finds a longest match. A longest match within the window must start in one of the two blocks b or b'. Since we only continue the search in T as long as $\max(e)$ is in the window, the match that we found starts at a position in the window. Similarly for T'.

2.2 Unbounded Phrase Length

We now consider the general case of unbounded phrase length and show how to extend the solution from the previous section to handle this case by exploiting a periodicity property of the sliding window [3].

Given a w-sliding window tree we now might reach a leaf, from where we need to continue the matching further. If there is only one substring stored at the leaf we can simply continue matching the corresponding substrings in S until the phrase ends. Unfortunately, we may have multiple strings stored at a leaf and thus we cannot afford naively matching against these.

We modify searching for longest match starting at position i as follows. We match in the w-sliding window trees T and T' just as before. If we reach a leaf in T we pick the maximum starting position x stored in the leaf ($x = \max(e)$ if e is the incoming edge) and continue matching from position $i + w$ and $x + w$ until we get a mismatch. If we reach a leaf in T' we pick the minimum starting position stored in the leaf (using $\min(e)$) and continue matching in the same fashion.

2.2.1 Correctness and Analysis

To show that the modification works correctly we will show the following. Let f_i be a phrase of length $> w$ starting at position p_i. If there are multiple strings in the w-sliding window tree that start in the window $S[p_i - w, p_i - 1]$ and match the first w characters of f_i, then *all* of these strings can be extended to longest matches with f_i. In particular, since the algorithm chooses one of these string to compare against (the maximum or minimum such string) this implies that the algorithm is correct.

We need the following lemma.

▶ **Lemma 5** (Breslauer and Galil [3], Lemma 3.1). *Let P and S be strings such that S contains at least three occurrences of P. Let $t_1 < t_2 < \cdots < t_h$ be the locations of all occurrences of P in S and assume that $t_{i+2} - t_i \leq |P|$, for $i = 1, \ldots, h-2$ and $h \geq 3$. Then, this sequence forms an arithmetic progression with difference $d = t_{i+1} - t_i$, for $i = 1, \ldots, h-1$, that is equal to the period length of P.*

Using Lemma 5 we show the following result.

▶ **Lemma 6.** *Assume we reach a leaf in the w-sliding window tree T (or T') when matching the phrase starting at p_i, and that there are $k \geq 2$ strings that are associated with the leaf and start in the window $S[p_i - w, p_i - 1]$. Let $t_1 < \cdots < t_k$ be the starting positions of the strings. Then $\mathrm{lcp}(p_i, t_j) = l_i$ for all $j = 1, \ldots, k$.*

Proof. The starting positions $t_1, \ldots, t_k$ correspond to starting positions of occurrences of the w-length substring $S[p_i, p_i + w - 1]$. Since they all start in the window $S[p_i - w, p_i - 1]$ we have $p_i - w \leq t_j < p_i$ for all $j = 1, \ldots, k$. Therefore $t_{j+2} - t_j \leq w$ for all $1 \leq j \leq k - 2$ and also $p_i - t_{k-1} \leq w$, and it follows from Lemma 5 that the sequence $t_1, t_2, \ldots, t_k, p_i$ forms an arithmetic progression, i.e., the substring $S[p_i, p_i + w - 1]$ is periodic with period length $d = p_i - t_k$. The suffix $S[p_i, n]$ starts with $r \geq 1$ whole repetitions of the period followed by possibly a prefix of the period of length $r' < d$. Let $l_i = rd + r'$. All the suffixes $S[t_1, n], \ldots S[t_k, n]$ start with strictly more than r repetitions of the period. Therefore, they all match with $S[p_i, n]$ up to position $p_i + l_i - 1$. Thus, continuing matching from any of these when we reach the leaf in T will give us the correct answer. The proof for the case where we reach a leaf in T' is similar. ◀

In summary, this proves that the algorithm finds the longest match and thus correctly computes the SWLZ77 parse.

2.3 Implementation and Analysis

The total space for the two w-sliding window trees stored at any time is $O(w)$. Building the w-sliding window trees requires building a suffix tree of size $2w$. If the alphabet size is polynomial ($\sigma = n^{O(1)}$) we can build all $\lceil n/w \rceil$ suffix trees in total $O(n)$ worst-case time [7]. If the alphabet size is larger we first hash to a polynomial sized alphabet and then build the suffix trees. This takes $O(n)$ expected time. Given the suffix tree for a $2w$ length substring we construct the w-sliding window tree in $O(w)$ time and use perfect hashing at each node [13] to index the first characters of outgoing edges and thus enable linear time matching (building the perfect hash tables takes expected $O(w)$ time). This concludes the proof of Lemma 4.

In total we get $O(n)$ expected time for constructing all w-sliding window trees, and $O(n)$ time for searching for all phrases. This sums to $O(n)$ expected time as desired. This proves Theorem 1(i).

To get deterministic bounds of Theorem 1(ii), we can instead build the suffix trees using Fischer and Gawrychowski [11]. These build suffix trees in sorting time complexity and support searches for a pattern of length m in $O(m + \log\log\sigma)$ time. We search for z phrases of total length n, and hence in total we use $O(\frac{n}{w}\text{sort}(w,\sigma) + z\log\log\sigma)$ time. In summary, this proves Theorem 1.

3 Approximate Rightmost Matching

We now show Theorem 2.

3.1 Algorithm

We assume that we have the leftmost LZ77 parse (defined analogously to the rightmost parse, see beginning of Section 1.2) of the input string. If not we can easily compute it within the bounds of Theorem 2. Our algorithm updates the references of the phrases in a left-to-right order.

For *levels* $i = 1, \ldots, \log_{(1+\epsilon)} n$ we build the w-sliding window trees for $S[j(1+\epsilon)^i, (j+1)(1+\epsilon)^i]$ for $j = 0, \ldots, \frac{n}{(1+\epsilon)^i} - 1$. That is, for a fixed i, we compute all the w-sliding window trees of size $(1+\epsilon)^i$ spaced by $(1+\epsilon)^i$ characters (remember $\epsilon > 0$ is an arbitrary constant).

▶ **Definition 7** (Covering w-sliding window tree)**.** Given a position k in S such that $j(1+\epsilon)^i \leq k \leq (j+1)(1+\epsilon)^i$ for some i and j, we say that the w-sliding window tree of the substring $S[j(1+\epsilon)^i, (j+1)(1+\epsilon)^i]$ is covering k on level i. We denote this tree by $T_{i,k}$.

We maintain references to the w-sliding window trees such that given k and i we can find the w-sliding window tree on level i covering k in constant time.

To update a reference of a phrase f_l beginning at position p_l to be the $(1+\epsilon)$-rightmost, we search the w-sliding window trees $T_{i,p_l-(1+\epsilon)^i}$ and T_{i,p_l}, $i = 1, \ldots, \log_{(1+\epsilon)} n$, for the occurrence of f_l closest to (but not after) p_l. We then update the phrase f_l. The search is done as described in Lemma 4.

The order in which the w-sliding window trees are searched is a binary search over the levels. If an occurrence is found at level i we continue the search on the smaller levels. If not, we continue the search on the bigger levels.

We assumed that all w-sliding window trees were built as the first step of the algorithm, but we can restrict the algorithm to only build the w-sliding window trees that are in fact needed and then discard them again when the algorithm progresses to a position not covered by it. In the proof of Theorem 2 we show that this improves the total time used to construct the w-sliding window trees from $O(n\log n)$ to $O(n\log z)$ and the space usage to $O(n)$.

3.2 Analysis

In this section we prove Theorem 2. We start by showing the approximation guarantees of our algorithm and then we analyze its space and time complexity.

3.2.1 Approximation

Here we prove that our algorithm produces a $(1+\epsilon)$-rightmost parsing.

▶ **Lemma 8.** *Let $f_1, \ldots, f_z = (r_1, l_1), \ldots, (r_z, l_z)$ be the parsing produced by our algorithm. For $k = 1, \ldots, z$ we have that $r_k \leq (1+\epsilon)\hat{r}_k$, i.e., our algorithm produces a $(1+\epsilon)$-rightmost parsing.*

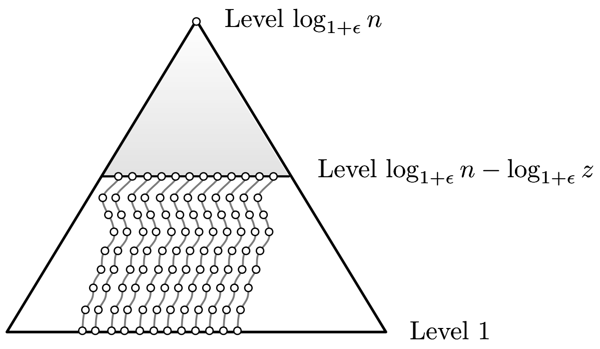

Level $\log_{1+\epsilon} n$
Level $\log_{1+\epsilon} n - \log_{1+\epsilon} z$
Level 1

Figure 2 Suppose the hierarchy of w-sliding window trees is represented by a $(1+\epsilon)$-ary tree as shown in this figure. Consider the case where all phrases are exactly of size $\frac{n}{z}$. In this case all w-sliding window trees of size $(1+\epsilon)^{\log_{1+\epsilon} n - \log_{1+\epsilon} z} = \frac{n}{z}$ (represented by the nodes on level $\log_{1+\epsilon} n - \log_{1+\epsilon} z$) have to be built. Furthermore, all trees represented by ancestor nodes (in the shaded part of the tree) are also built. The total time to do this is $O(n \log z)$. Now suppose that all w-sliding window trees built on the levels from 1 to $\log_{1+\epsilon} n - \log_{1+\epsilon} z - 1$ form disjoint paths in the tree as shown in the figure. We then have to build each tree represented by each node, but the sizes of these are exponentially decreasing as the levels decrease, and the total work therefore sums to $O(n)$.

Proof. Consider a phrase f_k starting at position p_k. Suppose the search for an occurrence of f_k terminates on level i. This means that there is an occurrence in T_{i,p_k}, but not in T_{i-1,p_k}. Since the algorithm disregards matches starting before $p_k - (1+\epsilon)^i$ we therefore have that $(1+\epsilon)^{i-1} < \hat{r}_k \leq r_k \leq (1+\epsilon)^i$ from which it follows that $r_k \leq (1+\epsilon)\hat{r}_k$. ◀

3.2.2 Space

The space used by our algorithm is dominated by the w-sliding window trees we construct. Once we are done using a w-sliding window tree we can discard it. When processing f_l we only need the w-sliding window trees covering p_l and $p_l - (1+\epsilon)^i$ for every level i. So at any point in time the total size of the maintained w-sliding window trees is bounded by $\sum_{i=1}^{\log_{1+\epsilon} n} O((1+\epsilon)^i) = O(n)$, hence the space usage by our algorithm is $O(n)$.

3.2.3 Time

First we analyze the time required for constructing the w-sliding window trees. Recall that a suffix tree is only constructed if we actually need to access it. For each phrase beginning we may have to access $\log \log_{1+\epsilon} n$ w-sliding window trees, however some of these may be reused. Our algorithm parses the string left to right, so if a w-sliding window tree is covering both p_l and p_{l+1} we only need to construct it once since we process f_{l+1} immediately after f_l.

In the worst case, we may be required to build all w-sliding window trees on level $\log_{1+\epsilon} n - \log_{1+\epsilon} z$, meaning that all possible w-sliding window trees on level $1 + \log_{1+\epsilon} n - \log_{1+\epsilon} z$ to $\log_{1+\epsilon} n$ will also have to be built. This requires $O(n \log z)$ time since the total size of the w-sliding window trees on any level is $O(n)$ and the number of levels is $\log_{1+\epsilon} z$. In the worst case the w-sliding window trees on the remaining levels are not subject to reuse. On level $\log_{1+\epsilon} n - \log_{1+\epsilon} z$ the total size of the w-sliding window trees is $O(n)$. On the previous level we also need to build at most z w-sliding window trees but the total size of these will be $O(n/(1+\epsilon))$. Therefore the total size of the w-sliding window trees on the lower levels is at most $\sum_{i=1}^{\log_{1+\epsilon} n - \log_{1+\epsilon} z} n/(1+\epsilon)^i = O(n)$. The total time for building the the w-sliding window trees is thus $O(n \log z + n) = O(n \log z)$ time. See also Figure 2.

We now look at the time it takes to search the w-sliding window trees. Consider phrase f_l and assume that we have all the w-sliding window trees covering p_l. We binary search for the w-sliding window tree having an occurrence of f_l as close to p_l as possible. Since there are $\log_{1+\epsilon} n$ levels this takes $O(|f_l| \log\log n)$ time, resulting in a total of $\sum_{i=1}^{z} |f_i| \log\log n = O(n \log\log n)$ time for this step. In total our algorithm uses $O(n(\log z + \log\log n))$ time. This concludes the proof of Theorem 2.

References

1 Djamal Belazzougui and Simon J. Puglisi. Range predecessor and Lempel-Ziv parsing. In Robert Krauthgamer, editor, *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2016)*, pages 2053–2071. SIAM, 2016. doi:10.1137/1.9781611974331.ch143.

2 Timothy C. Bell. Better OPM/L text compression. *IEEE Trans. Commun.*, 34(12):1176–1182, 1986. doi:10.1109/TCOM.1986.1096485.

3 Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Trans. Algorithms*, 10(4):22:1–22:12, 2014. doi:10.1145/2635814.

4 Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In James A. Storer and Michael W. Marcellin, editors, *Proceedings of the 2008 Data Compression Conference (DCC 2008)*, pages 482–488. IEEE Computer Society, 2008. doi:10.1109/DCC.2008.36.

5 Maxime Crochemore, Alessio Langiu, and Filippo Mignosi. The rightmost equal-cost position problem. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2013 Data Compression Conference (DCC 2013)*, pages 421–430. IEEE, 2013. doi:10.1109/DCC.2013.50.

6 Maxime Crochemore, Alessio Langiu, and Filippo Mignosi. Note on the greedy parsing optimality for dictionary-based text compression. *Theor. Comput. Sci.*, 525:55–59, 2014. doi:10.1016/j.tcs.2014.01.013.

7 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

8 Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of Lempel-Ziv compression. *SIAM J. Comput.*, 42(4):1521–1541, 2013. doi:10.1137/120869511.

9 Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989. doi:10.1145/63334.63341.

10 Johannes Fischer, Travis Gagie, Paweł Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In Nikhil Bansal and Irene Finocchi, editors, *Proc. of the 23rd Annual European Symposium on Algorithms (ESA 2015)*, volume 9294 of *LNCS*, pages 533–544. Springer, 2015. doi:10.1007/978-3-662-48350-3_45.

11 Johannes Fischer and Paweł Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proc. of the 26th Annual Symp. on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 160–171. Springer, 2015. doi:10.1007/978-3-319-19929-0_14.

12 Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel-Ziv computation in small space (LZ-CISS). In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proc. of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 172–184. Springer, 2015. doi:10.1007/978-3-319-19929-0_15.

13 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

14 Keisuke Goto and Hideo Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2014 Data Compression Conference (DCC 2014)*, pages 163–172. IEEE, 2014. doi:10.1109/DCC.2014.62.

15 Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. In John H. Reif, editor, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC 2002)*, pages 602–608. ACM, 2002. doi:10.1145/509907.509993.

16 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Proc. of the 12th International Symposium on Experimental Algorithms (SEA 2013)*, volume 7933 of *LNCS*, pages 139–150. Springer, 2013. doi:10.1007/978-3-642-38527-8_14.

17 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In Johannes Fischer and Peter Sanders, editors, *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013. doi:10.1007/978-3-642-38905-4_19.

18 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2014 Data Compression Conference (DCC 2014)*, pages 153–162. IEEE, 2014. doi:10.1109/DCC.2014.78.

19 Dominik Kempa and Simon J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In Peter Sanders and Norbert Zeh, editors, *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments (ALENEX 2013)*, pages 103–112. SIAM, 2013. doi:10.1137/1.9781611972931.9.

20 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2016 Data Compression Conference (DCC 2016)*, pages 3–12. IEEE, 2016. doi:10.1109/DCC.2016.38.

21 S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM J. Comput.*, 29(3):893–911, 1999. doi:10.1137/S0097539797331105.

22 Dmitry Kosolobov. Faster lightweight Lempel-Ziv parsing. In Giuseppe F. Italiano, Giovanni Pighizzini, and Donald Sannella, editors, *Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science (MFCS 2015)*, volume 9235 of *LNCS*, pages 432–444. Springer, 2015. doi:10.1007/978-3-662-48054-0_36.

23 Alessio Langiu. On parsing optimality for dictionary-based text compression – the Zip case. *J. Discrete Algorithms*, 20:65–70, 2013. doi:10.1016/j.jda.2013.04.001.

24 N. Jesper Larsson. Extended application of suffix trees to data compression. In James A. Storer and Martin Cohn, editors, *Proceedings of the 1996 Data Compression Conference (DCC 1996)*, pages 190–199. IEEE Computer Society, 1996. doi:10.1109/DCC.1996.488324.

25 Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

26 Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 1-3(304):87–101, 2003. doi:10.1016/S0304-3975(03)00053-7.

27 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In Raffaele Giancarlo and Giovanni Manzini, editors, *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*, volume 6661 of *LNCS*, pages 15–26. Springer, 2011. doi:10.1007/978-3-642-21458-5_4.

28 Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In Dan Halperin and Kurt Mehlhorn, editors, *Proceedings of the 16th Annual European Symposium on Algorithms (ESA 2008)*, volume 5193 of *LNCS*, pages 696–707. Springer, 2008. `doi:10.1007/978-3-540-87744-8_58`.

29 Alberto Policriti and Nicola Prezza. Fast online Lempel-Ziv factorization in compressed space. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *Proceedings of the 22nd International Symposium on String Processing and Information Retrieval (SPIRE 2015)*, volume 9309 of *LNCS*, pages 13–20. Springer, 2015. `doi:10.1007/978-3-319-23826-5_2`.

30 Alberto Policriti and Nicola Prezza. Computing LZ77 in run-compressed space. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2016 Data Compression Conference (DCC 2016)*, pages 23–32. IEEE, 2016. `doi:10.1109/DCC.2016.30`.

31 Julian Shun and Fuyao Zhao. Practical parallel Lempel-Ziv factorization. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2013 Data Compression Conference (DCC 2013)*, pages 123–132. IEEE, 2013. `doi:10.1109/DCC.2013.20`.

32 Tatiana Starikovskaya. Computing Lempel-Ziv factorization online. In Branislav Rovan, Vladimiro Sassone, and Peter Widmayer, editors, *Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science (MFCS 2012)*, volume 7464 of *LNCS*, pages 789–799. Springer, 2012. `doi:10.1007/978-3-642-32589-2_68`.

33 James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982. `doi:10.1145/322344.322346`.

34 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. `doi:10.1007/BF01206331`.

35 Jun-ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster compact on-line Lempel-Ziv factorization. In Ernst W. Mayr and Natacha Portier, editors, *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *LIPIcs*, pages 675–686. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2014. `doi:10.4230/LIPIcs.STACS.2014.675`.

36 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. `doi:10.1109/TIT.1977.1055714`.

Time-Space Trade-Offs for Lempel-Ziv Compressed Indexing

Philip Bille*[1], Mikko Berggren Ettienne†[2], Inge Li Gørtz‡[3], and Hjalte Wedel Vildhøj[4]

1 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
`phbi@dtu.dk`

2 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
`miet@dtu.dk`

3 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
`inge@dtu.dk`

4 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
`hwvi@dtu.dk`

Abstract

Given a string S, the *compressed indexing problem* is to preprocess S into a compressed representation that supports fast *substring queries*. The goal is to use little space relative to the compressed size of S while supporting fast queries. We present a compressed index based on the Lempel-Ziv 1977 compression scheme. Let n, and z denote the size of the input string, and the compressed LZ77 string, respectively. We obtain the following time-space trade-offs. Given a pattern string P of length m, we can solve the problem in

(i) $O(m + \mathrm{occ} \lg\lg n)$ time using $O(z \lg(n/z) \lg\lg z)$ space, or

(ii) $O(m(1 + \frac{\lg^\epsilon z}{\lg(n/z)}) + \mathrm{occ}(\lg\lg n + \lg^\epsilon z))$ time using $O(z \lg(n/z))$ space, for any $0 < \epsilon < 1$

In particular, (i) improves the leading term in the query time of the previous best solution from $O(m \lg m)$ to $O(m)$ at the cost of increasing the space by a factor $\lg\lg z$. Alternatively, (ii) matches the previous best space bound, but has a leading term in the query time of $O(m(1 + \frac{\lg^\epsilon z}{\lg(n/z)}))$. However, for any polynomial compression ratio, i.e., $z = O(n^{1-\delta})$, for constant $\delta > 0$, this becomes $O(m)$. Our index also supports extraction of any substring of length ℓ in $O(\ell + \lg(n/z))$ time. Technically, our results are obtained by novel extensions and combinations of existing data structures of independent interest, including a new batched variant of weak prefix search.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, E.4 Coding and Information Theory, E.1 Data Structures

Keywords and phrases compressed indexing, pattern matching, LZ77, prefix search

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.16

1 Introduction

Given a string S, the *compressed indexing problem* is to preprocess S into a compressed representation that supports fast *substring queries*, that is, given a string P, report all occurrences of substrings in S that match P. Here the compressed representation can be any

* Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).
† Supported by the Danish Research Council (DFF – 4005-00267).
‡ Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).

compression scheme or measure (kth order entropy, smallest grammar, Lempel-Ziv, etc.). The goal is to use little space relative to the compressed size of S while supporting fast queries. Compressed indexing is a key computational primitive for querying massive data sets and the area has received significant attention over the last decades with numerous theoretical and practical solutions, see e.g. [25, 12, 29, 23, 13, 14, 21, 22, 15, 34, 30, 9, 27, 18, 24, 4] and the surveys [34, 32, 33, 19].

The Lempel-Ziv 1977 compression scheme (LZ77) [37] is a classic compression scheme based on replacing repetitions by references in a greedy left-to-right order. Numerous variants of LZ77 have been developed and several widely used implementations are available (such as `gzip` [20]). Recently, LZ77 has been shown to be particularly effective at handling highly-repetitive data sets [30, 32, 27, 8, 3] and LZ77 compression is always at least as powerful as any grammar representation [36, 7].

In this paper, we consider compressed indexing based on LZ77 compression. Relatively few results are known for this version of the problem. Let n, z, and m denote the size of the input string, the compressed LZ77 string, and the pattern string, respectively. Kärkkäinen and Ukkonen introduced the problem in 1996 [25] and gave an initial solution that required read-only access to the uncompressed text. Interestingly, this work is among the first results in compressed indexing [34]. More recently, Gagie et al. [17, 18] revisited the problem and gave a solution using space $O(z \lg(n/z))$ and query time $O(m \lg m + \mathrm{occ} \lg \lg n)$, where occ is the number of occurrences of P in S. Note that these bounds assume a constant sized alphabet.

1.1 Our Results

We show the following main result.

▶ **Theorem 1.** *Given a string S of length n from a constant sized alphabet compressed using LZ77 into a string of length z we can build a compressed-index supporting substring queries in:*
(i) $O(m + \mathrm{occ} \lg \lg n)$ *time using* $O(z \lg(n/z) \lg \lg z)$ *space, or*
(ii) $O(m(1 + \frac{\lg^\epsilon z}{\lg(n/z)}) + \mathrm{occ}(\lg \lg n + \lg^\epsilon z))$ *time using* $O(z \lg(n/z))$ *space, for any* $0 < \epsilon < 1$

Compared to the previous bounds Theorem 1 obtains new interesting trade-offs. In particular, Theorem 1 (i) improves the leading term in the query time of the previous best solution from $O(m \lg m)$ to $O(m)$ at the cost of increasing the space by only a factor $\lg \lg z$. Alternatively, Theorem 1 (ii) matches the previous best space bound, but has a leading term in the query time of $O(m(1 + \frac{\lg^\epsilon z}{\lg(n/z)}))$. However, for any polynomial compression ratio, i.e., $z = O(n^{1-\delta})$, for constant $\delta > 0$, this becomes $O(m)$.

Gagie et al. [18] also showed how to extract an arbitrary substring of S of length ℓ in time $O(\ell + \lg n)$. We show how to support the same extraction operation and slightly improve the time to $O(\ell + \lg(n/z))$.

Technically, our results are obtained by new variants and extensions of existing data structures in novel combinations. In particular, we consider a batched variant of the *weak prefix search problem* and give the first non-trivial solution to it. We also generalize the well-known bidirectional compact trie search technique [28] to reduce the number of queries at the cost of increasing space. Finally, we show how to combine this efficiently with range reporting and fast random-access in a balanced grammar leading to the result.

As mentioned all of the above bounds hold for a constant size alphabet. However, Theorem 1 is an instance of full time-space trade-off that also supports general alphabets. We discuss the details in Section 8 and Appendix 8.1.

2 Preliminaries

We assume a standard unit-cost RAM model with word size $w = \Theta(\lg n)$ and that the input is from an integer alphabet $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$ and measure space complexity in words unless otherwise specified.

A string S of length $n = |S|$ is a sequence $S[1] \ldots S[n]$ of n characters drawn from Σ. The string $S[i] \ldots S[j]$ denoted $S[i, j]$ is called a *substring* of S. ϵ is the empty string and $S[i, i] = S[i]$ while $S[i, j] = \epsilon$ when $i > j$. The substrings $S[1, i]$ and $S[j, n]$ are the i^{th} *prefix* and the j^{th} *suffix* of S respectively. The reverse of the string S is denoted $\text{rev}(S) = S[n]S[n-1] \ldots S[1]$.

Let D be a set of k strings and let T_D be the compact trie storing all the strings of D. $\text{str}(v)$ denotes the prefix corresponding to the vertex v. The *depth* of vertex v is the number of edges on the path from v to the root. We assume each string in D is terminated by a special character $\$ \notin \Sigma$ such that each string in D corresponds to a leaf. The children of each vertex are sorted from left to right in increasing lexicographical order, and therefore the left to right order of the leaves corresponds to the lexicographical order of the strings in D. Let $\text{rank}(s)$ denote the rank of the string $s \in D$ in this order. The *skip interval* of a vertex $v \in \text{T}_D$ with parent u is $(|\text{str}(u)|, |\text{str}(v)|]$ denoted $\text{skip}(v)$ and $\text{skip}(v) = \emptyset$ if v is the root. The *locus* of a string s in T_D, denoted $\text{locus}(s)$, is the minimum depth vertex v such that s is a prefix of $\text{str}(v)$. If there is no such vertex, then $\text{locus}(s) = \perp$. In order to reduce the space used by T_D we only store the first character of every edge and in every vertex v we store $|\text{str}(v)|$ (This variation is also known as a PATRICIA tree [31]). We navigate T_D by storing a dictionary in every internal vertex mapping the first character of the label of an edge to the respective child. The size of T_D is $O(k)$.

A *Karp-Rabin fingerprinting function* [26] is a randomized hash function for strings. We use a variation of the original definition appearing in Porat and Porat [35]. The fingerprint for a string S of length n is defined as: $\phi(S) = \sum_{i=1}^{n} S[i] \cdot r^{i-1} \bmod p$, where p is a prime and r is a random integer in $\mathbb{Z}_p$ (the field of integers modulo p). Storing the values n, $r^n \bmod p$ and $r^{-n} \bmod p$ along with a fingerprint allows for efficient composition an subtraction of fingerprints. Using this we can compute and store the fingerprints of each of the prefixes of a string S of length n in $O(n)$ time and space such that we afterwards can compute the fingerprint of any substring $S[i, j]$ in constant time. We say that the fingerprints of the strings x and y *collide* when $\phi(x) = \phi(y)$ and $x \neq y$. A fingerprinting function ϕ is *collision-free* for a set of strings if there are no fingerprint collisions between any of the strings. Porat and Porat [35] show that if x and y are different strings of length at most n and $p = \Theta(n^{2+\alpha})$ for some $\alpha > 0$, then the probability that $\phi(x) = \phi(y)$ is less than $1/n^{1+\alpha}$.

The *LZ77 parse* of a string S of length n is a sequence Z of z subsequent substrings of S called *phrases* such that $S = Z[1]Z[2], \ldots, Z[z]$. Z is constructed in a left to right pass of S: Assume that we have found the sequence $Z[1, i]$ producing the string $S[1, j-1]$ and let $S[j, j'-1]$ be the longest prefix of $S[j, n-1]$ that is also a substring of $S[1, j'-2]$. Then $Z[i+1] = S[j, j']$. The occurrence of $S[j, j'-1]$ in $S[1, j'-2]$ is called the *source* of the phrase $Z[i]$. Thus a phrase is composed by the contents of its possibly empty source and a trailing character which we call the *phrase border* and is typically represented as a triple $Z[i] = (\mathit{start}, \mathit{len}, c)$ where *start* is the starting position of the source, *len* is the length of the source and $c \in \Sigma$ is the border. For a phrase $Z[i] = S[j, j']$ we denote the position of its border by $\text{border}(Z[i]) = j'$ and its source by $\text{source}(Z[i]) = S[j, j'-1]$. For example, the string $abcabcabc \ldots abc$ of length n has the LZ77 parse $|a|b|c|abcabcabc \ldots abc|$ of length 4 which is represented as $Z = (0, 0, a)(0, 0, b)(0, 0, c)(0, n-4, c)$.

3 Prefix Search

The *prefix search* problem is to preprocess a set of strings such that later, we can find all the strings in the set that are prefixed by some query string. Belazzougui et al. [2] consider the *weak prefix search* problem, a relaxation of the prefix search problem where we are only requested to output the ranks (in lexicographic order) of the strings that are prefixed by the query pattern and we only require no false negatives. Thus we may answer arbitrarily when no strings are prefixed by the query pattern.

▶ **Lemma 2** (Belazzougui et al. [2], appendix H.3). *Given a set D of k strings with average length l, from an alphabet of size σ, we can build a data structure using $O(k(\lg l + \lg\lg\sigma))$ bits of space supporting weak prefix search for a pattern P of length m in $O(m\lg\sigma/w + \lg m)$ time where w is the word size.*

The term $m\lg\sigma/w$ stems from preprocessing P with an incremental hash function such that the hash of any substring $P[i,j]$ can be obtained in constant time afterwards. Therefore we can do weak prefix search for h substrings of P in $O(m\lg\sigma/w + h\lg m)$ time. We now describe a data structure that builds on the ideas from Lemma 2 but obtains the following:

▶ **Lemma 3.** *Given a set D of k strings, we can build a data structure taking $O(k)$ space supporting weak prefix search for h substrings of a pattern P of length m in time $O(m + h(m/x + \lg x))$ where x is a positive integer.*

If we know h when building our data structure, we set x to h and obtain a query time of $O(m + h\lg h)$ with Lemma 3.

Before describing our data structure we need the following definition: The *2-fattest* number in a nonempty interval of strictly positive integers is the number in the interval whose binary representation has the highest number of trailing zeroes.

3.1 Data Structure

Let T_D be the compact trie representing the set D of k strings and let x be a positive integer. Denote by $\mathrm{fat}(v)$ the 2-fattest number in the skip interval of a vertex $v \in \mathrm{T}_D$. The *fat prefix* of v is the length $\mathrm{fat}(v)$ prefix of $\mathrm{str}(v)$. Denote by D^{fat} the set of fat prefixes induced by the vertices of T_D. The x-prefix of v is the shortest prefix of $\mathrm{str}(v)$ whose length is a multiple of x and is in the interval $\mathrm{skip}(v)$. If v's skip interval does not span a multiple of x, then v has no x-prefix. Let D^x be the set of x-prefixes induced by the vertices of T_D. The data structure is the compact trie T_D augmented with:

- A fingerprinting function ϕ.
- A dictionary $\mathcal{G}$ mapping the fingerprints of the strings in D^{fat} to their associated vertex.
- A dictionary $\mathcal{H}$ mapping the fingerprints of the strings in D^x to their associated vertex.
- For every vertex $v \in \mathrm{T}_D$ we store the rank in D of the string represented by the leftmost and rightmost leaf in the subtree of v, denoted l_v and r_v respectively.

The data structure is similar to the one by Belazzougui et al. [2] except for the dictionary $\mathcal{H}$, which we use in the first step of our search. There are at most k strings in each of D^{fat} and D^x thus the total space of the data structure is $O(k)$.

Let i be the start of the skip interval of some vertex $v \in \mathrm{T}_D$ and define the *pseudo-fat* numbers of v to be the set of 2-fattest numbers in the intervals $[i, p]$ where $i \leq p < \mathrm{fat}(v)$. We require that the fingerprinting function ϕ is collision-free for the strings in D^{fat}, the strings in D^x and all the length l-prefixes of the strings in D where l is a pseudo-fat number in the skip interval of some vertex $v \in \mathrm{T}_D$.

Observe that the range of strings in D that are prefixed by some pattern P of length m is exactly $[l_v, r_v]$ where $v = \text{locus}(P)$. Answering a weak prefix search query for P is comprised by two independent steps. First step is to find a vertex $v \in \mathrm{T}_D$ such that $\text{str}(v)$ is a prefix of P and $m - |\text{str}(v)| \leq x$. We say that v is in *x-range of* P. Next step is to apply a slightly modified version of the search technique from Belazzougui et al. [2] to find the *exit vertex* for P, that is, the deepest vertex $v' \in \mathrm{T}_D$ such that $\text{str}(v')$ is a prefix of P. Having found the exit vertex we can find the locus in constant time as it is either the exit vertex itself or one of its children.

Finding an x-range Vertex. We now describe how to find a vertex in x-range of P. If $m < x$ we simply report that the root of T_D is in x-range of P. Otherwise, let v be the root of T_D and for $i = 1, 2, \ldots \lfloor m/x \rfloor$ we check if $ix > |\text{str}(v)|$ and $\phi(P[1, ix])$ is in $\mathcal{H}$ in which case we update v to be the corresponding vertex. Finally, if $|\text{str}(v)| \geq m$ we report that v is $\text{locus}(P)$ and otherwise we report that v is in x-range of P. In the former case, we report $[l_v, r_v]$ as the range of strings in D prefixed by P. In the latter case we pass on v to the next step of the algorithm.

We now show that the algorithm is correct when P prefixes a string in D. It is easy to verify that the x-prefix of v prefixes P at all time during the execution of the algorithm. Assume that $|\text{str}(v)| \geq m$ by the end of the algorithm. We will show that in that case $v = \text{locus}(P)$, i.e., that v is the highest node prefixed by P. Since P prefixes a string in D, the x-prefix of v prefixes P, and $|\text{str}(v)| \geq m$, then P prefixes v. Since the x-prefix of v prefixes P, P does not prefix the parent of v and thus v is the highest node prefixed by P.

Assume now that $|\text{str}(v)| < m$. We will show that v is in x-range of P. Since P prefixes a string in D and the x-prefix of v prefixes P, then $\text{str}(v)$ prefixes P. Let $P[1, ix]$ be the x-prefix of v. Since v is returned, either $\phi(P[1, jx]) \notin \mathcal{H}$ or $jx \leq |\text{str}(v)|$ for all $i < j \leq \lfloor m/x \rfloor$. If $\phi(P[1, jx]) \notin \mathcal{H}$ then $P[1, jx]$ is not a x-prefix of any node in T_D. Since P prefixes a string in D this implies that jx is in the skip interval of v, i.e., $jx \leq |\text{str}(v)|$. This means that $jx \leq |\text{str}(v)|$ for all $i < j \leq \lfloor m/x \rfloor$. Therefore $\lfloor m/x \rfloor x \leq |\text{str}(v)| < m$ and it follows that $m - |\text{str}(v)| < x$. We already proved that $\text{str}(v)$ prefixes P and therefore v is in x-range of P.

In case P does not prefix any string in D we either report that $v = \text{locus}(P)$ even though $\text{locus}(P) = \perp$ or report that v is in x-range of P because $m - |\text{str}(v)| \leq x$ even though $\text{str}(v)$ is not a prefix of P due to fingerprint collisions. This may lead to a false positive. However, false positives are allowed in the weak prefix search problem.

Given that we can compute the fingerprint of substrings of P in constant time the algorithm uses $O(m/x)$ time.

From x-range to Exit Vertex. We now consider how to find the exit vertex of P hereafter denoted v_e. The algorithm is similar to the one presented in Belazzougui et al. [2] except that we support starting the search from not only the root, but from any ancestor of v_e.

Let v be any ancestor of v_e, let y be the smallest power of two greater than $m - |\text{str}(v)|$ and let z be the largest multiple of y no greater than $|\text{str}(v)|$. The search progresses by iteratively halving the search interval while using $\mathcal{G}$ to maintain a candidate for the exit vertex and to decide in which of the two halves to continue the search.

Let v_c be the candidate for the exit vertex and let l and r be the left and right boundary for our search interval. Initially $v_c = v$, $l = z$ and $r = z + 2y$. When $r - l = 1$, the search terminates and reports v_c. In each iteration, we consider the mid $b = (l + r)/2$ of the interval $[l, r]$ and update the interval to either $[b, r]$ or $[l, b]$. There are three cases:

1. b is out of bounds
   a. If $b > m$ set r to b.
   b. If $b \leq |\mathrm{str}(v_c)|$ set l to b.
2. $P[1,b] \in D^{\mathrm{fat}}$, let u be the corresponding vertex, i.e. $\mathcal{G}(\phi(P[1,b])) = u$.
   a. If $|\mathrm{str}(u)| < m$, set v_c to u and l to b.
   b. If $|\mathrm{str}(u)| \geq m$, report $u = \mathrm{locus}(P)$ and terminate.
3. $P[1,b] \notin D^{\mathrm{fat}}$ and thus $\phi(P[1,b])$ is not in $\mathcal{G}$, set r to b.

Observe that we are guaranteed that all fingerprint comparisons are collision-free in case P prefixes a string in D. This is because the length of the prefix fingerprints we consider are all either 2-fattest or pseudo-fat in the skip interval of locus(P) or one of its ancestors and we use a fingerprinting function that is collision-free for these strings.

Correctness. We now show that the invariant $l \leq |\mathrm{str}(v_c)| \leq |\mathrm{str}(v_e)| < r$ is satisfied and that $\mathrm{str}(v_c)$ is a prefix of P before and after each iteration. After $O(\lg x)$ iterations $r - l = 1$ and thus $l = |\mathrm{str}(v_e)| = |\mathrm{str}(v_c)|$ and therefore $v_c = v_e$. Initially v_c is an ancestor of v_e and thus $\mathrm{str}(v_c)$ is a prefix of P, $l = z \leq |\mathrm{str}(v_c)|$ and $r = z + 2y > m > |\mathrm{str}(v_e)|$ so the invariant is true. Now assume that the invariant is true at the beginning of some iteration and consider the possible cases:

1. b is out of bounds
   a. $b > m$ then because $|\mathrm{str}(v_e)| \leq m$, setting r to b preserves the invariant.
   b. $b \leq |\mathrm{str}(v_c)|$ then setting l to b preserves the invariant.
2. $P[1,b] \in D^{\mathrm{fat}}$, let $u = \mathcal{G}(\phi(P[1,b]))$.
   a. $|\mathrm{str}(u)| \leq m$ then $\mathrm{str}(u)$ is a prefix of P and thus $b = \mathrm{fat}(u) \leq |\mathrm{str}(u)| \leq |\mathrm{str}(v_e)|$ so setting l to b and v_c to u preserves the invariant.
   b. $|\mathrm{str}(u)| \geq m$ yet $u = \mathcal{G}(\phi(P[1,b]))$. Then u is the locus of P.
3. $P[1,b] \notin D^{\mathrm{fat}}$, and thus $\phi(P[1,b])$ is not in $\mathcal{G}$. As we are not in any of the out of bounds cases we have $|\mathrm{str}(v_c)| < b < m$. Thus, either $b > |\mathrm{str}(v_e)|$ and setting r to b preserves the invariant. Otherwise $b \leq |\mathrm{str}(v_e)|$ and thus b must be in the skip interval of some vertex u on the path from v_c to v_e excluding v_c. But skip(u) is entirely included in (l, r) and because b is 2-fattest in (l, r)[1] it is also 2-fattest in skip(u). It follows that $\mathrm{fat}(u) = b$ which contradicts $P[1,b] \notin D^{\mathrm{fat}}$ and thus the invariant is preserved.

Thus if P prefixes a string in D we find either the exit vertex v_e or the locus of P. In the former case the locus of P is the child of v_e identified by the character $P[|\mathrm{str}(v')|+1]$. Having found the vertex $u = \mathrm{locus}(P)$ we report $[l_u, r_u]$ as the range of strings in D prefixed by P. In case P does not prefix any strings in D, the fact that the fingerprint of a prefix of P match the fingerprint of some fat prefix in D^x does not guarantee equality of the strings. There are two possible consequences of this. Either the search successfully finds what it believes to be the locus of P even though $\mathrm{locus}(P) = \bot$ in which case we report a false positive. Otherwise, there is no child identified by $P[|\mathrm{str}(v')| + 1]$ in which case we can correctly report that no strings in D are prefixed by S, a true negative. Recall that false positives are allowed as we are considering the weak prefix search problem.

[1] If $b - a = 2^i$, $i > 0$ and a is a multiple of 2^{i-1} then the mid of the interval $(a + b)/2$ is 2-fattest in (a, b).

Complexity. The size of the interval $[l, r]$ is halved in each iteration, thus we do at most $O(\lg(m - |\text{str}(v)|))$ iterations, where v is the vertex from which we start the search. If we use the technique from the previous section to find a starting vertex in x-range of P, we do $O(\lg x)$ iterations. Each iteration takes constant time. Note that if P does not prefix a string in D we may have fingerprint collisions and we may be given a starting vertex v such that $\text{str}(v)$ does not prefix P. This can lead to a false positive, but we still have $m - |\text{str}(v)| \leq x$ and therefore the time complexity remains $O(\lg x)$.

Multiple Substrings. In order to answer weak prefix search queries for h substrings of a pattern P of length m, we first preprocess P in $O(m)$ time such that we can compute the fingerprint of any substring of P in constant time. We can then answer a weak prefix search query for any substring of P in total time $O(m/x + \lg x)$ using the techniques described in the previous sections. The total time is therefore $O(m + h(m/x + \lg x))$.

4 Distinguishing Occurrences

The following sections describe our compressed-index consisting of three independent data structures. One that finds long primary occurrences, one that finds short primary occurrences and one that finds secondary occurrences.

Let Z be the LZ77 parse of length z representing the string S of length n. If $S[i, j]$ is a phrase of Z then any substring of $S[i, j-1]$ is a *secondary substring* of S. These are the substrings of S that do not contain any phrase borders. On the other hand, a substring $S[i, j]$ is a *primary substring* of S when there is some phrase $S[i', j']$ where $i' \leq i \leq j' \leq j$, these are the substrings that contain one or more phrase borders. Any substring of S is either primary or secondary. A primary substring that match a query pattern P is a *primary occurrence* of P while a secondary substring that match P is a *secondary occurrence* [25].

5 Long Primary Occurrences

For simplicity, we assume that the data structure given in Lemma 3 not only solves the weak prefix problem, but also answers correctly when the query pattern does not prefix any of the indexed strings. Later in Section 5.3 we will see how to lift this assumption. The following data structure and search algorithm is a variation of the classical bidirectional search technique for finding primary occurrences [25].

5.1 Data Structure

For every phrase $S[i, j]$ the strings $S[i, j + k], 0 \leq k < \tau$ are relevant substrings unless there is some longer relevant substring ending at position $j + k$. If $S[i', j']$ is a relevant substring then the string $S[j' + 1, n]$ is the *associated suffix*. There are at most $z\tau$ relevant substrings of S and equally many associated suffixes. The primary index is comprised by the following:

- A prefix search data structure T_D on the set of reversed relevant substrings.
- A prefix search data structure $\text{T}_{D'}$ on the set of associated suffixes.
- An orthogonal range reporting data structure R on the $z\tau \times z\tau$ grid. Consider a relevant substring $S[i, j]$. Let x denote the rank of $\text{rev}(S[i, j])$ in the lexicographical order of the reversed relevant substrings, let y denote the rank of its associated suffix $S[j + 1, n]$ in the lexicographical order of the associated suffixes. Then (x, y) is a point in R and along with it we store the pair (j, b), where b is the position of the rightmost phrase border contained in $S[i, j]$.

Note that every point (x, y) in R is induced by some relevant substring $S[i, j]$ and its associated suffix $S[j+1, n]$. If some prefix $P[1, k]$ is a suffix of $S[i, j]$ and the suffix $P[k+1, m]$ is a prefix of $S[j+1, n]$ then $S[j-k+1, j-k+m]$ is an occurrence of P and we can compute its exact location from k and j.

5.2 Searching

The data structure can be used to find the primary occurrences of a pattern P of length m when $m > \tau$. Consider the $O(m/\tau)$ prefix-suffix pairs $(P[1, i\tau], P[i\tau + 1, m])$ for $i = 1, \ldots, \lfloor m/\tau \rfloor$ and the pair $(P[1, m], \epsilon)$ in case m is not a multiple of τ. For each such pair, we do a prefix search for $\text{rev}(P[1, i\tau])$ and $P[i\tau + 1, m]$ in T_D and $\mathrm{T}_{D'}$, respectively. If either of these two searches report no matches, we move on to the next pair. Otherwise, let $[l, r]$, $[l', r']$ be the ranges reported from the search in T_D and $\mathrm{T}_{D'}$ respectively. Now we do a range reporting query on R for the rectangle $[l, r] \times [l', r']$. For each point reported, let (j, b) be the pair stored with the point. We report $j - i\tau + 1$ as the starting position of a primary occurrence of P in S.

Finally, in case m is not a multiple of τ, we need to also check the pair $(P[1, m], \epsilon)$. We search for $\text{rev}(P[1, m])$ in in T_D and ϵ in $\mathrm{T}_{D'}$. If the search for $\text{rev}(P[1, m])$ reports no match we stop. Otherwise, we do a range reporting query as before. For each point reported, let (j, b) be the pair stored with the point. To check that the occurrence has not been reported before we do as follows. Let k be the smallest positive integer such that $j - m + k\tau > b$. If $k\tau > m$ we report $j - m + 1$ as the starting position of a primary occurrence.

Correctness. We claim that the reported occurrences are exactly the primary occurrences of P. We first prove that all primary occurrences are reported correctly. Let $P = S[i', j']$ be a primary occurrence. As it is a primary occurrence, there must be some phrase $S[i^*, j^*]$ such that $i^* \leq i' \leq j^* \leq j'$. Let k be the smallest positive integer such that $i' + k\tau - 1 \geq j^*$. There are two cases: $k\tau \leq m$ and $k\tau > m$. If $k\tau \leq m$ then $P[1, k\tau]$ is a suffix of the relevant substring ending at $i' + k\tau - 1$. Such a relevant substring exists since $i' + k\tau - 1 < j^* + \tau$. Thus its reverse $\text{rev}(P[1, k\tau])$ prefixes a string s in D, while $P[k\tau + 1, m]$ is a prefix of the associated suffix $S[i' + k\tau, n] \in D'$. Therefore, the respective ranks of s and $S[i' + k\tau, n]$ in D and D' are plotted as a point in R which stores the pair $(i' + k\tau - 1, b)$. We will find this point when considering the prefix-suffix pair $(P[1, k\tau]$, $P[k\tau + 1, m])$, and correctly report $(i' + k\tau - 1) - k\tau + 1 = i'$ as the starting position of a primary occurrence. If $k\tau > m$ then $P[1, m]$ is a suffix of the relevant substring ending in $i' + m - 1$. Such a relevant substring exists since $i' + m - 1 < i' + k\tau - 1 < j^* + \tau$. Thus its reverse prefixes a string in D and trivially ϵ is a prefix of the associated suffix. It follows as before that the ranks are plotted as a point in R storing the pair $(i' + m - 1, b)$ and that we find this point when considering the pair $(P[1, m], \epsilon)$. When considering $(P[1, m], \epsilon)$ we report $(i' + m - 1) - m + 1 = i'$ as the starting position of a primary occurrence if $k\tau > m$, and thus i' is correctly reported.

We now prove that all reported occurrences are in fact primary occurrences. Assume that we report $j - i\tau + 1$ for some i and j as the starting position of a primary occurrence in the first part of the procedure. Then there exist strings $\text{rev}(S[i', j])$ and $S[j + 1, n]$ in D and D' respectively such that $S[i', j]$ is suffixed by $P[1, i\tau]$ and $S[j + 1, n]$ is prefixed by $P[i\tau + 1, m]$. Therefore $j - i\tau + 1$ is the starting position of an occurrence of P. The string $S[i', j]$ is a relevant suffix and therefore there exists a border b in the interval $[j - \tau + 1, j]$. Since $i \geq 1$ the occurrence contains the border b and it is therefore a primary occurrence. If we report $j - m + 1$ for some j as the starting position of a primary occurrence in the second part of the procedure, then $\text{rev}(P[1, m])$ is a prefix of a string $\text{rev}(S[i', j])$ in D. It follows immediately that $j - m + 1$ is the starting point of an occurrence. Since $m > \tau$ we have $j - m + 1 < j - \tau + 1$, and by the definition of relevant substring there is a border in the interval $[j - \tau + 1, j]$. Therefore the occurrence contains the border and is primary.

Complexity. We now consider the time complexity of the algorithm described. First we will argue that any primary occurrence is reported at most once and that the search finds at most two points in R identifying it. Let $S[i', j']$ be a primary occurrence reported when we considered the prefix-suffix pair $(P[1, k\tau], P[k\tau + 1, m])$ as in the proof of correctness. None of the pairs $(P[1, i\tau], P[i\tau + 1, m])$, where $i < k$ will identify this occurrence as $i' + i\tau - 1 < j$. None of the pairs $(P[1, h\tau], P[h\tau + 1, m])$, where $h > k$, will identify this occurrence. This is the case since $i' + h\tau - 1 > j + \tau - 1$, and from the definition of relevant substrings it follows that if $S[i, j]$ is a phrase, $S[a, b]$ is a relevant substring and $a < i$, then $b < i + \tau - 1$. Thus there are no relevant substrings that end after $j + \tau - 1$ and start before $i' < j$. Therefore, only one of the pairs $(P[1, i\tau], P[i\tau + 1, m])$ for $i = 1, \ldots, \lfloor m/x \rfloor$ identifies the occurrence. If $(k + 1)\tau > m$ then we might also find the occurrence when considering the pair $(P[1, m], \epsilon)$, but we do not report i' as $k\tau \le m$.

After preprocessing P in $O(m)$ time, we can do the $O(m/\tau)$ prefix searches in total time $O(m + m/\tau(m/x + \lg x))$ where x is a positive integer by Lemma 3. Using the range reporting data structure by Chan et al. [6] each range reporting query takes $(1 + k) \cdot O(B \lg\lg(z\tau))$ time where $2 \le B \le \lg^{\epsilon}(z\tau)$ and k is the number of points reported. As each such point in one range reporting query corresponds to the identification of a unique primary occurrence of P, which happens at most twice for every occurrence we charge $O(kB \lg\lg(z\tau))$ to reporting the occurrences. The total time to find all primary occurrences is thus $O(m + \frac{m}{\tau}(\frac{m}{x} + \lg x + B \lg\lg(z\tau)) + \mathrm{occ}\ B \lg\lg(z\tau))$ where occ is the number of primary and secondary occurrences of P.

5.3 Prefix Search Verification

The prefix data structure from Lemma 3 gives no guarantees of correct answers when the query pattern does not prefix any of the indexed strings. If the prefix search gives false-positives, we may end up reporting occurrences of P that are not actually there. We show how to solve this problem after introducing a series of tools that we will need.

Straight line programs. A *straight line program* (SLP) for a string S is a context-free grammar generating the single string S.

▶ **Lemma 4** (Rytter [36], Charikar et al. [7]). *Given an LZ77 parse Z of length z producing a string S of length n we can construct a SLP for S of size $O(z \lg(n/z))$ in time $O(z \lg(n/z))$.*

The construction from Rytter [36] produces a balanced grammar for every consecutive substring of length n/z of S after a preprocessing step transforms Z such that no compression element is longer than n/z. The height of this balanced grammar is $O(\lg n)$ and this immediately yields extracting of any substring $S[i, j]$ in time $O(\lg(n) + j - i)$. We give a simple solution to reduce this to $O(\lg(n/z) + j - i)$, that also supports computation of the fingerprint of a substring in $O(\lg(n/z))$ time.

▶ **Lemma 5.** *Given an LZ77 parse Z of length z producing a string S of length n we can build a data structure that for any substring $S[i, j]$ can extract $S[i, j]$ in $O(\lg(n/z) + j - i)$ time and compute the fingerprint $\phi(S[i, j])$ in $O(\lg(n/z))$ time. The data structure uses $O(z \lg(n/z))$ space and $O(n)$ construction time.*

Proof. Assume for simplicity that n is a multiple of z. We construct the SLP producing S from Z. Along with every non-terminal of the SLP we store the size and fingerprint of its expansion. Let $s_1, s_2, \ldots s_z$ be consecutive length n/z substrings of S. We store the balanced grammar producing s_i along with the fingerprint $\phi(S[1,(i-1)n/z])$ at index i in a table A.

Now we can extract s_i in $O(n/z)$ time and any substring $s_i[j,k]$ in time $O(\lg(n/z)+k-j)$. Also, we can compute the fingerprint $\phi(s_i[j,k])$ in $O(\lg(n/z))$ time. We can easily do a constant time mapping from a position in S to the grammar in A producing the substring covering that position and the corresponding position inside the substring. But then any fingerprint $\phi(S[1,j])$ can be computed in time $O(\lg(n/z))$. Now consider a substring $S[i,j]$ that starts in s_k and ends in $s_l, k < l$. We extract $S[i,j]$ in $O(\lg(n/z) + j - i)$ time by extracting the appropriate suffix of s_k, all of s_m for $k < m < l$ and the appropriate prefix of s_l. Each of the fingerprints stored by the data structure can be computed in $O(1)$ time after preprocessing S in $O(n)$ time. Thus table A is filled in $O(z)$ time and by Lemma 4 the SLPs stored in A uses a total of $O(z\lg(n/z))$ space and construction time. ◀

Verification of fingerprints. We need the following lemma for the verification.

▶ **Lemma 6** (Bille et al. [5]). *Given a string S of length n, we can find a fingerprinting function ϕ that is collision-free for all length l substrings of S where l is a power of two in $O(n\lg n)$ expected time.*

5.3.1 Verification Technique

Our verification technique is identical to the one given by Gagie et al. [18] and involves a simple modification of the search for long primary occurrences. By using Lemma 5 instead of bookmarking [18] for extraction and fingerprinting and because we only need to verify $O(m/\tau)$ strings, the verification procedure takes $O(m + m/\tau\lg(n/z))$ time and uses $O(z\lg(n/z))$ space. See Appendix A.1 for details.

6 Short Primary Occurrences

We now describe a simple data structure that can find primary occurrences of P in time $O(m + \mathrm{occ})$ using space $O(z\tau)$ whenever $m \leq \tau$ where τ is a positive integer.

Let Z be the LZ77 parse of the string S of length n. Let $Z[i] = S[s_i, e_i]$ and define F to be the union of the strings $S[k, \min\{e_i + \tau, n\}]$ where $\max\{1, s_i, e_i - \tau\} \leq k \leq e_i$ for $i = 1, 2, \ldots z$. There are at most $z\tau$ such strings, each of length $O(\tau)$ and they are all suffixes of the z length 2τ substrings of S starting τ positions before each border position. We store these substrings along with the compact trie T_F over the strings in F. The edge labels of T_F are compactly represented by storing references into one of the substrings. Every leaf stores the starting position in S of the string it represents and the position of the leftmost border it contains.

The combined size of T_F and the substrings we store is $O(z\tau)$ and we simply search for P by navigating vertices using perfect hashing [16] and matching edge labels character by character. Now either $\mathrm{locus}(P) = \perp$ in which case there are no primary occurrences of P in S; otherwise, $\mathrm{locus}(P) = v$ for some vertex $v \in \mathrm{T}_F$ and thus every leaf in the subtree of v represents a substring of S that is prefixed by P. By using the indices stored with the leaves, we can determine the starting position for each occurrence and if it is primary or secondary. Because each of the strings in F start at different positions in S, we will only find an occurrence once. Also, it is easy to see that we will find all primary occurrences because of how the strings in F are chosen. It follows that the time complexity is $O(m + \mathrm{occ})$ where occ is the number of primary and secondary occurrences.

7 The Secondary Index

Let Z be the LZ77 parse of length z representing the string S of length n. We find the secondary occurrences by applying the most recent range reporting data structure by Chan et al. [6] to the technique described by Kärkkäinen and Ukkonen [25]. This gives us a secondary index using $O(z \lg\lg z)$ space and $O(\mathrm{occ} \lg\lg n)$ time for reporting all secondary occurrences. For details see Appendix A.2.

8 The Compressed Index

We obtain our final index by combining the primary index, the verification data structure and the secondary index. We use the transformed LZ77 parse generated by Lemma 4 when building our primary index. Therefore no phrase will be longer than n/z and therefore any primary occurrence of P will have a prefix $P[1, k]$ where $k \leq n/z$ that is a suffix of some phrase. It then follows that we need only consider the multiples $(P[1, i\tau], P[i\tau + 1, m])$ for $i < \lfloor \frac{n/z}{\tau} \rfloor$ when searching for long primary occurrences. This yields the following complexities:

- $O(m + \frac{\min\{m,n/z\}}{\tau}(\frac{m}{x} + \lg x + B \lg\lg(z\tau)) + \mathrm{occ}\ B \lg\lg(z\tau))$ time and $O(z\tau \lg_B \lg(z\tau))$ space for the index finding long primary occurrences where x and τ are positive integers and $2 \leq B \leq \lg^{\epsilon}(z\tau)$.
- $O(m + \mathrm{occ})$ time and $O(z \lg(n/z))$ space for the index finding short primary occurrences.
- $O(m + m/\tau \lg(n/z))$ time and $O(z \lg(n/z))$ space for the verification data structure.
- $O(\mathrm{occ} \lg\lg n)$ time and $O(z \lg\lg z)$ space for the secondary index.

If we fix x at n/z we have $\frac{\min\{m,n/z\}}{\tau} \frac{m}{x} \leq m$ in which case we obtain the following trade-off simply by combining the above complexities.

▶ **Theorem 7.** *Given a string S of length n from an alphabet of size σ compressed using LZ77 to a string of length z we can build a compressed-index supporting substring queries in $O(m + \frac{m}{\tau}(\lg(n/z) + B \lg\lg(z\tau)) + \mathrm{occ}(B \lg\lg(z\tau) + \lg\lg n))$ time using $O(z(\lg(n/z) + \tau \lg_B \lg(z\tau) + \lg\lg z))$ space for any query pattern P of length m where $2 \leq B \leq \lg^{\epsilon}(z\tau)$, $0 < \epsilon < 1$ and τ is a positive integer.*

We note that none of our data structures assume constant sized alphabet and thus Theorem 7 holds for any alphabet size.

Due to lack of space the description and analysis of the preprocessing have been moved to Appendix 8.2.

8.1 Trade-offs

Theorem 7 gives rise to a series of interesting time-space trade-offs.

▶ **Corollary 8.** *Given a string S of length n from an alphabet of size σ compressed using LZ77 into a string of length z we can build a compressed-index supporting substring queries in*

(i) $O(m(1 + \frac{\lg\lg z}{\lg(n/z)}) + \mathrm{occ} \lg\lg n)$ *time using* $O(z \lg(n/z) \lg\lg z)$ *space, or*
(ii) $O(m(1 + \frac{\lg^{\epsilon} z}{\lg(n/z)}) + \mathrm{occ}(\lg\lg n + \lg^{\epsilon} z))$ *time using* $O(z \lg(n/z))$ *space, or*
(iii) $O(m \lg^{\epsilon}(n/z) + \mathrm{occ} \lg\lg n)$ *time using* $O(z \lg(n/z))$ *space, or*
(iv) $O(m + \mathrm{occ} \lg\lg n)$ *time using* $O(z(\lg(n/z) \lg\lg z + \lg\lg^2 z))$ *space, or*

(v) *$O(m + \mathrm{occ}(\lg\lg n + \lg^{\epsilon} z))$ time using $O(z(\lg(n/z) + \lg^{\epsilon'} z))$ space.*
for any $0 < \epsilon < 1$ and $0 < \epsilon' < 1$.

Proof. For (i) set $B = 2$ and $\tau = \lg(n/z)$, for (ii) set $B = \lg^{\epsilon} z$ and $\tau = \lg(n/z)$, for (iii) set $B = 2$ and $\tau = \lg^{\epsilon'} n/z$ for some $0 < \epsilon' < 1$, for (iv) set $B = 2$ and $\tau = \lg(n/z) + \lg\lg z$, for (v) set $B = \lg^{\epsilon'}(z)$ and $\tau = \lg(n/z) + \lg^{\epsilon} z$. ◀

The leading term in the time complexity of Corollary 8 (i) is $O(m)$ whenever $\lg\lg(z) = O(\lg(n/z))$ which is true when $z = O(n/\lg n)$, i.e. for all strings that are compressible by at least a logarithmic fraction. For $\sigma = O(1)$ we have $z = O(n/\lg n)$ all strings [34] and thus Theorem 1 (i) follows immediately. Corollary 8 (ii) matches previous best space bounds but obtains a leading term of $O(m)$ for any polynomial compression rate. Theorem 1 (ii) is a weaker version of this because it assumes constant sized alphabet and therefore follows immediately. Corollary 8 (iii) matches the space and time for reporting occurrences of previous best bounds by Gagie et al. [18] but with a leading term of $O(m\lg^{\epsilon}(n/z))$ compared to a leading term of $O(m\lg m)$. Corollary 8 (iv) and (v) show how to guarantee the fast query times with leading term $O(m)$ without the assumptions on compression ratio that (i) and (ii) require to match this, but at the cost of increased space.

8.2 Preprocessing

We now consider the preprocessing time of the data structure. Let Z be the LZ77 parse of the string S of length n let T_D and $\mathrm{T}_{D'}$ be the compact tries used in the index for long primary occurrences. The compact trie T_D index $O(z\tau)$ substrings of S with overall length $O(n\tau)$. Thus we can construct the trie in $O(n\tau)$ time by sorting the strings and successively inserting them in their sorted order [1]. The compact tries $\mathrm{T}_{D'}$ index $z\tau < n$ suffixes of S and can be built in $O(n)$ time using $O(n)$ space [10]. The index for short primary occurrences is a generalized suffix tree over z strings of length $O(\tau)$ with total length $z\tau < n$ and is therefore also built in $O(n)$ time. The dictionaries used by the prefix search data structures and for trie navigation contain $O(z\tau)$ keys and are built in expected linear time using perfect hashing [16]. The range reporting data structures used by the primary and secondary index over $O(z\tau)$ points are built in $O(z\tau\lg(z\tau))$ expected time using Lemma 9.

Building the SLP for our verification data structure takes $O(z\lg(n/z))$ time using Lemma 4 and finding an appropriate fingerprinting function ϕ takes $O(n\lg n)$ expected time using Lemma 6. The prefix search data structures T_D and $\mathrm{T}_{D'}$ also require that ϕ is collision-free for the x-prefixes, fat prefixes and the prefixes with pseudo fat lengths. There are at most $O(z\tau\lg n)$ such prefixes [2]. If we compute these fingerprints incrementally while doing a traversal of the tries, we expect all the fingerprints to be unique. We simply check this by sorting the fingerprints in linear time and checking for duplicates by doing a linear scan. If we choose a prime $p = \Theta(n^5)$ for the fingerprinting function then the probability of a collision between any two strings is $O(1/n^4)$ [35] and by a union bound over the $O((n\lg n)^2)$ possible collisions the probability that ϕ is collision-free is at least $1-1/n$. Thus the expected time to find our required fingerprinting function is $O(n + n\lg n)$.

All in all, the preprocessing time for our combined index is therefore expected $O(n\lg n + n\tau)$.

References

1 Arne Andersson and Stefan Nilsson. A new efficient radix sort. In Shafi Goldwasser, editor, *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS 1994)*, pages 714–721. IEEE Computer Society, 1994. doi:10.1109/SFCS.1994.365721.

2 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In Mark de Berg and Ulrich Meyer, editors, *Proceedings of the 18th Annual European Symposium on Algorithms (ESA 2010)*, volume 6346 of *LNCS*, pages 427–438. Springer, 2010. doi:10.1007/978-3-642-15775-2_37.

3 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 26–39. Springer, 2015. doi:10.1007/978-3-319-19929-0_3.

4 Djamal Belazzougui, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Alberto Ordóñez Pereira, Simon J. Puglisi, and Yasuo Tabei. Queries on LZ-bounded encodings. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2015 Data Compression Conference (DCC 2015)*, pages 83–92. IEEE, 2015. doi:10.1109/DCC.2015.69.

5 Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time-space trade-offs for longest common extensions. In Juha Kärkkäinen and Jens Stoye, editors, *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM 2012)*, volume 7354 of *LNCS*. Springer, 2012. doi:10.1007/978-3-642-31265-6_24.

6 Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry (SocG 2011)*, pages 1–10. ACM, 2011. doi:10.1145/1998196.1998198.

7 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

8 Francisco Claude, Antonio Fariña, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Universal indexes for highly repetitive document collections. *Inf. Syst.*, 61:1–23, 2016. doi:10.1016/j.is.2016.04.002.

9 Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE 2012)*, volume 7608 of *LNCS*, pages 180–192. Springer, 2012. doi:10.1007/978-3-642-34109-0_19.

10 Martin Farach. Optimal suffix tree construction with large alphabets. In Anna Karlin, editor, *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997)*, pages 137–143, Washington, DC, USA, 1997. IEEE Computer Society. doi:10.1109/SFCS.1997.646102.

11 Martin Farach and Mikkel Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998. doi:10.1007/PL00009202.

12 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In Avrim Blum, editor, *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.

13 Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In S. Rao Kosaraju, editor, *Proceedings of the 12th Annual ACM-SIAM Symposium on

Discrete Algorithms (SODA 2001), pages 269–278. ACM/SIAM, 2001. URL: `http://dl.acm.org/citation.cfm?id=365411.365458`.

14 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. `doi:10.1145/1082036.1082039`.

15 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), 2007. `doi:10.1145/1240233.1240243`.

16 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. `doi:10.1145/828.1884`.

17 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A faster grammar-based self-index. In Adrian-Horia Dediu and Carlos Martín-Vide, editors, *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA 2012)*, volume 7183 of *LNCS*, pages 240–251. Springer, 2012. `doi:10.1007/978-3-642-28332-1_21`.

18 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. LZ77-based self-indexing with faster pattern matching. In Alberto Pardo and Alfredo Viola, editors, *Proceedings of the 11th Latin American Symposium on Theoretical Informatics (LATIN 2014)*, volume 8392 of *LNCS*, pages 731–742. Springer, 2014. `doi:10.1007/978-3-642-54423-1_63`.

19 Travis Gagie and Simon J. Puglisi. Searching and indexing genomic databases via kernelization. *Front. Bioeng. Biotechnol.*, 3:12, 2015. `doi:10.3389/FBIOE.2015.00012`.

20 Jean-Loup Gailly and Mark Adler. GNU zip, 1992. URL: `http://www.gzip.org/`.

21 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In Martin Farach-Colton, editor, *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 841–850. ACM/SIAM, 2003. URL: `http://dl.acm.org/citation.cfm?id=644108.644250`.

22 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In J. Ian Munro, editor, *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 636–645. SIAM, 2004. URL: `http://dl.acm.org/citation.cfm?id=982792.982888`.

23 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC 2000)*, pages 397–406. ACM, 2000. `doi:10.1145/335305.335351`.

24 Juha Kärkkäinen and Erkki Sutinen. Lempel-Ziv index for q-grams. *Algorithmica*, 21(1):137–154, 1998. `doi:10.1007/PL00009205`.

25 Juha Kärkkäinen and Esko Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In Nivio Ziviani, Ricardo Baeza-Yates, and Katia Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing (WSP 1996)*, pages 141–155. Carleton University Press, 1996.

26 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. `doi:10.1147/rd.312.0249`.

27 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013. `doi:10.1016/j.tcs.2012.02.006`.

28 Moshe Lewenstein. Orthogonal range searching for text indexing. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *LNCS*, pages 267–302. Springer, 2013. `doi:10.1007/978-3-642-40273-9_18`.

29 Veli Mäkinen. Compact suffix array. In Raffaele Giancarlo and David Sankoff, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, volume 1848 of *LNCS*, pages 305–319. Springer, 2000. `doi:10.1007/3-540-45123-4_26`.

30 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010. `doi:10.1089/cmb.2009.0169`.

31 Donald R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968. `doi:10.1145/321479.321481`.

32 Gonzalo Navarro. Indexing highly repetitive collections. In S. Arumugam and W. F. Smyth, editors, *Proceedings of the 23rd International Workshop on Combinatorial Algorithms (IWOCA 2012)*, volume 7643 of *LNCS*, pages 274–279. Springer, 2012. `doi:10.1007/978-3-642-35926-2_29`.

33 Gonzalo Navarro. *Compact Data Structures: A practical approach*. Cambridge University Press, 2016. `doi:10.1017/CBO9781316588284`.

34 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), April 2007. `doi:10.1145/1216370.1216372`.

35 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In Daniel A. Spielman, editor, *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2009)*, pages 315–323. IEEE Computer Society, 2009. `doi:10.1109/FOCS.2009.11`.

36 Wojciech Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. `doi:10.1016/S0304-3975(02)00777-6`.

37 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. `doi:10.1109/TIT.1977.1055714`.

A Appendix

A.1 Verification Technique

Consider the string S of length n that we wish to index and let Z be the $LZ77$ parse of S. The verification data structure is given by Lemma 5. Consider the prefix search data structure $\mathrm{T}_{D'}$ as given in Section 5.1 and let ϕ be the fingerprinting function used by the prefix search, the case for T_D is symmetric. We alter the search for primary occurrences such that it first does the $O(m/\tau)$ prefix searches, then verifies the results and discards false-positives before moving on to do the $O(m/\tau)$ range reporting queries on the verified results. We also modify ϕ using Lemma 6 to be collision-free for all substrings of the indexed strings which length is a power of two.

Let $Q_1, Q_2, \dots Q_j$ be the all the suffixes of P for which the prefix search found a locus candidate, let the candidates be $v_1, v_2, \dots v_j \in \mathrm{T}_{D'}$ and let p_i be $\mathrm{str}(v_i)[1, |Q_i|]$. Assume that $|Q_i| < |Q_{i+1}|$, and let 2-suf(Q) and 2-pre(Q) denote the fingerprints using ϕ of the suffix and prefix respectively of length $2^{\lfloor \lg |Q| \rfloor}$ of some string Q. The verification progresses in iterations. Initially, let $a = 1$, $b = 2$ and for each iteration do as follows:

1. 2-suf$(Q_a) \neq$ 2-suf(p_a) or 2-pre$(Q_a) \neq$ 2-pre(p_a): Discard v_a and set $a = a + 1$ and $b = b + 1$.
2. 2-suf$(Q_a) =$ 2-suf(p_a) and 2-pre$(Q_a) =$ 2-pre(p_a), let $R = p_b[|p_a| - |p_b| + 1, |p_a|]$.
   a. 2-suf$(R) =$ 2-suf(Q_a) and 2-pre$(R) =$ 2-pre(Q_a): set $a = a + 1$ and $b = b + 1$.
   b. 2-suf$(R) \neq$ 2-suf(Q_a) or 2-pre$(R) \neq$ 2-pre(Q_a): discard v_b and set $b = b + 1$.

3. $b = j + 1$: If all vertices have been discarded, report no matches. Otherwise, let v_f be the last vertex considered, that was not discarded. Compare p_f to Q_f and if equal, report all non-discarded vertices as verified. Otherwise discard all vertices and report no matches.

Consider the correctness and complexity of the algorithm. In case 1, clearly, p_a does not match Q_a and thus v_a must be a false-positive. Now observe that because Q_i is a suffix of P, it is also a suffix of $Q_{i'}$ for any $i < i'$. Thus in case 2 (b), if R does not match Q_a then v_b must be a false-positive. In case 2 (a), both v_a and v_b may still be false-positives, yet by Lemma 6, p_a is a suffix of p_b because 2-suf(p_a) = 2-suf(R) and 2-pre(p_a) = 2-pre(R). Finally, in case 3, v_f is a true positive if and only if $p_f = Q_f$. But any other non-discarded vertex $v_i \neq v_f$ is also only a true positive if $p_f = Q_f$ because p_i is a suffix of p_f and Q_i is a suffix of Q_p.

The algorithm does j iterations and fingerprints of substrings of P can be computed in constant time after $O(m)$ preprocessing. Every vertex $v \in \mathrm{T}_{D'}$ represents one or more substrings of S. If we store the starting index in S of one of these substrings in v when constructing $\mathrm{T}_{D'}$ we can compute the fingerprint of any substring $\mathrm{str}(v)[i, j]$ by computing the fingerprint of $S[i' + i - 1, i' + j - 1]$ where i' is the starting index of one of the substring of S that v represents. By Lemma 5, the fingerprint computations take $O(\lg(n/z))$ time and because $j \leq m/\tau$ the total time complexity of the algorithm is $O(m + m/\tau \lg(n/z))$.

A.2 Secondary Index

Let Z be the LZ77 parse of length z representing the string S of length n. We find the secondary occurrences by applying the most recent range reporting data structure by Chan et al. [6] to the technique described by Kärkkäinen and Ukkonen [25] which is inspired by the ideas of Farach and Thorup [11].

Let $X \subseteq \{0, \ldots, u\}^d$ be a set of points in a d-dimensional grid. The *orthogonal range reporting problem* in d-dimensions is to compactly represent X while supporting *range reporting queries*, that is, given a rectangle $R = [a_1, b_2] \times \cdots \times [a_d, b_d]$ report all points in the set $R \cap X$. We use the following results for 2-dimensional range reporting:

▶ **Lemma 9** (Chan et al. [6]). *For any set of n points in $[0, u] \times [0, u]$ and $2 \leq B \leq \lg^\epsilon n, 0 < \epsilon < 1$ we can solve 2-d orthogonal range reporting with $O(n \lg n)$ expected preprocessing time, $O(n \lg_B \lg n)$ space and $(1 + k) \cdot O(B \lg \lg u)$ query time where k is the number of occurrences inside the rectangle.*

Let $o_1, \ldots o_{\mathrm{occ}}$ be the starting positions of the occurrences of P in S ordered increasingly. Assume that o_h is a secondary occurrence such that $P = S[o_h, o_h + m - 1]$. Then by definition, $S[o_h, o_h + m - 1]$ is a substring the prefix $S[i, j - 1]$ of some phrase $S[i, j]$ and there must be an occurrence of P in the source of that phrase. More precise, let $S[k, l] = S[i, j - 1]$ be the source of the phrase $S[i, j]$ then $o_{h'} = k + o_h - i$ is an occurrence of P for some $h' < h$. We say that $o_{h'}$, which may be primary or secondary, is the source occurrence of the secondary occurrence o_h given the LZ77 parse of S. Thus every secondary occurrence has a source occurrence. Note that it follows from the definition that no primary occurrence has a source occurrence.

We find the secondary occurrences as follows: Build a range reporting data structure Q on the $n \times n$ grid and if $S[i, j]$ is a phrase with source $S[i', j']$ we plot a point (i', j') and along with it we store the phrase start i.

Now for each primary occurrence o found by the primary index, we query Q for the rectangle $[0, o] \times [o + m - 1, n]$. The points returned are exactly the occurrences having

o as source. For each point (x, y) and phrase start i reported, we report an occurrence $o' = i + o - x$ and recurse on o' to find all the occurrences having o' as source.

Because no primary occurrence have a source, while all secondary occurrences have a source, we will find exactly the secondary occurrences.

The range reporting structure Q is built using Lemma 9 with $B = 2$ and uses space $O(z \lg \lg z)$. Exactly one range reporting query is done for each primary and secondary occurrence each taking $O((1 + k) \lg \lg n)$ where k is the number of points reported. Each reported point identifies a secondary occurrence, so the total time is $O(\mathrm{occ} \lg \lg n)$.

From LZ77 to the Run-Length Encoded Burrows-Wheeler Transform, and Back

Alberto Policriti[1] **and Nicola Prezza**[*][2]

1 **University of Udine, Department of Informatics, Mathematics, and Physics, Udine, Italy; and Institute of Applied Genomics, Udine, Italy**
alberto.policriti@uniud.it

2 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
npre@dtu.dk

Abstract

The Lempel-Ziv factorization (LZ77) and the Run-Length encoded Burrows-Wheeler Transform (RLBWT) are two important tools in text compression and indexing, being their sizes z and r closely related to the amount of text self-repetitiveness. In this paper we consider the problem of converting the two representations into each other within a working space proportional to the input and the output. Let n be the text length. We show that $RLBWT$ can be converted to $LZ77$ in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space. Conversely, we provide an algorithm to convert $LZ77$ to $RLBWT$ in $\mathcal{O}\big(n(\log r + \log z)\big)$ time and $\mathcal{O}(r+z)$ words of working space. Note that r and z can be *constant* if the text is highly repetitive, and our algorithms can operate with (up to) *exponentially* less space than naive solutions based on full decompression.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Lempel-Ziv, Burrows-Wheeler transform, compressed computation, repetitive text collections

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.17

1 Introduction

The field of *compressed computation* – i.e. computation on compressed representations of the data without first fully decompressing it – is lately receiving much attention due to the ever-growing rate at which data is accumulating in archives such as the web or genomic databases. Being able to operate directly on the compressed data can make an enormous difference, considering that repetitive collections, such as sets of same-species genomes or software repositories, can be compressed at rates that often exceed 1000x. In such cases, this set of techniques makes it possible to perform most of the computation directly in primary memory and enables the possibility of manipulating huge datasets even on resource-limited machines.

Central in the field of compressed computation are *compressed data structures* such as compressed full-text indexes, geometry (e.g. 2D range search), trees, graphs. The compression of these structures (in particular those designed for unstructured data) is based on a set of techniques which include entropy compression, Lempel-Ziv parsings [16, 17] (LZ77/LZ78), grammar compression [6], and the Burrows-Wheeler transform [4] (BWT).

[*] Part of this work was done while the author was a PhD student at the University of Udine, Italy. Work supported by the Danish Research Council (DFF-4005-00267).

Grammar compression, Run-Length encoding of the BWT [14, 13] (RLBWT), and LZ77 have been shown superior in the task of compressing highly-repetitive data and, as a consequence, much research is lately focusing on these three techniques.

In this paper we address a central point in compressed computation: can we convert between different compressed representations of a text while using an amount of working space proportional to the input/output (i.e. sizes of the compressed files)? Being able to perform such task would, for instance, open the possibility of converting between compressed data structures (e.g. self-indexes) based on different compressors, all within compressed working space.

It is not the fist time that this problem has been addressed. In [12] the author shows how to convert the LZ77 encoding of a text into a grammar-based encoding, while in [2, 1] the opposite direction (though pointing to LZ78 instead of LZ77) is considered. In [15] the authors consider the conversions between LZ78 and run-length encoding of the text. Note that LZ77 and run-length encoding of the BWT are much more powerful than LZ78 and run-length encoding of the text, respectively, so methods addressing conversion between LZ77 and RLBWT would be of much higher interest. In this work we show how to efficiently solve this problem in space proportional to the sizes of these two compressed representations. See Basics section for a formal definition of $RLBWT(T)$ and $LZ77(T)$ as a list of r pairs and z triples, respectively. Let $RLBWT(T) \rightarrow LZ77(T)$ denote the computation of the list $LZ77(T)$ using as input the list $RLBWT(T)$ (analogously for the opposite direction). The following results are illustrated below:

(1) We can compute $RLBWT(T) \rightarrow LZ77(T)$ in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space

(2) We can compute $LZ77(T) \rightarrow RLBWT(T)$ in $\mathcal{O}\big(n(\log r + \log z)\big)$ time and $\mathcal{O}(r+z)$ words of working space

Result (1) is based on our own recent work [10] and requires space proportional to the input *only* as output is streamed to disk. Result (2) requires space proportional to the input *plus* the output, since data structures based on both compressors are used in main memory. In order to achieve result (2), we show how we can (locally) decompress $LZ77(T)$ while incrementally building a run-length BWT data structure of the reversed text. Extracting text from LZ77 is a computationally expensive task as it requires a time proportional to the parse height h per extracted character [8] (with h as large as n, in the worst case). The key ingredient of our solution is to use the run-length BWT data structure itself to efficiently extract text from $LZ77(T)$.

2 Basics

We assume that our text T is of the form $T = T'\# \in \Sigma^n$, with $T' \in (\Sigma \setminus \{\#\})^{n-1}$. Character $\#$ is lexicographically smaller than all elements in Σ and plays the role of both BWT and LZ77 terminators.

The *Burrows-Wheeler Transform* [4] $BWT(T)$ is a permutation of T defined as follows. Sort all cyclic permutations of T in a *conceptual* matrix $M \in \Sigma^{n\times n}$. $BWT(T)$ is the last column of M. With F and L we will denote the first and last column of M, respectively, and we will say *F-positions* and *L-positions* to refer to positions on these two columns. On compressible texts, $BWT(T)$ exhibits some remarkable properties that permit to boost compression. In particular, it can be shown [13] that repetitions in T generate equal-letter runs in $BWT(T)$. We can efficiently represent this transform as the list of pairs

$$RLBWT(T) = \langle \lambda_i, c_i \rangle_{i=1,\dots,r_T}$$

where $\lambda_i > 0$ is the length of the *maximal* i-th c_i-run, $c_i \in \Sigma$. Equivalently, $RLBWT(T)$ is the *shortest* list of pairs $\langle \lambda_i, c_i \rangle_{i=1,\dots,r_T}$ satisfying $BWT(T) = c_1^{\lambda_1} c_2^{\lambda_2} \dots c_{r_T}^{\lambda_{r_T}}$. Let $\overleftarrow{T}$ be the reverse of T. To simplify notation we define $r = \max\{r_T, r_{\overleftarrow{T}}\}$ (in practical cases $r_T \approx r_{\overleftarrow{T}}$ holds [3], and this definition simplifies notation).

▶ **Example 1.** Let `T = abcabbcaabcabcabbc#`. Then, `BWT(T) = ccccc#aaabbaaabbbbb` and `RLBWT(T) = ⟨5,c⟩⟨1,#⟩⟨3,a⟩⟨2,b⟩⟨3,a⟩⟨5,b⟩`. The Burrows-Wheeler transform has $r = 6$ equal-letter runs.

With $RLBWT^+(T)$ we denote a run-length encoded BWT *data structure* on the text T, taking $\mathcal{O}(r)$ words of space and supporting `insert`, `rank`, `select`, and `access` operation on the string $B = BWT(T)$. These operations are defined as follows:

- `insert(c,i)`, where $c \in \Sigma$ and $i < n$, turns B into $B[0,\dots,i-1]cB[i,\dots,n-1]$
- `rank(c,i)` returns the number of characters equal to c in $B[0,\dots,i-1]$
- `select(c,i)` returns the position j such that $B[c] = c$ and `rank(c,j)` $= i$
- `access(i)` returns $B[i]$

Using these operations, functions $RLBWT.LF(i)$ and $RLBWT.LF(j)$ (mapping L-positions to F-positions and *vice versa*) and function `extend` (turning $RLBWT^+(T)$ into $RLBWT^+(aT)$ for some $a \in \Sigma$) can be supported in $\mathcal{O}(\log r)$ time. We leave to the next sections details concerning the particular implementation of this data structure. With $RLBWT.LF^k(i)$ we denote the application of function LF k times starting from L-position i.

We recall that $BWT(\overleftarrow{T})$ can be built online with an algorithm that reads T-characters left-to-right and inserts them in a dynamic string data structure [7, 5]. Briefly, letting $a \in \Sigma$, the algorithm is based on the idea of backward-searching the extended reversed text $\overleftarrow{Ta}$ in the BWT index for $\overleftarrow{T}$. This operation leads to the F-position l where $\overleftarrow{Ta}$ should appear among all sorted $\overleftarrow{T}$'s suffixes. At this point, it is sufficient to insert $\#$ at position l in $BWT(\overleftarrow{T})$ and replace the old $\#$ with a to obtain $BWT(\overleftarrow{Ta})$.

The *LZ77 parsing* [16] (or *factorization*) of a text T is the sequence of *phrases* (or *factors*)

$$LZ77(T) = \langle \pi_i, \lambda_i, c_i \rangle_{i=1,\dots,z}$$

where $\pi_i \in \{0,\dots,n-1\} \cup \{\perp\}$ and $\perp$ stands for "undefined", $\lambda_i \in \{0,\dots,n-2\}$, $c_i \in \Sigma$, and:

1. $T = \omega_1 c_1 \dots \omega_z c_z$, with $\omega_i = \epsilon$ if $\lambda_i = 0$ and $\omega_i = T[\pi_i,\dots,\pi_i + \lambda_i - 1]$ otherwise.
2. For any $i = 1,\dots,z$, the string ω_i is the *longest* occurring at least twice in $\omega_1 c_1 \dots \omega_i$.

▶ **Example 2.** Let `T = abcabbcaabcabcabbc#`. The LZ77 factorization of the text is `a|b|c|abb|caa|bcabc|abbc#|`. This factorization can be compactly represented as the list of triples `LZ77(T) = ⟨⊥,0,a⟩⟨⊥,0,b⟩⟨⊥,0,c⟩⟨0,2,b⟩⟨2,2,a⟩⟨1,4,c⟩⟨3,4,#⟩`. The number of phrases is $z = 7$.

3 From RLBWT to LZ77

Our algorithm to compute $RLBWT(T) \to LZ77(T)$ is based on the result [10]: an algorithm to compute – in $\mathcal{O}(r)$ words of working space and $\mathcal{O}(n \log r)$ time – $LZ77(T)$ using T as input. The data structure at the core of this result is a dynamic run-length compressed string; we recall the bounds of such structure as we will use it several times in the rest of the paper:

▶ **Theorem 3** ([9, 10]). *Let $T \in \Sigma^n$ and let $\bar{r}$ be the number of equal-letter runs in T. There exists a data structure taking $\mathcal{O}(\bar{r})$ words of space and supporting `rank`, `select`, `access`, and `insert` operations on T in $\mathcal{O}(\log \bar{r})$ time.*

The algorithm described in [10] works in two steps, during the first of which builds $RLBWT^+(\overleftarrow{T})$ by inserting left-to-right T-characters in a *dynamic RLBWT* represented with the data structure of Theorem 3 – using the BWT construction procedure sketched in the previous section. In the second step, the procedure scans T once more left-to-right while searching (reversed) LZ77 phrases in $RLBWT^+(\overleftarrow{T})$. At the same time we store, for each BWT equal-letter run, the two most external (i.e. leftmost and rightmost in the run) text positions seen up to the current position; the key property proved in [10] is that this sampling is sufficient to locate LZ77 phrase boundaries and sources. LZ77 phrases are outputted in text order, therefore they can be directly streamed to output. The total size of the sampling of text positions never exceeds $2r$. From Theorem 3, all operations on $RLBWT^+(\overleftarrow{T})$ (`insert`, `LF`-mapping, `access`) are supported in $\mathcal{O}(\log r)$ time and the structure takes $\mathcal{O}(r)$ words of space. The claimed space/time bounds of the algorithm easily follow.

Note that, using the algorithm described in [10], we can only perform the conversion $RLBWT^+(\overleftarrow{T}) \to LZ77(T)$. Our full procedure to achieve conversion $RLBWT(T) \to LZ77(T)$ consists of the following three steps:

1. convert $RLBWT(T)$ to $RLBWT^+(T)$, i.e. add support for `rank`/`select`/`access` queries on $RLBWT(T)$;
2. compute $RLBWT^+(\overleftarrow{T})$ using $RLBWT^+(T)$;
3. run the algorithm described in [10] and compute $LZ77(T)$ using $RLBWT^+(\overleftarrow{T})$.

Let $RLBWT(T) = \langle \lambda_i, c_i \rangle_{i=1,\ldots,r}$ (see the previous section). Step 1 can be performed by just inserting characters $c_1^{\lambda_1} c_2^{\lambda_2} \ldots c_r^{\lambda_r}$ (in this order) in a dynamic run-length encoded string. Step 2 is performed by extracting characters $T[0], T[1], \ldots, T[n-1]$ from $RLBWT^+(T)$ and inserting them (in this order) in a dynamic $RLBWT$ data structure with the BWT construction algorithm sketched in the Section (2). Since this algorithm builds the $RLBWT$ of the *reversed* text, the final result is $RLBWT^+(\overleftarrow{T})$. We can state our first result:

▶ **Theorem 4.** *Conversion $RLBWT(T) \to LZ77(T)$ can be performed in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space.*

Proof. We use the dynamic RLBWT structure of Theorem 3 to implement components $RLBWT^+(T)$ and $RLBWT^+(\overleftarrow{T})$. Step 1 requires n `insert` operations in $RLBWT^+(T)$, and terminates therefore in $\mathcal{O}(n \log r)$ time. Since the string we are building contains r_T runs, this step uses $\mathcal{O}(r)$ words of working space. Step 2 calls n `extend` and `FL` queries on dynamic RLBWTs. `extend` requires a constant number of `rank` and `insert` operations [5]. FL function requires just an `access` and a `rank` on the F column and a `select` on the L column. From Theorem 3, all these operations are supported in $\mathcal{O}(\log r)$ time, so also step 2 terminates in $\mathcal{O}(n \log r)$ time. Recall that r is defined to be the maximum between the number of runs in $BWT(T)$ and $BWT(\overleftarrow{T})$. Since in this step we are building $RLBWT^+(\overleftarrow{T})$ using $RLBWT^+(T)$, the overall space is bounded by $\mathcal{O}(r)$ words. Finally, step 3 terminates in $\mathcal{O}(n \log r)$ time while using $\mathcal{O}(r)$ words of space [10]. The claimed bounds for our algorithm to compute $RLBWT(T) \to LZ77(T)$ follow. ◀

4 From LZ77 to RLBWT

Our strategy to convert $LZ77(T)$ to $RLBWT(T)$ consists of the following steps:

1. extract $T[0], T[1], \ldots, T[n-1]$ from $LZ77(T)$ and add them (one by one) in $RLBWT^+(\overleftarrow{T})$ (note: decompression is local. We discard $T[i]$ after inserting it in $RLBWT^+(\overleftarrow{T})$);
2. convert $RLBWT^+(\overleftarrow{T})$ to $RLBWT^+(T)$;
3. extract equal-letter runs from $RLBWT^+(T)$ and stream $RLBWT(T)$ to the output.

Step 2 is analogous to step 2 discussed in the previous section. Step 3 requires reading characters $RLBWT^+(T)[0]$, ..., $RLBWT^+(T)[n-1]$ (`access` queries on $RLBWT^+(T)$) and keeping in memory a character storing last run's head and a counter keeping track of last run's length. Whenever we open a new run, we stream last run's head and length to the output.

The problematic step is the first. As mentioned in the introduction, extracting a character from $LZ77(T)$ requires to follow a chain of character copies. In the worst case, the length h of this chain – also called the parse height (see [8] for a formal definition) – can be as large as n (even though in the average case h is small, see [8] for an experimental evaluation). Our observation is that, since we are building $RLBWT^+(\overleftarrow{T})$, we can use this component to extract text from $LZ77(T)$ *without* following the chain of LZ77-character copies: while decoding factor $\langle \pi_v, \lambda_v, c_v \rangle$, we convert text position π_v to its corresponding RLBWT position $j = RLBWT.LF^{\pi_v}(0)$ and extract λ_v characters by performing λ_v further LF queries from position j. Conceptually, this task could be achieved by directly performing π_v LF queries on the RLBWT starting from L-position 0. This is clearly not efficient as it would result in a quadratic-time strategy. In the next section we show how to compute $RLBWT.LF^{\pi_v}(0)$ in just $\mathcal{O}(\log z)$ time.

4.1 Dynamic functions

Considering that $RLBWT^+(\overleftarrow{T})$ is built incrementally, we need a data structure to encode a function $\mathcal{Z} : \{\pi_1, ..., \pi_z\} \rightarrow \{0, ..., n-1\}$ mapping those text positions that are the source of some LZ77 phrase to their corresponding $RLBWT$ positions. Moreover, the data structure must be *dynamic*, that is it must support the following three operations (see below the list for a description of how these operations will be used):

- `map`: $\mathcal{Z}(i)$. Compute the image of i
- `expand`: $\mathcal{Z}.expand(j)$. Set $\mathcal{Z}(i)$ to $\mathcal{Z}(i) + 1$ for every i such that $\mathcal{Z}(i) \geq j$
- `assign`: $\mathcal{Z}(i) \leftarrow j$. Call $\mathcal{Z}.expand(j)$ and set $\mathcal{Z}(i)$ to j

To keep the notation simple and light, we use the same symbol $\mathcal{Z}$ for the function as well as for the data structure representing it. We say that $\mathcal{Z}(i)$ is *defined* if, for some j, we executed an `assign` operation $\mathcal{Z}(i) \leftarrow j$ at some previous stage of the computation. For technical reasons that will be clear later, we restrict our attention to the case where we execute `assign` operations $\mathcal{Z}(i) \leftarrow j$ for increasing values of i, i.e. if $\mathcal{Z}(i_1) \leftarrow j_1, \ldots, \mathcal{Z}(i_q) \leftarrow j_q$ is the sequence (in temporal order) of the calls to `assign` on $\mathcal{Z}$, then $i_1 < \cdots < i_q$. This will be the case in our algorithm and, in particular, $i_1, \ldots, i_q$ will be the sorted non-null phrases sources $\pi_1, \ldots, \pi_z$. Finally, we assume that $\mathcal{Z}(i)$ is always called when $\mathcal{Z}(i)$ has already been defined – again, this will be the case in our algorithm.

Intuitively, $\mathcal{Z}.expand(j)$ will be used when we insert $T[i]$ at position j in the partial $RLBWT^+(\overleftarrow{T})$ and j is not associated with any phrase source (i.e. $i \neq \pi_v$ for all $v = 1, \ldots, z$). When we insert $T[i]$ at position j in the partial $RLBWT^+(\overleftarrow{T})$ and $i = \pi_v$ for some $v = 1, \ldots, z$ (possibly more than one), $\mathcal{Z}(i) \leftarrow j$ will be used.

The existence and associated query-costs of the data structure $\mathcal{Z}$ are proved in the following lemma.

▶ **Lemma 5.** *Letting z be the number of phrases in the LZ77 parsing of T, there exists a data structure taking $\mathcal{O}(z)$ words of space and supporting* `map`*,* `expand`*, and* `assign` *operations on $\mathcal{Z} : \{\pi_1, ..., \pi_z\} \rightarrow \{0, ..., n-1\}$ in $\mathcal{O}(\log z)$ time.*

Proof. First of all notice that, since $LZ77(T)$ is our input, we know beforehand the domain $\mathcal{D} = \{\pi \mid \langle \pi, \lambda, c\rangle \in LZ77(T) \wedge \pi \neq \bot\}$ of $\mathcal{Z}$. We can therefore map the domain to rank space and restrict our attention to functions $\mathcal{Z}' : \{0, ..., d-1\} \rightarrow \{0, ..., n-1\}$, with $d = |\mathcal{D}| \leq z$. To compute $\mathcal{Z}(i)$ we map text position $0 \leq i < n$ to a rank $0 \leq i' < d$ by binary-searching a precomputed array containing the sorted values of $\mathcal{D}$ and return $\mathcal{Z}'(i')$. Similarly, $\mathcal{Z}(i) \leftarrow j$ is implemented by executing $\mathcal{Z}'(i') \leftarrow j$ (with i' defined as above), and $\mathcal{Z}.expand(j)$ simply as $\mathcal{Z}'.expand(j)$.

We use a dynamic gap-encoded bitvector C marking (by setting a bit) those positions j such that $j = \mathcal{Z}(i)$ for some i. A dynamic gap-encoded bitvector with b bits set can easily be implemented using a red-black tree such that it takes $\mathcal{O}(b)$ words of space and supports `insert`, `rank`, `select`, and `access` operations in $\mathcal{O}(\log b)$ time; see [10] for such a reduction. Upon initialization of $\mathcal{Z}$, C is empty. Let k be the number of bits set in C at some step of the computation. We can furthermore restrict our attention to *surjective* functions $\mathcal{Z}'' : \{0, ..., d-1\} \rightarrow \{0, ..., k-1\}$ as follows. $\mathcal{Z}'(i')$ (`map`) returns $C.select_1(\mathcal{Z}''(i'))$. The `assign` operation $\mathcal{Z}'(i') \leftarrow j$ requires the `insert` operation $C.insert(1, j)$ followed by the execution of $\mathcal{Z}''(i') \leftarrow C.rank_1(j)$. Operation $\mathcal{Z}'.expand(j)$ is implemented with $C.insert(0, j)$.

To conclude, since we restrict our attention to the case where – when calling $\mathcal{Z}(i) \leftarrow j$ – argument i is greater than all i' such that $\mathcal{Z}(i')$ is defined, we will execute `assign` operations $\mathcal{Z}''(i') \leftarrow j''$ for increasing values of $i' = 0, 1, \ldots, d-1$. In particular, at each `assign` $\mathcal{Z}''(i') \leftarrow j''$, $i' = k$ will be the current domain size. We therefore focus on a new operation, `append`, denoted as $\mathcal{Z}''.append(j'')$ and whose effect is $Z''(k) \leftarrow j''$. We are left with the problem of finding a data structure for a *dynamic permutation* $\mathcal{Z}'' : \{0, ..., k-1\} \rightarrow \{0, ..., k-1\}$ with support for `map` and `append` operations. Note that both domain and codomain size (k) are incremented by one after every `append` operation.

▶ **Example 6.** Let $k = 5$ and $\mathcal{Z}''$ be the permutation $\langle 3, 1, 0, 4, 2\rangle$. After $\mathcal{Z}''.append(2)$, k increases to 6 and $\mathcal{Z}''$ turns into the permutation $\langle 4, 1, 0, 5, 3, 2\rangle$. Note that $\mathcal{Z}''.append(j'')$ has the following effect on the permutation: all numbers larger than or equal to j'' are incremented by one, and j'' is appended at the end of the permutation.

To implement the dynamic permutation $\mathcal{Z}''$, we use a red-black tree $\mathcal{T}$. We associate to each internal tree node x a counter storing the number of leaves contained in the subtree rooted in x. Let m be the size of the tree. The tree supports two operations:

- $\mathcal{T}.insert(j)$. Insert a new leaf at position j, i.e. the new leaf will be the j-th leaf to be visited in the in-order traversal of the tree. This operation can be implemented using subtree-size counters to guide the insertion. After the leaf has been inserted, we need to re-balance the tree (if necessary) and update at most $\mathcal{O}(\log m)$ subtree-size counters. The procedure returns (a pointer to) the tree leaf x just inserted. Overall, $\mathcal{T}.insert(j)$ takes $\mathcal{O}(\log m)$ time
- $\mathcal{T}.locate(x)$. Take as input a leaf in the red-black tree and return the (0-based) rank of the leaf among all leaves in the in-order traversal of the tree. $\mathcal{T}.locate(x)$ requires climbing the tree from x to the root and use subtree-size counters to retrieve the desired value, and therefore runs in $\mathcal{O}(\log m)$ time.

At this point, the dynamic permutation $\mathcal{Z}''$ is implemented using the tree described above and a vector N of red-black tree leaves supporting `append` operations (i.e. insert at the end of the vector). N can be implemented with a simple vector of words with initial capacity 1. Every time we need to add an element beyond the capacity of N, we re-allocate $2|N|$ words for the array. N supports therefore constant-time access and amortized constant-time append operations. Starting with empty $\mathcal{T}$ and N, we implement operations on $\mathcal{Z}''$ as follows:

- $\mathcal{Z}''.map(i)$ returns $\mathcal{T}.locate(N[i])$
- $\mathcal{Z}''.append(j)$ is implemented by calling $N.append(\mathcal{T}.insert(j))$

Taking into account all components used to implement our original dynamic function $\mathcal{Z}$, we get the bounds of our lemma. ◀

4.2 The algorithm

The steps of our algorithm to compute $RLBWT^+(\overleftarrow{T})$ from $LZ77(T)$ are the following:

1. sort $\mathcal{D} = \{\pi \mid \langle \pi, \lambda, c\rangle \in LZ77(T)\ \wedge \pi \neq \bot\}$;
2. process $\langle \pi_v, \lambda_v, c_v\rangle_{v=1,\ldots,z}$ from the first to last triple as follows. When processing $\langle \pi_v, \lambda_v, c_v\rangle$:
   a. use our dynamic function $\mathcal{Z}$ to convert text position π_v to RLBWT position $j' = \mathcal{Z}(\pi_v)$
   b. extract λ_v characters from RLBWT starting from position j' by using the LF function; at the same time, extend RLBWT with the extracted characters.
   c. when inserting a character at position j of the RLBWT, if j corresponds to some text position $i \in \mathcal{D}$, then update $\mathcal{Z}$ accordingly by setting $\mathcal{Z}(i) \leftarrow j$. If, instead, j does not correspond to any text position in $\mathcal{D}$, execute $\mathcal{Z}.expand(j)$.

Our algorithm is outlined below as Algorithm 1. Note that the pseudocode describes all 3 steps reported at the beginning of Section 4 (steps 2 and 3 are implicit in Line 26). Follows a detailed description of the pseudocode and a result stating its complexity.

In Lines 1–5 we initialize all structures and variables. In order: we compute and sort set $\mathcal{D}$ of phrase sources, we initialize current text position i (i is the position of the character to be read), we initialize an empty RLBWT data structure (we will build $RLBWT^+(\overleftarrow{T})$ online), and we create an empty dynamic function data structure $\mathcal{Z}$. In Line 6 we enter the main loop iterating over LZ77 factors. If the current phrase's source is not empty (i.e. if the phrase copies a previous portion of the text), we need to extract λ_v characters from the RLBWT. First, in Line 8 we retrieve the RLBWT position j' corresponding to text position π_v with a `map` query on $\mathcal{Z}$. Note that, if $\pi_v \neq \bot$, then $i > \pi_v$ and therefore $\mathcal{Z}(\pi_v)$ is defined (see next). We are ready to extract characters from RLBWT. For λ_v times, we repeat the following procedure (Lines 10–19). We read the l-th character from the source of the v-th phrase (Line 10) and insert it in the RLBWT (Line 11). Importantly, the `extend` operation at Line 11 returns the RLBWT position j at which the new character is inserted; RLBWT position j correspond to text position i. We now have to check if i is the source of some LZ77 phrase. If this is the case (Line 12), then we link text position i to RLBWT position j by calling a `assign` query on $\mathcal{Z}$ (Line 13). If, on the other hand, i is not the source of any phrase, then we call a `expand` query on $\mathcal{Z}$ on the codomain element j. Note that, after the `extend` query at Line 11, RLBWT positions after the j-th are shifted by one. If j' is one of such positions, then we increment it (Line 17). Finally, we increment text position i (Line 19). At this point, we finished copying characters from the v-th phrase's source (or we did not do anything if the v-th phrase consists of only one character). We therefore extend the RLBWT with the v-th trailing character (Line 20), and (as done before) associate text position i to RLBWT position j if i is the source of some phrase (Lines 21–24). We conclude the main loop by incrementing the current position i on the text (Line 25). Once all characters have been extracted from LZ77, RLBWT is a run-length BWT structure on

```
Algorithm 1: lz77_to_rlbwt($\langle \pi_v, \lambda_v, c_v \rangle_{v=1,\dots,z}$)
  input  : LZ77 factorization $LZ77(T) = \langle \pi_v, \lambda_v, c_v \rangle_{v=1,\dots,z}$ of a text $T$
  output : RLBWT representation $\langle \lambda_v, c_v \rangle_{v=1,\dots,r}$ of $T$

1  $\mathcal{D} \leftarrow \{\pi \mid \langle \pi, \lambda, c \rangle \in LZ77(T) \wedge \pi \neq \bot\}$;    /* Phrase sources */
2  $sort(\mathcal{D})$;                                   /* Sort phrase sources */
3  $i \leftarrow 0$;                                      /* Current position on $T$ */
4  $RLBWT \leftarrow \epsilon$;                           /* Init empty RLBWT of reversed text */
5  $\mathcal{Z} \leftarrow \emptyset$;                    /* Init empty dynamic function structure */
6  for $v = 1, \dots, z$ do
7      if $\pi_v \neq \bot$ then
8          $j' \leftarrow \mathcal{Z}(\pi_v)$;            /* Map text position to RLBWT position */
9          for $l = 1, \dots, \lambda_v$ do
10             $c \leftarrow RLBWT[j']$;                  /* read char from source */
11             $j \leftarrow RLBWT.extend(c)$;            /* left-extend reverse text's RLBWT */
12             if $i \in \mathcal{D}$ then
13                 $\mathcal{Z}(i) \leftarrow j$;         /* $j$ is the image of $i$ */
14             else
15                 $\mathcal{Z}.expand(j)$;               /* $j$ does not have counter-image */
16             if $j \leq j'$ then
17                 $j' \leftarrow j' + 1$;                /* new char falls before $j'$ */
18             $j' \leftarrow RLBWT.LF(j')$;
19             $i \leftarrow i + 1$;                      /* Advance text position */
20     $j \leftarrow RLBWT.extend(c_v)$;                  /* Extend with trailing character */
21     if $i \in \mathcal{D}$ then
22         $\mathcal{Z}(i) \leftarrow j$;
23     else
24         $\mathcal{Z}.expand(j)$;
25     $i \leftarrow i + 1$;                              /* Advance text position */
26 return $reverse(RLBWT)$;                               /* Build and return $RLBWT(T)$ */
```

$\overleftarrow{T}$. At Line 26 we convert it to $RLBWT^+(T)$ (see previous section) and return it as a series of pairs $\langle \lambda_v, c_v \rangle_{v=1,\dots,r}$.

▶ **Theorem 7.** *Algorithm 1 converts $LZ77(T) \rightarrow RLBWT(T)$ in $\mathcal{O}(n(\log r + \log z))$ time and $\mathcal{O}(r + z)$ words of working space.*

Proof. Sorting set $\mathcal{D}$ takes $\mathcal{O}(z \log z) \subseteq \mathcal{O}(n \log z)$ time. Overall, we perform $\mathcal{O}(z)$ `map`/`assign` and n `expand` queries on $\mathcal{Z}$. All these operations take globally $\mathcal{O}(n \log z)$ time. We use the structure of Theorem 3 to implement $RLBWT^+(T)$ and $RLBWT^+(\overleftarrow{T})$. We perform n `access`, `extend`, and `LF` queries on $RLBWT^+(\overleftarrow{T})$. This takes overall $\mathcal{O}(n \log r)$ time. Finally, inverting $RLBWT^+(\overleftarrow{T})$ at Line 26 takes $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of space (see previous section). We keep in memory the following structures: $\mathcal{D}$, $\mathcal{Z}$, $RLBWT^+(\overleftarrow{T})$, and $RLBWT^+(T)$. The bounds of our theorem easily follow. ◀

5 Conclusions

In this paper we presented space-efficient algorithms converting between two compressed file representations – the run-length Burrows-Wheeler transform (RLBWT) and the Lempel-Ziv 77 parsing (LZ77) – using a working space proportional to the input and the output. Both representations can be significantly (up to exponentially) smaller than the text; our solutions are therefore particularly useful in those cases in which the text does not fit in main memory but its compressed representation does. Another application of the results discussed in this paper is the optimal-space construction of compressed self-indexes based on these compression techniques (e.g. [3]) taking as input the RLBWT/LZ77 *compressed* file.

We point out two possible developments of our ideas. First of all, our algorithms rely heavily on dynamic data structures. On the experimental side, it has been shown (see, e.g., [11]) that algorithms based on compressed dynamic strings can be hundreds of times slower than others not making use of dynamism (despite offering very similar theoretical guarantees). This is due to factors ranging from cache misses to memory fragmentation; dynamic structures inherently incur into these problems as they need to perform a large number of memory allocations and de-allocations. A possible strategy for overcoming these difficulties could be to build the RLBWT by merging two static RLBWTs while using a working space proportional to the output size. A second improvement over our results concerns theoretical running times. We note that our algorithms perform a number of steps proportional to the size n of the text. Considering that the compressed file could be *exponentially* smaller than the text, it is natural to ask whether it is possible to perform the same tasks in a time proportional to $r + z$. This seems to be a much more difficult goal due to the intrinsic differences among the two compressors – one is based on suffix sorting, while the other on replacement of repetitions with pointers.

References

1 Hideo Bannai, Paweł Gawrychowski, Shunsuke Inenaga, and Masayuki Takeda. Converting SLP to LZ78 in almost linear time. In Johannes Fischer and Peter Sanders, editors, *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *LNCS*, pages 38–49. Springer, 2013. doi:10.1007/978-3-642-38905-4_6.

2 Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Efficient LZ78 factorization of grammar compressed text. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE 2012)*, volume 7608 of *LNCS*, pages 86–98. Springer, 2012. doi:10.1007/978-3-642-34109-0_10.

3 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 26–39. Springer, 2015. doi:10.1007/978-3-319-19929-0_3.

4 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Systems Research Center, 1994. URL: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf.

5 Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Trans. Algorithms*, 3(2):21, 2007. doi:10.1145/1240233.1240244.

6 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

7 Wing-Kai Hon, Tak-Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007. doi:10.1007/s00453-006-1228-8.

8 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.

9 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010. doi:10.1089/cmb.2009.0169.

10 Alberto Policriti and Nicola Prezza. Computing LZ77 in run-compressed space. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2016 Data Compression Conference (DCC 2016)*. IEEE, 2016. doi:10.1109/DCC.2016.30.

11 Nicola Prezza. A framework of dynamic data structures for string processing, 2017. arXiv:1701.07238.

12 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

13 Jouni Sirén. *Compressed full-text indexes for highly repetitive collections*. PhD thesis, University of Helsinki, June 2012. URL: http://urn.fi/URN:ISBN:978-952-10-8052-4.

14 Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In Amihood Amir, Andrew Turpin, and Alistair Moffat, editors, *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008)*, volume 5280 of *LNCS*, pages 164–175. Springer, 2009. doi:10.1007/978-3-540-89097-3_17.

15 Yuya Tamakoshi, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masanori Takeda. From run length encoding to LZ78 and back again. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2013 Data Compression Conference (DCC 2013)*, pages 143–152. IEEE, 2013. doi:10.1109/DCC.2013.22.

16 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.

17 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934.

Longest Common Extensions with Recompression[*]

Tomohiro I

Kyushu Institute of Technology, Fukuoka, Japan
tomohiro@ai.kyutech.ac.jp

Abstract

Given two positions i and j in a string T of length N, a *longest common extension (LCE) query* asks for the length of the longest common prefix between suffixes beginning at i and j. A compressed LCE data structure stores T in a compressed form while supporting fast LCE queries. In this article we show that the *recompression* technique is a powerful tool for compressed LCE data structures. We present a new compressed LCE data structure of size $O(z \lg(N/z))$ that supports LCE queries in $O(\lg N)$ time, where z is the size of Lempel-Ziv 77 factorization without self-reference of T. Given T as an uncompressed form, we show how to build our data structure in $O(N)$ time and space. Given T as a grammar compressed form, i.e., a straight-line program of size n generating T, we show how to build our data structure in $O(n \lg(N/n))$ time and $O(n + z \lg(N/z))$ space. Our algorithms are deterministic and always return correct answers.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Longest Common Extension (LCE) queries, compressed data structure, grammar compressed strings, Straight-Line Program (SLP)

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.18

1 Introduction

Given two positions i and j in a text T of length N, a *longest common extension (LCE) query* $\mathsf{LCE}(i, j)$ asks for the length of the longest common prefix between suffixes beginning at i and j. Since LCE queries play a central role in many string processing algorithms (see text book [6] for example), efficient LCE data structures have been extensively studied. If we are allowed to use $O(N)$ space, optimal $O(1)$ query time can be achieved by, e.g., lowest common ancestor queries [1] on the suffix tree of T. However, $O(N)$ space can be too expensive nowadays as the size of strings to be processed becomes quite large. Thus, recent studies focus on more space efficient solutions.

Roughly there are three scenarios: Several authors have studied tradeoffs among query time, construction time and data structure size [19, 5, 4, 21]; In [18], Prezza presented in-place LCE data structures showing that the memory space for storing T can be replaced with an LCE data structure while retaining optimal substring extraction time; LCE data structures working on grammar compressed representation of T were studied in [7, 3, 2, 17].

In this article we pursue the third scenario, which is advantageous when T is highly compressible. In grammar compression, T is represented by a Context Free Grammar (CFG) that generates T and only T. In particular CFGs in Chomsky normal form, called Straight Line Programs (SLPs), are often considered as any CFG can be easily transformed into an SLP without changing the order of grammar size. Let $\mathcal{S}$ be an arbitrary SLP of size n generating T. Bille et al. [2] showed a Monte Carlo randomized data structure of $O(n)$ space that supports LCE queries in $O(\lg N + \lg^2 \ell)$ time, where ℓ is the answer to the LCE query.

[*] I was supported by JSPS KAKENHI Grant Number 16K16009.

Because their algorithm is based on Karp-Rabin fingerprints, the answer is correct w.h.p (with high probability). If we always expect correct answers, we have to verify fingerprints in preprocessing phase, spending either $O(N \lg N)$ time (w.h.p.) and $O(N)$ space or $O(\frac{N^2}{n} \lg N)$ time (w.h.p.) and $O(n)$ space.

For a deterministic solution, I et al. [7] proposed an $O(n^2)$-space data structure, which can be built in $O(n^2 h)$ time and $O(n^2)$ space from $\mathcal{S}$, and supports LCE queries in $O(h \lg N)$ time, where h is the height of $\mathcal{S}$. As will be stated in Theorem 2, we outstrip this result.

Our work is most similar to that presented in [17]. They showed that the signature encoding [15] of T, a special kind of CFGs that can be stored in $O(z \lg N \lg^* N)$ space, can support LCE queries in $O(\lg N + \lg \ell \lg^* N)$ time, where z is the size of LZ77 factorization[1] of T and $\lg^*$ is the iterated logarithm. The signature encoding is based on the localy consistent parsing technique, which determines the parsing of a string by local surrounding. A key property of the signature encoding is that any occurrence of the same substring of length ℓ in T is guaranteed to be compressed in almost the same way leaving only $O(\lg \ell \lg^* N)$ discrepancies in its surrounding. As a result, an LCE query can be answered by tracing the $O(\lg \ell \lg^* N)$ surroundings created over two occurrences of the longest common extension. Since the cost $O(\lg N)$ is needed anyway to traverse the derivation tree of height $O(\lg N)$ from the root, an LCE query is supported in $O(\lg N + \lg \ell \lg^* N)$ time.

In this article we show that CFGs created by the *recompression* technique exhibit a similar property that can be used to answer LCE queries in $O(\lg N)$ time. In recent years recompression has been proved to be a powerful tool in problems related to grammar compression [8, 9, 10, 13] and word equations [11, 12]. The main component of recompression is to replace some pairs in a string with variables of the CFG. Although we use global information (like the frequencies of pairs in the string) to determine which pairs to be replaced, the pairing itself is done very locally, i.e., “all” occurrences of the pairs are replaced regardless of contexts. Then we can show that recompression compresses any occurrence of the same substring in T in almost the same way leaving only $O(\lg N)$ discrepancies in its surrounding. This leads to an $O(\lg N)$-time algorithm to answer LCE queries, improving the $O(\lg N + \lg \ell \lg^* N)$-time algorithm of [17]. We also improve the data structure size from $O(z \lg N \lg^* N)$ of [17][2] to $O(z \lg(N/z))$.

In [17], the authors proposed efficient algorithms to build their LCE data structure from various kinds of input as summarized in Table 1. We achieve a better and cleaner complexity to build our LCE data structure from SLPs. This has a great impact on compressed string processing, in which we are to solve problems on SLPs without decompressing the string explicitly. For instance, we can apply our result to the problems discussed in Section 7 of [17] and immediately improve the results (other than Theorem 17). It should be noted that the data structures in [17] also support efficient text edit operations. We are not sure if our data structures can be efficiently dynamized.

Theorems 1 and 2 show our main results. Note that our data structure is a simple CFG of height $O(\lg N)$ on which we can simulate the traversal of the derivation tree in constant time per move. Thus, it naturally supports $\mathsf{Extract}(i, \ell)$ queries, which asks for retrieving the substring $T[i..i + \ell - 1]$, in $O(\lg N + \ell)$ time.

[1] Note that there are several variants of LZ77 factorization. In this article we refer to the one that is known as the *f-factorization without self-reference* as LZ77 factorization unless otherwise noted.

[2] We believe that the space complexities of [17] can be improved to $O(z \lg(N/z) \lg^* N)$ by using the same trick we use in Lemma 13.

Table 1 Comparison of construction time and space between ours and [17], where N is the length of T, $\mathcal{S}$ is an SLP of size n generating T, z is the size of LZ77 factorization of T, and $f_{\mathcal{A}}$ is the time needed for predecessor queries on a set of $z \lg N \lg^* N$ integers from an N-element universe.

Input	Construction time	Construction space	Reference
T	$O(N f_{\mathcal{A}})$	$O(z \lg N \lg^* N)$	Theorem 3 (1a) of [17]
T	$O(N)$	$O(N)$	Theorem 3 (1b) of [17]
$\mathcal{S}$	$O(n f_{\mathcal{A}} \lg N \lg^* N)$	$O(n + z \lg N \lg^* N)$	Theorem 3 (3a) of [17]
$\mathcal{S}$	$O(n \lg\lg n \lg N \lg^* N)$	$O(n \lg^* N + z \lg N \lg^* N)$	Theorem 3 (3b) of [17]
LZ77	$O(z f_{\mathcal{A}} \lg N \lg^* N)$	$O(z \lg N \lg^* N)$	Theorem 3 (2) of [17]
T	$O(N)$	$O(N)$	this work, Theorem 1
$\mathcal{S}$	$O(n \lg(N/n))$	$O(n + z \lg(N/z))$	this work, Theorem 2
LZ77	$O(z \lg^2(N/z))$	$O(z \lg(N/z))$	this work, Corollary 3

▶ **Theorem 1.** *Given a string T of length N, we can compute in $O(N)$ time and space a compressed representation of T of size $O(z \lg(N/z))$ that supports* Extract(i, ℓ) *in $O(\lg N + \ell)$ time and* LCE *queries in $O(\lg N)$ time.*

▶ **Theorem 2.** *Given an SLP of size n generating a string T of length N, we can compute in $O(n \lg(N/n))$ time and $O(n + z \lg(N/z))$ space a compressed representation of T of size $O(z \lg(N/z))$ that supports* Extract(i, ℓ) *in $O(\lg N + \ell)$ time and* LCE *queries in $O(\lg N)$ time.*

Suppose that we are given the LZ77-compression of size z of T as an input. Since we can convert the input into an SLP of size $O(z \lg(N/z))$ [20], we can apply Theorem 2 to the SLP and get the next corollary.

▶ **Corollary 3.** *Given the LZ77-compression of size z of a string T of length N, we can compute in $O(z \lg^2(N/z))$ time and $O(z \lg(N/z))$ space a compressed representation of T of size $O(z \lg(N/z))$ that supports* Extract(i, ℓ) *in $O(\lg N + \ell)$ time and* LCE *queries in $O(\lg N)$ time.*

Technically, this work owes very much to two papers [10, 9]. For instance, our construction algorithm of Theorem 1 is essentially the same as the grammar compression algorithm [10], which produces an SLP of size $O(g^* \lg(N/g^*))$, where g^* is the smallest grammar size to generate T. Our contribution is in discovering the above mentioned property that can be used for fast LCE queries. Also, we use the property to upper bound the size of our data structure in terms of z rather than g^*. Since it is known that $z \leq g^*$ holds [20], an upper bound in terms of z is preferable. The technical issues in our construction algorithm of Theorem 2 have been tackled in [9], in which the recompression technique is used to solve the fully-compressed pattern matching problems. However, we make some contributions on top of it: We give a new observation that simplifies the implementation and analysis of a component of recompression called BComp (see Section 4.1.2). Also, we achieve a better construction time $O(n \lg(N/n))$ than what we obtain by straightforwardly applying the analysis in [9]—$O(n \lg N)$.

2 Preliminaries

An alphabet Σ is a set of characters. A string over Σ is an element in Σ^*. For any string $w \in \Sigma^*$, $|w|$ denotes the length of w. Let ε be the empty string, i.e., $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For any $1 \leq i \leq |w|$, $w[i]$ denotes the i-th character of w. For any $1 \leq i \leq j \leq |w|$, $w[i..j]$

denotes the substring of w beginning at i and ending at j. For convenience, let $w[i..j] = \varepsilon$ if $i > j$. For any $0 \leq i \leq |w|$, $w[1..i]$ (resp. $w[|w| - i + 1..|w|]$) is called the prefix (resp. suffix) of w of length i. We say that a string x *occurs* at position i in w iff $w[i..i + |x| - 1] = x$. A substring $w[i..j] = c^d$ ($c \in \Sigma, d \geq 1$) of w is called a *block* iff it is a maximal run of a single character, i.e., $(i = 1 \vee w[i-1] \neq c) \wedge (j = |w| \vee w[j+1] \neq c)$.

The text on which LCE queries are performed is denoted by $T \in \Sigma^*$ with $N = |T|$ throughout this paper. We assume that Σ is an integer alphabet $[1..N^{O(1)}]$ and the standard word RAM model with word size $\Omega(\lg N)$.

The size of our compressed LCE data structure is bounded by $O(z \lg(N/z))$, where z is the size of the LZ77 factorization of T defined as follows:

▶ **Definition 4** (LZ77 factorization). The factorization $T = f_1 f_2 \cdots f_z$ is the LZ77 factorization of T iff the following condition holds: For any $1 \leq i \leq z$, let $p_i = |f_1 f_2 \cdots f_{i-1}| + 1$, then $f_i = T[p_i]$ if $T[p_i]$ does not appear in $T[1..p_i - 1]$, otherwise f_i is the longest prefix of $T[p_i..N]$ that occurs in $T[1..p_i - 1]$.

In this article, we deal with grammar compressed strings, in which a string is represented by a Context Free Grammar (CFG) generating the string only. In particular, we consider *Straight-Line Programs (SLPs)* that are CFGs in Chomsky normal form. Formally, an SLP that generates a string T is a triple $\mathcal{S} = (\Sigma, \mathcal{V}, \mathcal{D})$, where Σ is the set of characters (terminals), $\mathcal{V}$ is the set of variables (non-terminals), $\mathcal{D}$ is the set of deterministic production rules whose righthand sides are in $\mathcal{V}^2 \cup \Sigma$, and the last variable derives T.[3] Let $n = |\mathcal{V}|$. We treat variables as integers in $[1..n]$ (which should be distinguishable from Σ by having extra one bit), and $\mathcal{D}$ as an injective function that maps a variable to its righthand side. We assume that given any variable X we can access in $O(1)$ time the information on X, e.g., $\mathcal{D}(X)$. We refer to n as the size of $\mathcal{S}$ since $\mathcal{S}$ can be encoded in $O(n)$ space. Note that N can be as large as 2^{n-1}, and so, SLPs have a potential to achieve exponential compression.

We extend SLPs by allowing run-length encoded rules whose righthand sides are of the form X^d with $X \in \mathcal{V}$ and $d \geq 2$, and call such CFGs *run-length SLPs (RLSLPs)*. Since a run-length encoded rule can be stored in $O(1)$ space, we still define the size of an RLSLP by the number of variables.

Let us consider the derivation tree $\mathcal{T}$ of an RLSLP $\mathcal{S}$ that generates a string T, where we delete all the nodes labeled with terminals for simplicity. That is, every node in $\mathcal{T}$ is labeled with a variable. The height of $\mathcal{S}$ is the height of $\mathcal{T}$. We say that a sequence $C = v_1 \cdots v_m$ of nodes is a *chain* iff the nodes are all adjacent in this order, i.e., the beginning position of v_{i+1} is the ending position of v_i plus one for any $1 \leq i < m$. C is labeled with the sequence of labels of $v_1 \cdots v_m$. For any sequence $p \in \mathcal{V}^*$ of variables, let $val_{\mathcal{S}}(p)$ denote the string obtained by concatenating the strings derived from all variables in the sequence. We omit $\mathcal{S}$ when it is clear from context. We say that p generates $val(p)$. Also, we say that p *occurs* at position i iff there is a chain that is labeled with p and begins at i.

The next lemma, which is somewhat standard for SLPs, also holds for RLSLPs.

▶ **Lemma 5.** *For any RLSLP $\mathcal{S}$ of height h generating T, by storing $|val(X)|$ for every variable X, we can support* Extract(i, ℓ) *in $O(h + \ell)$ time.*

[3] We treat the last variable as the starting variable.

3 LCE data structure built from uncompressed texts

In this section, we prove Theorem 1 by showing that the RLSLP obtained by grammar compression algorithm [9] based on recompression can be used for fast LCE queries. In Subsection 3.1 we review recompression and introduce notation we use. In Subsection 3.2 we present a new characterization of recompression, which is a key to our contributions.

3.1 TtoG: Grammar compression based on recompression

In [9] Jeż proposed an algorithm TtoG to compute an RLSLP of T in $O(N)$ time.[4] Let TtoG(T) denote the RLSLP of T produced by TtoG. We use the term *letters* for variables introduced by TtoG. In particular, we often refer to an occurrence of a sequence of letters, for which the readers should recall the definition of an occurrence of a sequence of variables. Also, we use c (rather than X) to represent a letter.

TtoG consists of two different types of compression, BComp and PComp, which stand for Block Compression and Pair Compression, respectively.

- BComp: Given a string w over $\Sigma = [1..|w|]$, BComp compresses w by replacing all blocks of length ≥ 2 with fresh letters. Note that BComp eliminates all blocks of length ≥ 2 in w. We can conduct BComp in $O(|w|)$ time and space (Lemma 6).
- PComp: Given a string w over $\Sigma = [1..|w|]$ that contains no block of length ≥ 2, PComp compresses w by replacing all pairs from $\acute{\Sigma}\grave{\Sigma}$ with fresh letters, where $(\acute{\Sigma}, \grave{\Sigma})$ is a partition of Σ, i.e., $\Sigma = \acute{\Sigma} \cup \grave{\Sigma}$ and $\acute{\Sigma} \cap \grave{\Sigma} = \emptyset$. We can deterministically compute in $O(|w|)$ time and space a partition of Σ by which at least $(|w| - 1)/4$ pairs are replaced (Lemma 7), and conduct PComp in $O(|w|)$ time and space (Lemma 8).

Let T_0 be a sequence of letters obtained by replacing every character c of T with a letter generating c. TtoG compresses T_0 by applying BComp and PComp by turns until the string gets shrunk into a single letter. Since PComp compresses a given string by a constant factor $3/4$, the height of TtoG(T) is $O(\lg N)$, and the total running time is bounded by $O(N)$.

In order to give a formal description we introduce some notation below. TtoG transforms level by level T_0 into strings, $T_1, T_2, \ldots, T_{\hat{h}}$, where $|T_{\hat{h}}| = 1$. For any $0 \leq h \leq \hat{h}$, we say that h is the *level* of T_h. If h is even, the transformation from T_h to T_{h+1} is performed by BComp, and production rules of the form $c \to \ddot{c}^d$ are introduced. If h is odd, the transformation from T_h to T_{h+1} is performed by PComp, and production rules of the form $c \to \acute{c}\grave{c}$ are introduced. Let Σ_h be the set of letters appearing in T_h. For any even h $(0 \leq h < \hat{h})$, let $\ddot{\Sigma}_h$ denote the set of letters with which there is a block of length ≥ 2 in T_h. For any odd h $(0 \leq h < \hat{h})$, let $(\acute{\Sigma}_h, \grave{\Sigma}_h)$ denote the partition of Σ_h used in PComp of level h.

Figure 1 shows an example of how TtoG compresses T_0.

The following four lemmas show how to conduct BComp, PComp, and thus, TtoG, efficiently, which are essentially the same as respectively Lemma 2, Lemma 5, Lemma 6, and Theorem 1, stated in [9]. We give the proofs in Appendix for the sake of completeness.

▶ **Lemma 6.** *Given a string w over $\Sigma = [1..|w|]$, we can conduct* BComp *in $O(|w|)$ time and space.*

For any string $w \in \Sigma^*$ that contains no block of length ≥ 2, let $\mathsf{Freq}_w(c, \tilde{c}, 0)$ (resp. $\mathsf{Freq}_w(c, \tilde{c}, 1)$) with $c > \tilde{c} \in \Sigma$ denote the number of occurrences of $c\tilde{c}$ (resp. $\tilde{c}c$) in w. We

[4] Indeed, the paper shows how to compute an "SLP" of size $O(g^* \lg(N/g^*))$, where g^* is the smallest SLP size to generate T. In order to estimate the number of SLP's variables needed to represent run-length encoded rules, its analysis becomes much involved.

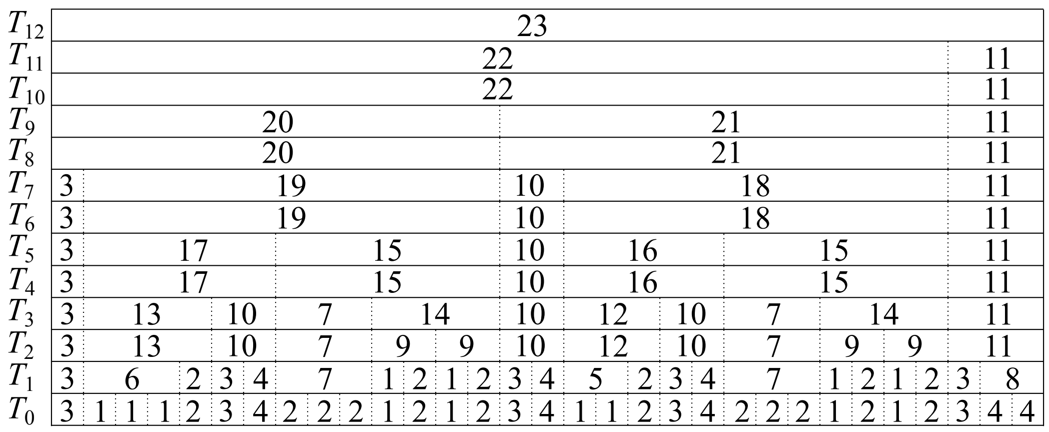

T_{12} 23
T_{11} 22 11
T_{10} 22 11
T_9 20 21 11
T_8 20 21 11
T_7 3 19 10 18 11
T_6 3 19 10 18 11
T_5 3 17 15 10 16 15 11
T_4 3 17 15 10 16 15 11
T_3 3 13 10 7 14 10 12 10 7 14 11
T_2 3 13 10 7 9 9 10 12 10 7 9 9 11
T_1 3 6 2 3 4 7 1 2 1 2 3 4 5 2 3 4 7 1 2 1 2 3 8
T_0 3 1 1 1 2 3 4 2 2 2 1 2 1 2 3 4 1 1 2 3 4 2 2 2 1 2 1 2 3 4 4

Figure 1 An example of how TtoG compresses T_0. Below we enumerate non-empty $\ddot{\Sigma}_h, \acute{\Sigma}_h, \grave{\Sigma}_h$ and production rules introduced in each level. From T_0 to T_1: $\ddot{\Sigma}_0 = \{1, 2, 4\}$, $\{5 \to 1^2, 6 \to 1^3, 7 \to 2^3, 8 \to 4^2\}$. From T_1 to T_2: $\acute{\Sigma}_1 = \{1, 3, 5, 6, 7\}$, $\grave{\Sigma}_1 = \{2, 4, 8\}$, $\{9 \to (1, 2), 10 \to (3, 4), 11 \to (3, 8), 12 \to (5, 2), 13 \to (6, 2)\}$. From T_2 to T_3: $\ddot{\Sigma}_2 = \{9\}$, $\{14 \to 9^2\}$. From T_3 to T_4: $\acute{\Sigma}_3 = \{3, 7, 12, 13\}$, $\grave{\Sigma}_3 = \{10, 14\}$, $\{15 \to (7, 14), 16 \to (12, 10), 17 \to (13, 10)\}$. From T_5 to T_6: $\acute{\Sigma}_5 = \{3, 10, 11, 16, 17\}$, $\grave{\Sigma}_5 = \{15\}$, $\{18 \to (16, 15), 19 \to (17, 15)\}$. From T_7 to T_8: $\acute{\Sigma}_7 = \{3, 10, 11\}$, $\grave{\Sigma}_7 = \{18, 19\}$, $\{20 \to (3, 19), 21 \to (10, 18)\}$. From T_9 to T_{10}: $\acute{\Sigma}_9 = \{11, 20\}$, $\grave{\Sigma}_9 = \{21\}$, $\{22 \to (20, 21)\}$. From T_{11} to T_{12}: $\acute{\Sigma}_{11} = \{22\}$, $\grave{\Sigma}_{11} = \{11\}$, $\{23 \to (22, 11)\}$.

refer to the list of non-zero $\mathsf{Freq}_w(c, \tilde{c}, \cdot)$ sorted in increasing order of c as the *adjacency list* of w. Note that it is a representation of the weighted directed graph in which there are exactly $\mathsf{Freq}_w(c, \tilde{c}, 0)$ (resp. $\mathsf{Freq}_w(c, \tilde{c}, 1)$) edges from c to $\tilde{c}$ (resp. from $\tilde{c}$ to c). Each occurrence of a pair in w is counted exactly once in the adjacency list. Then the problem of computing a good partition $(\acute{\Sigma}, \grave{\Sigma})$ of Σ reduces to maximum directed cut problem on the graph. Algorithm 1 is based on a simple greedy 1/4-approximation algorithm of maximum directed cut problem.

▶ **Lemma 7.** *Given the adjacency list of size m of a string $w \in \Sigma^*$, Algorithm 1 computes in $O(m)$ time a partition $(\acute{\Sigma}, \grave{\Sigma})$ of Σ such that the number of occurrences of pairs from $\acute{\Sigma}\grave{\Sigma}$ in w is at least $(|w| - 1)/4$.*

▶ **Lemma 8.** *Given a string w over $\Sigma = [1..|w|]$ that contains no block of length ≥ 2, we can conduct* PComp *in $O(|w|)$ time and space.*

▶ **Lemma 9.** *Given a string T over $\Sigma = [1..N^{O(1)}]$, we can compute* $\mathsf{TtoG}(T)$ *in $O(N)$ time and space.*

3.2 Popped sequences

We give a new characterization of recompression, which is a key to fast LCE queries as well as obtaining the upper bound $O(z \lg(N/z))$ for the size of $\mathsf{TtoG}(T)$. For any substring w of T, we define the *Popped Sequence (PSeq)*, denoted by *PSeq*(w), of w (formal definition is in the next paragraph). *PSeq*(w) is a sequence of letters such that *val*(*PSeq*$(w)) = w$ and consists of $O(\lg N)$ blocks of letters. It is not surprising that any substring can be represented by $O(\lg N)$ blocks of letters because the height of $\mathsf{TtoG}(T)$ is $O(\lg N)$. The significant property of *PSeq*(w) is that it occurs at "every" occurrence of w. A similar property has been observed in CFGs produced by locally consistent parsing and utilized for compressed indexes [14, 16]

Algorithm 1: How to compute a partition of Σ for PComp to compress w by 3/4.

```
Input: Adjacency list of $w \in \Sigma^*$.
Output: $(\acute{\Sigma}, \grave{\Sigma})$ s.t. # occurrences of pairs from $\acute{\Sigma}\grave{\Sigma}$ in $w$ is at least $(|w| - 1)/4$.
/* The information whether $c \in \Sigma$ is in $\acute{\Sigma}$ or $\grave{\Sigma}$ is written in the data
   space for $c$, which can be accessed in $O(1)$ time.                                  */
1 $\acute{\Sigma} \leftarrow \grave{\Sigma} \leftarrow \emptyset$;
2 foreach $c \in \Sigma$ in increasing order do
3     if $\sum_{\tilde{c} \in \grave{\Sigma}} \mathsf{Freq}_w(c, \tilde{c}, \cdot) \geq \sum_{\tilde{c} \in \acute{\Sigma}} \mathsf{Freq}_w(c, \tilde{c}, \cdot)$ then
4         add $c$ to $\acute{\Sigma}$;
5     else
6         add $c$ to $\grave{\Sigma}$;
7 if # occurrences of pairs from $\acute{\Sigma}\grave{\Sigma}$ < # occurrences of pairs from $\grave{\Sigma}\acute{\Sigma}$ then
8     switch $\acute{\Sigma}$ and $\grave{\Sigma}$;
9 return $(\acute{\Sigma}, \grave{\Sigma})$;
```

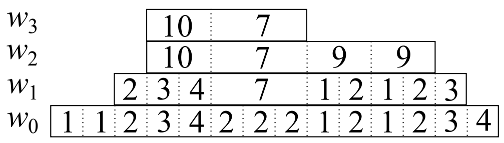

w_3 | 10 | 7
w_2 | 10 | 7 | 9 | 9
w_1 | 2 | 3 | 4 | 7 | 1 | 2 | 1 | 2 | 3
w_0 | 1 | 1 | 2 | 3 | 4 | 2 | 2 | 2 | 1 | 2 | 1 | 2 | 3 | 4

Figure 2 *PSeq* for $w_0 = [1, 1, 2, 3, 4, 2, 2, 2, 1, 2, 1, 2, 3, 4]$ under $\ddot{\Sigma}_h, \acute{\Sigma}_h, \grave{\Sigma}_h$ of Figure 1. At level 0, a block of 1 (resp. 4) is popped out from the leftend (resp. rightend) of w_0 because $1, 4 \in \ddot{\Sigma}_0$. At level 1, a letter 2 (resp. 3) is popped out from the leftend (resp. rightend) of w_1 because $2 \in \grave{\Sigma}_1$ and $3 \in \acute{\Sigma}_1$. At level 2, a block of 9 is popped out from the rightend of w_2 because $9 \in \ddot{\Sigma}_2$. At level 3, a letter 10 (resp. 7) is popped out from the leftend (resp. rightend) of w_3 because $10 \in \grave{\Sigma}_1$ and $7 \in \acute{\Sigma}_1$. Then, $PSeq(w_0) = [1, 1, 2, 10, 7, 9, 9, 3, 4]$. Observe that w_0 occurs twice in T_0 of Figure 1. and w_0, w_1, w_2 and w_3 are created over both occurrences. As a result, $PSeq(w_0)$ occurs everywhere w_0 occurs.

and a dynamic compressed LCE data structure [17]. For example, in [16, 17] the sequence having such a property is called the *common sequence* of w but its representation size is $O(\lg |w| \lg^* N)$ rather than $O(\lg N)$.

$PSeq(w)$ is the sequence of letters characterized by the following procedure. Let w_0 be the substring of T_0 that generates w. We consider applying BComp and PComp to w_0 exactly as we did to T but in each level we *pop* some letters out from both ends if the letters can be coupled with letters outside the scope. Formally, in increasing order of $h \geq 0$, we get w_{h+1} from w_h as follows:

- If h is even. We first pop out the leftmost and rightmost blocks of w_h if they are blocks of letter $c \in \ddot{\Sigma}_h$. Then we get w_{h+1} by applying BComp to the remaining string.
- If h is odd. We first pop out the leftmost letter and rightmost letter of w_h if they are letters in $\grave{\Sigma}_h$ and $\acute{\Sigma}_h$, respectively. Then we get w_{h+1} by applying PComp to the remaining string.

We iterate this until the string disappears. $PSeq(w)$ is the sequence obtained by concatenating the popped-out letters/blocks in an appropriate order, i.e., the order of the positions they occur. Note that for any occurrence of w the letters are compressed in the same way at least until they are popped out. Hence w_h is created for every occurrence of w and the occurrence of $PSeq(w)$ is guaranteed (see also Figure 2).

The next lemma formalizes the above discussion.

▶ **Lemma 10.** *For any substring w of T, PSeq(w) consists of $O(\lg N)$ blocks of letters. In addition, w occurs at position i iff PSeq(w) occurs at i.*

The next lemma and corollary are used to prove Lemmas 13 and 14.

▶ **Lemma 11.** *For any chain C whose label consists of m blocks of letters, the number of ancestor nodes of C is $O(m)$.*

▶ **Corollary 12.** *For any chain C corresponding to PSeq($T[b..e]$) for some interval $[b..e]$, the number of ancestor nodes of C is $O(\lg N)$.*

▶ **Lemma 13.** *The size of* $\mathsf{TtoG}(T)$ *is* $O(z \lg(N/z))$.

Proof. We first show the bound $O(z \lg N)$ and later improve the analysis to $O(z \lg(N/z))$.

Let $f_1 \dots f_z$ be the LZ77 factorization of T. For any $1 \le i \le z$, let L_i be the set of letters used in the ancestor nodes of leaves corresponding to the prefix $f_1 f_2 \dots f_i$. Clearly $|L_1| = O(\lg N)$. For any $1 < i \le z$, we estimate $|L_i \setminus L_{i-1}|$. Since f_i occurs in $f_1 \dots f_{i-1}$, we can see that the letters of *PSeq*(f_i) are in L_{i-1} thanks to Lemma 10. Let C_i be the chain corresponding to the occurrence $|f_1 \dots f_{i-1} + 1|$ of *PSeq*(f_i). Then, the letters in $L_i \setminus L_{i-1}$ are only in the labels of ancestor nodes of C_i. Since *PSeq*(f_i) consists of $O(\lg N)$ blocks of letters, $|L_i \setminus L_{i-1}|$ is bounded by $O(\lg N)$ due to Lemma 11. Therefore the size of $\mathsf{TtoG}(T)$ is $\sum_{i=1}^{z} |L_i \setminus L_{i-1}| = O(z \lg N)$.

In order to improve the bound to $O(z \lg(N/z))$, we employ the same trick that was used in [20, 9]. Let $h = 2 \lg_{4/3}(N/z) = 2 \lg_{3/4}(z/N)$. Recall that PComp compresses a given string by a constant factor 3/4. Since PComp has been applied $h/2$ times until the level h, $|T_h| \le N(3/4)^{h/2} = z$, and hence, the number of letters introduced in level $\ge h$ is bounded by $O(z)$. Then, we can ignore all the letters introduced in level $\ge h$ in the analysis of the previous paragraph, and by doing so, the bound $O(\lg N)$ of $|L_i \setminus L_{i-1}|$ is improved to $O(h) = O(\lg(N/z))$. This yields the bound $O(z \lg(N/z))$ for the size of $\mathsf{TtoG}(T)$. ◀

▶ **Lemma 14.** *Given* $\mathsf{TtoG}(T)$, *we can answer* $\mathsf{LCE}(i, j)$ *in* $O(\lg N)$ *time.*

Proof. We compute $\mathsf{LCE}(i, j)$ by matching the common sequence of letters occurring at i and j from left to right. First we traverse the derivation tree of $\mathsf{TtoG}(T)$ from the root down to the i-th and j-th leaves simultaneously while seeking the common block occurring at i and j. If there is no such block, $\mathsf{LCE}(i, j) = 0$, and we are done. Otherwise we stop at some internal nodes that contain the common block in their children. Let ℓ_1 be the length of the string generated by the block. Because $\mathsf{LCE}(i, j) \ge \ell_1$, we move on matching the next block by (possibly traversing up first and) traversing down to the $(i + \ell_1)$-th and $(j + \ell_1)$-th leaves. We iterate this procedure until we find no further common block. Then $\mathsf{LCE}(i, j) = \sum_{k=1}^{m} \ell_k$, where $\ell_1, \ell_2, \dots, \ell_m$ is the sequence of lengths of the common blocks we found.

Now we show that the above described algorithm runs in $O(\lg N)$ time. Note that it is bounded by the number of nodes we visit during the computation. In the light of Lemma 10, *PSeq*(w) occurs at both i and j, where w is the longest common prefix of two suffixes beginning at i and j. Let C_i (resp. C_j) be the chain that is labeled with *PSeq*(w) and begins at i (resp. j). Since the algorithm matches *PSeq*(w) or a succincter common sequence existing above C_i and C_j, we never go down below the parents of C_i or C_j during the computation. Hence, the number of visited nodes is bounded by the number of nodes that are ancestors of C_i or C_j, which is $O(\lg N)$ by Corollary 12. ◀

Theorem 1 is immediately from Lemmas 9, 5 and 14.

4 LCE data structure built from SLPs

In this section, we prove Theorem 2. Input is now an arbitrary SLP $\mathcal{S} = (\Sigma, \mathcal{V}, \mathcal{D})$ of size n generating T. Basically what we consider is to simulate TtoG on $\mathcal{S}$, namely, compute $\mathsf{TtoG}(T)$ without decompressing $\mathcal{S}$ explicitly. In Section 4.1, we present an algorithm $\mathsf{SimTtoG}$ that simulates TtoG in $O(n \lg^2(N/n))$ time and $O(n + z \lg(N/z))$ space. In Section 4.2, we present how to modify $\mathsf{SimTtoG}$ to get Theorem 2.

4.1 SimTtoG: Simulating TtoG on CFGs

We present an algorithm $\mathsf{SimTtoG}$ to simulate TtoG on $\mathcal{S}$. To begin with, we compute the CFG $\mathcal{S}_0 = (\Sigma_0, \mathcal{V}, \mathcal{D}_0)$ obtained by replacing, for all variables $X \in \mathcal{V}$ with $\mathcal{D}(X) \in \Sigma$, every occurrence of X in the righthand sides of $\mathcal{D}$ with the letter generating $\mathcal{D}(X)$. Note that Σ_0 is the set of terminals of $\mathcal{S}_0$, and $\mathcal{S}_0$ generates T_0. $\mathsf{SimTtoG}$ transforms level by level $\mathcal{S}_0$ into CFGs, $\mathcal{S}_1 = (\Sigma_1, \mathcal{V}, \mathcal{D}_1), \mathcal{S}_2 = (\Sigma_2, \mathcal{V}, \mathcal{D}_2), \ldots, \mathcal{S}_{\hat{h}} = (\Sigma_{\hat{h}}, \mathcal{V}, \mathcal{D}_{\hat{h}})$, where each $\mathcal{S}_h$ generates T_h. Namely, compression from T_h to T_{h+1} is simulated on $\mathcal{S}_h$. We can correctly compute the letters introduced in each level $h+1$ while modifying $\mathcal{S}_h$ into $\mathcal{S}_{h+1}$, and hence, we get all the letters of $\mathsf{TtoG}(T)$ in the end. We note that new "variables" are never introduced and the modification is done by rewriting righthand sides of the original variables.

Here we introduce the special formation of the CFGs $\mathcal{S}_h$ (it is a generalization of SLPs in a different sense from RLSLPs): For any $X \in \mathcal{V}$, $\mathcal{D}_h(X)$ consists of an "arbitrary number" of letters and at most "two" variables. More precisely, the following condition holds:

For any variable $X \in \mathcal{V}$ with $\mathcal{D}(X) = \acute{X}\grave{X}$, $\mathcal{D}_h(X)$ is either $w_1\acute{X}w_2\grave{X}w_3$, $w_1\acute{X}w_2$, $w_2\grave{X}w_3$ or w_2 with $w_1, w_2, w_3 \in \Sigma_h^*$, where $w_1 = w_3 = \varepsilon$ if X is not the starting variable.

As opposed to SLPs and RLSLPs, we define the size of $\mathcal{S}_h$ by the total lengths of righthand sides and denote it by $|\mathcal{S}_h|$.

4.1.1 PComp on CFGs

We firstly demonstrate that the adjacency list of T_h can be computed efficiently.

▶ **Lemma 15** (Lemma 6.1 of [10]). *For any odd h $(0 \leq h < \hat{h})$, the adjacency list of T_h, whose size is $O(|\mathcal{S}_h|)$, can be computed in $O(|\mathcal{S}_h| + n)$ time and space.*

Proof. For any variable $X \in \mathcal{V}$, let $\mathsf{VOcc}(X)$ denote the number of occurrences of the nodes labeled with X in the derivation tree of $\mathcal{S}$. It is well known that $\mathsf{VOcc}(X)$ for all variables can be computed in $O(n)$ time and space on the DAG representation of the tree.[5] Also, for any variable $X \in \mathcal{V}$, let $\mathsf{LML}(X)$ and $\mathsf{RML}(X)$ denote the leftmost letter and respectively rightmost letter of $val_{\mathcal{S}_h}(X)$. We can compute $\mathsf{LML}(X)$ for all variables in $O(|\mathcal{S}_h|)$ time by a bottom up computation, i.e., $\mathsf{LML}(X) = \mathsf{LML}(Y)$ if $\mathcal{D}_h(X)$ starts with a variable Y, and $\mathsf{LML}(X) = w[1]$ if $\mathcal{D}_h(X)$ starts with a non-empty string w. In a completely symmetric way $\mathsf{RML}(X)$ can be computed in $O(|\mathcal{S}_h|)$ time.

Now observe that any occurrence i of a pair $\acute{c}\grave{c}$ in T_h can be uniquely associated with a variable X that is the label of the lowest node covering the interval $[i..i+1]$ in the derivation tree of $\mathcal{S}_h$ (recall that $\mathcal{S}_h$ generates T_h). We intend to count all the occurrences of pairs associated with X in $\mathcal{D}_h(X)$. For example, let $\mathcal{D}_h(X) = \acute{X}w_2\grave{X}$ with $w_2 \in \Sigma_h^*$. Then $\acute{c}\grave{c}$

[5] It is sufficient to compute $\mathsf{VOcc}(X)$ once at the very beginning of $\mathsf{SimTtoG}$.

appears *explicitly* in w_2 or *crosses* the boundaries of $\acute{X}$ and/or $\grave{X}$. If $\acute{c}\grave{c}$ crosses the boundary of $\acute{X}$, $\mathsf{RML}(\acute{X})$ is $\acute{c}$ and $\grave{c}$ follows, i.e., $(w_2[1] = \grave{c}) \vee (w_2 = \varepsilon \wedge \mathsf{LML}(\grave{X}) = \grave{c})$. Using $\mathsf{RML}(\acute{X})$ and $\mathsf{LML}(\grave{X})$, we can compute in $O(|\mathcal{D}_h(X)|)$ time and space a $(|\mathcal{D}_h(X)| - 1)$-size multiset that lists all the explicit and crossing pairs in $\mathcal{D}_h(X)$. Each pair $\acute{c}\grave{c}$ with $\acute{c} > \grave{c}$ (resp. $\acute{c} < \grave{c}$) is listed by a quadruple $(\acute{c}, \grave{c}, 0, \mathsf{VOcc}(X))$ (resp. $(\grave{c}, \acute{c}, 1, \mathsf{VOcc}(X))$. $\mathsf{VOcc}(X)$ means that the pair has a weight $\mathsf{VOcc}(X)$ because the pair appears every time a node labeled with X appears in the derivation tree.

We compute such a multiset for every variable, which takes $O(|\mathcal{S}_h|)$ time and space in total. Next we sort the obtained list in increasing order of the first three integers in a quadruple. Note that the maximum value of letters is $O(z \lg(N/z))$ due to Lemma 13, and $O(z \lg(N/z)) = O(n^2)$ since $z \leq n$ and $\lg N \leq n$ hold. Thus the sorting can be done in $O(n)$ time and space by radix sort. Finally we can get the adjacency list of T_h by summing up weights of the same pair. The size of the list is clearly $O(|\mathcal{S}_h|)$. ◀

The next lemma shows how to implement PComp on CFGs:

▶ **Lemma 16.** *For any odd h $(0 \leq h < \hat{h})$, we can compute $\mathcal{S}_{h+1}$ from $\mathcal{S}_h$ in $O(|\mathcal{S}_h| + n)$ time and space. In addition, $|\mathcal{S}_{h+1}| \leq |\mathcal{S}_h| + 2n$.*

Proof. We first compute the partition $(\acute{\Sigma}_h, \grave{\Sigma}_h)$ of Σ_h, which can be done in $O(|\mathcal{S}_h| + n)$ time and space by Lemmas 15 and 7.

Given $(\acute{\Sigma}_h, \grave{\Sigma}_h)$, we can detect all the positions of the pairs from $\acute{\Sigma}_h\grave{\Sigma}_h$ in the righthands of $\mathcal{D}_h$, which are to be compressed. Some of the appearances of the pairs are explicit and the others are crossing. While explicit pairs can be compressed easily, crossing pairs need additional treatment. To deal with crossing pairs, we first *uncross* them by popping out every $\mathsf{LML}(Y) \in \grave{\Sigma}_h$ and $\mathsf{RML}(Y) \in \acute{\Sigma}_h$ from $val_{\mathcal{S}_h}(Y)$ and popping them into the appropriate positions in the other rules. More precisely, we do the followings (for technical reason, do PopInLet first):

PopInLet. For any variable X, if $\mathcal{D}_h(X)[i] = Y \in \mathcal{V}$ with $i > 1$ ($i \geq 1$ if X is the starting variable) and $\mathsf{LML}(Y) \in \grave{\Sigma}_h$, replace the occurrence of Y with $\mathsf{LML}(Y)Y$; if $\mathcal{D}_h(X)[i] = Y \in \mathcal{V}$ with $i < |\mathcal{D}_h(X)|$ ($i \leq |\mathcal{D}_h(X)|$ if X is the starting variable) and $\mathsf{RML}(Y) \in \acute{\Sigma}_h$, replace the occurrence of Y with $Y\mathsf{RML}(Y)$.

PopOutLet. For any variable X other than the starting variable, if $\mathcal{D}_h(X)[1] \in \grave{\Sigma}_h$, remove the first letter of $\mathcal{D}_h(X)$; and if $\mathcal{D}_h(X)[|\mathcal{D}_h(X)|] \in \acute{\Sigma}_h$, remove the last letter of $\mathcal{D}_h(X)$. In addition, if X becomes empty, we remove all the appearances of X in $\mathcal{D}_h$.

PopOutLet removes $\mathsf{LML}(Y) \in \grave{\Sigma}_h$ and $\mathsf{RML}(Y) \in \acute{\Sigma}_h$ from $val_{\mathcal{S}_h}(Y)$ (which can be a part of a crossing pair), and PopInLet introduces the removed letters into appropriate positions in $\mathcal{D}_h$ so that the modified $\mathcal{S}_h$ keeps to generate T_h. Notice that for each variable X the positions where letters popped in is at most two (four if X is the starting variable) and there is at least one variable that has no variables below, and hence, no letters popped in. Thus, the size of $\mathcal{S}_h$ increases at most $2n$. The uncrossing can be conducted in $O(|\mathcal{S}_h| + n)$ time.

Since all the pairs to be compressed become explicit, we can conduct BComp in $O(|\mathcal{S}_h|+n)$ time as follows. We scan righthand sides in $O(|\mathcal{S}_h|)$ time and list all the occurrences of pairs to be compressed. Each occurrence of pair $\acute{c}\grave{c} \in \acute{\Sigma}\grave{\Sigma}$ is listed by a triple $(\acute{c}, \grave{c}, p)$, where p is the pointer to the occurrence. Then we sort the list according to the pair of integers $(\acute{c}, \grave{c})$, which can be done in $O(|\mathcal{S}_h| + n)$ time and space by radix sort because $\acute{c}$ and $\grave{c}$ are $O(n^2)$. Finally, we replace each pair at position p with a fresh letter based on the rank of $(\acute{c}, \grave{c})$. ◀

4.1.2 BComp on CFGs

For any even h $(0 \leq h < \hat{h})$, BComp can be implemented in a similar way to PComp of Lemma 16. A block $T_h[b..e]$ of length ≥ 2 is uniquely associated with a variable X that is the label of the lowest node covering the interval $[b-1..e+1]$ in the derivation tree of $\mathcal{S}_h$ (if $b = 0$ or $e = |T_h|$, the block is associated with the starting variable). Here we take $[b-1..e+1]$ rather than $[b..e]$ to be sure that the block cannot extend outside the variable. Some blocks are explicitly written in $\mathcal{D}_h(X)$ and the others are crossing the boundaries of variables in $\mathcal{D}_h(X)$. The numbers of explicit blocks and crossing blocks in $\mathcal{D}_h$ is at most $|\mathcal{S}_h|$ and $2n$, respectively. The crossing blocks can be uncrossed in a similar way to uncrossing pairs. Then BComp can be done by replacing all the blocks with fresh letters on righthand sides of $\mathcal{D}_h$.

However here we have a problem. In order to give a unique letter to a block c^d, we have to sort the pairs of integers (c, d). Since d might be exponentially larger than $|\mathcal{S}_h| + n$, radix sort cannot be executed in $O(|\mathcal{S}_h| + n)$ time and space. In Section 6.2 of [10], Jeż showed how to solve this problem by tweaking the representation of lengths of long blocks, but its implementation and analysis are involved.[6]

We show in Lemma 17 our new observation, which leads to a simpler implementation and analysis of BComp. We say that a block c^d is *short* if $d = O(|\mathcal{S}_h| + n)$ and *long* otherwise. Also, we say that a variable is *unary* iff its righthand side consists of a single block.

▶ **Lemma 17.** *For any even h $(0 \leq h < \hat{h})$, a block $T_h[b..e] = c^d$ is short if it does not include a substring generated from a unary variable.*

Proof. Consider the derivation tree of $\mathcal{S}_h$ and the shortest path from $T_h[b]$ to $T_h[e]$. Let $X_1 X_2 \cdots X_{m'} \cdots X_m$ be the sequence of labels of internal nodes on the path, where $X_{m'}$ corresponds to the lowest common ancestor of $T_h[b]$ and $T_h[e]$. Since SLPs have no loops in the derivation tree, $X_1, \ldots, X_{m'}$ are all distinct. Similarly $X_{m'+1}, \ldots, X_m$ are all distinct. Since a unary variable is not involved to generate the block, it is easy to see that $d \leq \sum_{i=1}^{m} |\mathcal{D}_h(X_i)| \leq 2|\mathcal{S}_h|$ holds. ◀

Lemma 17 implies that most of blocks we find during the compression are short, which can be sorted efficiently by radix sort. If there is a long block in $\mathcal{D}_h$, an occurrence of a unary variable X must be involved to generate the block. Since BComp at level h pops out all the letters from X and removes the occurrences of X in $\mathcal{D}_h$, there are at most $2n$ long blocks in total. The number of long blocks can also be upper bounded by $2N/n$ with a different analysis based on the following fact:

▶ **Fact 18.** *If a substring of original text T generated from a long block overlaps with that generated from another long block, one substring must include the other, and moreover, the shorter block is completely included in "one" letter of the longer block. Hence the length of the substring of the longer block is at least n times longer than that of the shorter block.*

Let us consider the long blocks that generate substrings whose lengths are $[n^i..n^{i+1})$ for a fixed integer $i \geq 1$. By Fact 18, the substrings cannot overlap, and hence, the number of such long blocks is at most N/n^i. Therefore, the total number of long blocks is at most $\sum_{i \geq 1} N/n^i \leq 2N/n$. Thus we get the following lemma.

[6] Note that Section 6.2 of [10] also takes care of the case where the word size is $\Theta(\lg n)$ rather than $\Theta(\lg N)$. We do not consider the $\Theta(\lg n)$-bits model in this paper because using $\Theta(\lg N)$ bits to store the length of string generated by every letter is crucial for extract and LCE queries. However, we believe that our new observation stated in Lemma 17 will simplify the analysis for the $\Theta(\lg n)$-bits model, too.

▶ **Lemma 19.** *There are at most $O(\min(n, N/n))$ long blocks found during* SimTtoG*.*

By Lemma 19, we can employ a standard comparison-based sorting algorithm to sort all long blocks in $O(n \lg(\min(n, N/n)))$ time in total. In particular, BComp of one level can be implemented in the following complexities:

▶ **Lemma 20.** *For any even h ($0 \leq h < \hat{h}$), we can compute $\mathcal{S}_{h+1}$ from $\mathcal{S}_h$ in $O(|\mathcal{S}_h| + n + m \lg m))$ time and $O(|\mathcal{S}_h| + n)$ space, where m is the number of long blocks in $\mathcal{D}_h$. In addition, $|\mathcal{S}_{h+1}| \leq |\mathcal{S}_h| + 2n$.*

4.1.3 The complexities of SimTtoG

▶ **Theorem 21.** SimTtoG *runs in $O(n \lg^2(N/n))$ time and $O(n \lg(N/n))$ space.*

Proof. Using PComp and BComp implemented on CFGs (see Lemma 16 and 20), SimTtoG transforms level by level $\mathcal{S}_0$ into $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_{\hat{h}}$. In each level, the size of CFGs can increase at most $2n$ by the procedure of uncrossing. Since $|\mathcal{S}_h| = O(n \lg N)$ for any h ($0 \leq h < \hat{h}$), we get the time complexity $O(n \lg^2 N)$ by simply applying Lemmas 16 and 20.

We can improve it to $O(n \lg^2(N/n))$ by a similar trick used in the proof of Lemma 13. At some level h' where $|T_{h'}|$ becomes less than n, we decompress $\mathcal{S}_{h'}$ and switch to TtoG, which transforms $T_{h'}$ into $T_{\hat{h}}$ in $O(n)$ time by Lemma 9. We apply Lemmas 16 and 20 only for h with $0 \leq h < h'$. Since $h' = O(\lg(N/n))$, $|\mathcal{S}_h| = O(n \lg(N/n))$ for any h ($0 \leq h < h'$). Hence, we get the time complexity $O(n \lg^2(N/n))$. The space complexity is bounded by the maximum size of CFGs $\mathcal{S}_0, \mathcal{S}_1, \dots, \mathcal{S}_{h'}$, which is $O(n \lg(N/n))$. ◀

4.2 GtoG: $O(n \lg(N/n))$-time recompression

We modify SimTtoG slightly to run in $O(n \lg(N/n))$ time and $O(n + z \lg(N/z))$ space. The idea is the same as what has been presented in Section 6.1 of [10]. The problem of SimTtoG is that the sizes of intermediate CFGs $\mathcal{S}_h$ can grow up to $O(n \lg(N/n))$. If we can keep their sizes to $O(n)$, everything goes fine. This can be achieved by using two different types of partitions of Σ_h for PComp: One is for compressing T_h by a constant factor, and the other for compressing $|\mathcal{S}_h|$ by a constant factor (unless $|\mathcal{S}_h|$ is too small to compress). Recall that the former partition has been used in TtoG and SimTtoG, and the partition is computed from the adjacency list of T_h by Algorithm 1. Algorithm 1 can be extended to work on a set of strings by just inputting the adjacency list from a set of strings. Then, we can compute the partition for compressing $|\mathcal{S}_h|$ by a constant factor by considering the adjacency list from a set of strings in the righthand sides of $\mathcal{D}_h$. The adjacency list can be easily computed in $O(|\mathcal{S}_h| + n)$ time and space by modifying the algorithm described in the proof of Lemma 15: We just ignore the weight $\mathsf{VOcc}(X)$, i.e., use a unit weight 1 for every listed pair. Using the two types of partitions alternately, we can compress strings by a constant factor while keeping the sizes of the intermediate CFGs to $O(n)$.

We denote the modified algorithm by GtoG and the resulting RLSLP by GtoG($\mathcal{S}$). Note that GtoG($\mathcal{S}$) is not identical to TtoG(T) in general because the partitions used in GtoG depend on the input $\mathcal{S}$. Still the height of GtoG($\mathcal{S}$) is $O(\lg N)$ and the properties of *PSeq*s hold. Hence we can support LCE queries on GtoG($\mathcal{S}$) as we did on TtoG(T) by Lemma 14.

4.3 Proof of Theorem 2

Proof of Theorem 2. Let $\mathcal{S}$ be an input SLP of size n generating T. We compute GtoG($\mathcal{S}$) in $O(n \lg(N/n))$ time and $O(n + z \lg(N/z))$ space as described in Section 4.2. Since the height of $\mathsf{GtoG}(\mathcal{S})$ is $O(\lg N)$, we can support $\mathsf{Extract}(i, \ell)$ queries in $O(\lg N + \ell)$ time due to Lemma 5. $\mathsf{GtoG}(\mathcal{S})$ supports LCE queries in $O(\lg N)$ time in the same way as what was described in Lemma 14. ◀

References

1 Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005. doi:10.1016/j.jalgor.2005.08.001.

2 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. Finger search in grammar-compressed strings. In Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen, editors, *Proceedings of the 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016)*, volume 65 of *LIPIcs*, pages 36:1–36:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.FSTTCS.2016.36.

3 Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. In Frank Dehne, Roberto Solis-Oba, and Jörg-Rüdiger Sack, editors, *Proceedings of the 13th International Symposium on Algorithms and Data Structures (WADS 2013)*, volume 8037 of *LNCS*, pages 146–157. Springer, 2013. doi:10.1007/978-3-642-40104-6_13.

4 Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 65–76. Springer, 2015. doi:10.1007/978-3-319-19929-0_6.

5 Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time-space trade-offs for longest common extensions. *J. Discrete Algorithms*, 25:42–50, 2014. doi:10.1016/j.jda.2013.06.003.

6 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

7 Tomohiro I, Wataru Matsubara, Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Kazuyuki Narisawa, and Ayumi Shinohara. Detecting regularities on grammar-compressed strings. *Inf. Comput.*, 240:74–89, 2015. doi:10.1016/j.ic.2014.09.009.

8 Artur Jeż. Compressed membership for NFA (DFA) with compressed labels is in NP (P). In Christoph Dürr and Thomas Wilke, editors, *Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science (STACS 2012)*, volume 14 of *LIPIcs*, pages 136–147. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.STACS.2012.136.

9 Artur Jeż. Approximation of grammar-based compression via recompression. *Theor. Comput. Sci.*, 592:115–134, 2015. doi:10.1016/j.tcs.2015.05.027.

10 Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Trans. Algorithms*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.

11 Artur Jeż. One-variable word equations in linear time. *Algorithmica*, 74(1):1–48, 2016. doi:10.1007/s00453-014-9931-3.

12 Artur Jeż. Recompression: A simple and powerful technique for word equations. *J. ACM*, 63(1):4, 2016. doi:10.1145/2743014.

13 Artur Jeż and Markus Lohrey. Approximation of smallest linear tree grammar. In Ernst W. Mayr and Natacha Portier, editors, *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *LIPIcs*, pages 445–457. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.445.

14 Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. Esp-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms*, 18:100–112, 2013. `doi:10.1016/j.jda.2012.07.009`.

15 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. `doi:10.1007/BF02522825`.

16 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference (PSC 2016)*, pages 158–170, 2016. URL: `http://www.stringology.org/event/2016/p14.html`.

17 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *LIPIcs*, pages 72:1–72:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.MFCS.2016.72`.

18 Nicola Prezza. In-place longest common extensions, 2017. `arXiv:1608.05100v9`.

19 Simon J. Puglisi and Andrew Turpin. Space-time tradeoffs for longest-common-prefix array computation. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC 2008)*, volume 5369 of *LNCS*, pages 124–135. Springer, 2008. `doi:10.1007/978-3-540-92182-0_14`.

20 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1–3):211–222, 2003. `doi:10.1016/S0304-3975(02)00777-6`.

21 Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In Roberto Grossi and Moshe Lewenstein, editors, *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *LIPIcs*, pages 1:1–1:10. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.CPM.2016.1`.

A Appendix: Omitted proofs

A.1 Proof of Lemma 6

Proof. We first scan w in $O(|w|)$ time and list all the blocks of length ≥ 2. Each block c^d $(c \in \Sigma, d \geq 2)$ at position i is listed by a triple (c, d, i) of integers in Σ. Next we sort the list according to the pair of integers (c, d), which can be done in $O(|w|)$ time and space by radix sort. Finally, we replace each block c^d by a fresh letter based on the rank of (c, d). ◀

A.2 Proof of Lemma 7

Proof. In the foreach loop, we first run a 1/2-approximation algorithm of maximum "undirected" cut problem on the adjacency list, i.e., we ignore the direction of the edges here. For each c in increasing order, we greedily determine whether c is added to $\acute{\Sigma}$ or to $\grave{\Sigma}$ depending on $\sum_{\tilde{c} \in \grave{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot) \geq \sum_{\tilde{c} \in \acute{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot)$. Note that $\sum_{\tilde{c} \in \grave{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot)$ (resp. $\sum_{\tilde{c} \in \acute{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot)$) represents the number of edges between c and a character in $\grave{\Sigma}$ (resp. $\acute{\Sigma}$). By greedy choice, at least half of the edges in question become the ones connecting two characters each from $\acute{\Sigma}$ and $\grave{\Sigma}$. Hence, in the end, $|E|$ becomes at least $(|w| - 1)/2$, where let E denote the set of edges between characters from $\acute{\Sigma}$ and $\grave{\Sigma}$ (recalling that there are exactly $|w| - 1$ edges). Since each edge in E corresponds to an occurrence of a pair from $\acute{\Sigma}\grave{\Sigma} \cup \grave{\Sigma}\acute{\Sigma}$ in w, at least one of the two partitions $(\acute{\Sigma}, \grave{\Sigma})$ and $(\grave{\Sigma}, \acute{\Sigma})$ covers more than half of E. Hence we achieve our final bound $|E|/2 = (|w| - 1)/4$ by choosing an appropriate partition at Line 7.

In order to see that Algorithm 1 runs in $O(m)$ time, we only have to care about Line 3 and Line 7. We can compute $\sum_{\tilde{c} \in \grave{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot)$ and $\sum_{\tilde{c} \in \acute{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot)$ by going through all $\mathsf{Freq}(c, \cdot, \cdot)$ for fixed c in the adjacency list, which are consecutive in the sorted list. Since each element of the list is used only once, the cost for Line 3 is $O(m)$ in total. Similarly the computation at Line 7 can be done by going through the adjacency list again. Thus the algorithm runs in $O(m)$ time. ◀

A.3 Proof of Lemma 8

Proof. We first compute the adjacency list of w. This can be easily done in $O(|w|)$ time and space by sorting the $|w| - 1$ size multiset $\{(w[i], w[i+1], 0) \mid 1 \leq i < |w|, w[i] > w[i+1]\} \cup \{(w[i+1], w[i], 1) \mid 1 \leq i < |w|, w[i] < w[i+1]\}$ by radix sort. Then by Lemma 7 we compute a partition $(\acute{\Sigma}, \grave{\Sigma})$ in linear time in the size of the adjacency list, which is $O(|w|)$. Next we scan w in $O(|w|)$ time and list all the occurrences of pairs to be compressed. Each pair $\acute{c}\grave{c} \in \acute{\Sigma}\grave{\Sigma}$ at position i is listed by a triple $(\acute{c}, \grave{c}, i)$ of integers in Σ. Then we sort the list according to the pair of integers $(\acute{c}, \grave{c})$, which can be done in $O(|w|)$ time and space by radix sort. Finally, we replace each pair with a fresh letter based on the rank of $(\acute{c}, \grave{c})$. ◀

A.4 Proof of Lemma 9

Proof. We first compute T_0 in $O(N)$ by sorting the characters used in T and replacing them with ranks of characters. Then we compress T_0 by applying BComp and PComp by turns and get $T_1, T_2 \ldots T_{\hat{h}}$. One technical problem is that characters used in an input string w of BComp and PComp should be in $[1..|w|]$, which is crucial to conduct radix sort efficiently in $O(|w|)$ time (see Lemmas 6 and 8). However letters in T_h do not necessarily hold this property. To overcome this problem, during computation we maintain ranks of letters among those used in the current T_h, which should be in $[1..|T_h|]$, and use the ranks instead of letters for radix sort. If we have such ranks in each level, we can easily maintain them by radix sort for the next level. Now, in every level h $(0 \leq h < \hat{h})$ the compression from T_h to T_{h+1} can be conducted in $O(|T_h|)$ time and space. Since PComp compresses a given string by a constant factor, the total running time can be bounded by $O(N)$ time. ◀

A.5 Proof of Lemma 11

Proof. Since a block is compressed into one letter, the number of parent nodes of C is at most m. As every internal node has two or more children, it is easy to see that there are $O(m)$ ancestor nodes of the parent nodes of C. ◀

Fast and Simple Jumbled Indexing for Binary Run-Length Encoded Strings*

Luís Cunha[1], Simone Dantas[2], Travis Gagie[3], Roland Wittler[4], Luis Kowada[5], and Jens Stoye[6]

1 **Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brazil; and Universidade Federal Fluminense, Niterói, Brazil**
`lfignacio@cos.ufrj.br`
2 **Universidade Federal Fluminense, Niterói, Brazil**
`sdantas@im.uff.br`
3 **CeBiB – Center for Biotechnology and Bioengineering, University of Chile, Santiago, Chile; and School of Computer Science and Telecommunications, Diego Portales University, Santiago, Chile**
`travis.gagie@gmail.com`
4 **Universität Bielefeld, Bielefeld, Germany**
`roland.wittler@uni-bielefeld.de`
5 **Universidade Federal Fluminense, Niterói, Brazil**
`luis@ic.uff.br`
6 **Universidade Federal Fluminense, Niterói, Brazil; and Universität Bielefeld, Bielefeld, Germany**
`jens.stoye@uni-bielefeld.de`

Abstract

Important papers have appeared recently on the problem of indexing binary strings for jumbled pattern matching, and further lowering the time bounds in terms of the input size would now be a breakthrough with broad implications. We can still make progress on the problem, however, by considering other natural parameters. Badkobeh et al. (IPL, 2013) and Amir et al. (TCS, 2016) gave algorithms that index a binary string in $O(n + \rho^2 \log \rho)$ time, where n is the length and ρ is the number of runs, and Giaquinta and Grabowski (IPL, 2013) gave one that runs in $O(n + \rho^2)$ time. In this paper we propose a new and very simple algorithm that also runs in $O(n + \rho^2)$ time and can be extended either so that the index returns the position of a match (if there is one), or so that the algorithm uses only $O(n)$ bits of space instead of $O(n)$ words.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics

Keywords and phrases string algorithms, indexing, jumbled pattern matching, run-length encoding

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.19

1 Introduction

Since its introduction at the 2009 Prague Stringology Conference [6, 8], the problem of indexed binary jumbled pattern matching has been discussed in many top conferences and

* This work was partially supported by CAPES, CNPq and FAPERJ.

journals. It asks us to preprocess a binary string such that later, given a number of 0s and a number of 1s, we can quickly report whether there exists a substring with those numbers of 0s and 1s and, optionally, return the position of one such substring or possibly even all of them. The naïve preprocessing algorithm takes quadratic time but researchers have reduced that bound to $O(n^2/\log n)$ [5, 16], $O(n^2/\log^2 n)$ [17], $O(n^2/2^{\Omega(\sqrt{\log n/\log\log n})})$ [4, 14] and finally $O(n^{1.859})$ with randomization or $O(n^{1.864})$ without [7].

Researchers have also looked at indexing for approximate matching [9, 10], indexed jumbled pattern matching over larger alphabets [2, 15], indexing labelled trees and other structures [9, 11, 12], and how to index faster when the (binary) input string is compressible. Gagie et al. [12] gave an algorithm that runs in $O(g^{2/3}n^{4/3})$ when the input is represented as a straight-line program with g rules, and Badkobeh et al. [3] gave one that runs in $O(n + \rho^2 \log \rho)$ time when the input consists of ρ runs, i.e., maximal unary substrings (we will denote later as ρ the number of maximal substrings of 1s, for convenience). Giaquinta and Grabowski [13] gave two algorithms: one runs in $O(\rho^2 \log k + n/k)$ time, where k is a parameter, and produces an index that uses $O(n/k)$ extra space and answers queries in $O(\log k)$ time; the other runs in $O(n^2 \log^2(w)/w)$ time, where w is the size of a machine word. Amir et al. [1] gave an algorithm that runs in $O(\rho^2 \log \rho)$ time when the input is a run-length encoded binary string, or $O(n + \rho^2 \log \rho)$ time when it is a plain binary string; it builds an index that takes $O(\rho^2)$ words and answers queries in $O(\log \rho)$ time, however. Very recently, Sugimoto et al. [19] considered the related problems of finding Abelian squares, Abelian periods and longest common Abelian factors, also on run-length encoded strings.

We first review some preliminary notions in Section 2. We present our main result in Section 3: a new and very simple indexing algorithm that runs in $O(n + \rho^2)$ time, which matches Giaquinta and Grabowski's algorithm with the parameter $k = 1$ and is thus tied as the fastest known when $\rho = \Omega(n^{0.5}) \cap o(n^{0.932})$ and the smallest straight-line program for the input has $\omega(\rho^3/n^2)$ rules. For an input string of up to ten million bits, for example, if the average run-length is three or more then $\rho < n^{0.932}$. While Giaquinta and Grabowski found an efficient way to construct the Corner Index of Badkobeh et al. [3], our algorithm constructs a more direct index and takes only 17 lines of pseudocode, making it a promising starting point for investigating other possible algorithmic features. In Section 4, for example, we show how to extend our algorithm to store information that lets us report the position of a match (if there is one). Finally, in Section 5, we show how we can alternatively adapt it to use only $O(n)$ bits of space.

2 Preliminaries

Consider a string $s \in \{0,1\}^n$. We denote by $s[i \cdots j]$ the substring of s consisting of the ith through jth characters, for $1 \leq i \leq j \leq n$; if $i = j$, we can also write simply $s[i]$. Cicalese et al. [6, 8] observed that, if we slide a window of length k over s, the number of 1s in the window can change by at most 1 at each step. It follows that if $s[i \cdots i+k-1]$ contains x copies of 1 and $s[j \cdots j+k-1]$ contains z copies of 1 with $i \leq j$ then, for y between x and z (notice x could be smaller than, larger than, or equal to z), there is a substring of length k in $s[i \cdots j+k-1]$ with exactly y copies of 1. This immediately implies the following theorem:

▶ **Theorem 1.** *Let x and z be the minimum and maximum numbers of 1s in any substring of length k. There is a substring of length k with y copies of 1 if and only if $x \leq y \leq z$.*

By Theorem 1, if we compute and store, for $1 \leq k \leq n$, the minimum and maximum numbers of 1s in a substring of s of length k then later, given a number of 0s and a number of 1s, we can report in constant time whether there exists a substring with that many 0s and 1s. For example, if $s = 010101110011$ then, as k goes from 1 to $n = 12$, the minimum and maximum numbers of 1s are $0, 0, 1, 2, 2, 3, 4, 4, 5, 5, 6, 7$ and $1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7$, respectively. Since the fifth numbers in these lists are 2 and 4, we know that there are substrings of length 5 with exactly 2, 3 and 4 copies of 1, but none with 0, 1 or 5 (or more than 5, obviously).

Cicalese et al. [9] noted that, if we also store the positions of the substrings with the minimum and maximum numbers of 1s and a bitvector for s that supports constant time rank queries, then via binary search in $O(\log n)$ time we can find an example of a substring with any desired numbers of 0s and 1s, called a *witness* if such a substring exists. (The query $\text{rank}(i)$ returns the number of 1s in $s[1 \cdots i]$; see, e.g., [18] for more details of rank queries on bitvectors.) For example, suppose we want to find a substring of length 5 with exactly 3 copies of 1 in our example string s. We have stored that there are substrings of length 5 with 2 and 4 copies of 1 starting at positions 1 and 4, respectively, so we know there is a substring of length 5 with exactly 3 copies of 1 starting in $s[1 \cdots 4]$. We choose $\lfloor (1 + 4)/2 \rfloor = 2$ and check how many 1s there are in $s[2 \cdots 2 + 5 - 1 = 6]$ via two rank queries. In this case, the answer is 3, so we have found a witness in one step; otherwise, we would know there is a witness starting in $s[3 \cdots 4]$ and we would recurse on that interval.

The same authors noted that in each step, the lists of minimum and maximum numbers can only stay the same or increment, so we can represent each list as a bitvector of length n and support access to it using rank queries. For example, the bitvector for the list of minimum numbers in our example is 001101101011, so $\text{rank}(i)$ returns the ith number in the list. Since an n-bit bitvector takes $O(n)$ bits of space, it follows that we can store our index in $O(n)$ bits and still support constant-time queries, if we do not want a witness. We note, however, that even though the input s takes n bits and the resulting index takes $O(n)$ bits, all previous constructions have used $\Omega(n)$ *words* of workspace in the worst case.

A *run* in s is a maximal unary substring and the run-length encoding $\text{rle}(s)$ is obtained by replacing each run by a copy of the character it contains and its length. Although ρ is usually used to denote the number of runs, for convenience, we use it to denote only the number of runs of 1s – about half its normal value for binary strings – and consider s to begin and end with (possibly empty) runs of 0s. For example, for our example string the run-length encoding is $0^1 1^1 0^1 1^1 0^1 1^3 0^2 1^2 0^0$ and $\rho = 4$ (instead of 9). We denote the lengths of the runs of 0s and 1s as $z[0], \ldots, z[\rho]$ and $o[1], \ldots, o[\rho]$, respectively.

3 Basic Indexing

Since finding substrings with the minimum numbers of 1s is symmetric to finding substrings with the maximum numbers of 1s (e.g., by taking the complement of the string), we describe how, given a binary run-length encoded string $s[1 \cdots n]$, we can build a table $T[1 \cdots n]$ such that $T[k] = f(k)$, where $f(k)$ denotes the maximum number of 1s in a substring of s of length k.

The complete pseudo-code of our algorithm – only 17 lines – is shown as Algorithm 1. The starting point of our explanation and proof of correctness is the observation that, if the bit immediately to the left of a substring is a 1, we can shift the substring one bit left without decreasing the number of 1s; if the first bit of the substring is a 0, then we can shift the substring one bit right (shortening it on the right if necessary) without decreasing the number of 1s. It follows that, for $1 \leq k \leq n$, there is a substring of length at most k containing $f(k)$ copies of 1 and starting at the beginning of a run of 1s. Since we can remove

Algorithm 1: Building the index table T of string s.

```
for $i = 1, \ldots, n$ do
    $T[i] = 0$
for $i = 1, \ldots, \rho$ do
    $ones = o[i]$
    $zeros = 0$
    $T[ones] = ones$
    for $j = i + 1, \ldots, \rho$ do
        $ones \mathrel{+}= o[j]$
        $zeros \mathrel{+}= z[j - 1]$
        if $ones > T[ones + zeros]$ then
            $T[ones + zeros] = ones$
for $i = n - 1, \ldots, 1$ do
    if $T[i] < T[i + 1] - 1$ then
        $T[i] = T[i + 1] - 1$
for $i = 2, \ldots, n$ do
    if $T[i] < T[i - 1]$ then
        $T[i] = T[i - 1]$
```

any trailing 0s from such a substring also without changing the number of 1s, there is such a substring that also ends in a run of 1s. Therefore we have the following lemma:

▶ **Lemma 2.** *For $1 \leq k \leq n$, there is a substring of length at most k containing $f(k)$ copies of 1, starting at the beginning of a run of 1s and ending in a run of 1s.*

Applying Lemma 2 immediately yields an $O(n\rho)$-time algorithm: set $T[1 \cdots n]$ to all 0s; for each position i at the beginning of a run of 1s and each position $j \geq i$ in a run of 1s, set $T[j - i + 1] = \max(T[j - i + 1], s[i] + \cdots + s[j])$; finally, because f is non-decreasing, make a pass over T from $T[2]$ to $T[n]$ setting each $T[i] = \max(T[i], T[i - 1])$. Computing the number $s[i] + \cdots + s[j]$ of 1s in a substring $s[i \ldots j]$ starting at the beginning of a run of 1s and ending in a run of 1s is easy to do from the run-length encoding in amortized constant time.

To speed this preliminary algorithm up to run in $O(n + \rho^2)$ time, we first observe that, if ℓ is the length of a substring starting at the beginning of a run of 1s, ending in a run of 1s and containing $f(\ell)$ copies of 1, and $d > \ell$ is the length of a substring starting at the beginning of a run of 1s and ending at the end of a run of 1s, then $f(\ell) \geq f(d) - d + \ell$. (In fact this is true for any ℓ and $d \geq \ell$, simply because $f(x + 1) \leq f(x) + 1$ for all x.) We then observe that, for some such d, we have $f(\ell) = f(d) - d + \ell$. To see why, consider any substring $s[i \cdots j]$ of length ℓ starting at the beginning of a run of 1s, ending within a run of 1s and containing $f(\ell)$ copies of 1: let d be the length of the substring starting at $s[i]$ and ending at the end of the run of 1s containing $s[i + \ell - 1]$, so $f(\ell) = f(d) - d + \ell$.

▶ **Lemma 3.** *If ℓ is the length of a substring starting at the beginning of a run of 1s, ending in a run of 1s and containing $f(\ell)$ copies of 1, and $d > \ell$ is the length of a substring starting at the beginning of a run of 1s and ending at the end of a run of 1s, then $f(\ell) \geq f(d) - d + \ell$. Furthermore, for some such d, we have $f(\ell) = f(d) - d + \ell$.*

With Lemma 3, we can compute the number $s[i] + \cdots + s[j]$ of 1s in each substring $s[i \ldots j]$ starting at the beginning of a run of 1s and ending in a run of 1s, in a total of $O(n + \rho^2)$ time:

again, set $T[1 \cdots n]$ to all 0s; for each position i at the beginning of a run of 1s and each position $j \geq i$ at the end of a run of 1s, set $T[j-i+1] = \max(T[j-i+1], s[i]+\cdots+s[j])$; make a pass over T from $T[n-1]$ to $T[1]$ setting each $T[i] = \max(T[i], T[i+1]-1)$. Computing the number $s[i] + \cdots + s[j]$ of 1s in a substring $s[i \cdots j]$ starting at the beginning of a run of 1s and ending at the end of a run of 1s is again easy to do from the run-length encoding in amortized constant time.

Combining Lemmas 2 and 3, we have a complete algorithm for computing T in $O(n+\rho^2)$ time: set $T[1 \cdots n]$ to all 0s; for each position i at the beginning of a run of 1s and each position $j \geq i$ at the end of a run of 1s, set $T[j-i+1] = \max(T[j-i+1], s[i]+\cdots+s[j])$; make a pass over T from $T[n-1]$ to $T[1]$ setting each $T[i] = \max(T[i], T[i+1]-1)$ (which sets $T[\ell]$ correctly for every length ℓ of a substring starting at the beginning of a run of 1s, ending in a run of 1s and containing $f(\ell)$ copies of 1); and make a pass over T from $T[2]$ to $T[n]$ setting each $T[i] = \max(T[i], T[i-1])$ (which sets every entry in T correctly). Once we have T, we can convert it into a bitvector in $O(n)$ time. Summarizing our results so far, we have the following theorem, which we adapt in later sections:

▶ **Theorem 4.** *Given a binary string s of length n containing ρ runs of 1s, we can build an $O(n)$-bit index for constant-time jumbled pattern matching in $O(n+\rho^2)$ time.*

Now we examine how our algorithm works on our example $s = 010101110011$. First we set all entries of T to 0, then we loop through the runs of 1s and, for each, loop through the runs of 1s not earlier, computing distance from the start of the first to the end of the second and the number of 1s between those positions. While doing this, we set $T[1] = 1$, the number of 1s from the start to the end of the first run of 1s; $T[3] = 2$, the number of 1s from the start of the first run of 1s to the end of the second run of 1s; $T[7] = 5$, the number of 1s from the start of the first run of 1s to the end of the third run of 1s; $T[11] = 7$, the number of 1s from the start of the first run of 1s to the end of the fourth run of 1s; $T[5] = 4$, the number of 1s from the start of the second run of 1s to the end of the third run; etc. When we have finished this stage, $T = [1,2,3,0,4,0,5,0,6,0,7,0]$. We then make a pass over T from right to left, setting each $T[i] = \max(T[i], T[i+1]-1)$. After this stage, $T = [1,2,3,3,4,4,5,5,6,6,7,0]$. Finally, we make a pass over T from left to right, setting each $T[i] = \max(T[i], T[i-1])$. This fills in $T[12]$ and leaves T correctly computed as $T = [1,2,3,3,4,4,5,5,6,6,7,7]$.

4 Witnessing Index

As described in Section 2, if together with computing the minimum and maximum number of 1s in a substring of length k for $1 \leq k \leq n$, we also store the positions of substrings of length k with those numbers of 1s, and a single bitvector for s, then, together with confirming that s contains a substring with a given number of 0s and 1s (if it does), we can give the starting position of such a substring, still in constant time.

In this section, we show how to modify our algorithm from Section 3 to build also a table $P[1..n]$ such that $P[k]$ is the starting position of a substring of length k containing $f(k)$ copies of 1s. Computing and storing the starting position of a substring of length k with the minimum number of 1s is symmetric.

First, notice that during the first stage of Algorithm 1, whenever we set $T[k] = f(k)$, we have found a substring of length k containing $f(k)$ copies of 1, so we can set $P[k]$ at the same time. Now consider the second stage of the algorithm, in which we make a right-to-left pass over T setting $T[i] = \max(T[i], T[i+1]-1)$ for $1 \leq i \leq n-1$. When we start this stage, for every positive entry in T we have set the corresponding entry in P. Therefore, by induction, whenever we set $T[i] = T[i+1] - 1$, we have $P[i+1]$ set to the starting position of a substring of length $i+1$ containing $T[i+1]$ copies of 1. The substring of length i starting at $P[i+1]$ contains at least $T[i+1] - 1$ copies of 1, so we can set $P[i] = P[i+1]$. In the last stage of the algorithm, in which we make a left-to-right pass over T, we can almost use the same kind of argument and simply copy P values when we copy T values, except that we must ensure the starting positions we copy are far enough to the left of the end of the string (i.e., that the substrings have the correct lengths). Our modified algorithm is shown as Algorithm 2 – still only 25 lines – and we now have the following theorem:

▶ **Theorem 5.** *Given a binary string s of length n containing ρ runs of 1s, we can build an $O(n)$-word index for constant-time jumbled pattern matching with $O(\log n)$ time witnessing in $O(n + \rho^2)$ time.*

Running our modified algorithm on our example $s = 010101110011$, in the first stage we set $T = [1, 2, 3, 0, 4, 0, 5, 0, 6, 0, 7, 0]$ and, simultaneously, $P = [2, 11, 6, 0, 4, 0, 2, 0, 4, 0, 2, 0]$, where 0 indicates an unset value in P. In the second stage, we set $T = [1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 0]$ and $P = [2, 11, 6, 4, 4, 2, 2, 4, 4, 2, 2, 0]$. Finally, in the third stage, we fill in $T[12] = T[11]$, but we cannot just set $P[12] = P[11] = 2$ because $s[2 \cdots n = 12]$ has length only 11, so we set $P[12] = 1$.

5 Reducing Workspace

It is frustrating that both s and the index described in Theorem 4 take $O(n)$ bits, but we use $O(n)$ words to build the index. In this section, we show how to reduce this workspace to $O(n)$ bits also, without increasing the time bound for construction by more than a constant factor.

Suppose we divide T into blocks of size $\lg(n)/2$ and modify our algorithm such that, whenever we set a value $T[i]$, we ensure that each value $T[j]$ in the same block with $j < i$ is at least $T[i] - i + j$ and each value $T[j]$ in the same block with $i < j$ is at least $T[i]$. Since we would eventually set each such $T[j]$ to a value at least as great during the normal execution of the algorithm, this does not change its correctness, apart from perhaps slowing it down by an $O(\log n)$ factor.

For any two consecutive values $T[i]$ and $T[i+1]$ in the same block now, however, we have $T[i] \leq T[i+1] \leq T[i] + 1$. We can thus store each block by storing its first value and a binary string of length $\lg(n)/2$ whose bits indicate where the values in the block increase. Therefore, we need a total of only $O(n)$ bits to store all the blocks.

Notice that, if we increase a value $T[i]$ by more than $\lg(n)/2$, we reset the first value $T[h]$ of the block to be $T[i] - i + h$, set the leading bits of the block to 1s to indicate that the values increase until reaching $T[i]$, and set the later bits of the block to 0s to indicate that the values remain equal to $T[i]$ until the end of the block. Therefore, we can speed the algorithm up to run in $O(n + \rho^2)$ time again, by using a universal table of size $2^{\lg(n)/2} \log^{O(1)} n = o(n^{1/2+\epsilon})$ to decide how to update blocks when we set values in them.

▶ **Theorem 6.** *Given a binary string s of length n containing ρ runs of 1s, we can build an $O(n)$-bit index for constant-time jumbled pattern matching in $O(n + \rho^2)$ time using $O(n)$ bits of workspace.*

In fact, it seems possible to make the algorithm run in $O(n + \rho^2)$ time and $O(n)$ bits of space even without a universal table, using AC0 operations on words that are available on standard architectures.

Algorithm 2: Building the tables T and P for s.

```
for $i = 1, \ldots, n$ do
    $T[i] = 0$
$p = z[0]$
for $i = 1, \ldots, \rho$ do
    $ones = o[i]$
    $zeros = 0$
    $T[ones] = ones$
    $P[ones] = p$
    for $j = i + 1, \ldots, \rho$ do
        $ones \mathrel{+}= o[j]$
        $zeros \mathrel{+}= z[j - 1]$
        if $ones > T[ones + zeros]$ then
            $T[ones + zeros] = ones$
            $P[ones + zeros] = p$
    $p \mathrel{+}= ones[i] + zeros[i]$
for $i = n - 1, \ldots, 1$ do
    if $T[i] < T[i + 1] - 1$ then
        $T[i] = T[i + 1] - 1$
        $P[i] = P[i + 1]$
for $i = 2, \ldots, n$ do
    if $T[i] < T[i - 1]$ then
        $T[i] = T[i - 1]$
        $P[i] = P[i - 1]$
        if $P[i] + i > n$ then
            $P[i] = n - i$
```

This workspace reduction makes little sense for a string as small as our example $s =$ 010101110011 but, for the sake of argument, suppose we partition our array T for it into three blocks of length 4 each. We keep $T[1]$, $T[5]$ and $T[9]$ stored explicitly and represent the other entries of T implicitly with three 3-bit binary strings B_1, B_2 and B_3. Initially we set $T[1] = T[5] = T[9] = 0$ and $B_1 = B_2 = B_3 = 000$. Recall from Section 3 that we first set $T[1] = 1$, the number of 1s from the start to the end of the first run of 1s. At this point, we do not need to change B_1. We then set $T[3] = 2$ – the number of 1s from the start of the first run of 1s to the end of the second run of 1s – by setting $B_1 = 010$: starting from $T[1] = 1$, this encodes $T[2] = T[1] + 0 = 1$, $T[3] = T[1] + 0 + 1 = 2$ and $T[4] = T[1] + 0 + 1 + 0 = 2$. Next we set $T[7] = 5$ – the number of 1s from the start of the first run of 1s to the end of the third run of 1s – by setting $T[5] = 3$ and $B_2 = 110$: starting from $T[5] = 3$, this encodes $T[6] = T[5] + 1 = 4$, $T[7] = T[5] + 1 + 1 = 5$ and $T[8] = T[5] + 1 + 1 + 0 = 5$. Continuing like this, we set $T[11] = 7$ by setting $T[9] = 5$ and $B_3 = 110$; set $T[5] = 4$ and $B_2 = 010$; etc. When we are finished this stage, $T[1] = 1$, $T[5] = 4$ and $T[9] = 6$, and $B_1 = 110$, $B_2 = 010$ and $B_3 = 010$, encoding $T = [1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7]$. In this case the final right-to-left and left-to-right passes have no effect, but there are cases (e.g., when we do not set any values in a certain block) when they are still necessary.

References

1 Amihood Amir, Alberto Apostolico, Tirza Hirst, Gad M. Landau, Noa Lewenstein, and Liat Rozenberg. Algorithms for jumbled indexing, jumbled border and jumbled square on run-length encoded strings. *Theor. Comput. Sci.*, 656:146–159, 2016. doi:10.1016/j.tcs.2016.04.030.

2 Amihood Amir, Timothy M. Chan, Moshe Lewenstein, and Noa Lewenstein. On hardness of jumbled indexing. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP 2014)*, volume 8572 of *LNCS*, pages 114–125. Springer, 2014. doi:10.1007/978-3-662-43948-7_10.

3 Golnaz Badkobeh, Gabriele Fici, Steve Kroon, and Zsuzsanna Lipták. Binary jumbled string matching for highly run-length compressible texts. *Inf. Process. Lett.*, 113(17):604–608, 2013. doi:10.1016/j.ipl.2013.05.007.

4 David Bremner, Timothy M. Chan, Erik D. Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, Mihai Pătraşcu, and Perouz Taslakian. Necklaces, convolutions, and $X + Y$. *Algorithmica*, 69(2):294–314, 2014. doi:10.1007/s00453-012-9734-3.

5 Péter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. On table arrangements, scrabble freaks, and jumbled pattern matching. In Paolo Boldi and Luisa Gargano, editors, *Proceedings of the 5th International Conference on Fun with Algorithms (FUN 2010)*, volume 6099 of *LNCS*, pages 89–101. Springer, 2010. doi:10.1007/978-3-642-13122-6_11.

6 Péter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. On approximate jumbled pattern matching in strings. *Theory Comput. Syst.*, 50(1):35–51, 2012. doi:10.1007/s00224-011-9344-5.

7 Timothy M. Chan and Moshe Lewenstein. Clustered integer 3SUM via additive combinatorics. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the 47th Annual ACM on Symposium on Theory of Computing (STOC 2015)*, pages 31–40. ACM, ACM, 2015. doi:10.1145/2746539.2746568.

8 Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Searching for jumbled patterns in strings. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference (PSC 2009)*, pages 105–117, 2009. URL: http://www.stringology.org/event/2009/p10.html.

9 Ferdinando Cicalese, Travis Gagie, Emanuele Giaquinta, Eduardo Sany Laber, Zsuzsanna Lipták, Romeo Rizzi, and Alexandru I. Tomescu. Indexes for jumbled pattern matching in strings, trees and graphs. In Oren Kurland, Moshe Lewenstein, and Ely Porat, editors, *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE 2013)*, volume 8214 of *LNCS*, pages 56–63. Springer, 2013. doi:10.1007/978-3-319-02432-5_10.

10 Ferdinando Cicalese, Eduardo Laber, Oren Weimann, and Raphael Yuster. Near linear time construction of an approximate index for all maximum consecutive sub-sums of a sequence. In Juha Kärkkäinen and Jens Stoye, editors, *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM 2012)*, volume 7354 of *LNCS*, pages 149–158. Springer, 2012. doi:10.1007/978-3-642-31265-6_12.

11 Stephane Durocher, Robert Fraser, Travis Gagie, Debajyoti Mondal, Matthew Skala, and Sharma V. Thankachan. Indexed geometric jumbled pattern matching. In Alexander S. Kulikov, Sergei O. Kuznetsov, and Pavel A. Pevzner, editors, *Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM 2014)*, volume 8486 of *LNCS*, pages 110–119. Springer, Springer, 2014. doi:10.1007/978-3-319-07566-2_12.

12 Travis Gagie, Danny Hermelin, Gad M. Landau, and Oren Weimann. Binary jumbled pattern matching on trees and tree-like structures. *Algorithmica*, 73(3):571–588, 2015. doi:10.1007/s00453-014-9957-6.

13 Emanuele Giaquinta and Szymon Grabowski. New algorithms for binary jumbled pattern matching. *Inf. Process. Lett.*, 113(14):538–542, 2013. doi:10.1016/j.ipl.2013.04.013.

14 Danny Hermelin, Gad M. Landau, Yuri Rabinovich, and Oren Weimann. Binary jumbled pattern matching via all-pairs shortest paths, 2014. arXiv:1401.2065.

15 Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. In Hans L. Bodlaender and Giuseppe F. Italiano, editors, *Proceedings of the 21st Annual European Symposium on Algorithms (ESA 2013)*, volume 8125 of *LNCS*, pages 625–636. Springer, 2013. doi:10.1007/978-3-642-40450-4_53.

16 Tanaeem M. Moosa and M. Sohel Rahman. Indexing permutations for binary strings. *Inf. Process. Lett.*, 110(18-19):795–798, 2010. doi:10.1016/j.ipl.2010.06.012.

17 Tanaeem M. Moosa and M. Sohel Rahman. Sub-quadratic time and linear space data structures for permutation matching in binary strings. *J. Discrete Algorithms*, 10:5–9, 2012. doi:10.1016/j.jda.2011.08.003.

18 Gonzalo Navarro. *Compact Data Structures: A practical approach*. Cambridge University Press, 2016. doi:10.1017/CBO9781316588284.

19 Shiho Sugimoto, Naoki Noda, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing Abelian regularities on RLE strings, 2017. arXiv:1701.02836.

Faster STR-IC-LCS Computation via RLE[*]

Keita Kuboi[1], Yuta Fujishige[2], Shunsuke Inenaga[3], Hideo Bannai[4], and Masayuki Takeda[5]

1 **Department of Informatics, Kyushu University, Fukuoka, Japan**
`keita.kuboi@inf.kyushu-u.ac.jp`
2 **Department of Informatics, Kyushu University, Fukuoka, Japan**
`yuta.fujishige@inf.kyushu-u.ac.jp`
3 **Department of Informatics, Kyushu University, Fukuoka, Japan**
`inenaga@inf.kyushu-u.ac.jp`
4 **Department of Informatics, Kyushu University, Fukuoka, Japan**
`bannai@inf.kyushu-u.ac.jp`
5 **Department of Informatics, Kyushu University, Fukuoka, Japan**
`takeda@inf.kyushu-u.ac.jp`

Abstract

The constrained LCS problem asks one to find a longest common subsequence of two input strings A and B with some constraints. The STR-IC-LCS problem is a variant of the constrained LCS problem, where the solution must include a given constraint string C as a substring. Given two strings A and B of respective lengths M and N, and a constraint string C of length at most $\min\{M, N\}$, the best known algorithm for the STR-IC-LCS problem, proposed by Deorowicz (*Inf. Process. Lett.*, 11:423–426, 2012), runs in $O(MN)$ time. In this work, we present an $O(mN+nM)$-time solution to the STR-IC-LCS problem, where m and n denote the sizes of the run-length encodings of A and B, respectively. Since $m \leq M$ and $n \leq N$ always hold, our algorithm is always as fast as Deorowicz's algorithm, and is faster when input strings are compressible via RLE.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases longest common subsequence, STR-IC-LCS, run-length encoding

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.20

1 Introduction

Longest common subsequence (LCS) is one of the most basic measures of similarity between strings, and there is a vast amount of literature concerning its efficient computation. An LCS of two strings A and B of lengths M and N, respectively, is a longest string that is a subsequence of both A and B. There is a well known $O(MN)$ time and space dynamic programming (DP) algorithm [15] to compute an LCS between two strings. LCS has applications in bioinformatics [10, 16], file comparisons [9, 8], pattern recognition [13], etc.

Recently, several variants of the problem which try to find a longest common subsequence that satisfy some constraints have been considered. In 2003, Tsai [14] proposed the constrained LCS (CLCS) problem, where, given strings A, B with respective lengths M, N, and a constraint string C of length K, the problem is to find a longest string that contains C as a subsequence and is also a common subsequence of A and B. Tsai gave an $O(M^2N^2K)$ time

[*] This work was in part supported by JSPS KAKENHI Grant Numbers JP25240003, JP26280003, JP16H02783, JP17H01697 and by JST CREST Grant Number JPMJCR1402.

Table 1 Time complexities of best known solutions to various constrained LCS problems.

Problem	DP solution	DP solution using RLE
SEQ-IC-LCS	$O(MNK)$ [6]	$O(M + N + K\min\{mN, nM\})$ [12]
SEQ-EC-LCS	$O(MNK)$ [5]	–
STR-IC-LCS	$O(MN)$ [7]	$O(mN + nM)$ [this work]
STR-EC-LCS	$O(MNK)$ [17]	–

solution, which was improved in 2004 by Chin et al. to $O(MNK)$ time [6]. Variants of the constrained LCS problem called SEQ-IC-LCS, SEQ-EC-LCS, STR-IC-LCS, and STR-EC-LCS, were considered by Chen and Chao in 2011 [5]. Each problem considers as input, three strings A, B and C, and the problem is to find a longest string that includes (IC) or excludes (EC) C as a subsequence (SEQ) or substring (STR) and is a common subsequence of A and B (i.e., CLCS is equivalent to the SEQ-IC-LCS problem). The best solution for each of the problems is shown in Table 1.

In order to speed up the LCS computation, one direction of research that has received much attention is to apply compression, namely, run-length encoding (RLE) of strings. Bunke and Csirik [4] were one of the first to consider such a scenario, and proposed an $O(mN + nM)$ time algorithm. Here, m, n are the sizes of the RLE of the input strings of lengths M and N, respectively. Notice that since RLE can be computed in linear time, and $m \leq M$ and $n \leq N$, the algorithm is always asymptotically faster than the standard $O(NM)$ time dynamic programming algorithm, especially when the strings are compressible by RLE. Furthermore, Ahsan et al. proposed an algorithm which runs in $O((m+n)+R\log\log(mn)+R\log\log(M+N))$ time [1], where R is the total number of pairs of runs of the same character in the two RLE strings, i.e. $R \in O(mn)$, and the algorithm can be much faster when the strings are compressible by RLE.

For the constrained LCS problems, RLE based solutions for only the SEQ-IC-LCS problem have been proposed. In 2012, an $O(K(mN + nM))$ time algorithm was proposed by Ann et al. [2]. Later, in 2015, Liu et al. proposed a faster $O(M + N + K\min\{mN, nM\})$ time algorithm [12].

In this paper, we present the first RLE based solution for the STR-IC-LCS problem that runs in $O(mN + nM)$ time. Again, since RLE can be computed in linear time, and $m \leq M$ and $n \leq N$, the proposed algorithm is always asymptotically faster than the best known solution for the STR-IC-LCS problem by Deorowicz [7], which runs in $O(MN)$ time.

A common criticism against RLE based solutions is a claim that, although they are theoretically interesting, since most strings “in the real world” are not compressible by RLE, their applicability is limited and they are only useful in extreme artificial cases. We believe that this is not entirely true. There can be cases where RLE is a natural encoding of the data, for example, in music, a melody can be expressed as a string of pitches and their duration. Furthermore, in the data mining community, there exist popular preprocessing schemes for analyzing various types of time series data, which convert the time series to strings over a fairly small alphabet as an approximation of the original data, after which various analyses are conducted (e.g. SAX (Symbolic Aggregate approXimation) [11], *clipped* bit representation [3], etc.). These conversions are likely to produce strings which are compressible by RLE (and in fact, shown to be effective in [3]), indicating that RLE based solutions may have a wider range of application than commonly perceived.

2 Preliminaries

Let Σ be the finite set of characters, and Σ^* be the set of strings. For any string A, let $|A|$ be the length of A. For any $1 \leq i \leq i' \leq |A|$, let $A[i]$ be the ith character of A and let $A[i..i'] = A[i] \cdots A[i']$ denote a substring of A. Especially, $A[1..i']$ denotes a *prefix* of A, and $A[i..|A|]$ denotes a *suffix* of A. A string Z is a *subsequence* of A if Z can be obtained from A by removing zero or more characters. For two string A and B, a string Z is a *longest common subsequence* (LCS) of A, B, if Z is a longest string that is a subsequence of both A and B. For any $1 \leq i \leq |A|$ and $1 \leq j \leq |B|$, let $L^{pref}(i,j)$ denote the length of an LCS of $A[1..i]$, $B[1..j]$, and let $L^{suf}(i,j)$ denote the length of an LCS of $A[i..|A|]$, $B[j..|B|]$. The LCS problem is to compute the length of an LCS of given two strings A and B. A well known solution is dynamic programming, which computes in $O(MN)$ time, a table (which we will call *DP table*) of size $O(MN)$ that stores values of $L^{pref}(i,j)$ for all $1 \leq i \leq M$, $1 \leq j \leq N$. The DP table for $L^{suf}(i,j)$ can be computed similarly.

For two strings A, B and a constraint string C, a string Z is an STR-IC-LCS of A, B, C, if Z is a longest string that includes C as a substring and also is a subsequence of both A and B. The STR-IC-LCS problem is to compute the length of an STR-IC-LCS of any given three strings A, B and C. For example, if $A =$ `abacab`, $B =$ `babcaba`, $C =$ `bb`, then `abcab` and `bacab` are LCSs of A, B, and `abb` is an STR-IC-LCS of A, B, C.

The run-length encoding (RLE) of a string A is a kind of compressed representation of A where each maximal run of the same character is represented by a pair of the character and the length of the run. Let $RLE(A)$ denote the RLE of a string A. The *size* of $RLE(A)$ is the number of the runs in A, and is denoted by $|RLE(A)|$. By definition, $|RLE(A)|$ is always less than or equal to $|A|$.

In the next section, we consider the STR-IC-LCS problem of strings A, B and constraint string C. Let $|A| = M$, $|B| = N$, $|C| = K$, $|RLE(A)| = m$ and $|RLE(B)| = n$. We assume that $K \leq \min(M, N)$ and $|RLE(C)| \leq \min(m, n)$, since otherwise there is no solution. We also assume that $K > 0$, because in that case the problem becomes the normal LCS problem of A, B.

3 Algorithm

In this section, we will first introduce a slightly modified version of Deorowicz's $O(MN)$-time algorithm for the STR-IC-LCS problem [7], and then propose our $O(mN + nM)$-time algorithm, which is based on his dynamic programming approach, but uses RLE.

3.1 Deorowicz's $O(MN)$ Algorithm

We first define the notion of minimal C-intervals of a string.

▶ **Definition 1.** For any strings A and C, an interval $[s, f]$ is a *minimal C-interval of A* if

- C is a subsequence of $A[s..f]$, and
- C is not a subsequence of $A[s+1..f]$ or $A[s..f-1]$.

Deorowicz's algorithm is based on Lemma 2, which is used implicitly in [7].

▶ **Lemma 2** (implicit in [7]). *If Z is an STR-IC-LCS of A, B, C, then there exist minimal C-intervals $[s, f], [s', f']$ $(1 \leq s \leq f \leq M,\ 1 \leq s' \leq f' \leq N)$ respectively of A and B, such that $Z = XCY$, where X is an LCS of $A[1..s-1]$ and $B[1..s'-1]$ and Y is an LCS of $A[f+1..M]$ and $B[f'+1..N]$.*

Proof. From the definition of STR-IC-LCS, C is a substring of Z, and therefore, there exist (possibly empty) strings X, Y such that $Z = XCY$. Also, since Z is a common subsequence of A and B, there exist monotonically increasing sequences $i_1, \ldots, i_{|Z|}$ and $j_1, \ldots, j_{|Z|}$ such that $Z = A[i_1] \cdots A[i_{|Z|}] = B[j_1] \cdots B[j_{|Z|}]$, and $C = A[i_{|X|+1}] \cdots A[i_{|X|+K}] = B[j_{|X|+1}] \cdots B[j_{|X|+K}]$.

Now, since C is a subsequence of $A[i_{|X|+1}..i_{|X|+K}]$ and $B[j_{|X|+1}..j_{|X|+K}]$ there exist minimal C-intervals $[s, f]$, $[s', f']$ respectively of A and B that satisfy $i_{|X|+1} \leq s \leq f \leq i_{|X|+K}$ and $j_{|X|+1} \leq s' \leq f' \leq j_{|X|+K}$. Let X' be an LCS of $A[1..s-1]$ and $B[1..s'-1]$, and Y' an LCS of $A[f+1..M]$ and $B[f'+1..N]$. Since X must be a common subsequence of $A[1..s-1]$ and $B[1..s'-1]$, and Y a common subsequence of $A[f+1..M]$ and $B[f'+1..N]$, we have $|X'| \geq |X|$ and $|Y'| \geq |Y|$. However, we cannot have that $|X'| > |X|$ or $|Y'| > |Y|$ since otherwise, $X'CY'$ would be a string longer than Z that contains C as a substring, and is a common subsequence of A, B, contradicting that Z is an STR-IC-LCS of A, B, C. Thus, $|X| = |X'|$ and $|Y| = |Y'|$ implying that X is also an LCS of $A[1..s-1], B[1..s'-1]$, and Y is also an LCS of $A[f+1..M], B[f'+1..N]$, proving the lemma. ◀

The algorithm consists of the following two steps, whose correctness follows from Lemma 2.

Step 1. Compute all minimal C-intervals of A and B.

Step 2. For all pairs of a minimal C-interval $[s, f]$ of A and a minimal C-interval $[s', f']$ of B, compute the length of an LCS of the corresponding prefixes of A and B (i.e., $L^{pref}(s-1, s'-1)$) and that of the corresponding suffixes of A and B (i.e., $L^{suf}(f+1, f'+1)$). The largest sum of LCS lengths plus $|C|$ (i.e., $L^{pref}(s-1, s'-1) + L^{suf}(f+1, f'+1) + |C|$) is the length of an STR-IC-LCS.

The steps can be executed in the following running times. For Step 1, there are respectively at most M and N minimal C-intervals of A and B, which can be enumerated in $O(MK)$ and $O(NK)$ time. For Step 2, we precompute, in $O(MN)$ time, two dynamic programming tables which respectively contain the values of $L^{pref}(i, j)$ and $L^{suf}(i, j)$ for each $1 \leq i \leq M$ and $1 \leq j \leq N$. Using these tables, the value $L^{pref}(s-1, s'-1) + L^{suf}(f+1, f'+1) + |C|$ can be computed in constant time for any $[s, f]$ and $[s', f']$. There are $O(MN)$ possible pairs of minimal C-intervals, so Step 2 can be done in $O(MN)$ time. In total, since $K \leq M, K \leq N$, the STR-IC-LCS problem can be solved in $O(MN)$ time.

We note that in the original presentation of Deorowicz's algorithm, *right*-minimal C-intervals, that is, intervals $[s, f]$ where C is a subsequence of $A[s..f]$ but not of $A[s..f-1]$ are computed, instead of minimal C-intervals as defined in Definition 1. Although the number of considered intervals changes, this does not influence the asymptotic complexities in the non-RLE case. However, as we will see in Lemma 4 of Section 3.2, this is an essential difference for the RLE case, since, when $|RLE(C)| > 1$, the number of minimal C-intervals of A and B can be bounded by $O(m)$ and $O(n)$, but the number of right-minimal C-intervals of A and B cannot, and are only bounded by $O(M)$ and $O(N)$.

3.2 Our Algorithm via RLE

In this subsection, we propose an efficient algorithm based on Deorowicz's algorithm explained in Subsection 3.1, extended to strings expressed in RLE. There are two main cases to consider: when $|RLE(C)| = 1$, i.e., when C consists of only one type of character, and when $|RLE(C)| > 1$, i.e., when C contains at least two different characters.

3.2.1 Case $|RLE(C)| > 1$

▶ **Theorem 3.** *Let A, B, C be any strings and let $|A| = M$, $|B| = N$, $|RLE(A)| = m$ and $|RLE(B)| = n$. If $|RLE(C)| > 1$, we can compute the length of an STR-IC-LCS of A, B, C in $O(mN + nM)$ time.*

For Step 1, we execute the following procedure to enumerate all minimal C-intervals of A and B. Let $s_0 = 0$. First, find the right minimal C-interval starting at $s_0 + 1$, i.e., the smallest position f_1 such that C is a subsequence of $A[s_0 + 1..f_1]$. Next, starting from position f_1 of A, search backwards to find the left minimal C-interval ending at f_1, i.e., the largest position s_1 such that C is a subsequence of $A[s_1..f_1]$. The process is then repeated, i.e., find the smallest position f_2 such that C is a subsequence of $A[s_1 + 1..f_2]$, and then search backwards to find the largest position s_2 such that C is a subsequence of $A[s_2..f_2]$, and so on. It is easy to see that the intervals $[s_1, f_1], [s_2, f_2], \ldots$ obtained by repeating this procedure until reaching the end of A are all the minimal C-intervals of A, since each interval that is found is distinct, and there cannot exist another minimal C-interval between those found by the procedure. The same is done for B. For non-RLE strings, this takes $O((M + N)K)$ time. The lemma below shows that the procedure can be implemented more efficiently using RLE.

▶ **Lemma 4.** *Let A and C be strings where $|A| = M$, $|RLE(A)| = m$ and $|C| = K$. If $|RLE(C)| > 1$, the number of minimal C-intervals of A is $O(m)$ and can be enumerated in $O(M + mK)$ time.*

Proof. Because $|RLE(C)| > 1$, it is easy to see from the backward search in the procedure described above, that for any minimal C-interval of A, there is a unique run of A such that the last character of the first run of C corresponds to the last character of that run. Therefore, the number of minimal C-intervals of A is $O(m)$.

We can compute $RLE(A) = a_1^{M_1} \cdots a_m^{M_m}$ and $RLE(C) = c_1^{K_1} \cdots c_k^{K_k}$ in $O(M + K)$ time. What remains is to show that the forward/backward search procedure described above to compute all minimal C-intervals of A can be implemented in $O(mK)$ time. The pseudo-code of the algorithm described is shown in Algorithm 1.

In the forward search, we scan $RLE(A)$ to find a right minimal C-interval by greedily matching the runs of $RLE(C)$ to $RLE(A)$. We maintain the character c_q and exponent $rest$ of the first run c_q^{rest} of $RLE(C')$, where C' is the suffix of C that is not yet matched. When comparing a run $a_p^{M_p}$ of $RLE(A)$ and c_q^{rest}, if the characters are different (i.e., $a_p \neq c_q$), we know that the entire run $a_p^{M_p}$ will not match and thus we can consider the next run of A. Suppose the characters are the same. Then, if $M_p < rest$, the entire run $a_p^{M_p}$ of A is matched, and we can consider the next run $a_{p+1}^{M_{p+1}}$ of A. Also, $rest$ can be updated accordingly in constant time by simple arithmetic. Furthermore, since $c_q = a_p \neq a_{p+1}$, we can in fact skip to the next run $a_{p+2}^{M_{p+2}}$. If $M_p \geq rest$, the entire run c_q^{rest} is matched, and we consider the next run $c_{q+1}^{K_{q+1}}$ in C. Also, since $a_p = c_q \neq c_{q+1}$, we can skip the rest of $a_p^{M_p}$ and consider the next run $a_{p+1}^{M_{p+1}}$ of A. Thus, we spend only constant time for each run of A that is scanned in the forward search. The same holds for the backward search.

To finish the proof, we show that the total number of times that each run of A is scanned in the procedure is bounded by $O(K)$, i.e., the number of minimal C-intervals of A that intersects with a given run $a_p^{M_p}$ of A is $O(K)$. Since $|RLE(C)| > 1$, a minimal C-interval cannot be contained in $a_p^{M_p}$. Thus, for a minimal C-interval to intersect with the run $a_p^{M_p}$, it must cross either the left boundary of the run, or the right boundary of the run. For a minimal C-interval to cross the left boundary of the run, it must be that for some non-empty strings u, v such that $C = uv$, u occurs as a subsequence in $a_1^{M_1} \cdots a_{p-1}^{M_{p-1}}$ and v occurs as a

Algorithm 1: computing all minimal C-intervals of A.

```
   Input: strings $A$ and $C$
   Output: all minimal $C$-intervals $[s_1, f_1], \ldots, [s_l, f_l]$ of $A$
   // $RLE(A) = a_1^{M_1} \cdots a_m^{M_m}$, $RLE(C) = c_1^{K_1} \cdots c_k^{K_k}$
   // $M_{1..p} = M_1 + \cdots + M_p$
   // $p, q$ : index of run in $A, C$ respectively
   // $rest$ : number of rest of searching characters of $c_q^{K_q}$
   // $l$ : number of minimal $C$-intervals in $A$
1  $p \leftarrow 1$; $q \leftarrow 1$; $rest \leftarrow K_1$; $l \leftarrow 0$;
2  while $true$ do
3      while $p \le m$ and $q \le k$ do // forward search
4          if $a_p \ne c_q$ then $p \leftarrow p + 1$;
5          else
6              if $M_p \ge rest$ then
7                  $q \leftarrow q + 1$;
8                  if $q > k$ then $l \leftarrow l + 1$; $f_l \leftarrow M_{1..p-1} + rest$;
9                  else $p \leftarrow p + 1$; $rest \leftarrow K_q$;
10             else $rest \leftarrow rest - M_p$; $p \leftarrow p + 2$;
11     if $p > m$ then break;
12     $p \leftarrow p - 1$;
13     if $rest = K_k$ then $q \leftarrow q - 1$; $rest \leftarrow K_{k-1}$;
14     else $q \leftarrow k$; $rest \leftarrow K_k - rest$;
15     while $q \ge 1$ do // backward search
16         if $a_p \ne c_q$ then $p \leftarrow p - 1$;
17         else
18             if $M_p \ge rest$ then
19                 $q \leftarrow q + 1$;
20                 if $q < 1$ then $s_l \leftarrow M_{1..p} - rest + 1$;
21                 else $p \leftarrow p - 1$; $rest \leftarrow K_q$;
22             else $rest \leftarrow rest - M_p$; $p \leftarrow p - 2$;
23     $p \leftarrow p + 1$; $q \leftarrow 1$; $rest \leftarrow K_1 - rest + 1$;
24 return $[s_1, f_1], \ldots, [s_l, f_l]$;
```

subsequence in $a_p^{M_p} \cdots a_m^{M_m}$. The minimal C-interval corresponds to the union of the left minimal u-interval ending at the left boundary of the run and the right minimal v-interval starting at the left boundary of the run and is thus unique for u, v. Similar arguments also hold for minimal C-intervals that cross the right boundary of $a_p^{M_p}$. Since there are only $K - 1$ choices for u, v, the claim holds, thus proving the Lemma. ◀

In Deorowicz's algorithm, two DP tables were computed for Step 2, which took $O(MN)$ time. For our algorithm, we use a compressed representation of the DP table for A and B, proposed by Bunke and Csirik [4], instead of the normal DP table. We note that Bunke and Csirik actually solved the edit distance problem when the cost is 1 for insertion and deletion, and 2 for substitution, but this easily translates to LCS: $L^{pref}(i, j) = (i + j - ED^{pref}(i, j))/2$, where $ED^{pref}(i, j)$ denotes the edit distance with such costs, between $A[1..i]$ and $B[1..j]$.

		B								
		a	a	a	a	b	b	b	a	a
A	b				0			1		1
	b				0			2		2
	b	0	0	0	0	1	2	3	3	3
	a				1			3		4
	a				2			3		5
	a				3			3		5
	a	1	2	3	4	4	4	4	4	5

Figure 1 An example of a compressed L^{pref} DP table for strings $A =$ bbbaaaa and $B =$ aaaabbbaa.

▶ **Definition 5** ([4]). Let A, B be strings of length M, N respectively, where $RLE(A) = a_1^{M_1} \cdots a_m^{M_m}$ and $RLE(B) = b_1^{N_1} \cdots b_n^{N_n}$. The *compressed DP table (cDP table) of* A, B is an $O(mN + nM)$-space compressed representation of the DP table of A, B which holds only the values of the DP table for $(M_{1..p}, j)$ and $(i, N_{1..q})$, where, $1 \le i \le M$, $1 \le j \le N$, $1 \le p \le m$, $1 \le q \le n$, $M_{1..p} = M_1 + \cdots + M_p$, $N_{1..q} = N_1 + \cdots + N_q$.

Figure 1 illustrates the values stored in the cDP table for strings $A =$ bbbaaaa, $B =$ aaaabbbaa. Note that although the figure depicts a sparsely filled table of size $M \times N$, the values are actually stored in two (completely filled) tables: one of size $m \times N$, holding the values of $(M_{1..p}, j)$, and another of size $M \times n$, holding the values of $(i, N_{1..q})$, for a total of $O(mN + nM)$ space. Below are results adapted from [4] we will use.

▶ **Lemma 6** ([4, Theorem 7]). *Let A and B be any strings where $|A| = M$, $|B| = N$, $|RLE(A)| = m$ and $|RLE(B)| = n$. The compressed DP table of A and B can be computed in $O(mN + nM)$ time and space.*

▶ **Lemma 7** ([4, Lemma 3]). *Let $\alpha \in \Sigma$ and let A and B be any strings where $|A| = M$ and $|B| = N$. For any integer $d \ge 1$, if $A[M - d + 1..M] = B[N - d + 1..N] = \alpha^d$, then $L^{pref}(M, N) = L^{pref}(M - d, N - d) + d$.*

▶ **Lemma 8** ([4, Lemma 5]). *Let $\alpha, \beta \in \Sigma$, $\alpha \neq \beta$ and let A and B be any strings where $|A| = M$ and $|B| = N$. For any integers $d \ge 1$ and $d' \ge 1$, if $A[M - d + 1..M] = \alpha^d$ and $B[N - d' + 1..N] = \beta^{d'}$ then $L^{pref}(M, N) = \max\{L^{pref}(M - d, N), L^{pref}(M, N - d')\}$.*

From Lemmas 7 and 8, we easily obtain the following Lemma 9.

▶ **Lemma 9.** *Let A and B be any strings. Any entry of the DP table of A and B can be retrieved in $O(1)$ time by using the compressed DP table of A and B.*

From Lemma 6, we can compute in $O(mN + nM)$ time, two cDP tables of A, B which respectively hold the values of $L^{pref}(M_{1..p}, j)$, $L^{pref}(i, N_{1..q})$ and $L^{suf}(M_{1..p}, j)$, $L^{suf}(i, N_{1..q})$, each of them taking $O(mN + nM)$ space. From Lemma 9, we can obtain $L^{pref}(i, j)$, $L^{suf}(i, j)$ for any i and j in $O(1)$ time. Actually, to make Lemma 9 work, we also need to be able to convert the indexes between DP and cDP in constant time, i.e., for any $1 \le p \le m$, $1 \le q \le n$, the values $M_{1..p}$ and $N_{1..q}$, and for any $1 \le i \le M$, $1 \le j \le N$, the largest p, q such that $M_{1..p} \le i$, $N_{1..q} \le j$. This is easy to do by preparing some arrays in $O(M + N)$ time and space.

Now we are ready to show the running time of our algorithm for the case $|RLE(C)| > 1$. We can compute $RLE(A)$,$RLE(B)$,$RLE(C)$ from A, B, C in $O(M + N + K)$ time. In Step 1, we have from Lemma 4, that the number of all minimal C-intervals of A, B are respectively $O(m)$ and $O(n)$, and can be computed in $O(M+N+mK+nK)$ time. For the preprocessing of Step 2, we build the cDP tables holding the values of $L^{pref}(i,j)$, $L^{suf}(i,j)$ for $1 \le i \le M$, $1 \le j \le N$, which can be computed in $O(mN+nM)$ time and space from Lemma 6. With these tables, we can obtain for any i, j, the values $L^{pref}(i,j)$, $L^{suf}(i,j)$ in constant time from Lemma 9. Since there are $O(mn)$ pairs of a minimal C-interval of A and a minimal C-interval of B, the total time for Step 2, i.e. computing L^{pref} and L^{suf} for each of the pairs, is $O(mn)$. Since $n \le N, m \le M$, and we can assume that $K \le M, N$, the total time is $O(mN+nM)$. Thus Theorem 3 holds.

3.2.2 Case $|RLE(C)| = 1$

Next, we consider the case where $|RLE(C)| = 1$, and C consists of only one run.

▶ **Theorem 10.** *Let A, B, C be any strings and let $|A| = M$, $|B| = N$, $|RLE(A)| = m$ and $|RLE(B)| = n$. If $|RLE(C)| = 1$, we can compute the length of an STR-IC-LCS of A, B, C in $O(mN+nM)$ time.*

For Step 1, we compute all minimal C-intervals of A and B by Lemma 11. Note the difference from Lemma 4 in the case of $|RLE(C)| > 1$.

▶ **Lemma 11.** *If $|RLE(C)| = 1$, the number of minimal C-intervals of A and B are $O(M)$ and $O(N)$, respectively, and these can be enumerated in $O(M)$ and $O(N)$ time, respectively.*

Proof. Let $\alpha \in \Sigma$, $C = \alpha^K$, and let M_α be the number of times that α occurs in A. Then the number of minimal C-intervals of A is $M_\alpha - K + 1 \in O(M)$. The minimal C-intervals can be enumerated in $O(M)$ time by checking all positions of α in A. The same applies to B. ◀

From Lemma 11, we can see that the number of pairs of minimal C-intervals of A and B can be $\Theta(MN)$, and we cannot afford to consider all of those pairs for Step 2. We overcome this problem as follows. Let $U = \{[s_1, f_1], \ldots, [s_l, f_l]\}$ be the set of all minimal C-intervals of A. Consider the partition $G(1), \ldots, G(g)$ of U which are the equivalence classes induced by the following equivalence relation on U: For any $1 \le p \le q \le m$ and $[s_x, f_x], [s_y, f_y] \in U$,

$$[s_x, f_x] \equiv [s_y, f_y] \iff M_{1..p-1} < s_x, s_y \le M_{1..p} \text{ and } M_{1..q-1} < f_x, f_y \le M_{1..q}, \tag{1}$$

where, $M_{1..0} = 0$. In other words, $[s_x, f_x]$ and $[s_y, f_y]$ are in the same equivalence class if they start in the same run, and end in the same run. Noticing that minimal C-intervals cannot be completely contained in another minimal C-interval, we can assume that for $1 \le h < h' \le g$, $[s_x, f_x] \in G(h)$ and $[s_y, f_y] \in G(h')$, we have $s_x < s_y$ and $f_x < f_y$.

▶ **Lemma 12.** *Let $G(1), \ldots, G(g)$ be the partition of the set U of all minimal C-intervals of A induced by the equivalence relation (1). Then, $g \in O(m)$.*

Proof. Let $1 \le x < y \le l$ and $2 \le h \le g$. For any $[s_x, f_x] \in G(h-1)$ and $[s_y, f_y] \in G(h)$, let $1 \le p \le q \le m$ satisfy $M_{1..p-1} < s_x \le M_{1..p}$, $M_{1..q-1} < f_x \le M_{1..q}$. Since the intervals are not equivalent, either $M_{1..p} < s_y$ or $M_{1..q} < f_y$ must hold. Thus, $g \in O(m)$. ◀

Equivalently for B, we consider the set $U' = \{[s'_1, f'_1], \ldots, [s'_{l'}, f'_{l'}]\}$ of all minimal C-intervals of B, and the partition $G'(1), \ldots, G'(g')$ of U' based on the analogous equivalence relation, where $g' \in O(n)$.

		B							
		a	b	b	b	a	a	a	$\cdots$
	a	1			1	(1)	(1)		
	a	1			1	(2)	(2)		
	a	1			1	(2)	(3)		
A	a	1			1	(2)	(3)		
	a	1	1	1	1	2	3	4	$\cdots$
	b	1			2				
	$\vdots$	$\vdots$			$\vdots$				

		B						
		$\cdots$	a	a	a	b	b	b
	$\vdots$					$\vdots$		
	a			(4)	(3)	2		
	a			(4)	(3)	2		
A	a			(3)	(3)	2		
	b	$\cdots$	2	2	2	2	2	1
	b					1		
	a	$\cdots$	1	1	1	0	0	0

Figure 2 An example depicting the LCSs of corresponding prefixes (left) and suffixes (right) of all combinations of $G(2)$ and $G'(2)$ for strings $RLE(A) = \mathtt{a}^5\mathtt{b}^3\mathtt{a}^4\mathtt{b}^2\mathtt{a}^1$, $RLE(B) = \mathtt{a}^1\mathtt{b}^3\mathtt{a}^7\mathtt{b}^3$, and $RLE(C) = \mathtt{a}^5$. The values denoted inside parentheses are not stored in the cDP table, but each of them can be computed in $O(1)$ time.

For some h, let $[s_x, f_x]$, $[s_y, f_y]$ be the minimal C-intervals in $G(h)$ with the smallest and largest start positions. Since by definition, $A[s_x] = \cdots = A[s_y] = A[f_x] = \cdots = A[f_y]$, we have $G(h) = \{[s_x, f_x], [s_x + 1, f_x + 1], \ldots, [s_y, f_y]\}$. The same can be said for $G'(h')$ of B. From this observation, we can show the following Lemma 13.

▶ **Lemma 13.** *For any $1 \le h \le g$ and $1 \le h' \le g'$, let $[s, f]$, $[s+d, f+d] \in G(h)$ and $[s', f']$, $[s'+d, f'+d] \in G'(h')$, for some positive integer d. Then,*

$$L^{pref}(s-1, s'-1) + L^{suf}(f+1, f'+1) = L^{pref}(s+d-1, s'+d-1) + L^{suf}(f+d+1, f'+d+1).$$

Proof. Since $A[s..s+d] = A[f..f+d] = B[s'..s'+d] = B[f'..f'+d] = C[1]^d$, we have from Lemma 7, $L^{pref}(s+d-1, s'+d-1) = L^{pref}(s-1, s'-1) + d$, and $L^{suf}(f+1, f'+1) = L^{suf}(f+d+1, f'+d+1) + d$. ◀

From Lemma 13, we can see that for any $G(h), G'(h')$ $(1 \le h \le g,\ 1 \le h' \le g')$, we do not need to compute $L^{pref}(s-1, s'-1) + L^{suf}(f+1, f'+1)$ for all pairs of $[s, f] \in G(h)$ and $[s', f'] \in G'(h')$. Let $G_{\min}(h)$ and $G'_{\min}(h')$ be the minimal C-intervals respectively in $G(h)$ and $G'(h')$ with the smallest starting position. Then, we only need to consider the combination of $G_{\min}(h)$ with each of $[s', f'] \in G'(h')$, and the combination of each of $[s, f] \in G(h)$ with $G'_{\min}(h')$. Therefore, of all combinations of minimal C-intervals in U and U', we only need to consider for all $1 \le h \le g$ and $1 \le h' \le g'$, the combination of $G_{\min}(h)$ with each of U', and each of U with $G'_{\min}(h')$. The number of such combinations is clearly $O(mN + nM)$.

For example, consider $RLE(A) = \mathtt{a}^5\mathtt{b}^3\mathtt{a}^4\mathtt{b}^2\mathtt{a}^1$, $RLE(B) = \mathtt{a}^1\mathtt{b}^3\mathtt{a}^7\mathtt{b}^3$, $RLE(C) = \mathtt{a}^5$. For the minimal C-intervals of A, we have $G(1) = \{[1, 5]\}$, $G(2) = \{[2, 9], [3, 10], [4, 11], [5, 12]\}$, $G(3) = \{[9, 15]\}$. For the minimal C-intervals of B, we have $G'(1) = \{[1, 8]\}$, $G'(2) = \{[5, 9], [6, 10], [7, 11]\}$. Also, $G_{\min}(2) = [2, 9]$, $G'_{\min}(2) = [5, 9]$. Figure 2 shows the lengths of the LCS of prefixes and suffixes for each combination between minimal C-intervals in $G(2)$ and $G'(2)$. The gray part is the values that are referred to. The values denoted inside parentheses are not stored in the cDP table, but each of them can be computed in $O(1)$ time from Lemma 9. Figure 3 shows the sum of the LCS of prefixes and suffixes corresponding to the gray part. Due to Lemma 13, the values along the diagonal are equal. Thus, for the combinations of minimal C-intervals in $G(2)$, $G'(2)$, we only need to consider the six combinations: $([2, 9], [5, 9])$,$([2, 9], [6, 10])$,$([2, 9], [7, 11])$,$([3, 10], [5, 9])$,$([4, 11], [5, 9])$,$([5, 12], [5, 9])$.

5	4	3
5	5	4
4	5	5
3	4	5

Figure 3 Sum of the lengths of LCSs of corresponding prefixes and suffixes shown in Figure 2. Values along the diagonal are equal (each value is equal to the value to its upper left/lower right).

Now, we are ready to show the running time of our algorithm for the case $|RLE(C)| = 1$. We can compute $RLE(A)$, $RLE(B)$, $RLE(C)$ from A, B, C in $O(M + N + K)$ time. There are respectively $O(M)$ and $O(N)$ minimal C-intervals of A and B, and each of them can be assigned to one of the $O(m)$ and $O(n)$ equivalence classes G, G', in total of $O(M + N)$ time. The preprocessing for the cDP table is the same as for the case of $|RLE(C)| > 1$, which can be done in $O(mN + nM)$ time. By Lemma 13, we can reduce the number of combinations of minimal C-intervals to consider to $O(mN + nM)$. Finally, from Lemma 9, the LCS lengths for each combination can be computed in $O(1)$ using the cDP table. Therefore, the total running time is $O(mN + nM)$, proving Theorem 10.

From Theorems 3 and 10, the following Theorem 14 holds. The pseudo-code for our proposed algorithm is shown in Algorithm 2.

▶ **Theorem 14.** *Let A, B, C be any strings and let $|A| = M$, $|B| = N$, $|RLE(A)| = m$ and $|RLE(B)| = n$. We can compute the length of an STR-IC-LCS of A, B, C in $O(mN + nM)$ time.*

Although we only showed how to compute the length of an STR-IC-LCS, we note that the algorithm can be modified so as to obtain a RLE of an STR-IC-LCS in $O(m + n)$ time, provided that $RLE(C)$ is precomputed, simply by storing the minimal C-intervals $[s, f]$, $[s', f']$, respectively of A and B, that maximizes $L^{pref}(s-1, s'-1) + L^{suf}(f+1, f'+1) + |C|$. From Lemmas 7 and 8, we can simulate a standard back-tracking of the DP table for obtaining LCSs with the cDP table to obtain RLE of the LCSs in $O(m + n)$ time. Finally an RLE of STR-IC-LCS can be obtained by combining the three RLE strings (the two LCSs with $RLE(C)$ in the middle), appropriately merging the boundary runs if necessary.

4 Conclusion

In this work, we proposed a new algorithm to solve the STR-IC-LCS problem using an RLE representation. We can compute the length of an STR-IC-LCS of strings A, B, C in $O(mN + nM)$ time and space using this algorithm, where $|A| = M$, $|B| = N$, $|RLE(A)| = m$ and $|RLE(B)| = n$. This result is better than Deorowicz's $O(MN)$ time and space [7], which does not use RLE. If we want to know not only the length but also an STR-IC-LCS of A, B, C, we can retrieve it in $O(m + n)$ time.

Algorithm 2: Proposed $O(mN + Mn)$ time algorithm for STR-IC-LCS.

```
   Input: strings $A$, $B$ and $C$
   Output: length of an STR-IC-LCS of $A, B, C$
   // $[s_x, f_x]$ : a minimal $C$-interval in $A$
   // $[s'_y, f'_y]$ : a minimal $C$-interval in $B$
   // $l, l'$ : number of minimal $C$-intervals in $A, B$ respectively
   // $G_{\min}(h), G'_{\min}(h')$ : minimum element in $G(h), G'(h')$ respectively
   // $g, g'$ : number of sets $G, G'$ respectively
1  Make compressed DP tables of $A$ and $B$.;
2  if $|RLE(C)| > 1$ then
3      Compute all minimal $C$-intervals $[s_1, f_1], \dots, [s_l, f_l]$ of $A$ and $[s'_1, f'_1], \dots, [s'_{l'}, f'_{l'}]$ of
       $B$. (use Algorithm 1);
4      $L_{\max} \leftarrow 0$;
5      for $x = 1$ to $l$ do
6          for $y = 1$ to $l'$ do
7              $L_{\text{sum}} \leftarrow L^{pref}(s_x - 1, s'_y - 1) + L^{suf}(f_x + 1, f'_y + 1)$;
8              if $L_{\max} < L_{\text{sum}}$ then $L_{\max} \leftarrow L_{\text{sum}}$ ;
9  else
10     1 $l \leftarrow 1 - K$; $g \leftarrow 1$; $G_{\min}(1) \leftarrow 1$;
11     for $p = 1$ to $m$ do
12         if $a_p = C[1]$ then
13             for $p' = 1$ to $M_p$ do
14                 $l \leftarrow l + 1$; $s_{l+K} \leftarrow M_{1..p} + p'$;
15                 if $l \geq 1$ then $f_l \leftarrow M_{1..p} + p'$ ;
16                 if $l \geq 2$ then
17                     if $s_{l-1} + 1 \neq s_l$ or $f_{l-1} + 1 \neq f_l$ then $g \leftarrow g + 1$; $G_{\min}(g) \leftarrow l$ ;
18     $l' \leftarrow 1 - K$; $g' \leftarrow 1$; $G'_{\min}(1) \leftarrow 1$;
19     for $q = 1$ to $n$ do
20         if $b_q = C[1]$ then
21             for $q' = 1$ to $N_q$ do
22                 $l' \leftarrow l' + 1$; $s'_{l'+K} \leftarrow N_{1..q} + q'$;
23                 if $l' \geq 1$ then $f'_{l'} \leftarrow N_{1..q} + q'$ ;
24                 if $l' \geq 2$ then
25                     if $s'_{l'-1} + 1 \neq s'_{l'}$ or $f'_{l'-1} + 1 \neq f'_{l'}$ then $g' \leftarrow g' + 1$; $G'_{\min}(g') \leftarrow l'$ ;
26     $G_{\min}(g + 1) \leftarrow l + 1$; $G'_{\min}(g' + 1) \leftarrow l' + 1$;
27     $L_{\max} \leftarrow 0$;
28     for $h = 1$ to $g$ do
29         for $h' = 1$ to $g'$ do
30             for $x = G_{\min}(h)$ to $G_{\min}(h + 1) - 1$ do
31                 $L_{\text{sum}} \leftarrow L^{pref}(s_x - 1, s'_{G'_{\min}(h')} - 1) + L^{suf}(f_x + 1, f'_{G'_{\min}(h')} + 1)$;
32                 if $L_{\max} < L_{\text{sum}}$ then $L_{\max} \leftarrow L_{\text{sum}}$ ;
33             for $y = G'_{\min}(h')$ to $G'_{\min}(h' + 1) - 1$ do
34                 $L_{\text{sum}} \leftarrow L^{pref}(s_{G_{\min}(h)} - 1, s'_y - 1) + L^{suf}(f_{G_{\min}(h)} + 1, f'_y + 1)$;
35                 if $L_{\max} < L_{\text{sum}}$ then $L_{\max} \leftarrow L_{\text{sum}}$ ;
36 return $L_{\max} + K$;
```

References

1 Shegufta Bakht Ahsan, Syeda Persia Aziz, and M. Sohel Rahman. Longest common subsequence problem for run-length-encoded strings. *J. Comput.*, 9(8):1769–1775, 2014. doi:10.4304/jcp.9.8.1769-1775.

2 Hsing-Yen Ann, Chang-Biau Yang, Chiou-Ting Tseng, and Chiou-Yi Hor. Fast algorithms for computing the constrained LCS of run-length encoded strings. *Theor. Comput. Sci.*, 432:1–9, 2012. doi:10.1016/j.tcs.2012.01.038.

3 Anthony Bagnall, Chotirat "Ann" Ratanamahatana, Eamonn Keogh, Stefano Lonardi, and Gareth Janacek. A bit level representation for time series data mining with shape based similarity. *Data Min. Knowl. Discov.*, 13(1):11–40, 2006. doi:10.1007/s10618-005-0028-0.

4 Horst Bunke and János Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *Inf. Process. Lett.*, 54(2):93–96, 1995. doi:10.1016/0020-0190(95)00005-W.

5 Yi-Ching Chen and Kun-Mao Chao. On the generalized constrained longest common subsequence problems. *J. Comb. Optim.*, 21(3):383–392, 2011. doi:10.1007/s10878-009-9262-5.

6 Francis Y. L. Chin, Alfredo De Santis, Anna Lisa Ferrara, N. L. Ho, and S. K. Kim. A simple algorithm for the constrained sequence problems. *Inf. Process. Lett.*, 90(4):175–179, 2004. doi:10.1016/j.ipl.2004.02.008.

7 Sebastian Deorowicz. Quadratic-time algorithm for a string constrained LCS problem. *Inf. Process. Lett.*, 112(11):423–426, 2012. doi:10.1016/j.ipl.2012.02.007.

8 Paul Heckel. A technique for isolating differences between files. *Commun. ACM*, 21(4):264–268, 1978. doi:10.1145/359460.359467.

9 James W. Hunt and Malcolm Douglas McIlroy. An algorithm for differential file comparison. Technical Report 41, Bell Laboratories, 1976. URL: http://www.cs.dartmouth.edu/~doug/diff.pdf.

10 Dmitry Korkin and Lev Goldfarb. Multiple genome rearrangement: a general approach via the evolutionary genome graph. *Bioinformatics*, 18(suppl_1):S303–S311, 2002. doi:10.1093/bioinformatics/18.suppl_1.s303.

11 Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Min. Knowl. Discov.*, 15(2):107–144, 2007. doi:10.1007/s10618-007-0064-z.

12 Jia-Jie Liu, Yue-Li Wang, and Yu-Shan Chiu. Constrained longest common subsequences with run-length-encoded strings. *Comput. J.*, 58(5):1074–1084, 2015. doi:10.1093/comjnl/bxu012.

13 Helman Stern, Merav Shmueli, and Sigal Berman. Most discriminating segment – longest common subsequence (MDSLCS) algorithm for dynamic hand gesture classification. *Pattern Recognit. Lett.*, 34(15):1980–1989, 2013. doi:10.1016/j.patrec.2013.02.007.

14 Yin-Te Tsai. The constrained longest common subsequence problem. *Inf. Process. Lett.*, 88(4):173–176, 2003. doi:10.1016/j.ipl.2003.07.001.

15 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.

16 Congmao Wang and Dabing Zhang. A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Res.*, 39(7):e45, 2011. doi:10.1093/nar/gkr009.

17 Lei Wang, Xiaodong Wang, Yingjie Wu, and Daxin Zhu. A dynamic programming solution to a generalized LCS problem. *Inf. Process. Lett.*, 113(19-21):723–728, 2013. doi:10.1016/j.ipl.2013.07.005.

Gapped Pattern Statistics

Philippe Duchon[1], Cyril Nicaud[2], and Carine Pivoteau[3]

1 **CNRS and Université Bordeaux, LaBRI, Talence, France**
philippe.duchon@u-bordeaux.fr
2 **CNRS and Université Paris-Est, Marne-la-Vallée, France**
cyril.nicaud@u-pem.fr
3 **CNRS and Université Paris-Est, Marne-la-Vallée, France**
pivoteau@univ-mlv.fr

Abstract

We give a probabilistic analysis of parameters related to α-gapped repeats and palindromes in random words, under both uniform and memoryless distributions (where letters have different probabilities, but are drawn independently). More precisely, we study the expected number of maximal α-gapped patterns, as well as the expected length of the longest α-gapped pattern in a random word.

1998 ACM Subject Classification G.2.1 Combinatorics

Keywords and phrases combinatorics on words, α-gapped repeats, random words, memoryless sources, analytic combinatorics

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.21

1 Introduction

In this article, we are interested in the combinatorial aspects of the notion of α-gapped repeat and α-gapped palindromes [10, 7, 4]. An α-gapped repeat in a word is a factor of the form uvu, where u and v are words with $|uv| \leq \alpha|u|$. More precisely, such a pattern is essentially a repetition of u, but the second occurrence is not too far away from the first one. The definition for palindromes is similar, as we are looking for factors of the form $uv\overline{u}$ instead, where $\overline{u}$ is the reverse of u. The study of gapped patterns (see also [1, 12]) finds most of its motivation in bioinformatics. Recent works show that these patterns can be found in linear time [11, 17, 6], and there cannot be more than a linear number of them [2, 7]. Note that α-gapped repeats are also called fractional powers [16]: uvu is an α-gapped repeat if and only if it is a fractional power of uv with exponent at least $1 + \alpha^{-1}$.

When looking at patterns in words, there are usually two main categories of questions: providing efficient algorithms to find a specific set of patterns and studying the combinatorics of words with a focus on the appearance (or avoidance) of these patterns. These two points of view are of course directly related, as insights on the combinatorial properties often yield ideas for building new efficient algorithms.

In the sequel, we propose a probabilistic analysis of parameters related to α-gapped repeats and palindromes; more precisely, we answer the following questions:

- What is the expected number of α-gapped patterns in a random word?
- What is the expected length of the longest α-gapped pattern in a random word?

This only makes sense if one specifies what is meant by a random word, *i.e.*, what the distribution on words is. We first consider the uniform distribution, which often serves as an introductory example for the techniques we use and can provide, for instance, useful elements for average analysis of algorithms, while still being mathematically tractable. We also consider memoryless sources, which give a more general, yet simple distribution where all letters are not constrained to have identical frequencies. In this model, each letter is drawn independently following a fixed, but possibly biased, distribution on the alphabet. In particular, we exhibit a noteworthy behavior on the longest α-gapped repeat: if each letter a_i has probability π_i, then a long random word of length n has about $\pi_i n$ occurrences of each letter a_i; however, in a long α-gapped repeat uvu, the frequencies of the letters in the u parts of the factor do not follow this typical distribution (see section 4.2).

Our work follows several other combinatorial and probabilistic results obtained for different kinds of patterns in words, such as the expected number of runs [14], the expected total run length [8], the expected number of distinct palindromic factors [15], *etc.* We use both techniques from analytic combinatorics, based on the definition of generating series for gapped patterns in words, as in [13], and classical discrete probabilities.

2 Preliminaries

For any two nonnegative integers i, j, let $[i, j]$ denote the integer interval $\{i, \dots, j\}$. By convention, $[i, j] = \emptyset$ if $j < i$. Let also $[i]$ denote the integer interval $[1, i]$.

In the sequel we consider words on a finite alphabet A, of cardinality $k \geq 2$. We assume the reader is familiar with the classical definitions on words [3], such as prefixes, suffixes, and factors. For $w \in A^*$ of length n and $i \in [n]$, let w_i (or $w[i]$) denote the i-th letter of w, with the convention that positions start at 1. The last letter of w is therefore $w_{|w|}$. Let also $w[i, j] = w_i \cdots w_j$ denote the factor of w that starts at position i and ends at position j, with $w[i, j] = \varepsilon$ if $i > j$ or if i or j is not in $[n]$. The factor of w of length ℓ that starts at position i is $w[i, i + \ell - 1]$. For a given length ℓ, a position i in w is *valid* if $i + \ell - 1 \leq n$.

A *gapped repeat* in a word w of length n is a triple (i, u, v), where $i \in [n]$ and u and v are nonempty words, such that the factor of w of length $|uvu|$ starting at position i is uvu. For a given real $\alpha \geq 1$, it is an α-gapped repeat if $|uv| \leq \alpha |u|$. A gapped repeat (i, u, v) of w is *maximal* if, when the positions exist, $w_{i-1} \neq w_{i+|uv|-1}$ and $w_{i+|uvu|} \neq w_{i+|u|}$, *i.e.*, the gapped repeat cannot be extended to the left or to the right.

Similar notions can be defined for palindromes. Under the same conditions for i, u and v, a triple (i, u, v) is an α-*gapped palindrome* if the factor of length $|uv\overline{u}|$ starting at position i in w is $uv\overline{u}$, where $\overline{u} = u_{|u|} \cdots u_1$ denote the reverse of u. It is an α-*gapped palindrome* if $|uv| \leq \alpha |u|$ and maximal if $w_{i-1} \neq w_{i+|uvu|}$ (when they exist) and either $|v| = 1$ or $v_1 \neq v_{|v|}$.

▶ **Example 1.** Consider $w = aab\underline{abbbab}ab$ and $\alpha = 2$. The triple $(4, ab, bb)$ is an α-gapped repeat, but it is not maximal since it can be extended to the left to form $(3, bab, b)$.

▶ Remark. In the sequel, we only consider α-gapped patterns (repeats or palindromes) for rational $\alpha \geq 1$. This really matters for Section 3 only, as the other results hold for any real $\alpha \geq 1$. It is also convenient to consider $\beta := \alpha - 1$ in most computations, as it changes the condition into $|u| \leq \beta |v|$, and we therefore use this notation from now on.

The *uniform distribution* on a finite set E is the probability π defined for all $e \in E$ by $\pi(e) = \frac{1}{|E|}$. By a slight abuse of notation, we will speak of the *uniform distribution on* A^* to denote the sequence $(\pi_n)_{n \geq 0}$ of uniform distributions on A^n. For instance, if $A = \{a, b, c\}$, then each element of A^n has probability 3^{-n} under this distribution.

Another very classical distribution on A^n is the *memoryless distribution of probability* π, where π is a probability on the alphabet A. Under this distribution, the probability of a word $w = w_1 \cdots w_n \in A^n$ is $\mathbb{P}_n(w) = \pi(w_1) \cdots \pi(w_n)$. This distribution can also be seen as generating each letter of the word independently, following π.

It is convenient to fix a total order $a_1 < \ldots < a_k$ on A and to define $\pi_i = \pi(a_i)$, for all $i \in [k]$. We also see π as a vector $\vec{\pi} = (\pi_1, \ldots, \pi_k)$ of $[0,1]^k$. This notation will be used repeatedly in the sequel.

3 Number of gapped patterns

In this section, we compute the average number of maximal α-gapped patterns (repeats or palindromes) in random words of length n under a memoryless distribution. Our main tool is writing exact generating functions, which happen to be rational fractions; the asymptotic behavior is then obtained by using standard theorems of analytic combinatorics [5].

3.1 Framework

Let $A = \{a_1, \ldots, a_k\}$ be an alphabet and, for every $i \in [k]$, let z_i be a formal variable (associated with the letter a_i). To each word $w \in A^*$ we associate a monomial $c(w) = z_1^{|w|_1} \ldots z_k^{|w|_k}$, where $|w|_i$ is the number of occurrences of the letter a_i in w. In other words, the mapping c allows us to consider words as in the abelian world, where letters commute. Let $\vec{z} = (z_1, \ldots, z_k)$. If $\mathcal{X}$ is a set of words, its *formal power series* $X(\vec{z})$ is defined as the formal sum of the monomials associated with its words: $X(\vec{z}) = \sum_{w \in \mathcal{X}} c(w)$. As we shall see, this power series is a tool of choice to study the probabilistic properties of the set $\mathcal{X}$.

First, the *symbolic method* [5] can be used to build $X(\vec{z})$, directly from a nonambiguous regular description of $\mathcal{X}$: if $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ are three sets of words whose respective series are $X(\vec{z})$, $Y(\vec{z})$ and $Z(\vec{z})$, then

- if $\mathcal{X}$ is the disjoint union of $\mathcal{Y}$ and $\mathcal{Z}$, then $X(\vec{z}) = Y(\vec{z}) + Z(\vec{z})$;
- if $\mathcal{X}$ is the nonambiguous concatenation of $\mathcal{Y}$ and $\mathcal{Z}$, then $X(\vec{z}) = Y(\vec{z})Z(\vec{z})$;
- if $\mathcal{X}$ is the nonambiguous Kleene star of $\mathcal{Y}$, then $X(\vec{z}) = \frac{1}{1-Y(\vec{z})}$.

Second, for a given probability $\vec{\pi} = (\pi_1, \ldots, \pi_k)$ on A, one can build the formal power series in a single variable $\overline{X}(z)$, by substituting $\pi_i\, z$ to each z_i. After the substitution, the contribution of each word of length n to the coefficient of z^n in $\overline{X}(z)$, in the memoryless model, is exactly its probability. By marking a certain set of patterns with a copy of the alphabet, one can effectively multiply the contribution of a word by its number of patterns, and hence compute the expected number of such patterns using this technique (another approach is to control the unambiguity of the description [13]). Once $\overline{X}(z)$ is known, analytic combinatorics can be used to estimate the quantities under study.

Let us illustrate this technique on a toy example. Assume that we want to compute the expected number of occurrences of the pattern aba in a random word of length n under the memoryless distribution on the alphabet $\{a,b\}$, with[1] $\pi_a = \frac{1}{3}$ and $\pi_b = \frac{2}{3}$. Observe that the word $w = bbababaaab$ contains two (overlapping) occurrences of the pattern. The *marking technique* consists in distinguishing these two occurrences by using another alphabet, say $\{\overline{a}, \overline{b}\}$ for the letters of the pattern. The associated regular language is $\mathcal{L} = (a+b)^*\overline{aba}(a+b)^*$. The two words $w = bb\overline{aba}baaab$ and $w = bbab\overline{aba}aab$ correspond to w, which therefore contributes twice, as the pattern occurs twice. Using the symbolic method directly yields that the generating series of $\mathcal{L}$ is

$$L(\vec{z}) = \frac{1}{1-(z_a+z_b)} \cdot z_a z_b z_a \cdot \frac{1}{1-(z_a+z_b)} = \frac{z_a^2 z_b}{(1-z_a-z_b)^2}.$$

[1] For readability, we use π_a, π_b, z_a and z_b instead of π_1, π_2, z_1 and z_2.

Then, we compute $\overline{L}(z)$ by performing the substitutions $z_a \mapsto \pi_a z$ and $z_b \mapsto \pi_b z$:

$$\overline{L}(z) = \frac{\pi_a^2 \pi_b z^3}{(1 - \pi_a z - \pi_b z)^2} = \frac{\pi_a^2 \pi_b z^3}{(1-z)^2} = \frac{2z^3}{27\,(1-z)^2}.$$

The coefficient of z^n in $\overline{L}(z)$ is the expected number of occurrences of the pattern in a random word of length n. The expression above is amenable to the analytic technique presented below (see Section 3.3), yielding the (natural) estimate of $\frac{2n}{27}$ occurrences on average.

3.2 Generating series for the expected number of patterns

We now use this general framework to compute the expected number of maximal α-gapped patterns. To simplify the notations, for any positive integer i and any vector $\vec{z}$, let $N_i(\vec{z}) = z_1^i + \ldots + z_k^i$. In particular, $N_1(\vec{z}) = z_1 + \ldots + z_k$.

A gapped pattern is equivalent to a triple of words (u, v, u'), with a condition $u' = u$ (for gapped repeats) or $u' = \overline{u}$ (for gapped palindromes), and a length condition $1 \leq |v| \leq \beta|u|$, which we rewrite into the equivalent $|u| \geq |v|/\beta$ and $|v| \geq 1$. Because we are ultimately interested in *maximal* patterns, we need to keep track of the first and last letters of v; this, in turn, forces us to distinguish between the subcases $|v| = 1$ and $|v| \geq 2$.

- In the simpler case $|v| = 1$, a pattern is just given by a single letter $a \in A$, and an arbitrary word u of length at least $\lceil 1/\beta \rceil$. The generating series for words of length at least ℓ is $N_1(\vec{z})^\ell/(1 - N_1(\vec{z}))$. In our patterns, the letters of u are to be counted twice, once in u and once in u'. This is taken into account by just changing $N_1(\vec{z})$ into $N_2(\vec{z})$ into the formula. Hence, the generating series for α-gapped patterns with $v = a_i$ is $\frac{z_i N_2(\vec{z})^{\lceil 1/\beta \rceil}}{1 - N_2(\vec{z})}$.

  We now want to add a prefix and a suffix (both possibly empty) to the patterns. To avoid ambiguity in the description, we duplicate the alphabet and consider that patterns are written using this newly introduced copy. We are therefore considering words with one *marked* pattern, clearly identified. We also want the marked patterns to be maximal; this adds a condition on the prefix (resp. suffix) when it is not empty. This condition is slightly different for gapped repeats and gapped palindromes; we deal with gapped repeats first. Then the condition is that both prefix and suffix can be empty, but if they are not, the last letter of the prefix and the first letter of the suffix must be different from a_i. The generating series for both the possible prefixes and suffixes are the same, and equal to $(1 - z_i)/(1 - N_1(\vec{z}))$. Summing over all possible i, the generating series for all words with a marked maximal α-gapped pattern having a gap of length exactly 1 is therefore

  $$U_\alpha(\vec{z}) = \frac{(N_1(\vec{z}) - 2N_2(\vec{z}) + N_3(\vec{z}))N_2(\vec{z})^{\lceil 1/\beta \rceil}}{(1 - N_1(\vec{z}))^2(1 - N_2(\vec{z}))}.$$

  For gapped palindromes, there is a condition on the prefix and suffix when they are both nonempty: the last letter of the prefix must be different from the first letter of the suffix. This leads to multiplying the generating series for all patterns by the generating series for this set of pairs of words, which is $\frac{1 - N_2(\vec{z})}{(1 - N_1(\vec{z}))^2}$. We thus get as the generating series for all words with a marked maximal α-gapped palindrome having a gap of length exactly 1,

  $$\overline{U}_\alpha(\vec{z}) = \frac{(1 - N_2(\vec{z}))\, N_1(\vec{z})\, N_2(\vec{z})^{\lfloor 1/\beta \rfloor}}{(1 - N_1(\vec{z}))^2\,(1 - N_2(\vec{z}))}.$$

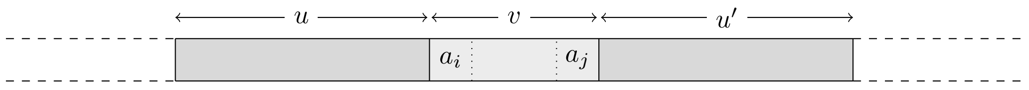


Figure 1 A gapped pattern uvu' with the first and last letters in v distinguished.

- We now turn to the case $|v| \geq 2$. For any two letters a_i and a_j, we consider the possible gapped patterns (see Figure 1) such that v starts with a_i and ends with a_j (for maximal gapped palindromes, an additional condition is $i \neq j$). Let $\ell + 2$ be the length of such a word v; the α-gapped condition is thus $|u| \geq (\ell+2)/\beta$. Writing $\beta = p/q$ with positive integers p and q, and writing the Euclidean division of ℓ by p as $\ell = tp + m$, the condition becomes $|u| \geq tq + (m+2)/\beta$.
  Thus, in the pattern uvu', u is obtained by concatenation of t arbitrary words of length q, plus one arbitrary word of length $\lceil (m+2)/\beta \rceil$, plus an arbitrary (possibly empty) word; and v starts with a_i, concatenated with t arbitrary words of length p, plus one arbitrary word of length m, and ends with a_j. In the pattern composition, the composition of u has to be counted twice since u' also contributes and has the same composition. Summing over all possible values of t and m, we get the generating series for all α-gapped patterns such that v starts with a_i and ends with a_j:

$$G_{\alpha,i,j}(\vec{z}) = \frac{z_i z_j Q_\alpha(\vec{z})}{(1 - N_2(\vec{z}))(1 - N_1(\vec{z})^p N_2(\vec{z})^q)}, \text{ with } Q_\alpha(\vec{z}) = \sum_{m=0}^{p-1} N_1(\vec{z})^m N_2(\vec{z})^{\lceil (m+2)/\beta \rceil}.$$

  Writing the generating functions for all words with a marked maximal gapped pattern again corresponds to adding a prefix and suffix, but leads to different generating functions for repeats and palindromes because the conditions on the suffix and prefix are slightly different.
  For gapped repeats, both the prefix and the suffix can be empty, or an arbitrary word that does not end with a_j (for the prefix), or that does not start with a_i (for the suffix). This is done by multiplying the generating series $G_{\alpha,i,j}(\vec{z})$ by $(1-z_i)(1-z_j)/(1-N_1(\vec{z}))^2$. Taking the sum over all possible j yields that the generating series of all words with a marked maximal α-gapped repeat and $|v| \geq 2$ is

$$V_\alpha(\vec{z}) = \frac{(N_1(\vec{z}) - N_2(\vec{z}))^2 \, Q_\alpha(\vec{z})}{(1 - N_1(\vec{z}))^2 (1 - N_2(\vec{z}))(1 - N_1(\vec{z})^p N_2(\vec{z})^q)}.$$

  For gapped palindromes, maximality induces two conditions. First the last letter of v must be different from its first letter; this is taken into account by summing $G_{\alpha,i,j}(\vec{z})$ over all possible $i \neq j$. Second, the prefix and suffix must also satisfy the same conditions as for the case $|v| = 1$, which leads to multiply by $\frac{1-N_2(\vec{z})}{(1-N_1(\vec{z}))^2}$ as before. Hence, the generating series of all words with a marked maximal α-gapped palindrome and $|v| \geq 2$ is

$$\overline{V}_\alpha(\vec{z}) = \frac{(N_1(\vec{z})^2 - N_2(\vec{z})) \, Q_\alpha(\vec{z})}{(1 - N_1(\vec{z}))^2 (1 - N_1(\vec{z})^p N_2(\vec{z})^q)}.$$

We can now proceed with the substitution $z_i \to \pi_i z$, which changes $N_1(\vec{z})$ into z, $N_2(\vec{z})$ into $\lambda_2 z^2$ and $N_3(\vec{z})$ into $\lambda_3 z^3$, with $\lambda_j = \sum_i \pi_i^j$.
Let $\chi(w)$ (resp. $\xi(w)$) denote the number of maximal α-gapped repeats (resp. palindromes) in a word w. Let $R(z) = \sum_w \chi(w)\mathbb{P}(w)z^{|w|}$ and $P(z) = \sum_w \xi(w)\mathbb{P}(w)z^{|w|}$ be the generating series of the expectations of χ and ξ, that is, the coefficients of z^n of $R(z)$ and $P(z)$ are $\mathbb{E}_n[\chi]$ and $\mathbb{E}_n[\xi]$, respectively. These series $R(z)$ and $P(z)$ are obtained by

the previous substitutions $z_i \to \pi_i z$ from the series $U_\alpha(\vec{z}) + V_\alpha(\vec{z})$ and $\overline{U}_\alpha(\vec{z}) + \overline{V}_\alpha(\vec{z})$, respectively. From the computations above, we obtain the following statement.

▶ **Theorem 2.** *For $\beta = \alpha - 1 = \frac{p}{q}$, the series $R(z)$ and $P(z)$ for the memoryless model of probability $\vec{\pi}$ are given by*

$$R(z) = \frac{(z - 2\lambda_2 z^2 + \lambda_3 z^3)\lambda_2^{\lceil 1/\beta \rceil} z^{2\lceil 1/\beta \rceil}}{(1-z)^2(1-\lambda_2 z^2)} + \frac{(z-\lambda_2 z^2)^2\, \overline{Q_\alpha}(z)}{(1-z)^2(1-\lambda_2 z^2)(1-\lambda_2^q\, z^{p+2q})},$$

$$P(z) = \frac{\lambda_2^{\lceil 1/\beta \rceil} z^{1+2\lceil 1/\beta \rceil}}{(1-z)^2} + \frac{(z^2 - \lambda_2 z^2)\, \overline{Q_\alpha}(z)}{(1-z)^2(1-\lambda_2^q\, z^{p+2q})},$$

with

$$\overline{Q_\alpha}(z) = \sum_{j=0}^{p-1} \lambda_2^{\lceil (j+2)/\beta \rceil} z^{j+2\lceil (j+2)/\beta \rceil}, \ \lambda_2 = \sum_{i=1}^{k} \pi_i^2, \ \textit{and}\ \lambda_3 = \sum_{i=1}^{k} \pi_i^3.$$

3.3 From generating series to asymptotics

Analytic combinatorics links asymptotic behavior of counting sequences to singularities of the corresponding generating functions, viewed as analytic functions of a complex variable. For rational generating series of one variable, as in Theorem 2, the situation is quite simple, and we use this simplified version of the Transfer Theorem [5] for rational functions:

▶ **Theorem 3** (Simplified Transfer Theorem [5]). *Assume $A(z) = F(z)(1-z)^{-\ell}$, where ℓ is a positive integer, $F(z)$ is a rational function with no pole in the closed disc of radius 1 and $F(1) \neq 0$. Then the n-th coefficient of $A(z)$ is asymptotically equivalent to $\frac{F(1)}{(\ell-1)!}\, n^{\ell-1}$.*

The series $R(z)$ and $P(z)$ of Theorem 2 both have a dominant pole of order 2 at $z = 1$. Applying Theorem 3 yields the following statement. Note that, though the generating series $R(z)$ and $P(z)$ are different, they lead to the same asymptotics for the coefficients; the difference is in lower order terms.

▶ **Theorem 4.** *Under the memoryless distribution of probability $\vec{\pi}$, and for any rational $\alpha = 1 + p/q$, the expected number of maximal α-gapped repeats (respectively, palindromes) in a random word of length n is asymptotically equivalent to $r_\alpha n$ (respectively, $p_\alpha n$) defined by*

$$r_\alpha = \frac{(1 - 2\lambda_2 + \lambda_3)\lambda_2^{\lceil q/p \rceil}}{1-\lambda_2} + \frac{(1-\lambda_2)}{1-\lambda_2^q} \sum_{j=2}^{p+1} \lambda_2^{\lceil jq/p \rceil} \ \textit{and}\ p_\alpha = \lambda_2^{\lceil q/p \rceil} + \frac{(1-\lambda_2)}{1-\lambda_2^q} \sum_{j=2}^{p+1} \lambda_2^{\lceil jq/p \rceil}.$$

In particular, when α is a positive integer, these reduce to

$$r_\alpha = (\alpha - 1)\lambda_2 + \frac{\lambda_2(\lambda_3 - \lambda_2^2)}{1-\lambda_2} \ \textit{and}\ p_\alpha = (\alpha-1)\lambda_2 + \lambda_2^2.$$

For the uniform distribution, we have $\lambda_2 = 1/k$ and $\lambda_3 = 1/k^2$, yielding the following result.

▶ **Corollary 5.** *For the uniform distribution on an alphabet of size $k \geq 2$, we have*

$$r_\alpha = \frac{k-1}{k}\left(k^{-\lceil q/p \rceil} + \frac{\sum_{j=2}^{p+1} k^{-\lceil jq/p \rceil}}{1-k^{-q}} \right) \ \textit{and}\ p_\alpha = r_\alpha + k^{-1-\lceil q/p \rceil}.$$

In particular, if α is a positive integer, then $r_\alpha = \frac{\alpha-1}{k}$ and $p_\alpha = \frac{\alpha-1}{k} + \frac{1}{k^2}$.

▶ Remark. As a function of $\alpha = 1 + \frac{p}{q}$, the value of r_α is not very regular. It is increasing, as expected, but there are some large variations when reaching a value with a small denominator (typically integers or half-integers). This also gives hints on the difficulty of giving a formula if α is not rational. Some examples are given in the table below, for $k = 4$.

α	5/4	3/2	7/4	2	9/4	5/2	11/4	3	13/4	7/2	15/4	4	17/4
r_α	0.002	0.05	0.061	0.25	0.252	0.3	0.311	0.5	0.502	0.55	0.561	0.75	0.752

4 Longest pattern

In this section we focus on the typical and expected length of the longest α-gapped patterns (repeat or palindromes) in a random word. Contrarily to the previous section, our analysis relies on discrete probabilities rather than on generating series.

Let L_n denote the random variable associated with the length of the longest α-gapped patterns in a random word of length n. We first focus on the uniform distribution, in order to introduce the main techniques of this section. For memoryless distributions, the computations are more involved, but the general idea remains the same.

If X_n is a random variable, we say that it is *concentrated around its mean* if there exists a sequence $(\nu_n)_{n\geq 1}$ that tends to 0 such that $\mathbb{P}(|X_n - \mathbb{E}[X_n]| > \nu_n \, \mathbb{E}[X_n]) \xrightarrow[n\to\infty]{} 0$.

In this whole section, whenever we say that some property holds *with asymptotic probability* 1, the property implicitly depends on some integer n, which denotes the length of the random words considered; and we mean that, as n goes to infinity, the probability tends to 1. The details in the text typically make it possible to derive a more explicit bound on the speed of convergence.

4.1 Uniform distribution

We establish the following theorem. Its proof is obtained by computing tight lower and upper bounds for the typical value of L_n, for the uniform distribution.

▶ **Theorem 6.** *For the uniform distribution on words of length n, on an alphabet of size k, the expected length of the longest α-gapped repeat (or palindrome) is asymptotically equivalent to $(\alpha + 1)\log_k n$. Moreover, the random variable L_n is concentrated around its mean.*

Observe that a longest α-gapped repeat is necessarily maximal. Moreover, the proof is exactly the same for palindromes, so we focus on repeats only.

To establish the lower bound, we prove that with asymptotic probability 1 there is an α-gapped repeat of length t_0 in a random word of length n, where t_0 is a well chosen value, which is asymptotically equivalent to $(\alpha + 1)\log_k n$. This property is proved to hold by just looking for α-gapped repeats lying at very specific positions: the word is split into roughly n/t_0 factors of length t_0, and we only compute the probability that at least one of these factors is an α-gapped repeat of a particular $|v|/|u|$ ratio. This fairly rough estimation is enough to establish a lower bound that is asymptotically tight.

For any $\ell \geq 1$, let $\mathcal{M}_\beta(\ell)$ denote the set of words of the form uvu, with $u \in A^\ell$ and $v \in A^{\lfloor \beta\ell \rfloor}$. The set $\mathcal{M}_\beta(\ell)$ therefore contains all the α-gapped repeats where u has size ℓ and v is of maximal length. Let $\ell_0 = \lfloor \log_k n - 2\log_k \log_k n \rfloor$. Every word of $\mathcal{M}_\beta(\ell_0)$ has length $t_0 = 2\ell_0 + \lfloor \beta\ell_0 \rfloor$, and t_0 is asymptotically equivalent to $(\alpha + 1)\log_k n$, as required.

The probability for an element of $\mathcal{M}_\beta(\ell_0)$ to be a factor of a random word of length n is exactly the probability that, for some $i \in [n]$, the factor of length t_0 starting at position i belongs to $\mathcal{M}_\beta(\ell_0)$. Thus, it is at least the probability that the factor of length t_0 starting at position $1 + jt_0$ is in $\mathcal{M}_\beta(\ell_0)$ for some $j \geq 1$ such that $(j+1)t_0 \leq n$. For such a given j, the probability that the factor starting at position $1 + jt_0$ is in $\mathcal{M}_\beta(\ell_0)$ is $k^{-\ell_0}$, since $|\mathcal{M}_\beta(\ell_0)| = k^{\ell_0 + \lfloor \beta \ell_0 \rfloor}$ and each possible factor has probability $k^{-\ell_0 - 2\lfloor \beta \ell_0 \rfloor}$. Since the integer intervals $[1 + jt_0, (j+1)t_0]$ do not overlap, the factors they define are independent, and the probability that none of them is in $\mathcal{M}_\beta(\ell_0)$ is $(1 - k^{-\ell_0})^{\lfloor n/t_0 \rfloor}$. Therefore, with probability at least $1 - (1 - k^{-\ell_0})^{\lfloor n/t_0 \rfloor}$, a random word of length n contains an α-gapped repeat of length t_0.

Straightforward computations yield that $(1 - k^{-\ell_0})^{\lfloor n/t_0 \rfloor} \leq \exp(-\log_k n)$, which tends to 0 as $n \to \infty$. Thus, with asymptotic probability 1, a random uniform word of length n contains an α-gapped repeat of length t_0, which is asymptotically equivalent to $(\alpha + 1) \log_k n$.

We now proceed with the upper bound. Let $\mathcal{R}_\beta(t)$ denote the set of all words uvu such that $|uvu| = t$ and $|v| \leq \beta |u|$. The set $\mathcal{R}_\beta(t)$ contains all the possible α-gapped repeats of length t. Observe that, by summing over all the possible lengths ℓ for u, we have

$$|\mathcal{R}_\beta(t)| = \sum_{\ell = \lceil t/(2+\beta) \rceil}^{\lfloor (t-1)/2 \rfloor} k^{t-\ell} \leq k^{t - \lceil t/(2+\beta) \rceil} \sum_{j=0}^{\infty} k^{-j} \leq 2k^{(\beta+1)t/(\beta+2)}.$$

Let $t_1 = \lceil (\beta + 2) \log_k n + 2(\beta + 2) \log_k \log_k n \rceil + 1$. The probability that a random word contains a factor in $\mathcal{R}_\beta(t_1)$ at a given valid position is $|\mathcal{R}_\beta(t_1)| k^{-t_1} \leq 2k^{-t_1/(\beta+2)}$. Since the number of valid positions is no more than n, by the union bound, the probability that a uniform random word of length n contains an element of $\mathcal{R}_\beta(t_1)$ (as a factor in any position) is at most $2nk^{-t_1/(\beta+2)}$. These computations also hold if one substitutes $t_1 + i$ for t_1. This yields that the probability that a uniform random word of length n contains an element of $\mathcal{R}_\beta(t_1 + i)$ is bounded from above by $\frac{2k^{-i/(\beta+2)}}{\log_k^2 n}$.

Using the union bound again, we sum these bounds for $i \geq 0$, and obtain that, with asymptotic probability 1, a uniform random word of length n contains no α-gapped repeat of length greater than or equal to t_1, which is asymptotically equivalent to $(\alpha + 1) \log_k(n)$.

A bit more is required to estimate the expectation of L_n, but this can be easily done from the computations above: they yield that the contribution to the expectation of the values that are not between t_0 and t_1 is negligible, and $t_0 \sim t_1 \sim (\alpha + 1) \log_k n$. The concentration around the mean can be proved by taking any sequence ν_n that tends to 0 and such that $\frac{\nu_n \log_k n}{\log_k \log_k n}$ tends to infinity.

4.2 Memoryless sources

In this section, we associate to each letter $a_i \in A = \{a_1, \ldots, a_k\}$ a probability $\pi_i = p(a_i)$ as described in Section 2. We assume all these probabilities to be positive (otherwise, reduce the alphabet size accordingly).

From the probability $\vec{\pi}$, we build another probability $\vec{\tau}$ proportional to the square of π: for every $i \in [k]$, $\tau_i = \pi_i^2 / \lambda_2$, where $\lambda_2 = \sum_{i \in [k]} \pi_i^2$ is the *coincidence probability* of $\vec{\pi}$ (as in Section 3). The result for memoryless sources, which generalizes Theorem 6, is the following.

▶ **Theorem 7.** *For the memoryless source of probability $\vec{\pi}$, the expected length of the longest α-gapped repeat (or palindrome) in a random word of length n is asymptotically $\mathbb{E}[L_n] \sim (\alpha + 1) \log_{1/\lambda_2} n$, where $\lambda_2 = \sum_{i \in [k]} \pi_i^2$. Moreover, L_n is concentrated around its mean.*

Though it follows the same main ideas as in the proof of Theorem 6, the proof of Theorem 7 is more technical. Due to lack of space, we only focus on the main steps in this extended abstract. We will focus on the most probable words, and the most probable longest

α-gapped repeat. For this purpose, for a probability vector $\vec{s}$ on A and $\delta \geq 0$, we consider the set $\mathcal{W}_n(\vec{s}, \delta)$ of words whose letters roughly follow the distribution of $\vec{s}$, defined by

$$\mathcal{W}_n(\vec{s}, \delta) = \left\{ u \in A^n : \left| |u|_a - s(a)\, n \right| \leq \delta, \ \forall a \in A \right\}.$$

To establish the lower bound, we define the set $\mathcal{M}_\beta(\vec{\pi}, \ell)$ of α-gapped repeats uvu where the letters are distributed following $\vec{\pi}$ in v and following $\vec{\tau}$ in u. More formally:

$$\mathcal{M}_\beta(\vec{\pi}, \ell) = \left\{ uvu \in A^* : \ u \in \mathcal{W}_\ell(\vec{\tau}, \sqrt{\log n}) \text{ and } v \in \mathcal{W}_{\lfloor \beta \ell \rfloor}(\vec{\pi}, \sqrt{\log n}) \right\}.$$

The set $\mathcal{M}_\beta(\vec{\pi}, \ell)$ will play the same role as the set $\mathcal{M}_\beta(\ell)$ of the previous section. They do not coincide if the distribution is uniform, but they still have the same order of size.

We now proceed with the lower bound. We define ℓ_0 and t_0 by

$$\ell_0 = \left\lfloor \frac{\log n}{\log(1/\lambda_2)} - \frac{(\log n)^{2/3}}{\log(1/\lambda_2)} \right\rfloor \text{ and } t_0 = 2\ell_0 + \lfloor \beta \ell_0 \rfloor,$$

then prove that long random words have a factor in $\mathcal{M}_\beta(\vec{\pi}, \ell_0)$ with high probability.

For this purpose, we need to estimate the probability that a factor of length t_0 at a given position is in $\mathcal{M}_\beta(\vec{\pi}, \ell_0)$. The computations are done as follows. Let $\vec{n} = (n_1, \ldots, n_k)$ with $n_1 + \ldots + n_k = \ell_0$ and let $\vec{m} = (m_1, \ldots, m_k)$ with $m_1 + \ldots + m_k = \lfloor \beta \ell_0 \rfloor$. We are interested in the set of words $\mathcal{E}(\vec{n}, \vec{m})$, with fixed compositions for u and v, defined by

$$\mathcal{E}(\vec{n}, \vec{m}) = \{ uvu : \ |u|_{a_i} = n_i \text{ and } |v|_{a_i} = m_i, \ \forall i \in [k] \}.$$

Observe that $\mathcal{M}_\beta(\ell_0)$ can be written as a union of $\mathcal{E}(\vec{n}, \vec{m})$ for properly chosen ranges for $\vec{n}$ and $\vec{m}$. The probability that the factor of length t_0 at a given valid position lies in $\mathcal{E}(\vec{n}, \vec{m})$ is

$$\mathbb{P}_{t_0}(\mathcal{E}(\vec{n}, \vec{m})) = \binom{\ell_0}{n_1, \ldots, n_k} \prod_{i \in [k]} \pi_i^{2n_i} \binom{\lfloor \beta \ell_0 \rfloor}{m_1, \ldots, m_k} \prod_{i \in [k]} \pi_i^{m_i}.$$

By estimating this quantity and summing for all $\vec{n}$ and $\vec{m}$ such that $\mathcal{E}(\vec{n}, \vec{m}) \subseteq \mathcal{M}_\beta(\ell_0)$, we obtain that the probability of the factor of length t_0 at a given valid position not being in $\mathcal{M}_\beta(\vec{\pi}, \ell_0)$ is $O(\frac{1}{\log^2 n})$. At this point, the proof continues exactly as in Section 4.1.

We now turn to the upper bound. As in Section 4.1 let $\mathcal{R}_\beta(t)$ be the set of all uvu such that $|uvu| = t$ and $1 \leq |v| \leq \beta |u|$. We want to compute an upper bound for the probability that the factor of length t at a given valid position lies in $\mathcal{R}_\beta(t)$.

We need to partition the set $\mathcal{R}_\beta(t)$ for our computations. Let $\vec{\ell} = (\ell_1, \ldots, \ell_k)$ be a vector of non-negative integers such that $N_1(\vec{\ell}) = \ell_1 + \cdots + \ell_k = \ell$. Let $\mathcal{R}_\beta(\vec{\ell}, t)$ be the set of all words uvu such that $|uvu| = t$ and $|u|_{a_i} = \ell_i$ for every $i \in [k]$. Observe that $\mathcal{R}_\beta(t)$ can be written as the following disjoint union:

$$\mathcal{R}_\beta(t) = \bigcup_{\ell = \lceil t/(\beta+2) \rceil}^{\lfloor (t-1)/2 \rfloor} \ \bigcup_{N_1(\vec{\ell}) = \ell} \mathcal{R}(\vec{\ell}, t).$$

Moreover, $\mathbb{P}_t(\mathcal{R}(\vec{\ell}, t)) = \binom{\ell}{\ell_1, \ldots, \ell_k} \prod_{i \in [k]} \pi_i^{2\ell_i}$. But $\sum_{N_1(\vec{\ell}) = \ell} \binom{\ell}{\ell_1, \ldots, \ell_k} \prod_{i \in [k]} \tau_i^{\ell_i} = 1$, as it is the sum of the probabilities of all the words of length ℓ for the memoryless distribution of probability vector $\vec{\tau}$. Hence, $\mathbb{P}_\ell(\cup_{N_1(\vec{\ell}) = \ell} \mathcal{R}(\vec{\ell}, t)) = \lambda_2^\ell$. Therefore, $\mathbb{P}_t(\mathcal{R}_\beta(t)) \leq t \, \lambda_2^{t/(\beta+2)}$. In particular, for $t_1 = \lceil (\beta + 2) \log_{1/\lambda_2} n + 3(\beta + 2) \log_{1/\lambda_2} \log n \rceil$, we have

$$\mathbb{P}(\mathcal{R}_\beta(t_1 + i)) \leq \frac{2 \, \lambda_2^{i/(\beta+2)}}{\log^2 n},$$

and the proof continues as in the uniform case.

▶ Remark. As a byproduct of the proof, we obtain the following interesting result on the probabilistic nature of the longest α-gapped repeat. Though a sufficiently large random word in the memoryless model contains roughly a proportion π_i of each letter a_i, the letters are distributed differently in the arms (the u's of uvu) of a typical longest α-gapped repeat: the proportion of each letter is roughly τ_i instead of π_i. This phenomenon is completely hidden in the uniform case, where $\tau_i = \pi_i = 1/k$ for every $i \in [k]$.

5 A remark on the number of distinct factors

In [15], Rubinchik and Shur estimated the expected number of distinct palindromes in a random word: these factors are counted only once, even if they have multiple occurrences. They established that for the uniform distribution, a random word contains $\Theta(\sqrt{n})$ distinct palindromes, and several refinements of this result.

In this short section we explain how their proof can be extended to estimate the expected number of distinct α-gapped repeats and palindromes, for the uniform distribution. There is no new idea, one just has to take care of the possibilities for the additional v part in the pattern. The result, however, is interesting on its own. It is stated as followed.

▶ **Theorem 8.** *For the uniform distribution over words of length n, the expected number of distinct α-gapped repeats (or palindromes) is in $\Theta(n^{\alpha/(\alpha+1)})$.*

We only consider repeats in our proof sketch; gapped palindromes are treated the same way. The lower bound is obtained using Guibas and Odlyzko's result on pattern avoidance [9]: the number of words of length n that avoid a given pattern w of length $m > 3$ is equal to $C_w\theta_w^n + O(1.7^n)$. In [15], the authors prove that the constants are maximal when $w = a^m$:

$$\theta_w \le \theta_{a^m} = k\left(1 - \frac{k-1}{k^{m+1}} + O\left(\frac{m}{k^{2m+2}}\right)\right) \text{ and } C_w \le C_{a^m} = 1 + O\left(\frac{m}{k^m}\right). \tag{1}$$

Let $\S_\beta(n)$ be the set of all uvu such that $|u| = \ell = \lfloor \frac{1}{\beta+2}\log_k n\rfloor$ and $|v| = \lfloor \beta\ell \rfloor$. Let $m = 2\ell + \lfloor \beta\ell\rfloor$. As a direct application of Equation (1), for a given $w \in \S_\beta(n)$, the probability that a random word of length n avoids w satisfies

$$\mathbb{P}_n(\text{avoiding } w) \le \left(1 - \frac{k-1}{k^{m+1}} + O\left(\frac{m}{k^{2m+2}}\right)\right)^n \left(1 + O\left(\frac{m}{k^m}\right)\right),$$

which is at most C for some positive constant $C < 1$ and n sufficiently large (for our choice of ℓ, and thus of m). Hence, the probability for w to be a factor in a random word of length n is at least $1 - C$, and by linearity of the expectation, the expected number of distinct α-gapped repeats is greater than or equal to $(1-C)|\S_\beta(n)| = (1-C)k^{m-\ell} = \Omega(n^{\alpha/(\alpha+1)})$.

For the upper bound, let $\mathcal{R}_\beta(t)$ denote all the uvu such that $|uvu| = t$ and $|v| \le \beta|u|$. Let also $t_0 = \lceil \log_k n\rceil$. We count differently the α-gapped repeats, depending on whether they are shorter or longer than t_0.

We count all the α-gapped repeats of length at most t_0 as contributing to the upper bound. By summing over all possible values for the length ℓ of the arms, we have

$$|\mathcal{R}_\beta(t)| = \sum_{\ell=\lceil t/(\beta+1)\rceil}^{\lfloor (t-1)/2\rfloor} k^{t-\ell} \le k^t \sum_{\ell=\lceil t/(\beta+1)\rceil}^{\infty} k^{t-\ell} \le 2k^{(\beta+1)t/(\beta+2)}.$$

Hence,

$$\sum_{t=1}^{t_0} |\mathcal{R}_\beta(t)| \le \sum_{t=1}^{t_0} 2k^{(\beta+1)t/(\beta+2)} \le 4k^{(\beta+1)t_0/(\beta+2)} \le 4n^{\alpha/(\alpha+1)}.$$

To obtain an upper bound for the expected number of patterns of length greater than t_0, we observe as in [15] that the probability for a given α-gapped repeat uvu to be a factor is at most the expected number of occurrences of uvu. As we shall see, this rough upper bound is enough to conclude. Let $\mathcal{R}_\beta(t,\ell)$ be the set of uvu such that $|uvu| = t$, $|u| = \ell$ and $|v| \leq \beta\ell$. The probability that there is pattern of $\mathcal{R}_\beta(t,\ell)$ at a given valid position in a random word is $k^{-\ell}$. Hence, the expected number of occurrences of such patterns is at most $nk^{-\ell}$, for given t and ℓ. By summing over all $t > t_0$ and all valid ℓ for each t we obtain the following upper bound:

$$\sum_{t=t_0}^{n} \sum_{\ell=\lceil t/(\beta+2)\rceil}^{\lfloor t/2\rfloor} n\,k^{-\ell} \leq \sum_{t=t_0}^{n} 2k^{-t/(\beta+2)} \leq 4k^{-t_0/(\beta+2)} \leq 4n^{\alpha/(\alpha+1)}.$$

Combining both results, for short and long α-gapped repeats, we get that the expected number of distinct such factors is bounded from above by $8n^{\alpha/(\alpha+1)}$, concluding the proof.

▶ Remark. Theorem 8 also holds for *maximal* patterns. The proof simply needs to be adapted for the lower bound, and there are sufficiently many of them to obtain the same result.

6 Conclusions

In this article we establish results about some statistics of random words related to the notion of α-gapped patterns, for both the uniform and memoryless distributions. We propose different techniques, generating series and discrete probabilities, to provide some tools for further analysis of statistics of interest. Amongst them, if would be natural to consider gapped patterns as a whole, *i.e.* if $uvu = u'v'u'$ then it is considered as one pattern instead of two different ones.

The biased distribution on letters in the arms of a typical α-gapped pattern, in the memoryless model, is something worth noticing (see Section 4.2). It may provide some leverage for speeding up algorithms, though the difference might be too thin to be exploited.

Finally, generalizing Theorem 8 to memoryless proves quite difficult. This is ongoing work, and the techniques involved are more advanced than what is presented in this article.

References

1 Gerth Stølting Brodal, Rune B. Lyngsø, Christian N. S. Pedersen, and Jens Stoye. Finding maximal pairs with bounded gap. In Maxime Crochemore and Mike Paterson, editors, *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM 1999)*, volume 1645 of *LNCS*, pages 134–149. Springer, 1999. doi:10.1007/3-540-48452-3_11.

2 Maxime Crochemore, Roman Kolpakov, and Gregory Kucherov. Optimal bounds for computing α-gapped repeats. In Adrian-Horia Dediu, Jan Janousek, Carlos Martín-Vide, and Bianca Truthe, editors, *Proceedings of the 10th International Conference on Language and Automata Theory and Applications (LATA 2016)*, volume 9618 of *LNCS*, pages 245–255. Springer, 2016. doi:10.1007/978-3-319-30000-9_19.

3 Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.

4 Marius Dumitran and Florin Manea. Longest gapped repeats and palindromes. In Giuseppe F. Italiano, Giovanni Pighizzini, and Donald Sannella, editors, *Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science (MFCS 2015)*, volume 9234 of *LNCS*, pages 205–217. Springer, 2015. doi:10.1007/978-3-662-48057-1_16.

5 Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009. doi:10.1017/CBO9780511801655.

6 Paweł Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Efficiently finding all maximal α-gapped repeats. In Nicolas Ollinger and Heribert Vollmer, editors, *Proceedings of the 33rd Symposium on Theoretical Aspects of Computer Science (STACS 2016)*, volume 47 of *LIPIcs*, pages 39:1–39:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.STACS.2016.39.

7 Paweł Gawrychowski and Florin Manea. Longest α-gapped repeat and palindrome. In Adrian Kosowski and Igor Walukiewicz, editors, *Proceedings of the 20th International Symposium on Fundamentals of Computation Theory (FCT 2015)*, volume 9210 of *LNCS*, pages 27–40. Springer, 2015. doi:10.1007/978-3-319-22177-9_3.

8 Amy Glen and Jamie Simpson. The total run length of a word. *Theor. Comput. Sci.*, 501:41–48, 2013. doi:10.1016/j.tcs.2013.06.004.

9 Leonidas J. Guibas and Andrew M. Odlyzko. String overlaps, pattern matching, and nontransitive games. *J. Comb. Theory, Ser. A*, 30(2):183–208, 1981. doi:10.1016/0097-3165(81)90005-4.

10 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theor. Comput. Sci.*, 410(51):5365–5373, 2009. doi:10.1016/j.tcs.2009.09.013.

11 Roman Kolpakov, Mikhail Podolskiy, Mikhail Posypkin, and Nickolay Khrapov. Searching of gapped repeats and subrepetitions in a word. In Alexander S. Kulikov, Sergei O. Kuznetsov, and Pavel A. Pevzner, editors, *Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM 2014)*, volume 8486 of *LNCS*, pages 212–221. Springer, 2014. doi:10.1007/978-3-319-07566-2_22.

12 Roman M. Kolpakov and Gregory Kucherov. Finding repeats with fixed gap. In Pablo de la Fuente, editor, *Proceedings of the 7th International Symposium on String Processing and Information Retrieval (SPIRE 2000)*, pages 162–168. IEEE Computer Society, 2000. doi:10.1109/SPIRE.2000.878192.

13 Cyril Nicaud. Estimating statistics on words using ambiguous descriptions. In Roberto Grossi and Moshe Lewenstein, editors, *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *LIPIcs*, pages 9:1–9:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.9.

14 Simon J. Puglisi and Jamie Simpson. The expected number of runs in a word. *Australas. J. Comb.*, 42:45–54, 2008. URL: https://ajc.maths.uq.edu.au/pdf/42/ajc_v42_p045.pdf.

15 Mikhail Rubinchik and Arseny M. Shur. The number of distinct subpalindromes in random words. *Fundam. Inform.*, 145(3):371–384, 2016. doi:10.3233/FI-2016-1366.

16 Arseny M. Shur. Growth properties of power-free languages. *Comput. Sci. Rev.*, 6(5-6):187–208, 2012. doi:10.1016/j.cosrev.2012.09.001.

17 Yuka Tanimura, Yuta Fujishige, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. A faster algorithm for computing maximal α-gapped repeats in a string. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *Proceedings of the 22nd International Symposium on String Processing and Information Retrieval (SPIRE 2015)*, volume 9309 of *LNCS*, pages 124–136. Springer, 2015. doi:10.1007/978-3-319-23826-5_13.

Computing All Distinct Squares in Linear Time for Integer Alphabets[*]

Hideo Bannai[1], Shunsuke Inenaga[2], and Dominik Köppl[3]

1 **Department of Informatics, Kyushu University, Fukuoka, Japan**
`bannai@inf.kyushu-u.ac.jp`

2 **Department of Informatics, Kyushu University, Fukuoka, Japan**
`inenaga@inf.kyushu-u.ac.jp`

3 **Department of Computer Science, TU Dortmund, Dortmund, Germany**
`dominik.koeppl@tu-dortmund.de`

Abstract

Given a string on an integer alphabet, we present an algorithm that computes the set of all distinct squares belonging to this string in time linear in the string length. As an application, we show how to compute the tree topology of the minimal augmented suffix tree in linear time. Asides from that, we elaborate an algorithm computing the longest previous table in a succinct representation using compressed working space.

1998 ACM Subject Classification G.2.1 Combinatorial Algorithms

Keywords and phrases tandem repeats, distinct squares, counting algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.22

1 Introduction

A square is a string of the form SS, where S is some non-empty string. It is well-known that a string of length n contains at most $n^2/4$ squares. This bound is the number of *all* squares, i.e., we count multiple occurrences of the same square, too. If we consider the number of all *distinct* squares, i.e., we count *exactly one* occurrence of each square, then it becomes linear in n: The first linear upper bound was given by Fraenkel and Simpson [17] who proved that a string of length n contains at most $2n$ distinct squares. Later, Ilie [26] showed the slightly improved bound of $2n - \Theta(\lg n)$. Recently, Deza et al. [10] refined this bound to $\lfloor 11n/6 \rfloor$. In the light of these results one may wonder whether future results will "converge" to the upper bound of n: The *distinct square conjecture* [17, 27] is that a string of length n contains at most n distinct squares; this number is known to be independent of the alphabet size [37]. However, there still is a big gap between the best known bound and the conjecture. While studying a combinatorial problem like this, it is natural to think about ways to actually compute the exact number.

This article focuses on a computational problem on distinct squares, namely, we wish to compute (a compact representation of) the set of all distinct squares in a given string. Gusfield and Stoye [23] tackled this problem with an algorithm running in $\mathcal{O}(n\sigma_T)$ time, where σ_T denotes the number of different characters contained in the input text T of length n.

[*] This work was mainly done while Dominik Köppl visited Kyushu University in Japan under the support by the JSPS Summer Program SP16305. Hideo Bannai and Shunsuke Inenaga were supported in part by JSPS KAKENHI Grant Numbers JP26280003, JP16H02783, JP17H01697.

Although its running time is optimal $\mathcal{O}(n)$ for a constant alphabet, it becomes $\mathcal{O}(n^2)$ for a large alphabet since σ_T can be as large as $\mathcal{O}(n)$.

We present an algorithm (Section 4.1) that computes this set in $\mathcal{O}(n)$ time for a given string of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$. Like Gusfield and Stoye, we can use the computed set to decorate the suffix tree with all squares (Section 5.1). As an application, we provide an algorithm that computes the tree topology of the minimal augmented suffix tree [1] in linear time (Section 5.2). The fastest known algorithm computing this tree topology takes $\mathcal{O}(n \lg n)$ time [5].

For our approach, we additionally need the longest previous factor table [18, 8]. As a side result of independent interest, we show in Section 3 how to store this table in $2n + o(n)$ bits, and give an algorithm that computes it using compressed working space.

2 Definitions

Our computational model is the word RAM model with word size $\Omega(\lg n)$ for some natural number n. Let Σ denote an integer alphabet of size $\sigma = |\Sigma| = n^{\mathcal{O}(1)}$. An element w in Σ^* is called a ***string***, and $|w|$ denotes its length. We denote the i-th character of w with $w[i]$, for $1 \leq i \leq |w|$. When w is represented by the concatenation of $x, y, z \in \Sigma^*$, i.e., $w = xyz$, then x, y and z are called a ***prefix***, ***substring*** and ***suffix*** of w, respectively. For i, j with $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j in w.

The ***longest common prefix (LCP)*** of two strings is the longest prefix shared by both strings. The ***longest common extension (LCE)*** query asks for the longest common prefix of two suffixes of the *same* string. The time for an LCE query is denoted by t_{LCE}.

A ***factorization*** of a string T is a sequence of non-empty substrings of T such that the concatenations of the substrings is T. Each substring in the factorization is called a ***factor***.

In the rest of this paper, we take a string T of length $n > 0$, and call it ***the text***. We assume that $T[n] = \$$ is a special character that appears nowhere else in T, so that no suffix of T is a prefix of another suffix of T. We further assume that T is read-only; accessing a character costs constant time. We sometimes need the ***reverse*** of T, which is given by the concatenation $T[n-1] \cdots T[1] \cdot T[n] = T[n-1] \cdots T[1]\$$.

The ***suffix tree*** of T is the tree obtained by compacting the trie of all suffixes of T; it has n leaves and at most $n - 1$ internal nodes. The leaf corresponding to the i-th suffix $T[i..n]$ is labeled with i. Each edge e is associated with a non-empty substring x of T called the ***edge label*** of e. Each edge label x is represented by tuple (i, ℓ) of integers such that $T[i..i + \ell - 1] = x$. This way the suffix tree of T takes $\mathcal{O}(n)$ *words* of space, and it can be computed in $\mathcal{O}(n)$ time for strings of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$ [11]. The ***string label*** of a node v is defined as the concatenation of all edge labels on the path from the root to v; the ***string depth*** of a node is the length of its string label.

SA and ISA denote the suffix array and the inverse suffix array of T, respectively [36]. The access time to an element of SA is denoted by t_{SA}. LCP is an array such that $\mathsf{LCP}[i]$ is the length of the longest common prefix of $T[\mathsf{SA}[i]..n]$ and $T[\mathsf{SA}[i-1]..n]$ for $i = 2, \ldots, n$. For our convenience, we define $\mathsf{LCP}[1] := 0$. The arrays SA, ISA, and LCP can be constructed in $\mathcal{O}(n)$ time [30, 32, 31].

A ***range minimum query (RMQ)*** asks for the smallest value in a sub-array of an integer array. There are data structures that can answer RMQs on an integer array of length n in constant time while taking $2n + o(n)$ bits of space [15]. An LCE query for the suffixes $T[s..n]$ and $T[t..n]$ can be answered with an RMQ data structure on LCP with the range $[\min(\mathsf{ISA}[s], \mathsf{ISA}[t]) + 1.. \max(\mathsf{ISA}[s], \mathsf{ISA}[t])]$ in constant time.

A ***bit vector*** is a string on the binary alphabet $\{0,1\}$. A ***select query*** on a bit vector asks the position of the i-th '0' or '1' in the bit vector. There is a data structure that can be built in $\mathcal{O}(n)$ time with $\mathcal{O}(n)$ bits of working space such that it takes $o(n)$ bits on top of the bit vector, and can answer a select query in constant time [6].

We identify occurrences of substrings with their position and length in the text, i.e., if x is a substring of T, then there is an i with $1 \le i \le n$ and an ℓ with $0 \le \ell \le n-i+1$ such that $T[i..i+\ell-1] = x$. In the following, we will represent the occurrences of substrings by tuples of position and length. When storing these tuples in a set, we call the set ***distinct***, if there are no two tuples (i,ℓ) and (i',ℓ) such that $T[i..i+\ell-1] = T[i'..i'+\ell-1]$. A special kind of substring is a square: A ***square*** is a string of the form SS for $S \in \Sigma^+$; we call S and $|S|$ the ***root*** and the ***period*** of the square SS, respectively. Like with substrings, we can generate a set containing some occurrences of squares. A set of ***all distinct squares*** is a distinct set of occurrences of squares that is maximal under inclusion.

3 A Compact Representation of the LPF Array

The longest previous factor table LPF of T is formally defined as

$$\mathsf{LPF}[j] := \max\{\ell \mid \text{there exists an } i \in [1..j-1] \text{ such that } T[i..i+\ell-1] = T[j..j+\ell-1]\}\,.$$

It is useful for computing the ***Lempel-Ziv factorization*** of $T = f_1 \cdots f_z$, which is defined as $f_i = T[k..k+\max(1,\mathsf{LPF}[k])]$ with $k := \sum_{j=1}^{i-1}|f_j|+1$ for $1 \le i \le z$.

In the following, we will use the text $T = \overset{0}{\mathtt{a}}\overset{0}{\mathtt{b}}\overset{3}{\mathtt{a}}\overset{2}{\mathtt{b}}\overset{1}{\mathtt{a}}\overset{2}{\mathtt{a}}\overset{5}{\mathtt{a}}\overset{4}{\mathtt{b}}\overset{3}{\mathtt{a}}\overset{2}{\mathtt{b}}\overset{1}{\mathtt{a}}\overset{0}{\mathtt{\$}}$ as our running example whose LPF array is represented by the small numbers above the characters. The Lempel-Ziv factorization of T is given by $\overset{1}{\mathtt{a}}|\overset{2}{\mathtt{b}}|\overset{3}{\mathtt{a}}\mathtt{ba}|\overset{4}{\mathtt{a}}\mathtt{a}|\overset{5}{\mathtt{b}}\mathtt{aba}|\overset{6}{\mathtt{\$}}$, where the small numbers denote the factor indices, and the vertical bars denote the factor borders.

▶ **Corollary 1.** *Given* LPF*, we can compute the Lempel-Ziv factorization in $\mathcal{O}(n)$ time. If the factorization consists of z factors, the factorization can be represented by an array of $z \lg n$ bits, where the x-th entry stores the beginning of the x-th factor. Alternatively, it can be represented by a bit vector of length n in which we mark the factor beginnings. A select data structure on top of the bit vector can return the length and the position of a factor in constant time.*

Since we will need LPF in Section 4, we are interested in the time and space bounds for computing LPF. We start with the (to the best of our knowledge) state of the art algorithm with respect to time and space requirements.

▶ **Lemma 2** ([9, Theorem 1])**.** *Given* SA *and* LCP*, we can compute* LPF *in $\mathcal{O}(nt_{\mathsf{SA}})$ time. Besides the output space of $n\lg n$ bits, we only need constant working space.*

Apart from this algorithm, we are only aware of some practical improvements [40, 28].

Let us consider the size of LCP needed in Lemma 2. Sadakane [41] showed a $2n+o(n)$-bits representation of LCP. Thereto he stores the ***permuted longest-common-prefix array*** PLCP defined as $\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i]$ in a bit vector in the following way (also described in [13]): Since $\mathsf{PLCP}[1]+1, \mathsf{PLCP}[2]+2, \ldots, \mathsf{PLCP}[n]+n$ is a non-decreasing sequence with $1 \le \mathsf{PLCP}[1]+1 \le \mathsf{PLCP}[n]+n = n$ ($\mathsf{PLCP}[i] \le n-i$ since the terminal $\$$ is a unique character in T) the values $I[1] := \mathsf{PLCP}[1]$ and $I[i] := \mathsf{PLCP}[i] - \mathsf{PLCP}[i-1]+1$ $(2 \le i \le n)$ are non-negative. By writing $I[i]$ in the unary code $\mathtt{0}^{I[i]}\mathtt{1}$ to a bit vector S subsequently for each $2 \le i \le n$, we can compute $\mathsf{PLCP}[i] = \mathrm{select}_1(S,i) - 2i$ and $\mathsf{LCP}[i] = \mathrm{select}_1(S,\mathsf{SA}[i]) - 2\mathsf{SA}[i]$. Moreover, $\sum_{i=1}^{n} I[i] \le n$ and therefore S is of length at most $2n$.

Table 1 Algorithms computing LPF; space is counted in bits. The output space $|\mathsf{LPF}|$ is not considered as working space. $0 < \epsilon \leq 1$ is a constant.

algorithm	time	working space	$\|\mathsf{LPF}\|$
Lemma 2,[9]	$\mathcal{O}(nt_{\mathsf{SA}})$	$\|\mathsf{SA}\| + \|\mathsf{LCP}\| + \mathcal{O}(\lg n)$	$n \lg n$
Corollary 3,[35, 24]	$\mathcal{O}(n)$	$n \lg n + 2n + \mathcal{O}(\lg n)$	$n \lg n$
Lemma 6,[34]	$\mathcal{O}(n/\epsilon)$	$(1+\epsilon)n \lg n + \mathcal{O}(n)$	$2n + o(n)$
Lemma 6,[16]	$\mathcal{O}(nt_{\mathsf{SA}})$	$\mathcal{O}(n \lg \sigma)$	$2n + o(n)$

By using Sadakane's LCP-representation, we get LPF with the algorithm of Crochemore et al. [9] in the following time and space bounds:

▶ **Corollary 3.** *Having* SA *and* LCP *stored in $n \lg n$ bits (this allows $t_{\mathsf{SA}} = \mathcal{O}(1)$) and $2n+o(n)$ bits, respectively, we can compute* LPF *with $\mathcal{O}(\lg n)$ additional bits of working space (not counting the space for* LPF*) in $\mathcal{O}(n)$ time.*

By plugging in a suffix array construction algorithm like the in-place construction algorithm by Goto [21], we get the bounds shown in Table 1.

Although this result seems compelling, this approach stores SA and LPF in plain arrays (the former for getting constant time access). In the following, we will show that the LPF array can be stored more compactly. We start with a new representation of LPF, for which we use the same trick as for PLCP due to the following property (which is crucial for squeezing PLCP into $2n + o(n)$ bits).

▶ **Lemma 4.** $n - j \geq \mathsf{LPF}[j] \geq \mathsf{LPF}[j-1] - 1$ *for* $2 \leq j \leq n$.

Proof. There is an i with $1 \leq i < j-1$ such that $T[i..i + \mathsf{LPF}[j-1] - 1] = T[j-1..j-1+\mathsf{LPF}[j-1]-1]$. Hence $T[i+1..i+\mathsf{LPF}[j-1]-1] = T[j..j-1+\mathsf{LPF}[j-1]-1]$. ◀

We conclude that the sequence $\mathsf{LPF}[1]+1, \mathsf{LPF}[2]+2, \ldots, \mathsf{LPF}[n]+n$ is non-decreasing with $1 \leq \mathsf{LPF}[1]+1 \leq \mathsf{LPF}[n]+n \leq n$. We immediately get:

▶ **Corollary 5.** LPF *can be represented by a bit vector with a select data structure such that accessing an* LPF *value can be performed in constant time. The data structures use $2n+o(n)$ bits.*

To get a better working space bound, we have to come up with a new algorithm since the algorithm of Lemma 2 creates a plain array to get constant time random write-access for computing the entries of LPF. To this end, we present two algorithms that compute LPF in this representation with the aid of the suffix tree. The two algorithms are derivatives of the algorithms [34, 16] that compute the Lempel-Ziv factorization, either in $\mathcal{O}(n \lg \lg \sigma)$ time using $\mathcal{O}(n \lg \sigma)$ bits, or in $\mathcal{O}(n/\epsilon^2)$ time using $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits, for a constant $0 < \epsilon \leq 1$. The current bottleneck of both algorithms is the suffix tree implementation with respect to space and time. Due to current achievements [39, 35], the algorithms now run in $\mathcal{O}(n)$ time using $\mathcal{O}(n \lg \sigma)$ bits, or in $\mathcal{O}(n/\epsilon)$ time using $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits, respectively.

We aim at building the LPF-representation of Corollary 5 directly such that we do not need to allocate the plain LPF array using $n \lg n$ bits in the first place. To this end we create a bit vector of length $2n$ and store the LPF values in it successively. In more detail, we follow the description of the Lempel-Ziv factorization algorithms presented in [34, 16]. There, the algorithms are divided into several passes. In each pass we successively visit leaves in text order (determined by the labels of the leaves). To compute LPF, we only have to do a single pass. Similarly to the first passes of the two Lempel-Ziv algorithms, we use a bit vector B_{V} to mark already visited internal nodes. On visiting a leaf we climb up the tree until reaching the root or an already marked node. In the former case (we climbed up to the root) we output zero. In the latter case, we output the string depth of the marked node. By doing so, we have computed $\mathsf{LPF}[1..j]$ after having processed the leaf with label j.

▶ **Lemma 6.** *We can compute* LPF *in $\mathcal{O}(nt_{\mathsf{SA}})$ time with $\mathcal{O}(n \lg \sigma)$ bits of working space, or in $\mathcal{O}(n/\epsilon)$ time using $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits of working space, for a constant $0 < \epsilon \leq 1$. Both variants include the space of the output in their working spaces.*

Proof. Computing the string depth of a node needs access to an RMQ data structure of LCP, and an access to SA. Both accesses can be emulated by the compressed suffix array in t_{SA} time, given that we have computed PLCP in the above representation. ◀

4 The Set of All Distinct Squares

Given a string T, our goal is to compute all distinct squares of T. Thereto we return a set of pairs, where each pair (s, ℓ) consists of a starting position s and a length ℓ such that $T[s..s+\ell-1]$ is the leftmost occurrence of a square. The size of this set is linear due to

▶ **Lemma 7** (Fraenkel and Simpson [17])**.** *A string of length n can contain at most $2n$ distinct squares.*

We follow the approach of Gusfield and Stoye [23]. Their idea is to compute a set of squares (the set stores pairs of position and length like described in Section 2)[1] with which they can generate all distinct squares. They call this set of squares a ***leftmost covering set***. A leftmost covering set obeys the property that every square of the text can be constructed by right-rotating a square of this set. A square (k, ℓ) is constructed by ***right-rotating*** a square (i, ℓ) with $i \leq k$ iff each tuple $(i+j, \ell)$ with $1 \leq j \leq k-i$ represents a square $T[i+j..i+\ell+j-1] = T[i+j..i+\ell-1]T[i..i+j-1]$.

The set of the leftmost occurrences of all squares is a set of all distinct squares. Unfortunately, the leftmost covering set computed in [23] is not necessarily a set of all distinct squares since (a) it does not have to be distinct, and (b) a square might be missing that can be constructed by right-rotating a square of the computed leftmost covering set.

For illustration, the squares of our running example $T = \texttt{ababaaababa\$}$ are highlighted with bars. The set of all squares is $\{(1,4),(2,4),(5,2),(6,2),(7,4),(8,4)\}$. If we take the leftmost occurrences of all squares, we get $\{(1,4),(2,4),(5,2)\}$; this set comprises all squares marked by the solid bars, i.e., the dotted bars correspond to occurrences of squares that are not leftmost. In this example, the dotted bars form the set $\{(6,2),(7,4),(8,4)\}$, which is a set of all distinct squares. A leftmost covering set is $\{(1,4),(5,2)\}$.

Our goal is to compute the set of all leftmost occurrences directly by modifying the algorithm of [23]. To this end, we briefly review how their approach works: They compute their leftmost covering set by examining the borders between all Lempel-Ziv factors $f_1 \cdots f_z = T$. That is because of

[1] It differs to the set we want to compute by the fact that they allow, among others, occurrences of the same square in their set.

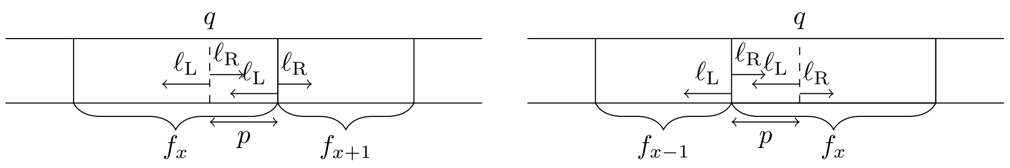


Figure 1 Search for squares on Lempel-Ziv borders. The left image corresponds to squares of type Lemma 8(1), the right image to the type Lemma 8(2). Given two adjacent factors, we determine a position q that is p positions away from the border (the direction is determined by the type of square we want to search for). By two LCE queries we can determine the lengths ℓ_{L} and ℓ_{R} that indicate the presence of a square if $\ell_{\mathrm{L}} + \ell_{\mathrm{R}} \geq p$.

▶ **Lemma 8** ([23, Theorem 5]). *The leftmost occurrence of a square $T[i..i+2p-1]$ touches at least two Lempel-Ziv factors. Let f_x $(1 \leq x \leq z)$ be the factor that contains the center of the square $i+p-1$. Then either*

(a) *the square has its left end (position i) inside f_x and its right end (position $i+2p-1$) inside f_{x+1}, or*

(b) *the left end of the square extends into f_{x-1} (or even further left). The right end can be contained inside f_x or f_{x+1}.*

Having a data structure for computing LCE queries on the text and on its inverse, they can probe at the borders of two consecutive factors whether there is a square. Roughly speaking, they have to check at most $|f_x| + |f_{x+1}|$ many periods at the borders of every two consecutive factors f_x and f_{x+1} due to the above lemma ($1 \leq x \leq z$, set f_{z+1} to the empty string). This gives $\sum_{x=1}^{z} t_{\mathrm{LCE}} (|f_x| + |f_{x+1}|) = \mathcal{O}(n t_{\mathrm{LCE}})$ time, during which they can compute a leftmost covering set L. Figure 1 visualizes how the checks are done. Applying the algorithm on our running example will yield the set $L = \{(1,4),(5,2),(7,4)\}$. To transform this set into a set of all distinct squares, their algorithm runs the so-called Phase II that uses the suffix tree. It begins with computing the locations of the squares belonging to a subset $L' \subseteq L$ in the suffix tree in $\mathcal{O}(n)$ time. This subset L' is still guaranteed to be a leftmost covering set. Finally, their algorithm computes all distinct squares of the text by right-rotating the squares in L'. In their algorithm, the right-rotations are done by *suffix link walks* over the suffix tree. Their running time analysis is based on the fact that each node has at most σ_T incoming suffix links, where σ_T denotes the number of different characters occurring in the text T. Given that the number of distinct squares is linear, Phase II runs in $\mathcal{O}(n\sigma_T)$ time.

4.1 Algorithm Computing the Set of All Distinct Squares

In the following, we will present our modification of the above sketched algorithm. To speed up the computation, we discard the idea of using the suffix links for right-rotating squares (i.e., we skip Phase II completely). Instead, we compute a list of all distinct squares directly. To this end, we show a modification of the sketched algorithm such that it outputs this list sorted first by the lengths (of the squares), and second by the starting position.

First, we want to show that we can change the original algorithm to output its leftmost covering set in the above described order. To this end, we iterate over all possible periods, and search not yet reported squares at all Lempel-Ziv borders, for each period. To achieve linear running time, we want to skip a factor f_x when the period becomes longer than $|f_x| + |f_{x+1}|$. We can do this with an array Z of $z \lg z$ bits that is zero initialized. When the currently tested period p exceeds $|f_x| + |f_{x+1}|$, we write $Z[x] \leftarrow \min \{y > x : |f_y| + |f_{y+1}| \geq p\}$ such that $Z[x]$ refers to the next factor whose length is sufficiently large. By doing so, if $Z[x] \neq 0$, we can skip all factors f_y with $y \in [x..Z[x]-1]$ in constant time. This allows us running the modified algorithm still in linear time.

We have to show that the modified algorithm still computes the same set. To this end, let us fix the period p (over which we iterate in the outer loop). By [23, Lemma 7], processing squares satisfying Lemma 8(1) before processing squares satisfying Lemma 8(2) (all squares have the same period p) produces the desired output for period p.

Finally, we show the modification that computes all distinct squares (instead of the original leftmost covering set). On a high level, we use an RMQ data structure on LPF to filter already found squares. The filtered squares are used to determine the leftmost occurrences of all squares by right-rotation. In more detail, we modify Algorithm 1 of [23] by filtering the squares in the following way (see Algorithm 1 in the full version [2]): For each period p, we use a bit vector B marking the beginning positions of all found squares with period p. On reporting a square, we additionally mark its starting position in B. By doing so, an invariant of the algorithm below is that all right-rotated squares of a marked square are already reported.

Let us assume that we are searching for the leftmost occurrences of all squares whose periods are equal to p. Given the starting position s of a square returned by [23, Algorithm 1], we consider the square $(s, 2p)$ and its right-rotations as candidates of our list: If $B[s] = 1$, then this square and its right-rotations have already been reported. Otherwise, we report $(s, 2p)$ if $\mathsf{LPF}[s] < 2p$. In order to find the leftmost occurrences of all not yet reported right-rotated squares efficiently, we first compute the rightmost position e of the repetition of period p containing the square $(s, 2p)$ by an LCE query. Second, we check the interval $I := [s+1.. \min(s+p-1, e-2p+1)]$ for the starting positions of the squares whose LPF values are less than $2p$. To this end, we perform an RMQ query on LPF to find the position j whose LPF value is minimal in I. If $\mathsf{LPF}[j] > 2p$, then there is no leftmost occurrence of a square with the period p in the considered range. Otherwise, we report $(j, 2p)$ and recursively search for the text position with the minimal LPF value within the intervals $[s+1..j-1]$ and $[j+1.. \min(s+p-1, e-2p+1)]$. In overall, the time of the recursion is bounded by twice the number of distinct squares starting in the interval I, since a recursion step terminates if it could not report any square.

▶ **Theorem 9.** *Given an LCE data structure with t_{LCE} access time and LPF, we can compute all distinct squares in $\mathcal{O}(nt_{LCE} + \mathrm{occ}) = \mathcal{O}(nt_{LCE})$ time, where occ is the number of distinct squares.*

Proof. We show that the returned list is the list of all distinct squares. No square occurs in the list twice since we only report the occurrence of a square (i, ℓ) if $\mathsf{LPF}[i] < \ell$. Assume that there is a square missing in the list; let (i, ℓ) be its leftmost occurrence. There is a square (j, ℓ) reported by the (original) algorithm [23] such that $i - \ell/2 < j \leq i$ and right-rotating (j, ℓ) yields (i, ℓ). Since we right-rotate all found squares, we obviously have reported (j, ℓ).

The occ term in the running time is dominated by the nt_{LCE} term due to Lemma 7. ◀

The next corollary, which is immediate from Theorem 9, yields the main result.

▶ **Corollary 10.** *Given a string T of length n over an integer alphabet of size $n^{\mathcal{O}(1)}$, we can compute all distinct squares in T in $\mathcal{O}(n)$ time.*

4.2 Need for RMQ on LPF

Our algorithm performs right-rotations of a square $(s, 2p)$ with an RMQ on the interval $I := [s + 1..\min(s + p - 1, e - 2p + 1)]$, where e is the last position of the maximal repetition of period p that contains the square. Without an RMQ data structure, we could linearly scan all LPF values in I, giving $\mathcal{O}(p) = \mathcal{O}(n)$ time. We cannot do better since the LPF values are arbitrary in general. For instance, consider the text $T =$ `abaaabaababaaabaaa$`. The text aligned with LPF is shown in the table below.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T	a	b	a	a	a	b	a	a	b	a	b	a	a	a	b	a	a	a	$
LPF	0	0	1	2	4	3	4	3	2	8	7	6	5	5	4	3	2	1	0

The square `abaaabaa` has two occurrences starting at positions 1 and 10. The square `baaabaaa` at position 11 is found by right-rotating the occurrence of `abaaabaa` at position 10. It is found by a linear scan over LPF or an RMQ on LPF. A slight modification of this example can change the LPF values around this occurrence. This shows that we cannot perform a shortcut in general (like stopping the search when the LPF value is at least twice as large as p).

4.3 Practical Evaluation

We have implemented the algorithm computing the leftmost occurrences of all squares in C++11 [33]. The primary focus was on the execution time, rather than on a small memory footprint: We have deliberately chosen plain 32-bit integer arrays for storing all array data structures like SA, LCP and LPF. These data structures are constructed as follows: First, we generate SA with `divsufsort` [38]. Subsequently, we generate LCP with the Φ-algorithm [29], and LPF with the simple algorithm of [9, Proposition 1]. Finally, we use the bit vector class and the RMQ data structure provided by the sdsl-lite library [20]. In practice, it makes sense to use an RMQ only for very large LCP values and periods (i.e., RMQs on LPF) due to its long execution time. For small values, we naively compared characters, or scanned LPF linearly.

We ran the algorithm on all 200MiB collections of the Pizza&Chili Corpus [12]. The Pizza&Chili Corpus is divided in a real text corpus with the prefix PC, and in a repetitive corpus with the prefix PCR. The experiments were conducted on a machine with 32 GB of RAM and an Intel® Xeon® CPU `E3-1271 v3`. The operating system was a 64-bit version of Ubuntu Linux 14.04 with the kernel version 3.13. We used a single execution thread for the experiments. The source code was compiled using the GNU compiler `g++ 6.2.0` with the compile flags `-O3 -march=native -DNDEBUG`.

Table 2 shows the running times of the algorithm on the described datasets. It seems that large factors tend to slow down the computation, since the algorithm has to check all periods up to $\max_x(|f_x| + |f_{x+1}|)$. This seems to have more impact on the running time than the number of Lempel-Ziv factors z.

4.4 Online Variant

In this section, we consider the *online* setting, where new characters are appended to the end of the text T. Given the text $T[1..i]$ up to position i with the Lempel-Ziv factorization $f_1 \cdots f_y = T[1..i]$, we consider computing the set of all distinct squares of $f_1 \cdots f_{y-2}$, i.e., up to the last two Lempel-Ziv factors. For this setting, we show that we can compute the set of all distinct

Table 2 Practical evaluation of the algorithm computing all distinct squares on the datasets described in Section 4.3. Execution time is in seconds, $\mathrm{K} = 10^3$. It is the median of several conducted experiments, whose variance in time was small. The expression $\mathrm{avg}_{\mathsf{LCP}}$ is the average of all LCP values, and z is the number of Lempel-Ziv factors.

collection	σ	$\mathrm{avg}_{\mathsf{LCP}}$	z	$\max_x \lvert f_x \rvert$	$\max_x \lvert f_x f_{x+1} \rvert$	$\lvert \mathrm{occ} \rvert$	time
PC-DBLP.XML	97	44	7035K	1K	1K	7K	70
PC-DNA	17	60	13,970K	98K	98K	133K	310
PC-ENGLISH	226	9390	13,971K	988K	1094K	13K	2639
PC-PROTEINS	26	278	20,875K	46K	68K	3108K	245
PC-SOURCES	231	373	11,542K	308K	308K	340K	792
PCR-CERE	6	3541	1447K	176K	185K	47K	535
PCR-EINSTEIN.EN	125	45,983	50K	907K	1634K	18,193K	3953
PCR-KERNEL	161	149,872	775K	2756K	2756K	9K	6608
PCR-PARA	6	2268	1927K	71K	74K	37K	265

squares in $\mathcal{O}\left(n \min\left(\lg^2 \lg n / \lg\lg\lg n, \sqrt{\lg n / \lg\lg n}\right)\right)$ time using $\mathcal{O}(n)$ words of space. To this end, we adapt the algorithm of Theorem 9 to the online setting. We need an algorithm computing LPF online, and a semi-dynamic LCE data structure (answering LCE queries on the text *and* on the reversed text while supporting appending characters to the text).

The main idea of our solution is to build suffix trees with two online suffix tree construction algorithms. The first is Ukkonen's algorithm that computes the suffix tree online in $\mathcal{O}(nt_{\mathrm{nav}})$ time [43], where t_{nav} is the time for inserting a node and navigating (in particular, selecting the child on the edge starting with a specific character). We can adapt this algorithm to compute LPF online: Assume that we have computed the suffix tree of $T[1..i-1]$. The algorithm processes the new character $T[i]$ by (1) taking the suffix links of the current suffix tree, and (2) adding new leaves where a branching occurs. On adding a new leaf with suffix number i, we additionally set $\mathsf{LPF}[i]$ to the string depth of its parent. By doing so, we can update the LPF values in time linear in the update time of the suffix tree. We build the semi-dynamic RMQ data structure of Fischer [14] (or of [42] if n is known beforehand) on top of LPF. This data structure takes $\mathcal{O}(n)$ words and can perform query and appending operations in constant amortized time.

The second suffix tree construction algorithm is a modified version [4] of Weiner's algorithm [44] that builds the suffix tree in the reversed order of Ukkonen's algorithm in $\mathcal{O}(nt_{\mathrm{nav}})$ time. Since Weiner's algorithm incrementally constructs the suffix tree of a given text from right to left, we can adapt this algorithm to compute the suffix tree of the reversed text online in $\mathcal{O}(nt_{\mathrm{nav}})$ time.

To get a suffix tree construction time of $\mathcal{O}\left(n \min\left(\lg^2 \lg n / \lg\lg\lg n, \sqrt{\lg n / \lg\lg n}\right)\right)$, we use the predecessor data structure of Beame and Fich [3]. We create a predecessor data structure to store the children of each suffix tree node, such that we get the navigation time $t_{\mathrm{nav}} = \mathcal{O}\left(\min\left(\lg^2 \lg n / \lg\lg\lg n, \sqrt{\lg n / \lg\lg n}\right)\right)$ for both suffix trees. We also create a predecessor data structure to store the out-going suffix link of each node of the suffix tree constructed by Weiner's algorithm. Overall, these take a total of $\mathcal{O}(n)$ words of space.

Finally, our last ingredient is a dynamic lowest common ancestor data structure with $\mathcal{O}(n)$ words that performs querying and modification operations in constant time [7]. The lowest common ancestor of two suffix tree leaves with the labels j and k is the node whose string depth is equal to the longest common extension of $T[j..i]$ and $T[k..i]$ — remember that we consider the text T up to the position i, hence $T[j..i]$ is (currently) the j-th suffix. Building this data structure on the suffix tree of the text T and on the suffix tree of the reversed text allows us to compute LCE queries in both directions in constant time.

Given the text $T[1..i] = f_1 \cdots f_y$ up to the i-th character, the entries of

$\mathsf{LPF}[1..|f_1 \cdots f_{y-2}| - 1]$ are fixed (i.e., they will not change when appending new characters) due to the properties of the Lempel-Ziv factorization. We let the semi-dynamic RMQ data structure grow with LPF, but only up to the fixed range of LPF. Similarly, the text positions from 1 up to $|f_1 \cdots f_{y-2}| - 1$ are represented as leaves in both suffix trees that are fixed, i.e., these leaves will always be leaves representing their respective suffixes. To sum up, our data structures support LCE queries and RMQs on LPF in the range $[1..|f_1 \cdots f_{y-2}| - 1]$ in constant time.

We adapt the algorithm of Section 4.1 by switching the order of the loops (again). The algorithm first fixes a Lempel-Ziv factor f_x and then searches for squares with a period between one and $|f_x| + |f_{x+1}|$. Unfortunately, we would need an extra bit vector for each period so that we can track all found leftmost occurrences. Instead, we use the predecessor data structure of [3] storing the found occurrences of squares as pairs of starting positions and lengths. These pairs can be stored in lexicographic order (first sorted by starting position, then by length). The predecessor data structure will contain at most occ elements, hence takes $\mathcal{O}(\text{occ}) = \mathcal{O}(n)$ words of space. An insertion or a search costs us $\mathcal{O}\left(\min\left(\lg^2 \lg n / \lg\lg\lg n, \sqrt{\lg n / \lg\lg n}\right)\right)$ time.

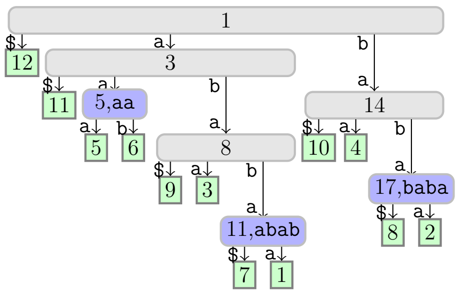
Let us assume that we have computed the set for $T[1..i-1]$, and that the Lempel-Ziv factorization of $T[1..i-1]$ is $f_1 \cdots f_y$. If appending a new character $T[i]$ will result in a new factor f_{y+1}, we check for squares of type Lemma 8(1) and Lemma 8(2) at the borders of f_{y-1}. Duplicates are filtered by the predecessor data structure storing all already reported leftmost occurrences. The algorithm outputs only the leftmost occurrences with the aid of LPF, whose entries are fixed up to the last two factors (this is sufficient since we search for the starting position of the leftmost occurrence of a square with type Lemma 8(1) only in $T[1..|f_1 \cdots f_{y-1}|]$, including right-rotations). In overall, we need $\mathcal{O}\left((|f_{y-1}| + |f_y|)\min\left(\lg^2 \lg n / \lg\lg\lg n, \sqrt{\lg n / \lg\lg n}\right)\right)$ time.

5 Applications

In this section, we provide two applications of the (offline) variant.

5.1 Decorating the Suffix Tree with All Squares

Gusfield and Stoye described a representation of the set of all distinct squares by a decoration of the suffix tree, like the highlighted nodes (additionally annotated with its respective square) shown in the suffix tree of our running example below.



This representation asks for a set of tuples of the form (node, length) such that each square $T[i..i+\ell-1]$ is represented by a tuple (v, ℓ), where v is the highest node whose string label has $T[i..i+\ell-1]$ as a (not necessarily proper) prefix. We show that we can compute this set of tuples in linear time by applying the Phase II algorithm [23] described in Section 4 to our computed set of all distinct squares. The Phase II algorithm takes a list L_i storing squares starting at text position i, for each $1 \leq i \leq n$. Each of these lists has to be sorted in descending order with respect to the squares' lengths. It is easy to adapt our algorithm to produce these lists: On reporting a square (i, ℓ), we insert it at the front of L_i. By doing so, we can fill the lists *without* sorting, since we iterate over the period length in the outer loop, while we iterate over all Lempel-Ziv factors in the inner loop.

Finally, we can conduct Phase II. In the original version, the goal of Phase II was to decorate the suffix tree with the endpoints of a subset of the original leftmost covering set. We will show that performing exactly the same operations with the set of the leftmost occurrences of all squares will decorate the suffix tree with all squares directly. In more detail, we first augment the suffix tree leaf having label i with the list L_i, for each $1 \leq i \leq n$. Subsequently, we follow Gusfield and Stoye [23] by processing every node of the suffix tree with a bottom-up traversal. During this traversal we propagate the lists of squares from the leaves up to the root: An internal node u inherits the list of the child whose subtree contains the leaf with the smallest label among all leaves in the subtree rooted at u. If the edge to the parent node contains the ending position of one or more squares in the list (these candidates are stored at the front of the list), we decorate the edge with these squares, and pop them off from the list. By [23, Theorem 8], there is no square of the set L' (defined in Section 4) neglected during the bottom-top traversal. The same holds if we exchange L' with our computed set of all distinct squares:

▶ **Lemma 11.** *By feeding the algorithm of Phase II with the above constructed lists L_i containing the leftmost occurrences of the squares starting at the text position i, it will decorate the suffix tree with all distinct squares.*

Proof. We adapt the algorithm of Section 4.1 to build the lists L_i. These lists contain the leftmost occurrences of all squares. In the following we show that no square is left out during the bottom-up traversal. Let us take a suffix tree node u with its children v and w. Without loss of generality, assume that the smallest label among all leaves contained in the subtree of v is smaller than the label of every leaf contained in w's subtree. For the sake of contradiction, assume that the list of w contains the occurrence of a square (i, ℓ) at the time when we pass the list of v to its parent u. The length ℓ is smaller than v's string depth, otherwise it would already have been popped off from the list. But since v's subtree contains a leaf whose label j is the smallest among all labels contained in the subtree of w, the square occurs before at $T[j..j + \ell - 1] = T[i..i + \ell - 1]$, a contradiction to the distinctness. ◀

This concludes the correctness of the modified algorithm. We immediately get:

▶ **Theorem 12.** *Given* LPF*, an LCE data structure on the reversed text, and the suffix tree of T, we can decorate the suffix tree with all squares of the text in $\mathcal{O}(nt_{\mathrm{LCE}})$ time. Asides from these data structures, we use $(\mathrm{occ} + n)\lg n + z \lg z + \min(n + o(n), z \lg n) + \mathcal{O}(\lg n)$ bits of additional working space.*

▶ **Corollary 13.** *We can compute the suffix tree and decorate it with all squares of the text in $\mathcal{O}(n/\epsilon)$ time using $(3n + \mathrm{occ} + 2n\epsilon)\lg n + z \lg z + \mathcal{O}(n)$ bits, for a constant $0 < \epsilon \leq 1$.*

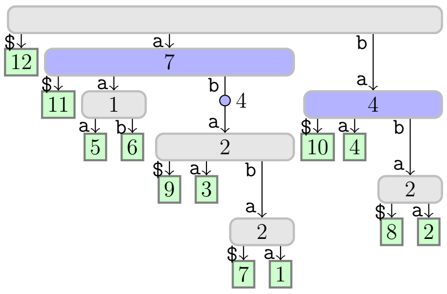
As an application, we consider the common squares problem: Given a set of non-empty strings with a total length n, we want to find all squares that occur in every string in $\mathcal{O}(n)$ time. We solve this problem by first decorating the generalized suffix tree built on all strings with the distinct squares of all strings. Subsequently, we apply the $\mathcal{O}(n)$ time solution of

Hui [25] that annotates each internal suffix tree node v with the number of strings that contain v's string label. This solves our problem since we can simply report all squares corresponding to nodes whose string labels are found in all strings. This also solves the problem asking for the longest common square of all strings in $\mathcal{O}(n)$ time, analogously to the longest common substring problem [22].

The last subsection is dedicated to another application of our suffix tree decoration:

5.2 Computing the Tree Topology of the MAST in Linear Time

A modification of the suffix tree is the ***minimal augmented suffix tree (MAST)*** [1]. This tree can answer the number of the non-overlapping occurrences of a substring S of T in $\mathcal{O}(|S|)$ time. The MAST can be built in $\mathcal{O}(n \lg n)$ time [5].



In this section, we show how to compute the tree topology of the MAST in linear time. The topology of the MAST differs to the suffix tree topology by the fact that the root of each square is the string label of an MAST node. Our goal is to compute a list storing the information about where to insert the missing nodes. The list stores tuples consisting of a node v and a length ℓ; we use this information later to create a new node w splitting the edge (u, v) into (u, w) and (w, v), where u is the (former) parent of v. We will label (w, v) with the last ℓ characters and (u, v) with the rest of the characters of the edge label of (u, v).

To this end, we explore the suffix tree with a top-down traversal while locating the roots of the squares in the order of their lengths. To locate the roots of the squares in linear time we use two data structures. The first one is a semi-dynamic lowest marked ancestor data structure [19]. It allows marking a node and querying for the lowest marked ancestor of a node in constant amortized time. We will use it to mark the area in the suffix tree that has already been processed for finding the roots of the squares.

The second data structure is the list of tuples of the form (node, length) computed in Section 5.1, where each tuple (v, ℓ) consists of the length ℓ of a square $T[i..i + \ell - 1]$ and the highest suffix tree node v whose string label has $T[i..i + \ell - 1]$ as a (not necessarily proper) prefix. We sort this list, which we now call L, with respect to the square lengths with a linear time integer sorting algorithm.

Finally, we explain the algorithm locating the roots of all squares. We successively process all tuples of L, starting with the shortest square length. Given a tuple of L containing the node v and the length ℓ, we want to split an edge on the path from the root to v and insert a new node whose string depth is $\ell/2$. To this end, we compute the lowest marked ancestor u of v. If u's string depth is smaller than $\ell/2$, we mark all descendants of u whose string depths are smaller than $\ell/2$, and additionally the children of those nodes (this can be done by a DFS or a BFS). If we query for the lowest marked ancestor of u again, we get an ancestor w whose string depth is at least $\ell/2$, and whose parent has a string depth less than $\ell/2$. We report w and the subtraction of $\ell/2$ from w's string depth (if $\ell/2$ is equal to the string depth of w, then w's string label is equal to the root of v's string label, i.e., we do not have to report it).

▶ **Theorem 14.** *We can compute the tree topology of the MAST in linear time using linear number of words.*

Proof. By using the semi-dynamic lowest marked ancestor data structure, we visit a node as many times as we have to insert nodes on the edge to its parent, plus one. This gives $\mathcal{O}(n + 2\text{occ}) = \mathcal{O}(n)$ time. ◀

Open Problems. It is left open to compute the number of the non-overlapping occurrences of the string labels of the MAST nodes in linear time. Since RMQ data structures are practically slow, we wonder whether we can avoid the use of any RMQ without loosing linear running time. The current bottleneck of the online algorithm is the predecessor data structure in terms of the running time. Future integer dictionary data structures can improve the overall performance of this algorithm.

Acknowledgements. We thank Thomas Schwentick for the question whether we can run our algorithm online, for which we provided a solution in Section 4.4.

References

1 Alberto Apostolico and Franco P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15(5):481–494, 1996. `doi:10.1007/BF01955046`.

2 Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl. Computing all distinct squares in linear time for integer alphabets, 2016. `arXiv:1610.03421`.

3 Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002. `doi:10.1006/jcss.2002.1822`.

4 Anselm Blumer, Janet A. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985. `doi:10.1016/0304-3975(85)90157-4`.

5 Gerth Stølting Brodal, Rune B. Lyngsø, Anna Östlin, and Christian N. S. Pedersen. Solving the string statistics problem in time $\mathcal{O}(n \log n)$. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming (ICALP 2002)*, volume 2380 of *LNCS*, pages 728–739. Springer, 2002. `doi:10.1007/3-540-45465-9_62`.

6 David R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996. URL: `http://hdl.handle.net/10012/64`.

7 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4), 2005. `doi:10.1137/S0097539700370539`.

8 Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008. `doi:10.1016/j.ipl.2007.10.006`.

9 Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter, and Tomasz Waleń. LPF computation revisited. In Jirí Fiala, Jan Kratochvíl, and Mirka Miller, editors, *Proceedings of the 20th International Workshop on Combinatorial Algorithms (IWOCA 2009)*, volume 5874 of *LNCS*, pages 158–169. Springer, 2009. `doi:10.1007/978-3-642-10217-2_18`.

10 Antoine Deza, Frantisek Franek, and Adrien Thierry. How many double squares can a string contain? *Discrete Appl. Math.*, 180:52–69, 2015. `doi:10.1016/j.dam.2014.08.016`.

11 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. `doi:10.1145/355541.355547`.

12 Paolo Ferragina and Gonzalo Navarro. The Pizza & Chili Corpus. Available at `http://pizzachili.di.unipi.it` and `http://pizzachili.dcc.uchile.cl`, 2005.

13 Johannes Fischer. Wee LCP. *Inf. Process. Lett.*, 110(8–9):317–320, 2010. `doi:10.1016/j.ipl.2010.02.010`.

14 Johannes Fischer. Inducing the LCP-array. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Proceedings of the 12th International Symposium on Algorithms and Data Structures (WADS 2011)*, volume 6844 of *LNCS*, pages 374–385. Springer, 2011. `doi:10.1007/978-3-642-22300-6_32`.

15 Johannes Fischer and Volker Heun. Space efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. `doi:10.1137/090779759`.

16 Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel-Ziv computation in small space (LZ-CISS). In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 172–184. Springer, 2015. `doi:10.1007/978-3-319-19929-0_15`.

17 Aviezri S. Fraenkel and Jamie Simpson. How many squares can a string contain? *J. Comb. Theory, Ser. A*, 82(1):112–120, 1998. `doi:10.1006/jcta.1997.2843`.

18 Frantisek Franek, Jan Holub, William F. Smyth, and Xiangdong Xiao. Computing quasi suffix arrays. *J. Autom. Lang. Comb.*, 8(4):593–606, 2003.

19 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. `doi:10.1016/0022-0000(85)90014-5`.

20 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA 2014)*, volume 8504 of *LNCS*, pages 326–337. Springer, 2014. `doi:10.1007/978-3-319-07959-2_28`.

21 Keisuke Goto. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets, 2017. `arXiv:1703.01009`.

22 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997. `doi:10.1017/CBO9780511574931`.

23 Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004. `doi:10.1016/j.jcss.2004.03.004`.

24 Wing-Kai Hon and Kunihiko Sadakane. Space-economical algorithms for finding maximal unique matches. In Alberto Apostolico and Masayuki Takeda, editors, *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, volume 2373 of *LNCS*, pages 144–152. Springer, 2002. `doi:10.1007/3-540-45452-7_13`.

25 Lucas Chi Kwong Hui. Color set size problem with application to string matching. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM 1992)*, volume 644 of *LNCS*, pages 230–243. Springer, 1992. `doi:10.1007/3-540-56024-6_19`.

26 Lucian Ilie. A note on the number of squares in a word. *Theor. Comput. Sci.*, 380(3):373–376, 2007. `doi:10.1016/j.tcs.2007.03.025`.

27 Natasa Jonoska, Florin Manea, and Shinnosuke Seki. A stronger square conjecture on binary words. In Viliam Geffert, Bart Preneel, Branislav Rovan, Julius Stuller, and A Min

Tjoa, editors, *Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2014)*, volume 8327 of *LNCS*, pages 339–350. Springer, 2014. `doi:10.1007/978-3-319-04298-5_30`.

28 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In Johannes Fischer and Peter Sanders, editors, *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013. `doi:10.1007/978-3-642-38905-4_19`.

29 Juha Kärkkäinen, Giovanni Manzini, and Simon John Puglisi. Permuted longest-common-prefix array. In Gregory Kucherov and Esko Ukkonen, editors, *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009. `doi:10.1007/978-3-642-02441-2_17`.

30 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. `doi:10.1145/1217856.1217858`.

31 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir and Gad M. Landau, editors, *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001. `doi:10.1007/3-540-48194-X_17`.

32 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005. `doi:10.1016/j.jda.2004.08.002`.

33 Dominik Köppl. Computing all distinct squares efficiently, 2017. URL: `https://github.com/koeppl/distinct_squares`.

34 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2016 Data Compression Conference (DCC 2016)*, pages 3–12. IEEE Computer Society, 2016. `doi:10.1109/DCC.2016.38`.

35 Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting, 2016. `arXiv:1610.08305`.

36 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. `doi:10.1137/0222058`.

37 Florin Manea and Shinnosuke Seki. Square-density increasing mappings. In Florin Manea and Dirk Nowotka, editors, *Proceedings of the 10th International Conference on Combinatorics on Words (WORDS 2015)*, volume 9304 of *LNCS*, pages 160–169. Springer, 2015. `doi:10.1007/978-3-319-23660-5_14`.

38 Yuta Mori. libdivsufsort, 2015. URL: `https://github.com/y-256/libdivsufsort`.

39 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In Philip N. Klein, editor, *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 408–424. SIAM, 2017. `doi:10.1137/1.9781611974782.26`.

40 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In Raffaele Giancarlo and Giovanni Manzini, editors, *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*, volume 6661 of *LNCS*, pages 15–26. Springer, 2011. `doi:10.1007/978-3-642-21458-5_4`.

41 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. `doi:10.1007/s00224-006-1198-x`.

42 Yohei Ueki, Diptarama, Masatoshi Kurihara, Yoshiaki Matsuoka, Kazuyuki Narisawa, Ryo Yoshinaka, Hideo Bannai, Shunsuke Inenaga, and Ayumi Shinohara. Longest common subsequence in at least k length order-isomorphic substrings. In Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria, editors, *Proceedings of the 43rd International Conference on Current Trends in Theory and Practice* *of Computer Science (SOFSEM 2017)*, volume 10139 of *LNCS*, pages 363–374. Springer, 2017. doi:10.1007/978-3-319-51963-0_28.

43 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

44 Peter Weiner. Linear pattern matching algorithms. In H. Raymond Strong, editor, *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

Small Observation. In [23, Line 6 of Algorithm 1b], the condition $start + k < h_1$ has to be changed to $start + k \leq h_1$. Otherwise, given the text $T =$ abaabab\$, the algorithm would find only the square aa, but not abaaba.

A Algorithm Execution with one Step at a Time

In this section, we process the running example $T =$ ababaaababa\$ with the algorithm devised in Section 4.1 step by step. SA, LCP, PLCP, and LPF are given in the table below (the LZ row partitions the text into factors, their borders are represented by the vertical bars):

i	1	2	3	4	5	6	7	8	9	10	11	12
T	a	b	a	b	a	a	a	b	a	b	a	\$
SA	12	11	5	6	9	3	7	1	10	4	8	2
LCP	0	0	1	2	1	3	3	5	0	2	2	4
PLCP	5	4	3	2	1	2	3	2	1	0	0	0
LPF	0	0	3	2	1	2	5	4	3	2	1	0
LZ	f_1	f_2	f_3			f_4		f_5				f_6

The text $T = \overset{1}{\texttt{a}}|\overset{2}{\texttt{b}}|\overset{3}{\texttt{aba}}|\overset{4}{\texttt{aa}}|\overset{5}{\texttt{baba}}|\overset{6}{\texttt{\$}} = f_1 \cdots f_6$ is factorized in six Lempel-Ziv factors. We call $T[1+|f_1 \cdots f_{i-1}|]$ (first position of the i-th factor) and $T[1+|f_1 \cdots f_i|]$ (position after the i-th factor) the ***left border*** and the ***right border*** of f_i, respectively. The idea of the algorithm is to check the presence of a square at a factor border and at an offset value q of the border with LCE queries. q is either the *addition* of p to the *left* border, or the *subtraction* of p from the *right* border (see Figure 1).
The algorithm finds the leftmost occurrences of all squares in the order (first) of their lengths and (second) of their starting positions. We start with the period $p = 1$ and try to detect squares at each Lempel-Ziv factor border. To this end, we create a bit vector B marking all found squares with period $p = 1$. A square of this period is found at the right border of f_3. It is of type Lemma 8(1), since its starting position is in f_3. To find it, we take the right border $b = 6$ of f_3, and the position $q := b - p = 5$. We perform an LCE query at b and q in the forward and backward direction. Only the forward query returns the non-zero value of one. But this is sufficient to find the square aa of period one. Its LPF value is smaller than $2p = 2$, so it is the leftmost occurrence. It is not yet marked in B, thus we have not yet reported it. Right-rotations are not necessary for period 1. Having found all squares with period 1, we clear B.

Next, we search for squares with period 2. We find a square of type Lemma 8(2) at the left border $b = 2$ of f_2. To this end, we perform an LCE query starting from b and $q := b + p = 4$ in both directions. Both LCE queries show that $T[1..5]$ is a repetition with period $p = 2$. Thus we know that $T[1..4]$ is a square. It is not yet marked in B, and has an LPF value smaller than $2p = 4$, i.e., it is a not yet reported leftmost occurrence. On finding a leftmost occurrence of a square, we right-rotate it, and report all right-rotations whose LPF values are below $2p$. This is the case for $T[2..5]$, which is the leftmost occurrence of the square baba.

After some unsuccessful checks at the next factor borders, we come to factor f_5 and search for a square of type Lemma 8(2). Two LCE queries in both directions at the left border $b = 8$ of f_5 and $q := b + p = 10$ reveal that $T[7..11]$ is a repetition of period 2. The substring $T[7..10]$ is a square, but its LPF value is $5(\geq 2p)$, i.e., we have already reported this square. Although we have already reported it, some right-rotation of it might not have been reported yet (see Section 4.2 for an example). This time, all right-rotations (i.e., $T[8..12]$) have an LPF value $\geq 2p$, i.e., there is no leftmost occurrence of a square of period 2 found by right-rotations. In overall, we have found and reported the leftmost occurrences of all squares *once*.

B More Evaluation

Table 3 Running times in seconds, evaluated on different input sizes. We took prefixes of 1MiB, 10MiB, 50MiB, and 100MiB of all collections.

collection	1MiB	10MiB	50MiB	100MiB	200MiB
PC-DBLP.XML	0.2	3	16	33	70
PC-DNA	0.3	3	23	56	310
PC-ENGLISH	0.2	5	42	500	2639
PC-PROTEINS	0.3	4	25	74	245
PC-SOURCES	0.2	3	31	286	792
PCR-CERE	0.6	6	30	79	535
PCR-EINSTEIN.EN	0.4	12	83	1419	3953
PCR-KERNEL	0.2	8	233	1274	6608
PCR-PARA	0.4	4	26	98	265

C Proofs

Proof of Theorem 12

Proof. We need $(\text{occ} + n) \lg n$ bits for storing the lists L_i ($\text{occ} \lg n$ bits for storing the lengths of all squares in an integer array, and $n \lg n$ bits for the pointers to the first element of each list). The array Z uses $z \lg z$ bits. The Lempel-Ziv factors are represented as in Corollary 1. The time t_{LCE} is the maximum time of the LCE data structure and the suffix tree for answering an LCE query. ◀

Proof of Theorem 13

Proof. We use Theorem 6 to store SA, ISA, LCP, and LPF in $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits. Subsequently, we build an RMQ data structure on LCP such that LCE queries can be answered in constant time. We additionally need the suffix array, its inverse, and the LCP array (with an RMQ data structure) of the reversed text to answer LCE queries on the reversed text. Finally, we endow LPF with an RMQ data structure for the right-rotations. An LCE query on the text can be answered by the string depth of a lowest common ancestor in the suffix tree in constant time. ◀

D Pseudo Code

Algorithm 1: Modified Algorithm 1 of [23].

```
b($f$) denotes the left end of a factor $f = T[\mathsf{b}(f)..\mathsf{b}(f) + |f| - 1]$, $lcp$ and $lcs$ compute the LCE
  in $T$ and the LCE in the reverse of $T$ (mirroring the input indices by $i \mapsto n - i$ for
  $1 \leq i \leq n - 1$), respectively.
Let $f_1, \ldots, f_z$ be the factors of the Lempel-Ziv factorization
$f_{z+1} \leftarrow T[n]$                                            // dummy factor
Function recursive-rotate($s$ : starting position, $e$: ending position)
    $m \leftarrow$ LPF.$RMQ[s..e]$
    if $m > 2p$ then return
    report($m, 2p$) and $B[m] \leftarrow 1$
    recursive-rotate($s$,$m - 1$) and recursive-rotate($m + 1$,$e$)
Function right-rotate($s$ : starting position of square, $p$: period of square)
    if $B[s] = 1$ then return
    if LPF$[s] < 2p$ then report($s, 2p$) and $B[s] \leftarrow 1$
    $\ell \leftarrow lcp(s, s + p)$
    recursive-rotate($s + 1, s + p - 1, s + \ell - p$)
$Z \leftarrow$ array of size $z \lg z$ bits, zero initialized
$m \leftarrow \max(|f_1| + |f_2|, \ldots, |f_{z-1}| + |f_z|)$
for $p = 1, \ldots, m$ do
    $B \leftarrow$ bit vector of length $n$, zero initialized
    for $x = 1, \ldots, z$ do
        if $|f_x| + |f_{x+1}| < p$ then
            $y \leftarrow x$
            while $|f_y| + |f_{y+1}| < p$ do
                if $Z[y] \neq 0$ then $y \leftarrow Z[y]$
                else incr $y$
            $Z[x] \leftarrow y$ and $x \leftarrow y$
        if $|f_x| \geq p$ then // probe for squares satisfying Lemma 8(1)
            $q \leftarrow \mathsf{b}(f_{x+1}) - p$
            $\ell_{\mathrm{R}} \leftarrow lcp(\mathsf{b}(f_{x+1}), q)$ and $\ell_{\mathrm{L}} \leftarrow lcs(\mathsf{b}(f_{x+1}) - 1, q - 1)$
            if $\ell_{\mathrm{R}} + \ell_{\mathrm{L}} \geq p$ and $\ell_{\mathrm{R}} > 0$ then // found a square of length $2p$ with its
              right end in $f_{x+1}$
                $s \leftarrow \max(q - \ell_{\mathrm{L}}, q - p + 1)$ // square starts at $s$
                right-rotate($s, p$)
        $q \leftarrow \mathsf{b}(f_x) + p$ // probe for squares satisfying Lemma 8(2)
        $\ell_{\mathrm{R}} \leftarrow lcp(\mathsf{b}(f_x), q)$ and $\ell_{\mathrm{L}} \leftarrow lcs(\mathsf{b}(f_x) - 1, q - 1)$
        $s \leftarrow \max(\mathsf{b}(f_x) - \ell_{\mathrm{L}}, \mathsf{b}(f_x) - p + 1)$ // square starts in a factor preceding $f_x$
        if $\ell_{\mathrm{R}} + \ell_{\mathrm{L}} \geq p$ and $\ell_{\mathrm{R}} > 0$ and $s + p \leq \mathsf{b}(f_{x+1})$ and $\ell_{\mathrm{L}} > 0$ then // found a square
          of length $2p$ whose center is in $f_x$
            right-rotate($s, p$)
```

Palindromic Length in Linear Time

Kirill Borozdin[1], Dmitry Kosolobov[2], Mikhail Rubinchik[3], and Arseny M. Shur[4]

1 **Ural Federal University, Ekaterinburg, Russia**
borozdin.kirill,@gmail.com
2 **University of Helsinki, Helsinki, Finland**
dkosolobov@mail.ru
3 **Ural Federal University, Ekaterinburg, Russia**
mikhail.rubinchik@gmail.com
4 **Ural Federal University, Ekaterinburg, Russia**
arseny.shur@urfu.ru

Abstract

Palindromic length of a string is the minimum number of palindromes whose concatenation is equal to this string. The problem of finding the palindromic length drew some attention, and a few $O(n \log n)$ time online algorithms were recently designed for it. In this paper we present the first linear time online algorithm for this problem.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics

Keywords and phrases palindrome, palindromic length, palindromic factorization, online

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.23

1 Introduction

Algorithmic and combinatorial problems involving palindromes attracted the attention of researchers since the first days of stringology. Recall that a string $w = a_0a_1 \cdots a_{n-1}$ is a *palindrome* if it is equal to the string $\overleftarrow{w} = a_{n-1} \cdots a_1a_0$. The early works [4, 6, 8, 11] considered palindromes as structures that might provide examples of (context-free) languages that are impossible to recognize in linear time, thus provably restricting the computational power of some models (RAM, in particular). Subsequently, it was shown that many of such languages are, in fact, linear recognizable. Recently it was proved [7] that the language $\mathcal{P}^k$, where $\mathcal{P}$ is the set of all palindromes on a given alphabet, is recognizable online in $O(kn)$ time, where n is the length of the input string. Roughly at the same time, a closely related notion of *palindromic length* of a string was introduced: this is the minimal number k such that the string belongs to $\mathcal{P}^k$. In 2014–2015 three different algorithms that compute the palindromic length of a string of length n in $O(n \log n)$ time were presented in [3, 5, 10] (however, they all are based on similar principles). In this paper we present the first linear algorithm computing the palindromic length. Moreover, our algorithm is *online*, i.e., it reads the input string sequentially from left to right and computes the palindromic length for each prefix after reading the rightmost letter of that prefix. Thus, we prove the following theorem.

▶ **Theorem 1.** *Palindromic length of a string is computable online in linear time.*

The implementation of our algorithm and tests for it can be found in [9]. Due to a large constant under the big-O, it is slower in practice (for 32/64 bit machine words) than the existing $O(n \log n)$ solutions; the fastest algorithm is the one of [10].

The paper is organized as follows. Section 2 contains a high-level description of the algorithm: it starts with a naive $O(n^2)$ algorithm, then improves the time to $O(n \log n)$, and, finally, describes on a high level a modified $O(n)$-time version of the $O(n \log n)$ algorithm. In Section 3 we discuss the main components of the linear algorithm in details.

1.1 Preliminaries

Let w be a string of length $n = |w|$. We write $w[i]$ for the ith letter of w ($i = 0, \ldots, n{-}1$) and $w[i..j]$ for $w[i]w[i{+}1]\cdots w[j]$. A string u is a *substring* of w if $u = w[i..j]$ for some i, j. Such pair (i, j) is not necessarily unique; i specifies an *occurrence* of u at *position* i. A substring $w[0..j]$ (resp., $w[i..n{-}1]$) is a *prefix* (resp. *suffix*) of w. The *empty string* is denoted by ε. For any i, j, $[i..j]$ denotes the set $\{k \in \mathbb{Z}\colon i \le k \le j\}$; let $(i..j] = [i..j] \setminus \{i\}$, $[i..j) = [i..j] \setminus \{j\}$, $(i..j) = [i..j) \cap (i..j]$. Our notation for arrays is the same as for strings.

A substring (resp. suffix, prefix) that is a palindrome is called a *subpalindrome* (resp. *suffix-palindrome*, *prefix-palindrome*). If $w[i..j]$ is a subpalindrome of w, then the number $(j + i)/2$ is the *center* of $w[i..j]$ and the number $\lfloor (j - i + 1)/2 \rfloor$ is the *radius* of $w[i..j]$. The following remarkable property of palindromic lengths is crucial for our algorithm.

▶ **Lemma 2** (see [10, Lemma 11]). *Denote by $\ell_0, \ell_1, \ldots, \ell_{n-1}$, resp., the palindromic lengths of the prefixes $w[0..0], w[0..1], \ldots, w[0..n{-}1]$ of a string w. Then, for any $i \in (0..n)$, $|\ell_i{-}\ell_{i-1}| \le 1$.*

An integer p is a *period* of w if $w[i] = w[i{+}p]$ for any $i \in [0..n{-}p)$. As the previous results [3, 5, 10], our approach relies on a number of periodic properties of palindromes.

▶ **Lemma 3** (see [7, Lemmas 2, 3]). *For any palindrome w and any $p \in (0..|w|]$, the following conditions are equivalent: (1) p is a period of w, (2) there are palindromes u, v such that $|uv| = p$ and $w = (uv)^k u$ for some $k \ge 1$, (3) $w[p..|w|{-}1]$ ($w[0..|w|{-}p{-}1]$) is a palindrome.*

▶ **Lemma 4** (see [7, Lemma 7]). *Suppose that $w = (uv)^k u$ for $k \ge 1$ and for palindromes u and v such that $|uv|$ is the minimal period of w; then, the center of any subpalindrome x of w such that $|x| \ge |uv|{-}1$ coincides with the center of some u or v from the decomposition.*

Henceforth, let s denote the input string of length n. We assume that the algorithm works in the unit-cost word-RAM model with $\Theta(\log n)$-bit machine words (an assumption justified in, e.g., [2]) and standard operations like in the C programming language.

2 High-Level Description of the Algorithm

Our aim is to maintain an array $\mathsf{ans}[0..n{-}1]$ in which each element $\mathsf{ans}[i]$ is the palindromic length of $s[0..i]$. We always assume n to be the length of the string s processed so far (i.e., $s = s[0..n{-}1]$). Processing the next letter $s[n]$, we compute $\mathsf{ans}[n]$ and then increment n.

2.1 Naive approach

An easy quadratic-time approach is to maintain the list of all non-empty suffix-palindromes $u_1, \ldots, u_k$ of the string s and calculate $\mathsf{ans}[n] = 1 + \min_{i \in [1..k]} \mathsf{ans}[n{-}|u_i|]$. The list can be updated in linear time: the suffix-palindromes of wa have the form aua, where u is a suffix palindrome of w, plus the palindrome a and, optionally, aa. As a first speedup to this basic approach, we utilize the (palindromic) *iterator*, introduced in [7]; this data structure contains a string s and supports the following operations:

1. add(a) appends the letter a to the end of s;
2. maxPal returns the center of the longest suffix-palindrome of s;
3. rad(x) returns the radius of the longest subpalindrome of s with the center x;
4. nextPal(x) returns the center of the longest proper suffix-palindrome of the suffix-palindrome of s with the center x; undefined if x is not the center of a suffix-palindrome.

The iterator can be implemented so that all its operations work in $O(1)$ time (amortized, for add) [7, Prop. 1]. The same time bound applies to computing length of the longest subpalindrome centered at x: $\mathsf{len}(x) = 2 \cdot \mathsf{rad}(x) + \lfloor x \rfloor - \lfloor x - \frac{1}{2} \rfloor$. Still, the iterator alone cannot lower the asymptotic time of the naive algorithm; its improved version looks as follows:

```
1: add($s[n]$); ans[$n$] $\leftarrow +\infty$
2: for ($x \leftarrow$ maxPal; $x \neq n + \frac{1}{2}$; $x \leftarrow$ nextPal($x$)) do ans[$n$] $\leftarrow \min\{$ans[$n$]$, 1 +$ ans[$n -$ len($x$)]$\}$
```

2.2 Algorithm working in $O(n \log n)$ time

All subquadratic algorithms for palindromic length heavily use grouping of suffix-palindromes into *series*. Let $u_1, \ldots, u_k$ be all non-empty suffix-palindromes of a string s in the order of decreasing length. Since u_j is a suffix of u_i for any $i < j$, any period of u_i is a period of u_j; hence the sequence of minimal periods of $u_1, \ldots, u_k$ is non-increasing. The groups of suffix-palindromes with the same minimal period are *series of palindromes* (of s):

$$\underbrace{u_1, \ldots, u_{i_1}}_{p_1}, \underbrace{u_{i_1+1}, \ldots, u_{i_2}}_{p_2}, \ldots, \underbrace{u_{i_{t-1}+1}, \ldots, u_k}_{p_t}.$$

We refer to the longest and the shortest palindrome in a series as its *head* and *baby* respectively (they coincide in the case of a 1-element series); we enumerate the elements of a series from the head to the baby. Given an integer p, the *p-series* is the series with period p. A very useful observation [3, 5, 7] is that the length of a head is multiplicatively smaller than the length of the baby from the previous series, and thus every string of length n has $O(\log n)$ series. (As it was shown in [3], strings with $\Omega(\log n)$ series for $\Omega(n)$ prefixes do exist.)

The idea of the $O(n \log n)$ solution is to use the dynamic programming rule $\mathsf{ans}[n] = 1 + \min_U \min_{u \in U} \mathsf{ans}[n-|u|]$, where U runs through the series of s, and compute the internal minimum in $O(1)$ time using precalculations based on the structure of series. The structure of any series is described in the following lemma, which is easily implied by Lemmas 3, 4.

▶ **Lemma 5.** *For a string s and $p \geq 1$, let U be a p-series of palindromes. There exist $k \geq 1$ and unique palindromes u, v with $|uv| = p$, $v \neq \varepsilon$ such that one of three conditions hold:*

- *$U = \{(uv)^{k+1}u, (uv)^k u, \ldots, (uv)^2 u\}$ and the next series begins with uvu,*
- *$U = \{(uv)^k u, (uv)^{k-1} u, \ldots, uvu\}$ and the next series begins with u,*
- *$U = \{v^k, v^{k-1}, \ldots, v\}$, $p = 1$, $|v| = 1$, $u = \varepsilon$, and U is the last series for s.*

Let U be a p-series for $s[0..n]$ with $k > 1$ palindromes (w.l.o.g., $U = \{(uv)^k u, \ldots, uvu\}$). Updating $\mathsf{ans}[n]$ using this series, we compute $m = \min\{\mathsf{ans}[n{-}kp{-}|u|], \ldots, \mathsf{ans}[n{-}p{-}|u|]\}$. Now note that $s[0..n]$ ends with $(uv)^k u$ but not with $(uv)^{k+1}u$: otherwise, the latter string would belong to U. Then $s[0..n{-}p]$ ends with $(uv)^{k-1}u$ but not with $(uv)^k u$ and thus has the p-series $U' = \{(uv)^{k-1}u, \ldots, uvu\}$. Thus, at that iteration we computed $m' = \min\{\mathsf{ans}[n{-}kp{-}|u|], \ldots, \mathsf{ans}[n{-}2p{-}|u|]\}$ for updating $\mathsf{ans}[n{-}p]$. If we save m' into an auxiliary array, then $m = \min\{m', \mathsf{ans}[n - p - |u|]\}$ is computable in constant time, as required. Let us implement this construction using the iterator.

We start an iteration calling $\mathsf{add}(s[n])$. Let x be the center of a suffix-palindrome u. By Lemma 3, the minimal period p of u equals $\mathsf{len}(x) - \mathsf{len}(\mathsf{nextPal}(x))$. Let $\mathsf{cntr}(d)$ denote the center of the length d suffix-palindrome of $s[0..n]$ (i.e., $\mathsf{cntr}(d) = n - (d-1)/2$). Let $x' = \mathsf{cntr}(p + (\mathsf{len}(x) \bmod p))$. By Lemma 5, x' is the center either of the baby of the p-series or of the head of the next series, depending on the period $\mathsf{len}(x') - \mathsf{len}(\mathsf{nextPal}(x'))$ of this suffix-palindrome. All these computations take $O(1)$ amortized time using the iterator.

Our algorithm maintains an array $\mathsf{left}[1..n]$: for $p \in [1..n]$, if there is a p-series, then $s[\mathsf{left}[p]{+}1..n]$ is the longest suffix (which is not necessarily a palindrome) of $s[0..n]$ with period p; otherwise, $\mathsf{left}[p]$ is undefined. E.g., if $s[0..n] = \cdots aaabaaba$ and $p = 3$, then the mentioned suffix is $s[n{-}6..n] = aabaaba$ and $\mathsf{left}[3] = n - 7$. Computing $\mathsf{left}[p]$ in $O(1)$ time is done as follows. Let $w = (uv)^k u$ be the head of the p-series (see Lemma 5), x be the center of w, and $z(uv)^k u$ be the longest suffix of $s[0..n]$ with period p (in our example, $u = \varepsilon$, $v = aba$, $x = n - 5/2$, $z = a$). Then z is a proper suffix of uv. Hence $\mathsf{len}(x_1) = 2|z| + |u|$, where x_1 is the center of the prefix-palindrome u of w (in the example, $x_1 = n - 11/2$, $\mathsf{len}(x_1) = |aa|$). Note that $|u| = \mathsf{len}(x) \bmod p$ and $x_1 = 2x - x_2$, where $x_2 = \mathsf{cntr}(\mathsf{len}(x) \bmod p)$ is the center of the suffix u of w. Thus, $|z|$ and $\mathsf{left}[p] = n{-}\mathsf{len}(x){-}|z|$ are computed in $O(1)$ time.

All precalculated minimums are stored in an array $\mathsf{pre}[1..n]$, where each $\mathsf{pre}[p]$ is, in turn, an array $\mathsf{pre}[p][0..p{-}1]$ (we discuss in the next subsection why only $O(n)$ of possible $O(n^2)$ elements of pre are actually stored). For each j such that $n - j > \mathsf{left}[p]$, the string $s[0..n{-}j]$ usually has a suffix-palindrome with period p and thus can have a p-series; the array $\mathsf{pre}[p][0..p{-}1]$ contains the precalculations made for all these series. Formally,

$$\begin{aligned}&\mathsf{pre}[p][i] = \min\{\mathsf{ans}[t]:\\ &\quad (t - \mathsf{left}[p]) \bmod p = i \text{ and } s[t{+}1..n] \text{ has a prefix-palindrome of minimal period } p\};\end{aligned}$$

$\mathsf{pre}[p][i]$ is undefined if there is no such t (i.e., no p-series for the corresponding string). So if $u_1, \ldots, u_k$ is a p-series for $s[0..n]$, then $\mathsf{pre}[p][n{-}|u_1|{-}\mathsf{left}[p]] = \min\{\mathsf{ans}[n{-}|u_i|] : i \in [1..k]\}$. Hence, given a new letter $s[n]$, we compute $\mathsf{ans}[n]$ as follows:

```
1: add($s[n]$); ans[$n$] ← $+\infty$;
2: for ($x$ ← maxPal; $x \neq n + \frac{1}{2}$; $x$ ← nextPal(cntr($d$))) do      ▷ goes to next head each time
3:     $p$ ← len($x$) − len(nextPal($x$));                    ▷ min. period of the suf.-pal. centered at $x$
4:     $d$ ← $p$ + (len($x$) mod $p$);                              ▷ length of the baby in the $p$-series
5:     if len(cntr($d$)) − len(nextPal(cntr($d$))) ≠ $p$ then $d$ ← $d + p$;            ▷ corrected length
6:     compute left[$p$];                                                ▷ in $O(1)$ time, see above
7:     if len($x$) = $d$ then pre[$p$][$n$−len($x$)−left[$p$]] ← ans[$n-d$];
8:     pre[$p$][$n$−len($x$)−left[$p$]] ← min{pre[$p$][$n$−len($x$)−left[$p$]], ans[$n-d$]};
9:     ans[$n$] ← min{ans[$n$], 1 + pre[$p$][$n$−len($x$)−left[$p$]]};
```

Let $u_1, \ldots, u_k$ be a p-series, $i = n{-}|u_1|{-}\mathsf{left}[p]$. If $k = 1$, there was no p-series p iterations ago, so we set the undefined value $\mathsf{pre}[p][i]$ to $\mathsf{ans}[n{-}|u_k|]$ in line 7. Otherwise, by the definition of pre, we have $\mathsf{pre}[p][i] = \min\{\mathsf{ans}[n{-}|u_1|], \ldots, \mathsf{ans}[n{-}|u_{k-1}|]\}$. We update this value using $\mathsf{ans}[n{-}|u_k|]$ in line 8. So pre is correctly maintained, and the above algorithm computes the array ans in $O(n \log n)$ time due to logarithmic number of series.

2.3 Sketch of the linear algorithm

The idea of the linear solution is to perform the above log-time processing of all series of the current string not n times, but only $O(\frac{n}{\log n})$ times during the run of the algorithm. (However, we are able to make $\Theta(n)$ calls to the iterator.) To achieve this, during the processing of a series we replace each computation of the minimum $\mathsf{ans}[n] \leftarrow \min\{\mathsf{ans}[n], 1 + z\}$, for a precomputed value z from pre, with the simultaneous computation ("prediction") for the range of values $\mathsf{ans}[n..n{+}b]$, where $b = \lfloor\frac{\log n}{8}\rfloor$: we compute in advance $\mathsf{ans}[j] \leftarrow \min\{\mathsf{ans}[j], 1 + z_j\}$

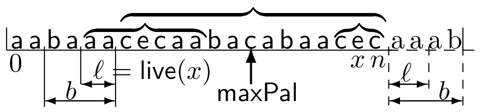


Figure 1 Predictable extensions.

for all $j \in [n..n{+}b]$ and corresponding precomputed z_j from pre. It is proved below that the arrays ans and pre can be organized so that, after a linear time preprocessing, such range operations on $O(b)$ elements of ans will take $O(1)$ time (this type of bit compression techniques is referred to as the four Russians' trick [1]).

Let us extend $s[0..n{-}1]$ with $s[n] = a$. We say that a suffix-palindrome u of $s[0..n{-}1]$, centered at x, *survives* if $s[0..n]$ has the suffix aua (i.e, x remains the center of a suffix-palindrome), and *dies* otherwise. We say that an extension of $s[0..n{-}1]$ by $s[n]$ is *predictable* if it retains maxPal, i.e., if the longest suffix-palindrome survives. From maxPal it can be calculated which of the other suffix-palindromes survive. If a suffix-palindrome of s centered at x survives $d \geq 0$ consecutive predictable extensions but dies after the $(d{+}1)$th such extension (or the $(d{+}1)$th predictable extension is not possible), we write $\mathsf{live}(x) = d$. We have $\mathsf{live}(\mathsf{maxPal}) = n - \mathsf{len}(\mathsf{maxPal})$ and $\mathsf{live}(x) = \mathsf{rad}(\mathsf{refl}(x)) - \mathsf{rad}(x)$ for $x \neq \mathsf{maxPal}$; here $\mathsf{refl}(x) = 2 \cdot \mathsf{maxPal} - x$ is the position symmetric to x w.r.t. maxPal. (See Fig. 1 for clarification; e.g., in Fig. 1 $\mathsf{live}(x) = 2$ and $\mathsf{live}(\mathsf{maxPal}) = 6$.)

Suppose that $\mathsf{ans}[n{+}j] = +\infty$ for $j \in [0..b]$. Having performed $\mathsf{add}(s[n])$, we get access to the suffix-palindromes of $s[0..n]$. If, for the center x of each such palindrome, we perform

$$\mathsf{ans}[n{+}j] \leftarrow \min\{\mathsf{ans}[n{+}j], 1 + \mathsf{ans}[n{-}\mathsf{len}(x){-}j]\} \text{ for all } j \in [0..\min\{b, \mathsf{live}(x)\}], \tag{1}$$

then we accumulate all information we can obtain from these palindromes during the next b predictable extensions. Thus we get an approximation of $\mathsf{ans}[n..n{+}b]$, which later will be updated using suffix-palindromes with the centers $x \geq n{+}\frac{1}{2}$. One phase of our algorithm is roughly as follows:

- append $s[n]$ to the iterator, update precalculations, and "predict" $\mathsf{ans}[n..n{+}b]$ with the assignments (1), using operations on blocks of bits ($\mathsf{ans}[n]$ is computed exactly);
- append subsequent letters, each time updating the predictions with either one or two new palindromes (after processing $s[n{+}j]$, $\mathsf{ans}[n..n{+}j]$ contains correct values);
- stop after b iterations or at the moment when an unpredicted letter is encountered;
- discard unused predictions and start a new phase with the first unpredicted letter.

For arrays α, β and numbers $i, j, \ell \geq 0$, denote by $\alpha[i..i{+}\ell] \overset{\min}{\leftarrow} \beta[j..j{+}\ell]$ the sequence of assignments $\alpha[i{+}k] \leftarrow \min\{\alpha[i{+}k], \beta[j{+}k]\}$ for all $k \in [0..\ell]$. Let $\mathsf{increv}(i, j)$ be the function returning an array $a[0..j{-}i]$ such that $a[k] = 1 + \mathsf{ans}[j{-}k]$ for $k \in [0..j{-}i]$ ("increment & reverse"). The predictions are made by the function $\mathsf{predict}$ that uses precalculations stored in pre to perform in a fast way the assignments $\mathsf{ans}[n..n{+}c] \overset{\min}{\leftarrow} \mathsf{increv}(n{-}\mathsf{len}(x){-}c, n{-}\mathsf{len}(x))$, where $c = \min\{b, \mathsf{live}(x)\}$, for all centers x of suffix-palindromes. (Hence $\mathsf{predict}$ computes the value $\mathsf{ans}[n]$ correctly even if $c = 0$ for some x.) Let $\mathsf{precalc}$ be a function that updates (possibly once in several iterations) the array pre to the actual state. The implementations of $\mathsf{predict}$ and $\mathsf{precalc}$ are discussed in Section 3. Our algorithm is as follows:

```
1: for ($n \leftarrow 0, end \leftarrow 0$; not($end\_of\_input$); $n \leftarrow n + 1$) do
2:     if $n = end$ or $\mathsf{len}(\mathsf{maxPal}) = n$ or $s[n] \neq s[n{-}\mathsf{len}(\mathsf{maxPal}){-}1]$ then      ▷ new phase
3:         $\mathsf{add}(s[n])$; $\mathsf{precalc}$; $\mathsf{predict}$; $end \leftarrow n + b$
4:     else $\mathsf{add}(s[n])$                              ▷ old phase continues, $s[n]$ is predictable
5:     $c \leftarrow \min\{b, \mathsf{live}(n)\}$; $\mathsf{ans}[n..n{+}c] \overset{\min}{\leftarrow} \mathsf{increv}(n{-}1{-}c, n{-}1)$
6:     if $s[n] = s[n{-}1]$ then $c \leftarrow \min\{b, \mathsf{live}(n{-}\frac{1}{2})\}$; $\mathsf{ans}[n..n{+}c] \overset{\min}{\leftarrow} \mathsf{increv}(n{-}2{-}c, n{-}2)$
```

This algorithm computes the same values $\mathsf{ans}[n]$ as the $O(n \log n)$ algorithm above, because finally all suffix-palindromes of $s[0..n]$ are used. So, the algorithm is correct.

Let t be the number of series in the current string $s[0..n]$ and q is the time required to perform all the calls $\mathsf{add}(s[n]), \mathsf{add}(s[n{-}1]), \dots, \mathsf{add}(s[n'{+}1])$, where $s[0..n']$ is the string for which $\mathsf{precalc}$ was called last time. Below we show that $\mathsf{predict}$ and $\mathsf{precalc}$ work in $O(t)$ and $O(t + q)$ time respectively, and the array ans can be organized so that the range operations in lines 5–6 can be performed in $O(1)$ time using the four Russians' trick. Let us estimate the running time of the algorithm under these assumptions.

During predictable extensions, line 3 is reached iff $n = \mathit{end}$, i.e., at most $O(\frac{n}{b})$ times. Since add works in $O(1)$ amortized time (see [7, Prop. 1]), the sum of all q's in the working time of $\mathsf{precalc}$ is $O(n)$. Since $O(t) = O(\log n)$, all predictable extensions take $O(n + \frac{n}{b}\log n) = O(n)$ overall time. To estimate the running time of unpredictable extensions, consider the value $\gamma_i = \mathsf{live}[\mathsf{maxPal}] = i - \mathsf{len}(\mathsf{maxPal})$ after processing $s[0..i]$. If $s[i{+}1]$ is predictable, one has $\gamma_{i+1} = (i + 1) - (\mathsf{len}(\mathsf{maxPal}) + 2) = \gamma_i - 1$. If $s[i{+}1]$ is unpredictable, $\gamma_{i+1} \ge (i + 1) - (\mathsf{len}(\mathsf{nextPal}(\mathsf{maxPal})) + 2)$; by Lemma 5, $\gamma_{i+1} - \gamma_i \ge p - 1$, where p is the minimal period of the longest suffix-palindrome of $s[0..i]$. By Lemmas 4 and 5, the length of the longest suffix-palindrome whose minimal period differs from p is less than $2p$. Therefore, $\mathsf{predict}$ and $\mathsf{precalc}$ take $O(p + q)$ time during this unpredictable extension (actually, $O(\log p + q)$). Since $\gamma_n - \gamma_1 < n$, the sum of the working times of all calls to $\mathsf{predict}$ and $\mathsf{precalc}$ is $O(n)$.

2.4 Organization of the arrays ans and pre

Informally, the four Russians' trick allows us to compute any operation on structures of size $\le \varepsilon \log n$ bits in $O(1)$ time using a precomputed table of size $O(n^{\varepsilon} \log^{O(1)} n)$ bits. For example, let a $\lfloor \frac{\log n}{2} \rfloor$-bit integer x encode a sequence $x_1, \dots, x_{\lfloor \log n/4 \rfloor}$ so that $x_j = 1 - (\lfloor x/2^{2j-2} \rfloor \bmod 4)$, i.e., $(2j{-}1)$th and $(2j{-}2)$th bits of x encode x_j. We can compute, for $j \in [1..\log n/4]$, the sum $x_1 + \cdots + x_j$ in $O(1)$ time using a table $T[0..\lfloor\sqrt{n}\rfloor][1..\lfloor \log n/4 \rfloor]$ such that $T[x][j] = x_1 + \cdots + x_j$ for any $x \in [0..2^{\log n/2}] = [0..\sqrt{n}]$ and $j \in [1..\log n/4]$. The table T can be precomputed in $O(\sqrt{n} \log^{O(1)} n)$ time.

In our case, we split ans into blocks of length b. By Lemma 2, adjacent elements of ans differ by at most one. This allows us to encode each block $\mathsf{ans}[ib{+}1..(i{+}1)b]$ as the number $\mathsf{ans}[ib{+}1]$ and the sequence $x_1, x_2, \dots, x_b$ such that $x_j \in \{-1, 0, 1\}$ and $\mathsf{ans}[ib{+}j] = \mathsf{ans}[ib{+}1] + x_1 + \cdots + x_j$ for any $j \in [1..b]$. This sequence $x_1, x_2, \dots, x_b$ is encoded in a $2b$-bit integer exactly as in the example above (note that $2b \le \lfloor \frac{\log n}{4} \rfloor$). Using a precomputed table of size $O(\sqrt[4]{n}b)$, we can extract any element $\mathsf{ans}[j]$ in $O(1)$ time. It is shown in Sect. 3 that arrays in pre can be encoded in a similar way (with some additional complications).

Applying a similar trick, one can perform many other operations. Let $c[0..\ell]$ be an array of integers such that $\ell \in [0..b]$, $|c[i{-}1] - c[i]| \le 1$, and c is encoded, like a block of ans, by $c[0]$ and a $2b$-bit integer. Let us show how to perform in $O(1)$ time the operation $\mathsf{ans}[n..n{+}\ell] \overset{\min}{\leftarrow} c[0..\ell]$ as in lines 5–6 of the algorithm (similar operations are also performed in $\mathsf{predict}$). We first check whether $c[0] > \mathsf{ans}[n] + 2\ell$: if so, then ans remains unchanged. It is guaranteed by the algorithm that $c[0] \ge \mathsf{ans}[n{-}1] - 1$. Then, we concatenate bit representations of all required components: the (at most) two blocks $\mathsf{ans}[ib{+}1..(i{+}1)b]$ and $\mathsf{ans}[(i{+}1)b{+}1..(i{+}2)b]$ encoding the subarray $\mathsf{ans}[n..n{+}\ell]$ are stored as two $2b$-bit sequences (encoding the differences $\mathsf{ans}[i] - \mathsf{ans}[i{-}1]$ for $i \in [ib{+}2..(i{+}2)b]$ as above), $c[0..\ell]$ is also stored as a $2b$-bit sequence, the offset $(n{-}ib)$ and the difference $c[0] - \mathsf{ans}[n{-}1]$ are stored as $O(\log b)$-bit integers; $6b + O(\log b)$ bits in total. We precompute a table T that, for a given combined bit representation, stores two $2b$-bit sequences encoding two blocks that represent the resulting modified $\mathsf{ans}[n..n{+}\ell]$ array. It should be noted that the information provided in the given representation suffices to compute the result and, since the resulting array ans satisfies Lemma 2, we may put $\mathsf{ans}[n{+}\ell{+}j] = \min\{\mathsf{ans}[n{+}\ell{+}j], \mathsf{ans}[n{+}\ell] + j\}$ for $j \geq 1$ so that the structure of the last block is preserved. Since $6b {\leq} \frac{3}{4}\log n$, the size of the table T is $O(b \cdot 2^{6b+O(\log b)}) = O(n^{3/4}\log^k n)$ for some $k = O(1)$. Obviously, T can be precomputed in $O(n^{3/4}\log^{k+O(1)} n)$ time. Analogously, we precompute tables that allow us to calculate $\mathsf{increv}(i, j)$ in $O(1)$ time if $j - i \leq b$; the resulting array of increv is encoded, like the array c, by the first element and a $2b$-bit integer. Thus, all range operations in lines 5–6 of the algorithm can be performed in $O(1)$ time.

We use a number of different range operations on the arrays ans and pre in Section 3 but all of them are similar to the discussed ones, so we omit detailed descriptions.

3 Implementation of the Main Functions

Now it remains to describe the functions $\mathsf{predict}$ and $\mathsf{precalc}$ and prove their time complexity.

3.1 Function predict

At the beginning, the function $\mathsf{predict}$ sets $\mathsf{ans}[n{+}j] \leftarrow \mathsf{ans}[n{-}1]{+}j{+}1$ for $j \in [0..b]$. By Lemma 2, the assigned values are upper bounds for the elements of $\mathsf{ans}[n..n{+}b]$. The assignments are performed in $O(1)$ time using range operations. Then $\mathsf{predict}$ processes each of the t series; let us describe precisely how we process a p-series $u_1, \ldots, u_k$.

Let u, v be the palindromes described in Lemma 5, x_i be the center of u_i for $i \in [1..k]$. If $\mathsf{len}(x_i) < n - \mathsf{left}[p]$ (i.e., either $i > 1$ or $i = 1$ and u_1 is not the longest suffix of s with period p), then x_i will remain the center of a new suffix-palindrome after the appending of $s[n]$ iff $s[n] = s[n{-}p] = v[0]$. In this case, the period p "extends" and x_i remains the center of a suffix-palindrome with the minimal period p. In the remaining case $\mathsf{len}(x_1) = n - \mathsf{left}[p]$ (u_1 is the longest suffix with period p) x_1 will remain the center of a suffix-palindrome iff $s[n] = s[\mathsf{left}[p]]$; the period p breaks and the palindrome $s[n]u_1s[n]$ will belong to a different series.

Suppose that d upcoming predictable extensions extend the period p of the suffix $s[\mathsf{left}[p]{+}1..n{-}1]$ and the $(d{+}1)$st predictable extension breaks this period. It follows from the previous paragraph that the only suffix-palindrome u_i that can survive the $(d{+}1)$st extension (in other words, for which $\mathsf{live}(x_i) > d$) must have length $n - \mathsf{left}[p] - d$ (see Fig. 2). So if d is known, we check whether $x = \mathsf{cntr}(n{-}\mathsf{left}[p]{-}d)$ is the center of a suffix-palindrome (i.e., $\mathsf{cntr}(\mathsf{len}(x)) = x$) and, if so, we compute $\mathsf{ans}[n..n{+}c] \overset{\min}{\longleftarrow} \mathsf{increv}(n{-}\mathsf{len}(x){-}c, n{-}\mathsf{len}(x))$, where $c = \min\{b, \mathsf{live}(x)\}$, in $O(1)$ time using range operations.

Now it remains to find d and change $\mathsf{ans}[n..n{+}\min\{b, d\}]$ taking $u_1, \ldots, u_k$ into account. Since predictable extensions append the letters $s[n{-}\mathsf{len}(\mathsf{maxPal})], s[n{-}\mathsf{len}(\mathsf{maxPal}){-}1], \ldots$ to the right of s, we can approximately find d looking at the string $s[0..n{-}\mathsf{len}(\mathsf{maxPal}){-}1]$. Put $d' = \min\{\mathsf{live}(\mathsf{cntr}(|u|)), \mathsf{live}(\mathsf{cntr}(|uvu|))\}$ (see Fig. 2). Let us show that we can use d' instead of d. If $d' < n - \mathsf{left}[p] - |uvu|$, then the longest suffix-palindrome is preceded by the reversed prefix of $(vu)^\infty$ of length d'. In turn, this prefix either is preceded by a letter that breaks the period p of this prefix (the letter e in Fig. 2) or is a prefix of the whole string. In either case, $d' = d$. If $d' \geq n - \mathsf{left}[p] - |uvu|$, then the longest suffix-palindrome is also preceded by the reversed prefix of $(vu)^\infty$ of length d' but $d \geq d'$ in general. However, even in this case, we can use d' in the sequel since none of the suffix-palindromes from our series survives after $n - \mathsf{left}[p] - |uvu|$ predictable extensions; therefore, also, the possible processing of a suffix-palindrome of length $n - \mathsf{left}[p] - d'$ mentioned above is not required.

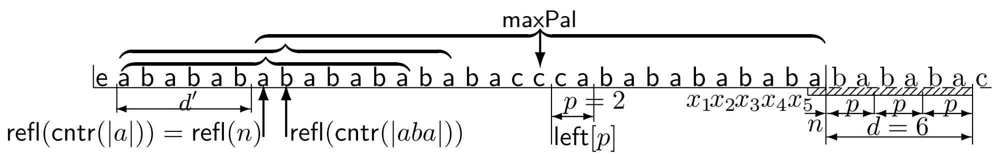

maxPal
e a b a b a b a b a b a b a b a b a c c c a b a b a b a b a b a b a b a b a c
d'
refl(cntr($|a|$)) = refl(n)
refl(cntr($|aba|$))
$p = 2$
left[p]
$x_1x_2x_3x_4x_5$
n
p
p
p
$d = 6$

Figure 2 A series $x_1, \dots, x_5$: $\mathsf{live}(x_4) > d$, the shaded region corresponds to $\mathsf{ans}[n..n{+}d]$.

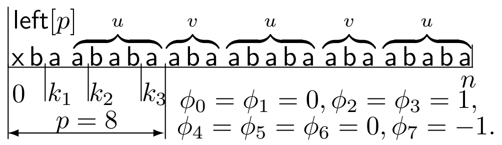

left[p]
u v u v u
x b a a b a b a a b a a b a b a a b a a b a b a
0 k_1 k_2 k_3
$p = 8$
n
$\phi_0 = \phi_1 = 0, \phi_2 = \phi_3 = 1,$
$\phi_4 = \phi_5 = \phi_6 = 0, \phi_7 = -1.$

Figure 3 Partition of $[0..p)$ in Lemma 6.

Let us track the set $S = \{\ell_1 = n{-}\mathsf{len}(x_1){+}1, \dots, \ell_k = n{-}\mathsf{len}(x_k){+}1\}$ of the leftmost positions of the suffix-palindromes centered at $x_1, x_2, \dots, x_k$ in the d' predictable extensions: all these positions shift to the left by one after each extension; if a position reaches $\mathsf{left}[p]$, the corresponding palindrome dies and this position is excluded from S. By Lemma 3, for any $i \in [1..k]$, if ℓ_i is in the set after $f \in [0..d']$ predictable extensions, then the suffixes $s[\ell_i{+}jp..n]$ (here n is increased by f) are palindromes for all integers $j \ge 0$ such that $\ell_i{+}jp \le n$; therefore, along with the assignments $\mathsf{ans}[n] \overset{\min}{\leftarrow} 1 + \mathsf{ans}[\ell_i{-}1]$ (here n is increased by f) that we are intended to perform, we can occasionally perform $\mathsf{ans}[n] \overset{\min}{\leftarrow} 1 + \mathsf{ans}[\ell_i{+}jp{-}1]$ for any such j.

Obviously, $|u_1| + p > n - \mathsf{left}[p]$ since otherwise uvu_1 would be a longer suffix-palindrome with the minimal period p. Based on the above observation, we perform the assignments $\mathsf{ans}[n{+}j] \overset{\min}{\leftarrow} 1{+}\mathsf{pre}[p][r(j)]$ for all $j \in [0..\min\{b, d'\}]$, where $r(j) = (n{-}|u_k|{-}\mathsf{left}[p]{-}j) \bmod p$ (see Fig. 3; $r(j)$ cyclically runs through the range $[0..p)$ from right to left when j increases). Recall that, immediately before the execution of $\mathsf{predict}$, the function $\mathsf{precalc}$ recalculates the array pre. After this recalculation $\mathsf{pre}[p]$ stores an array $\mathsf{pre}[p][0..p{-}1]$ for each $p \in [1..n]$ such that p is the minimal period of a suffix-palindrome of $s[0..n]$. For $i \in [0..p)$ we have $\mathsf{pre}[p][i] = \min\{\mathsf{ans}[\mathsf{left}[p]{+}i{+}jp] \colon j \in [0..\phi_i]\}$, where $\phi_i \ge 0$ is the maximal integer such that the string $s[\mathsf{left}[p]{+}i{+}\phi_i p{+}1..n]$ has a prefix-palindrome with the minimal period p; if there is no such ϕ_i, we put $\mathsf{pre}[p][i] = +\infty$ and $\phi_i := -1$.

We perform $\mathsf{ans}[n{+}j] \overset{\min}{\leftarrow} 1 + \mathsf{pre}[p][r(j)]$, for all $j \in [0..\min\{b, d'\}]$, in $O(1)$ time using range operations on the arrays pre and ans. (These operations are discussed below.) It follows from Lemma 5 that, after $f \in [0..d']$ predictable extensions, the strings $s[\ell_i..n']$ (here n' denotes the value of n before the extensions), for $i \in [1..k)$ such that ℓ_i is still in the set S, have prefix-palindromes with the minimal period p. Therefore, the above assignments will really process the palindromes $u_1, \dots, u_{k-1}$ for the upcoming d' predictable extensions (see Fig. 2) but will, probably, perform some additional unnecessary assignments for suffix-palindromes with period p that will appear only after a number of predictable extensions; but this does not harm since such assignments will be performed anyway in the future. For the baby u_k, we compute explicitly $\mathsf{ans}[n..n{+}c] \overset{\min}{\leftarrow} \mathsf{increv}(n{-}\mathsf{len}(x_k){-}c, n{-}\mathsf{len}(x_k))$, where $c = \min\{b, \mathsf{live}(x_k)\}$, in $O(1)$ time using range operations. It remains to describe the structure of the array pre that allows us to perform constant time range operations on subarrays of length $\le b$.

▶ **Lemma 6.** *For each $i \in [0..p)$, let ϕ_i be the minimal integer such that the string $s[\mathsf{left}[p]{+}i{+}(\phi_i{+}1)p{+}1..n]$ has no prefix-palindromes with the minimal period p. Then, the segment $[0..p)$ can be split into subsegments $[k_0..k_1), \dots, [k_6..k_7)$, for $0 = k_0 \le \dots \le k_7 = p$, such that, for $i \in (0..p)$, we have $\phi_i = \phi_{i-1}$ whenever i and $i{-}1$ belong to the same subsegment (see Fig. 3).*

Proof. For $i \in [0..p)$ and $j \geq 0$, denote $i(j) = \mathsf{left}[p]{+}i{+}jp$. Let k be an integer such that, for $i = p-1$, we have $i(k) < n$ and $i(k) + p \geq n$. So, for $i \in [0..p)$, we obtain $\phi_i = j'_i - 1$, where j'_i is the minimal integer such that $j'_i \in [0..k]$ and the string $s[i(j'_i){+}1..n]$ has no prefix-palindromes with the minimal period p. While i descends from $p{-}1$ to 0 with step one, some of the suffixes $s[i(j){+}1..n]$ may acquire prefix-palindromes with the minimal period p and some may lose such prefix-palindromes thus changing the value of ϕ_i (see Fig. 3).

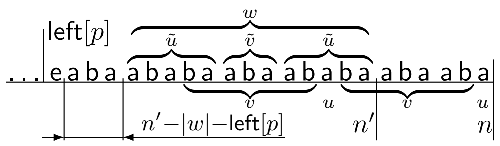
Let us choose $i \in [0..p)$ that maximizes the value of ϕ_i. Denote $j' = \phi_i$ for this i. If $j' \geq 0$, then $s[i(j'){+}1..n]$ has a prefix-palindrome w with the minimal period p; by Lemma 3, there are palindromes u and v such that $|uv| = p$ and $w = (uv)^r u$ for $r \geq 1$. Thus, for any $j'' \in [0..j'{-}2]$, the suffix $s[i(j''){+}1..n]$ has prefix-palindromes $(uv)^3u$ and $(uv)^2u$ both having the minimal period p. When i further decreases to 0, the prefix-palindrome $(uv)^2u$ "grows" together with $s[i(j''){+}1..n]$ and, when i increases, $(uv)^3u$ "shrinks"; in both cases $s[i(j''){+}1..n]$ retains a prefix-palindrome with the minimal period p while $i \in [0..p)$. Hence, only suffixes $s[i(j'{-}1){+}1..n]$ and $s[i(j'){+}1..n]$ may lose or acquire a prefix-palindrome with the minimal period p while i changes from $p{-}1$ to 0, i.e., ϕ_i varies in the range $[j'{-}2..j']$.

Let us prove that any suffix $s[i(j){+}1..n]$ can lose a prefix-palindrome with the minimal period p at most once during the descending of i from $p{-}1$ to 0. Then, the existence of the desired numbers $k_0, k_1, \ldots, k_7$ follows from a simple analysis of possible cases.

Suppose that $s[i(j){+}1..n]$ has a prefix-palindrome centered at x with the minimal period p. When i decreases, $s[i(j){+}1..n]$ grows and the prefix-palindrome "grows" simultaneously. Then, before $s[i(j){+}1..n]$ loses the prefix-palindrome, we have $|s[i(j){+}1..n]| = \mathsf{len}(x)$ for some $i \in [0..p)$. By Lemma 4, there are palindromes u' and v' such that $|u'v'| = p$ and $s[n{-}\mathsf{len}(x){+}1..n] = (u'v')^{r'}u'$ for $r' \geq 1$. If, for some smaller $i \in [0..p)$, $s[i(j){+}1..n]$ again acquires a prefix-palindrome with the minimal period p, then, by Lemma 4, the center x' of this prefix-palindrome must coincide with the center of u' or v' from the decomposition. Hence $x' \leq x - p/2$. Then, this prefix-palindrome can be lost only after p decrements of i once we have had $|s[i(j){+}1..n]| = \mathsf{len}(x)$. This proves the claim. ◀

We partition $\mathsf{pre}[p][0..p{-}1]$ into subarrays $\mathsf{pre}[p][k_0..k_1{-}1], \ldots, \mathsf{pre}[p][k_6..k_7{-}1]$ according to Lemma 6. Consider a segment $[a..b] \subset [0..p)$ such that $\phi_{i_1} = \phi_{i_2}$ and $\phi_{i_1} \neq -1$ whenever $i_1, i_2 \in [a..b]$. Since $\mathsf{pre}[p][i] = \min\{\mathsf{ans}[\mathsf{left}[p]{+}i{+}jp] : j \in [0..\phi_i]\}$ and, by Lemma 2, $|\mathsf{ans}[j] - \mathsf{ans}[j{-}1]| \leq 1$ for any $j \in (0..n)$, we easily obtain $|\mathsf{pre}[p][i] - \mathsf{pre}[p][i{-}1]| \leq 1$ for any $i \in (a..b]$. Therefore, by Lemma 6, each of the subarrays of pre either contains only $+\infty$ or has a structure similar to the structure of ans described in Lemma 2. We do not store the subarrays that contain $+\infty$ and encode all other subarrays in a way described for ans in Sect. 2.4: we split them into blocks of length b and encode each block as its starting element and a $2b$-bit integer encoding the differences between adjacent elements (the last block may contain less than b elements). The linear size of pre measured in machine words (but not in the number of elements) follows from the overall linear running time of the function $\mathsf{precalc}$ maintaining pre; this analysis is given below.

To perform $\mathsf{ans}[n{+}j] \overset{\min}{\leftarrow} \mathsf{pre}[p][r(j)]$ for all $j \in [0..\min\{b, d'\}]$, we concatenate $2b$-bit integers from the blocks covering the subarray $\mathsf{ans}[n..n{+}\min\{b, d'\}]$, $2b$-bit integers from a constant number of blocks covering the subarrays of $\mathsf{pre}[p][0..p{-}1]$ containing positions $r(j)$ for $j \in [0..\min\{b, d'\}]$, and some other lightweight auxiliary data similar to the data used in the operation $\overset{\min}{\leftarrow}$ considered above; then we compute the resulting array $\mathsf{ans}[n..n{+}\min\{b, d'\}]$ using the obtained bit string and a precomputed table of size $o(n)$. This might require to duplicate the content of $\mathsf{pre}[p]$ if $p < \min\{b, d'\}$ (see the shaded region in Fig. 2); these duplications must be already precalculated in the tables. Note that thus defined changes of ans may affect the whole subarray $\mathsf{ans}[n..n{+}b]$ and not only $\mathsf{ans}[n..n{+}\min\{b, d'\}]$: e.g., if we



left[p]
w
$\tilde{u}$ $\tilde{v}$ $\tilde{u}$
. . . e a b a a b a b a a b a a b a b a a b a a b a
v u v u
$n'-|w|-$left[p]
n'
n

Figure 4 Palindrome w with the center x, the minimal period $p = 8$; for $i = 1, 2, 3$, $j = 0$.

perform ans[n] $\overset{\min}{\leftarrow}$ x, then, to maintain the property of ans described in Lemma 2, we must also perform ans[$n{+}j$] $\overset{\min}{\leftarrow}$ $x + j$ for $j \in [1..b]$ (it is guaranteed by the algorithm that the elements of ans[$0..n{-}1$] cannot be affected analogously since always $|$ans[$n{-}1$] $-$ ans[n]$| \le 1$). Similar "normalizations" must be included in the precomputed assignments ans[$n{+}j$] $\overset{\min}{\leftarrow}$ pre[p][$r(j)$] for all $j \in [0..\min\{b, d'\}]$. Thus, the structure of ans is preserved.

The computations seem to be quite sophisticated but, nevertheless, since all involved structures occupy $\varepsilon \log n$ bits, for $\varepsilon < 1$, all required precalculations can be performed in $O(n^{\varepsilon} \log^{O(1)} n)$ time at the beginning of our algorithm. The tedious details are omitted here and can be retrieved from the implementation [9].

3.2 Function precalc

Denote by n' the value of n at the moment of the last call of precalc. (The first call of precalc for $n = 0$ is trivial.) Our goal is to compute the array pre[p][$0..p{-}1$] for each p for which there exists a p-series in $s[0..n]$. Note that since, as described above, pre[p][$0..p{-}1$] is stored as a constant number of pointers to subarrays containing non-infinite values, we can fill pre[p][$0..p{-}1$] with $+\infty$ in $O(1)$ time simply removing all these pointers.

The function precalc loops through all series in $s[0..n]$ and processes each p-series as follows: precalc computes the new value of left[p] in $O(1)$ time and, if left[p] has changed since $s[0..n']$ (this is where we really use the array left), then fills pre[p][$0..p{-}1$] with $+\infty$ in $O(1)$ time; otherwise, precalc uses the array pre[p][$0..p{-}1$] calculated for $s[0..n']$. In either case, for each $i \in [0..p)$, if there is an integer $j \ge 0$ such that s[left[p]$+i+jp..n'$] does not have a prefix-palindrome with the minimal period p and s[left[p]$+i+jp..n$] has such a prefix-palindrome, then pre[p][i] is updated by performing pre[p][i] $\overset{\min}{\leftarrow}$ ans[left[p]$+i+jp-1$]. The methods by which we find such $i \in [0..p)$ and really perform the later assignments are described below. It follows from the definition of pre that thus defined precalc computes the arrays pre[p][$0..p{-}1$] for $s[0..n]$.

Let us process all centers x for which there are $i \in [0..p)$ and $j \ge 0$ such that x is the center of a prefix-palindrome of s[left[p]$+i+jp..n$] with the minimal period p but s[left[p]$+i+jp..n'$] does not have a prefix-palindrome with the minimal period p. We consider two cases.

Case 1. Suppose that such x is less than $n'{+}1$ and the longest subpalindrome w in $s[0..n']$ centered at x has the minimal period p. Clearly, the leftmost position of w is greater than left[p] $+ i + jp$ and w must be a suffix-palindrome of $s[0..n']$. Let us describe all positions $h_m = n'{-}|w|{-}m$ such that x is the center of a prefix-palindrome of $s[h_m{+}1..n]$ and is not the center of a prefix-palindrome of $s[h_m{+}1..n']$. Obviously $m > 0$. After $n'{-}|w|{-}$left[p]$+1$ extensions of $s[0..n']$, the suffix-palindrome centered at x dies because it reaches left[p] by its leftmost position (see Fig. 4). So, since w grows at most $n - n'$ times, we obtain $m \in [1..\min\{n{-}n', n'{-}|w|{-}$left[$p$]$\}]$. For each such m, the prefix-palindrome of $s[h_m{+}1..n]$ centered at x has length $|w| + 2m$ and the minimal period p since the minimal period of w, centered at x, is p and the palindrome with the length $|w| + 2m$ and the center x, for the given m, is a substring of the suffix of s with period p. Hence, we can perform

$\mathsf{pre}[p][(r{-}m) \bmod p] \overset{\min}{\leftarrow} \mathsf{ans}[n'{-}|w|{-}m]$, where $r = n' - |w| - \mathsf{left}[p]$, for all such m. Among these assignments there is the required $\mathsf{pre}[p][i] \overset{\min}{\leftarrow} \mathsf{ans}[\mathsf{left}[p]{+}i{+}jp{-}1]$ (see Fig. 4). Since $n - n' \le b$, once x is known, we can perform all these assignments in $O(1)$ time using range operations and precomputed tables; the boundaries of subarrays of pre can be adjusted appropriately after these calculations. Now it remains to find all such centers x.

By Lemma 3, there are palindromes $\tilde{u}$ and $\tilde{v}$ such that $p = |\tilde{u}\tilde{v}|$ and $w = (\tilde{u}\tilde{v})^r\tilde{u}$ for $r \ge 1$ (see Fig. 4). If the minimal period of $(\tilde{u}\tilde{v})^{r-1}\tilde{u}$ is p, then all strings $s[h..n']$, for $h \in (\mathsf{left}[p]..n'{-}|w|]$, have prefix-palindromes of the form $\alpha(\tilde{u}\tilde{v})^{r-1}\tilde{u}\overleftarrow{\alpha}$, where α is a suffix of $\tilde{u}\tilde{v}$, with the minimal period p. But, by our assumption, $s[\mathsf{left}[p]{+}i{+}jp..n']$ cannot have such a prefix-palindrome. Therefore, w is the baby in the p-series of the string $s[0..n']$, i.e., either $w = \tilde{u}\tilde{v}\tilde{u}$ or $w = \tilde{u}\tilde{v}\tilde{u}\tilde{v}\tilde{u}$. We find the baby in $O(1)$ time by the techniques described above using an instance of the iterator and the list of all series of suffix-palindromes for the string $s[0..n']$; these iterator and list are further discussed below.

Case 2. It remains to detect all x such that x is the center of a prefix-palindrome of $s[\mathsf{left}[p]{+}i{+}jp..n]$ with the minimal period p, for some $i \in [0..p)$ and $j \ge 0$, and either $x > n'$ or the minimal period of any subpalindrome in $s[0..n']$ centered at x is not p. Hence, a subpalindrome with the minimal period p and the center x appeared after several extensions of $s[0..n']$ and, thus, was a suffix-palindrome at that moment. To catch the moments when growing suffix-palindromes acquire new minimal periods, we need a device tracking changes of periods in all suffix-palindromes after extensions. The iterator can serve as such a device.

Let w be a suffix-palindrome of $s[0..n']$ with the minimal period p'. By Lemma 3, we have $p' = |w| - |u|$, where u is the longest proper suffix-palindrome of w. Suppose that $s[0..n']$ is extended by the letter $a = s[n'{+}1]$ and awa is a suffix-palindrome of the new string. By Lemma 3, awa has period p' iff aua is a suffix-palindrome of $s[0..n'{+}1]$. Thus, to detect new suffix-palindromes with a given period p, we can track, during the extensions of s, changes in the list of the centers of all suffix-palindromes. The iterator maintains such list. The following lemma is a straightforward corollary of the proof of [7, Prop. 1].

▶ **Lemma 7.** *The iterator maintains a linked list of the centers of all suffix-palindromes of $s[0..n]$. The function $\mathsf{add}(a)$ removes a number of centers from the list, adds the centers $n{+}\frac{1}{2}$ (if $a = s[n]$) and $n{+}1$ to the end of the list, and thus obtains a new list for the string $s[0..n]a$; all in $\Omega(1 + c)$ time, where c is the number of removed centers.*

We maintain an instance of the iterator for the previously processed string $s[0..n']$ and store the list of the centers of all suffix-palindromes of $s[0..n']$ since the last call of $\mathsf{precalc}$. The function $\mathsf{precalc}$ performs $\mathsf{add}(s[n'{+}1]), \dots, \mathsf{add}(s[n])$ and thus consecutively obtains the lists of the centers of all suffix-palindromes of $s[0..n'{+}1], \dots, s[0..n]$.

Consider, for $n'' \in (n'..n]$, such list $x_1, \dots, x_k$ for $s[0..n''{-}1]$ so that $x_1 < \dots < x_k$. By Lemma 7, the call to $\mathsf{add}(s[n''])$ gives us a sublist $x_{i_1}, \dots, x_{i_c}$ of the centers removed from $x_1, \dots, x_k$. By Lemma 3, for $x_i \notin \{x_{i_1}, \dots, x_{i_c}\}$ the minimal period of the suffix-palindrome with the center x_i has changed iff $x_{i+1} \in \{x_{i_1}, \dots, x_{i_c}\}$. We easily find all such x_i parsing the list $x_{i_1}, \dots, x_{i_c}$ from left to right and compute the new period as $p = \mathsf{len}(x_i) - \mathsf{len}(\mathsf{nextPal}(x_i))$. Denote by ℓ the number that is equal to $\mathsf{len}(x_i)$ for $s[0..n'']$. By the definition of pre, we must perform $\mathsf{pre}[p][r] \overset{\min}{\leftarrow} \mathsf{ans}[n''{-}\ell]$, where $r = (n'' - \ell - \mathsf{left}[p]) \bmod p$, if the string $s[0..n]$ has a p-series. In this case, we must also perform $\mathsf{pre}[p][(r{-}m) \bmod p] \overset{\min}{\leftarrow} \mathsf{ans}[n''{-}\ell{-}m]$ for all $m \in [0..\min\{n - n'', n'' - \ell - \mathsf{left}[p]\}]$ because the strings $s[n''{-}\ell{-}m..n]$ have prefix-palindromes of length $\ell{+}2m$ centered at x_i with the minimal period p; after $n'' - \ell - \mathsf{left}[p] + 1$ extensions, such palindrome dies since it reaches $\mathsf{left}[p]$ by its leftmost position and thus its period breaks. Since $n - n'' \le b$, these assignments, for all such m, can be performed by range operations on pre and ans in $O(1)$ time using precomputed tables like those described in Sect. 2.4 (subarrays of pre$[p]$ can be also adjusted appropriately).

Thus, precalc works in $O(t+q)$ time as required, where t is the number of series in $s[0..n]$ and q is the time required to perform the sequence of calls add$(s[n'+1]), \ldots,$ add$(s[n])$. This finishes the proof of the linear time complexity of main algorithm.

References

1 Vladimir Arlazarov, Efim Dinic, Mikhail Kronrod, and Igor Faradzev. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR*, 194(11):1209–1210, 1970.

2 Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Proc. of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *LNCS*, pages 55–69. Springer, 2003. doi:10.1007/3-540-44888-8_5.

3 Gabriele Fici, Travis Gagie, Juha Kärkkäinen, and Dominik Kempa. A subquadratic algorithm for minimum palindromic factorization. *J. Discrete Algorithms*, 28:41–48, 2014. doi:10.1016/j.jda.2014.08.001.

4 Zvi Galil and Joel I. Seiferas. A linear-time on-line recognition algorithm for “palstar”. *J. ACM*, 25(1):102–111, 1978. doi:10.1145/322047.322056.

5 Tomohiro I, Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing palindromic factorizations and palindromic covers on-line. In Alexander S. Kulikov, Sergei O. Kuznetsov, and Pavel A. Pevzner, editors, *Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM 2014)*, volume 8486 of *LNCS*, pages 150–161. Springer, 2014. doi:10.1007/978-3-319-07566-2_16.

6 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

7 Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Pal^k is linear recognizable online. In Giuseppe F. Italiano, Tiziana Margaria-Steffen, Jaroslav Pokorný, Jean-Jacques Quisquater, and Roger Wattenhofer, editors, *Proceedings of the 41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2015)*, volume 8939 of *LNCS*, pages 289–301. Springer, 2015. doi:10.1007/978-3-662-46078-8_24.

8 Glenn K. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.

9 Palindromic Length. Sources and tests for linear palindromic length problem, 2017. URL: https://github.com/kborozdin/palindromic-length.

10 Mikhail Rubinchik and Arseny M. Shur. EERTREE: an efficient data structure for processing palindromes in strings. In Zsuzsanna Lipták and William F. Smyth, editors, *Proc. of the 26th International Workshop on Combinatorial Algorithms (IWOCA 2015)*, volume 9538 of *LNCS*, pages 321–333. Springer, 2015. doi:10.1007/978-3-319-29516-9_27.

11 Anatol O. Slisenko. A simplified proof of the real-time recognizability of palindromes on turing machines. *J. Soviet. Math.*, 15(1):68–77, 1977. doi:10.1007/BF01404109.

Tight Bounds on the Maximum Number of Shortest Unique Substrings*

Takuya Mieno[1], Shunsuke Inenaga[2], Hideo Bannai[3], and Masayuki Takeda[4]

1 **Department of Informatics, Kyushu University, Japan**
`takuya.mieno@inf.kyushu-u.ac.jp`
2 **Department of Informatics, Kyushu University, Japan**
`inenaga@inf.kyushu-u.ac.jp`
3 **Department of Informatics, Kyushu University, Japan**
`bannai@inf.kyushu-u.ac.jp`
4 **Department of Informatics, Kyushu University, Japan**
`takeda@inf.kyushu-u.ac.jp`

Abstract

A substring Q of a string S is called a shortest unique substring (SUS) for interval $[s,t]$ in S, if Q occurs exactly once in S, this occurrence of Q contains interval $[s,t]$, and every substring of S which contains interval $[s,t]$ and is shorter than Q occurs at least twice in S. The SUS problem is, given a string S, to preprocess S so that for any subsequent query interval $[s,t]$ all the SUSs for interval $[s,t]$ can be answered quickly. When $s = t$, we call the SUSs for $[s,t]$ as *point SUSs*, and when $s \leq t$, we call the SUSs for $[s,t]$ as *interval SUSs*. There exist optimal $O(n)$-time preprocessing scheme which answers queries in optimal $O(k)$ time for both point and interval SUSs, where n is the length of S and k is the number of outputs for a given query. In this paper, we reveal structural, combinatorial properties underlying the SUS problem: Namely, we show that the number of intervals in S that correspond to point SUSs for all query positions in S is less than $1.5n$, and show that this is a matching upper and lower bound. Also, we consider the maximum number of intervals in S that correspond to interval SUSs for all query intervals in S.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases shortest unique substrings, maximal unique substrings

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.24

1 Introduction

1.1 Shortest unique substring (SUS) problems

A substring Q of a string S is called a *shortest unique substring* (*SUS*) for interval $[s,t]$ in S, if (1) Q occurs exactly once in S, (2) this occurrence of Q contains interval $[s,t]$, and (3) every substring of S which contains interval $[s,t]$ and is shorter than Q occurs at least twice in S. The *SUS problem* is to preprocess a given string S so that for any subsequent query interval $[s,t]$, SUSs for interval $[s,t]$ can be answered quickly. When $s = t$, a query $[s,t]$ refers to a single position in the string S, and the problem is specifically called the *point SUS problem*. For clarity, when $s \leq t$, the problem is called the *interval SUS problem*.

* This work was in part supported by JSPS KAKENHI Grant Numbers JP25240003, JP26280003, JP16H02783, JP17H01697.

Pei et al. [5] were the first to consider the point SUS problem, motivated by some applications in bioinformatics. They considered two versions of this problem, depending on whether a single point SUS has to be returned (the *single point SUS problem*) or all point SUSs have to be returned (the *all point SUSs problem*) for a query position.

There is a series of research for the single point SUS problem. Pei et al. [5] gave an $O(n^2)$-time preprocessing scheme which returns a single point SUS for a query position in $O(1)$ time, where n is the length of the input string. Tsuruta et al. [6] and Ileri et al. [3] independently showed optimal $O(n)$-time preprocessing schemes which return a single point SUS for a query position in $O(1)$ time. Hon et al. [1] proposed an *in-place* algorithm for the same version of the problem, achieving the same bounds as the above solutions.

For the all point SUS problem which is more difficult, Tsuruta et al. [6] and Ileri et al. [3] also showed optimal algorithms achieving $O(n)$ preprocessing time and $O(k)$ query time, where k is the number of all point SUSs for a query point.

Hu et al. [2] were the first to consider the interval SUS problem, and they proposed an optimal algorithm for the interval SUS problem, using $O(n)$ time for preprocessing and $O(k')$ time for queries, where k' is the number of interval SUSs for a query interval. Recently, Mieno et al. [4] proposed an algorithm which solves the interval SUS problem on strings represented by *run-length encoding* (RLE). If r is the size of the RLE of a given string of length n, then $r \leq n$ always holds. Mieno et al.'s algorithm uses $O(r)$ space, requires $O(r \log r)$ time to construct, and answers all SUSs for a query interval in $O(k' + \sqrt{\log r / \log\log r})$ time.

A substring X of a string S is said to be a *minimal unique substring* (*MUS*) of S, if (i) X occurs in S exactly once and (ii) every proper substring of X occurs at least twice in S. All the above algorithms for the SUS problems pre-compute all MUSs of the input string S (or some data structure which is essentially equivalent to MUSs), and extensively use MUSs to return the SUSs for a query position or interval.

Tsuruta et al. [6] showed that the maximum number of MUSs contained in a string of length n is at most n. This immediately follows from the fact that MUSs do not nest. Mieno et al. [4] proved that the maximum number of MUSs in a string is bounded by $2r - 1$, where r is the size of the RLE of the string. They also showed a series of strings which have $2r - 1$ MUSs, and hence this bound is tight. These properties played significant roles in designing efficient algorithms for the SUS problems.

On the other hand, structural properties of SUSs are not well understood. A trivial upperbound for the maximum number of intervals that correspond to point SUSs is $3n$, since every MUS can be a SUS for some position of the input string S, and for each query position p ($1 \leq p \leq n$), there can be at most 2 SUSs that are not MUSs (one that ends at position p and the other that begins at position p).

1.2 Our contribution

The main contribution of this paper is matching upper and lower bounds for the maximum number of SUSs for the point SUS problem, which translate to "less than $1.5n$ point SUSs". Namely, we prove that any string of length n contains at most $(3n - 1)/2$ SUSs for the point SUS problem. We give a series of strings which contains $(3n - 1)/2$ SUSs for any odd number $n \geq 5$. Therefore, our bound is tight, and to our knowledge, this is the first non-trivial result for structural properties of SUSs.

We also consider the maximum number of SUSs for the interval SUS problem. In so doing, we exclude a special case where a query interval $[s, t]$ itself is a unique substring that occurs exactly once in S. This is because we have $\Theta(n^2)$ bounds for such trivial SUSs. We then prove that any string of length n contains less than $2n$ *non-trivial* SUSs for the interval

SUS problem. We also prove that there exists a string of length n which contains $(2-\varepsilon)n$ non-trivial SUSs for any small number $\varepsilon > 0$.

1.3 Related work

Xu [7] introduced the *longest repeat* (*LR*) problem. An interval $[i, j]$ of a string S is said to be an LR for interval $[s, t]$ if (a) the substring $R = S[i..j]$ occurs at least twice in S, (b) the occurrence $[i, j]$ of R contains $[s, t]$ and (c) there does not exist an interval $[i', j']$ of S such that $j' - i' > j - i$, the substring $S[i'..j']$ occurs at least twice in S, and the interval $[i', j']$ contains interval $[s, t]$. The point and interval LR problems are defined analogously as the point and interval SUS problems, respectively.

Xu [7] presented an optimal algorithm which, after $O(n)$-time preprocessing, returns all LRs for a given interval in $O(k'')$ time, where k'' is the number of output LRs. He claimed that although the point/interval SUS problems and the point/interval LR problems look alike, these problems are actually quite different, with a support from an example where an SUS and LR for the same query point seem rather unrelated.

Our $(3n-1)/2$ bound for the maximum number of SUSs for the point SUS problem also supports his claim in the following sense: In the preprocessing, Xu's algorithm computes the set of *maximal repeats* (*MR*). An interval $[i, j]$ of a string S is said to be an MR if (A) the substring $W = S[i..j]$ occurs at least twice in S, and (B) for any $1 \leq i' \leq i \leq j \leq j' \leq n$ with $j' - i' > j - i$, the superstring $Y = S[i'..j']$ of W occurs once in S. It is easy to see that the maximum number of MRs is bounded by n, since for any position in S, there can be at most one MR that begins at that position. This bound is also tight: any even palindrome consisting of $n/2$ distinct characters contains n intervals for which the corresponding substrings are MRs (e.g., for even palindrome `abcdeedcba` of length 10, any interval $[i, i]$ for $1 \leq i \leq 10$ is an MR). By definition, any LR of string S is also an MR of S. Hence, the maximum number of LRs is also bounded by n. Since the above lower bound for MRs with palindromes also applies to LRs, this upper bound for LRs is also tight. Thus, there is a gap of $(n-1)/2$ between the maximum numbers of SUSs and LRs.

2 Preliminaries

2.1 Notations

Let Σ be the alphabet. An element of Σ^* is called a string. We denote the length of string S by $|S|$. The empty string is the string of length 0. For any string S of length n and integer $1 \leq i \leq n$, let $S[i]$ denote the ith character of S. For any $1 \leq i \leq j \leq n$, let $S[i..j]$ denote the substring of S that starts at position i and ends at position j in S. For convenience, $S[i..j]$ is the empty string if $i > j$. For any strings S and w, let $\#occ_S(w)$ denote the number of occurrences of w in S, namely, $\#occ_S(w) = |\{i : S[i..i + |w| - 1] = w\}|$.

2.2 MUSs and SUSs

Let S be any string of length n, and w be any non-empty substring of S. We say that w is a *repeating substring* of S iff $\#occ_S(w) \geq 2$, and that w is a *unique substring* of S iff $\#occ_S(w) = 1$. Since any unique substring w of S occurs exactly once in S, we will sometimes identify w with its corresponding interval $[i, j]$ such that $w = S[i..j]$. We also say that interval $[i, j]$ is unique iff the corresponding $S[i..j]$ is a unique substring of S.

A unique substring $w = S[i..j]$ of S is said to be a *minimal unique substring* (*MUS*) iff any proper substring of w is a repeating substring, namely, $\#occ_S(S[i'..j']) \geq 2$ for any i'

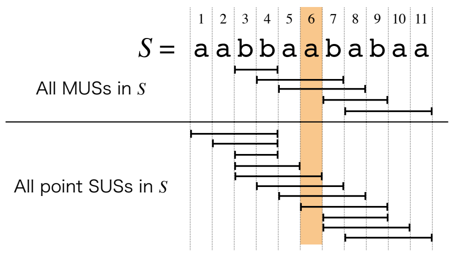


Figure 1 For string S = aabbaababaa, the set $\mathcal{M}_S = \{[3..4], [4..7], [5..8], [7..9], [8..11]\} = \{$bb, baab, aaba, bab, abaa$\}$ of all MUSs of S is shown in the upper part of the diagram. The set $\mathcal{PS}_S$ of all SUSs for all positions of string S is shown in the lower part of the diagram. For example, the intervals $[3..6]$ = bbaa, $[4..7]$ = baab, $[5..8]$ = aaba, and $[6..9]$ = abab are SUSs for query position 6, where the first SUS $[3..6]$ is obtained by extending the right-end of MUS $[3..4]$ up to position 6, the second SUS $[4..7]$ and the third $[5..8]$ are MUSs of S, and the fourth SUS $[6..9]$ is obtained by extending the left-end of MUS $[8..11]$ up to position 6.

and j' with $i' \geq i$, $j' \leq j$, and $j' - i' < j - i$. Let $\mathcal{M}_S$ be the set of all MUSs in S, namely, $\mathcal{M}_S = \{[i, j] : S[i..j] \text{ is a MUS of } S\}$. The next lemma follows from the definition of MUSs.

▶ **Lemma 1** ([6]). *No element of $\mathcal{M}_S$ is nested in another element of $\mathcal{M}_S$, namely, any two MUSs $[i, j], [k, \ell] \in \mathcal{M}_S$ satisfy $[i, j] \not\subset [k, \ell]$ and $[k, \ell] \not\subset [i, j]$. Therefore, $0 < |\mathcal{M}_S| \leq n$.*

For any substring $S[i..j]$ and an interval $[s, t]$ in S, $S[i..j]$ is said to be a *shortest unique substring* (*SUS*) for interval $[s, t]$ iff

1. $S[i..j]$ is a unique substring of S,
2. $[s, t] \subset [i, j]$, and
3. $S[i'..j']$ is a repeating substring of S for any i', j' with $[s, t] \subset [i', j']$ and $j' - i' < j - i$.

In particular, a SUS for some interval $[p, p]$ of length 1 is said to be a SUS for position p and is sometimes referred to as a *point* SUS in S. Also, a SUS for some interval (including those of length 1) is sometimes referred to as an *interval* SUS in S.

Since any SUS $S[i..j]$ occurs in S exactly once, we will sometimes identify it with the interval $[i, j]$ which corresponds to its unique occurrence in S.

Clearly, if $[i, j]$ is unique, then $[i, j]$ is the only SUS for the interval $[i, j]$. For any interval $[i, j]$ with $i < j$, if $[i, j]$ is unique and there is no other interval $[s, t] \subset [i, j]$ for which $[i, j]$ is a SUS, then we say that $[i, j]$ is a *trivial* interval SUS. Also, we say that $[i, j]$ is a *non-trivial* interval SUS if $[i, j]$ is not a trivial SUS.

For any interval $[s, t] \subset [1, |S|]$, let $\mathsf{SUS}_S([s, t])$ denote the set of interval SUSs of S that contain query interval $[s, t]$, and $\mathcal{IS}_S$ the set of all non-trivial interval SUSs of S. Also, for any position $p \in [1, |S|]$, let $\mathsf{SUS}_S(p)$ denote the set of point SUSs of S that contain query position p, and $\mathcal{PS}_S$ the set of all point SUSs of S, namely, $\mathcal{PS}_S = \bigcup_{p=1}^{n} \mathsf{SUS}_S(p)$. Figure 1 shows examples of MUSs and SUSs.

Hu et al. [2] showed that it is possible to preprocess a given string S of length n in $O(n)$ time so that later, we can return all SUSs that contain a query interval $[s, t]$ in $O(k)$ time, where k is the number of such SUSs.

As is shown in Lemma 1, the number of MUSs in any string S of length n is bounded by n. In this paper, we show that the number of point SUSs in S is less than $1.5n$, more precisely, $|\mathcal{PS}_S| \leq (3n - 1)/2$. We will do so by first showing two different bounds on $|\mathcal{PS}_S|$ in terms of the number $|\mathcal{M}_S|$ of MUSs in the string S, and then merging these two results that lead to the claimed bound. Moreover, this bound is indeed tight, namely, we show a series of strings containing $(3n-1)/2$ SUSs. In addition, we show that the number of non-trivial SUSs in S is less than $2n$, namely, $|\mathcal{IS}_S| < 2n$. We also prove that there exists a string of length n which contains $(2-\varepsilon)n$ non-trivial SUSs for any small number $\varepsilon > 0$.

3 Bounds on the number of point SUSs

Here we show a tight bound for the maximum number of point SUSs in a string. In this section, whenever we speak of SUSs, we mean point SUSs (those for the point SUS problem).

3.1 Upperbound A

In this subsection, we show our first upperbound on the number of SUSs in a string S. In so doing, we define the subsets $\mathcal{LS}_S$, $\mathcal{MS}_S$, and $\mathcal{RS}_S$ of the set $\mathcal{PS}_S$ of all SUS of string S by

$$\begin{aligned}\mathcal{LS}_S &= \mathcal{PS}_S \cap \{[x,y] \notin \mathcal{M}_S : x < \exists i \le y\ [i,y] \in \mathcal{M}_S\},\\ \mathcal{MS}_S &= \mathcal{PS}_S \cap \mathcal{M}_S, \text{ and}\\ \mathcal{RS}_S &= \mathcal{PS}_S \cap \{[x,y] \notin \mathcal{M}_S : x \le \exists j < y\ [x,j] \in \mathcal{M}_S\}.\end{aligned}$$

Intuitively, $\mathcal{LS}_S$ is the set of SUSs of S which are *not* MUSs of S and can be obtained by extending the beginning positions of some MUSs to the left up to query positions, $\mathcal{MS}_S$ is the set of SUSs of S which are also MUSs of S, and $\mathcal{RS}_S$ is the set of SUSs of S which are *not* MUSs of S and can be obtained by extending the ending positions of some MUSs to the right up to query positions.

It follows from their definitions that $\mathcal{LS}_S \cap \mathcal{MS}_S = \phi$, $\mathcal{MS}_S \cap \mathcal{RS}_S = \phi$, $\mathcal{RS}_S \cap \mathcal{LS}_S = \phi$ and that $\mathcal{PS}_S = \mathcal{LS}_S \cup \mathcal{MS}_S \cup \mathcal{RS}_S$.

Figure 3 in the next subsection shows examples of $\mathcal{LS}_S$, $\mathcal{MS}_S$, and $\mathcal{RS}_S$ for string $S =$ `aabbaababaa`. Also compare it with Figure 1 which shows $\mathcal{PS}_S$ for the same string S.

In the proof of the following theorem, we will evaluate the sizes of these three sets $\mathcal{LS}_S$, $\mathcal{MS}_S$, and $\mathcal{RS}_S$ separately.

▶ **Theorem 2.** *For any string S, $|\mathcal{PS}_S| \le 2|S| - |\mathcal{M}_S|$.*

Proof. Let $n = |S|$ and $m = |\mathcal{M}_S|$. For any $1 \le i \le m$, let $[b_i, e_i]$ denote the MUS of S that has the ith smallest beginning position in $\mathcal{M}_S$.

It is clear that $|\mathcal{MS}_S| \le m$. Note that the inequality is due to that fact that some MUS may not be a point SUS for any position in S (such a MUS is called *meaningless* in the literature [6]).

Next, we consider the size of $\mathcal{RS}_S$. By definition, for any $[x,y] \in \mathcal{RS}_S$, x is equal to the beginning position of a MUS of S. Therefore, we can bound $|\mathcal{RS}_S|$ by summing up the number of SUSs that begin with b_i for every $[b_i, e_i] \in \mathcal{M}_S$. For any $1 \le i \le m-1$, consider two adjacent MUSs $[b_i, e_i], [b_{i+1}, e_{i+1}] \in \mathcal{M}_S$. Recall that $b_i < b_{i+1}$. Then, for any $j \ge e_{i+1}$, the interval $[b_i, j]$ contains both MUSs $[b_i, e_i]$ and $[b_{i+1}, e_{i+1}]$. This implies that $[b_i, j] \notin \mathcal{PS}_S$ (see Figure 2), since otherwise both $[b_i, j]$ and $[b_{i+1}, j]$ are SUSs for position j, a contradiction. Thus, for any $[b_i, e_i] \in \mathcal{M}_S$ with $1 \le i \le m-1$, the number of SUSs that begin with b_i and belong to $\mathcal{RS}_S$ is at most $e_{i+1} - e_i - 1$. Also, the number of SUSs that begin with b_m and belong to $\mathcal{RS}_S$ is at most $n - e_m$. Consequently, we get $|\mathcal{RS}_S| = \sum_{i=1}^{m-1}(e_{i+1} - e_i - 1) + n - e_m = e_m - e_1 - (m-1) + n - e_m \le n - m$.

A symmetric argument gives us the same bound for $|\mathcal{LS}_S|$, namely, $|\mathcal{LS}_S| \le n - m$. Overall, we obtain $|\mathcal{PS}_S| = |\mathcal{LS}_S| + |\mathcal{MS}_S| + |\mathcal{RS}_S| \le 2(n-m) + m = 2n - m$. ◀

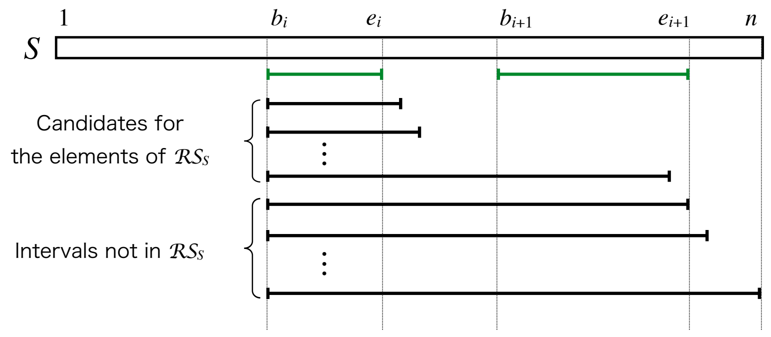

1
b_i
e_i
b_{i+1}
e_{i+1}
n
S
Candidates for the elements of $\mathcal{RS}_S$
Intervals not in $\mathcal{RS}_S$

Figure 2 Illustration for Theorem 2. Consider two adjacent MUSs $[b_i, e_i]$ and $[b_{i+1}, e_{i+1}]$ depicted as the two intervals on the top. For any $e_i < e < e_{i+1}$, $[b_i, e]$ can be an element of $\mathcal{RS}_S$. On the other hand, for any $e' \geq e_{i+1}$, $[b_i, e']$ can never be an element of $\mathcal{PS}_S$ since $[b_i, e']$ contains two distinct MUSs $[b_i, e_i]$ and $[b_i, e_{i+1}]$, and hence $[b_i, e']$ can never be an element of $\mathcal{RS}_S$ as well.

3.2 Upperbound B

In this subsection, we provide another upperbound on the size of $\mathcal{PS}_S$.

▶ **Theorem 3.** *For any string S, $|\mathcal{PS}_S| \leq |S| + |\mathcal{M}_S| - 1$.*

In order to show Theorem 3, we will use a function $f : \mathcal{PS}_S \to \{1, 2, \ldots, n\}$ and its inverse image $f^{-1} : \{1, 2, \ldots, n\} \to 2^{\mathcal{PS}_S}$. The next lemma is useful to define f and f^{-1}.

▶ **Lemma 4.** *For any string S and interval $[x, y]$ such that $1 \leq x \leq y \leq |S|$, if $[x, y] \in \mathcal{RS}_S$ then $[x, y] \in \mathsf{SUS}_S(y)$, and if $[x, y] \in \mathcal{LS}_S$ then $[x, y] \in \mathsf{SUS}_S(x)$.*

Proof. We first prove the former case. Assume on the contrary that some $[x, y] \in \mathcal{RS}_S$ satisfies $[x, y] \notin \mathsf{SUS}_S(y)$. This implies that there exists a position p in S such that $x \leq p < y$ and $[x, y] \in \mathsf{SUS}_S(p)$. In addition, since $[x, y] \in \mathcal{RS}_S$, there exists a position q such that $x \leq q < y$ and $[x, q] \in \mathcal{M}_S$. Let $z = \max\{p, q\}$. Then, $S[x..z]$ is a unique substring of S which is shorter than $S[x..y]$ and contains position p. However, this contradicts that $S[x..y]$ is a SUS for position p. Thus, if $[x, y] \in \mathcal{RS}_S$ then $[x, y] \in \mathsf{SUS}_S(y)$. The latter case is symmetric and thus can be shown similarly. ◀

We are now ready to define f:

$$f([x, y]) = \begin{cases} x & \text{if } [x, y] \in \mathcal{LS}_S \cup \mathcal{MS}_S, \\ y & \text{if } [x, y] \in \mathcal{RS}_S. \end{cases}$$

Intuitively, the function f charges a given interval $[x, y]$ to its beginning position x if $[x, y]$ is an element of $\mathcal{M}_S \cap \mathcal{PS}_S$ or if $[x, y]$ is an element of $\mathsf{SUS}_S(p)$ for some query position p which is obtained by extending the left-end of a MUS to the left up to p. On the other hand, it charges $[x, y]$ to its ending position y if the interval is an element of $\mathsf{SUS}_S(p)$ for some query position p which is obtained by extending the right-end of a MUS to the right up to p. Figure 3 shows examples for how the function f charges given interval $[x, y] \in \mathcal{PS}_S$.

We also define the inverse image f^{-1} of f as follows:

$$f^{-1}(u) = \{[x, y] \in \mathcal{PS}_S \ : \ f([x, y]) = u\}.$$

For positions u for which there is no element $[x, y]$ in $\mathcal{PS}_S$ satisfying $f([x, y]) = u$, let $f^{-1}(u) = \emptyset$. See also Figure 3 for examples of f^{-1}.

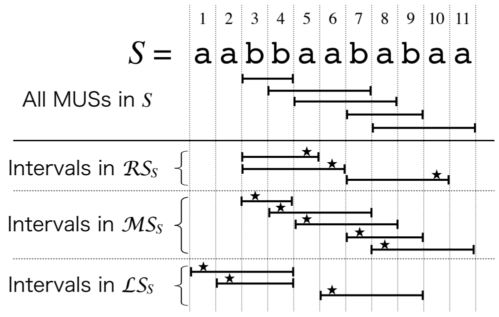

1 2 3 4 5 6 7 8 9 10 11
S = aabbaababaa
All MUSs in S
Intervals in $\mathcal{RS}_S$
Intervals in $\mathcal{MS}_S$
Intervals in $\mathcal{LS}_S$

Figure 3 Illustration for functions f and f^{-1} of string S = aabbaababaa. The upper part of this diagram shows all MUSs in S, and the lower part shows all SUSs for all positions in S. Each star shows the position to which the function f maps the corresponding interval. Here, $\mathcal{RS}_S = \{[3,5],[3,6],[7,10]\}$, $\mathcal{MS}_S = \{[3,4],[4,7],[5,8],[7,9],[8,11]\}$, and $\mathcal{LS}_S = \{[1,4],[2,4],[6,10]\}$. Hence, we have $f([3,5]) = 5$, $f([3,6]) = 6$, $f([7,10]) = 10$, $f([3,4]) = 3$, $f([4,7]) = 4$, $f([5,8]) = 5$, $f([7,9]) = 7$, $f([8,11]) = 8$, $f([1,4]) = 1$, $f([2,4]) = 2$, and $f([6,10]) = 6$. For the inverse image, f^{-1}, we have $f^{-1}(1) = \{[1,4]\}$, $f^{-1}(2) = \{[2,4]\}$, $f^{-1}(3) = \{[3,4]\}$, $f^{-1}(4) = \{[4,7]\}$, $f^{-1}(5) = \{[3,5],[5,8]\}$, $f^{-1}(6) = \{[3,6],[6,10]\}$, $f^{-1}(7) = \{[7,9]\}$, $f^{-1}(8) = \{[8,11]\}$, $f^{-1}(9) = f^{-1}(11) = \emptyset$, and $f^{-1}(10) = \{[7,10]\}$.

By the definition of f^{-1}, it is clear that $|\mathcal{PS}_S| = \sum_{u=1}^{|S|} |f^{-1}(u)|$. Hence, in what follows we analyze $|f^{-1}(u)|$ for all positions u in string S.

▶ **Lemma 5.** *For any string and position $1 \le u \le |S|$, $|f^{-1}(u)| \le 2$.*

Proof. Assume on the contrary that $|f^{-1}(u)| \ge 3$ for some position u in S. Let $[x_1, y_1]$, $[x_2, y_2]$ be any distinct elements of $f^{-1}(u)$. We firstly consider the following cases.

1. Case where $[x_1, y_1], [x_2, y_2] \in \mathcal{LS}_S$: It follows from the definition of f^{-1} that $f([x_1, y_1]) = f([x_2, y_2]) = u$, and it follows from the definition of f that $x_1 = x_2 = u$. Since $[x_1, y_1]$ and $[x_2, y_2]$ are distinct, $y_1 \ne y_2$. Assume w.l.o.g. that $y_1 < y_2$. Then, $[x_2, y_2] = [u, y_2]$ is a SUS for position u but it is longer than another SUS $[x_1, y_1] = [u, y_1]$ for position u, a contradiction.
2. Case where $[x_1, y_1], [x_2, y_2] \in \mathcal{MS}_S$: It follows from the definition of f^{-1} that $f([x_1, y_1]) = f([x_2, y_2]) = u$, and it follows from the definition of f that $x_1 = x_2 = u$. Since $[x_1, y_1]$ and $[x_2, y_2]$ are distinct, $y_1 \ne y_2$. Assume w.l.o.g. that $y_1 < y_2$. Then, $[x_2, y_2] = [u, y_2]$ is a MUS, but it contains another MUS $[x_1, y_1] = [u, y_1]$, a contradiction.
3. Case where $[x_1, y_1], [x_2, y_2] \in \mathcal{RS}_S$: This is symmetric to Case (1) and thus we can obtain a contradiction in a similar way.

Hence, none of the above three cases is possible, and thus the remaining possibility is the case where $|f^{-1}(u)| = 3$ and each element of $f^{-1}(u)$ belongs to a different subset of $\mathcal{PS}_S$, namely, $f^{-1}(u) = \{[x_1, y_1], [x_2, y_2], [x_3, y_3]\}$ for some $[x_1, y_1] \in \mathcal{LS}_S$, $[x_2, y_2] \in \mathcal{MS}_S$, and $[x_3, y_3] \in \mathcal{RS}_S$. It follows from the definition of f^{-1} that $f([x_1, y_1]) = f([x_2, y_2]) = u$, and it follows from the definition of f that $x_1 = x_2 = u$. Since $[x_1, y_1]$ and $[x_2, y_2]$ are distinct, $y_1 \ne y_2$. There are two sub-cases.

(i) If $y_1 < y_2$, then a MUS $[x_2, y_2] = [u, y_2]$ contains a shorter SUS $[x_1, y_1] = [u, y_1]$ for position u, a contradiction.

(ii) If $y_1 > y_2$, then a SUS $[x_1, y_1] = [u, y_1]$ for position u contains a shorter MUS $[x_2, y_2] = [u, y_2]$, a contradiction.

Hence, neither of the sub-cases is possible.

Overall, we conclude that $|f^{-1}(u)| \le 2$. ◀

By Lemma 5, for any position u in string S we have $|f^{-1}(u)| \leq 2$. Now let us consider any position u for which $|f^{-1}(u)| = 2$. We have the next lemma.

▶ **Lemma 6.** *For any position u in string S for which $|f^{-1}(u)| = 2$, let $f^{-1}(u) = \{[x_1, y_1], [x_2, y_2]\}$ and assume w.l.o.g. that $x_1 \leq x_2$. Then, $x_1 \neq x_2$, $[x_1, y_1] \in \mathcal{RS}_S$ and $[x_2, y_2] \in \mathcal{LS}_S \cup \mathcal{MS}_S$.*

Proof. Suppose $x_1 = x_2$ and assume w.l.o.g. that $y_1 < y_2$. Then, from the definition of f, we have that ($x_1 = u$ or $y_1 = u$) and ($x_2 = u$ or $y_2 = u$) and thus $x_1 = x_2 = u$. Since $[x_2, y_2] \in f^{-1}(u)$ is not a MUS since it includes $[x_1, y_1]$, it must be that $[x_2, y_2] \in \mathsf{SUS}_S(u)$. This is a contradiction, because there exists a shorter unique substring $[x_1, y_1]$ that contains u. Thus we have $x_1 \neq x_2$. Assume on the contrary that $[x_1, y_1] \in \mathcal{LS}_S \cup \mathcal{MS}_S$. Then, it follows from the definition of f that $f([x_1, y_1]) = x_1$. In addition, since $[x_1, y_1] \in f^{-1}(u)$, we have $u = x_1$. This implies that $u = x_1 < x_2$, but it contradicts that $[x_2, y_2] \in f^{-1}(u)$. Thus, $[x_1, y_1] \notin \mathcal{LS}_S \cup \mathcal{MS}_S$, namely, $[x_1, y_1] \in \mathcal{RS}_S$. Now, it follows from the arguments in the proof of Lemma 5 that $[x_2, y_2] \notin \mathcal{RS}_S$, and hence $[x_2, y_2] \in \mathcal{MS}_S \cup \mathcal{LS}_S$. ◀

Let $m = |\mathcal{M}_S|$, and $\mathcal{M}_S = \{[b_1, e_1], \ldots, [b_m, e_m]\}$. The next corollary immediately follows from Lemmas 4 and 6.

▶ **Corollary 7.** *For any position u in string S with $|f^{-1}(u)| = 2$, there exist two integers $1 \leq i < j \leq m$ such that $\mathsf{SUS}_S(u) = \{[b_i, u], [u, e_j]\}$.*

For any position u in string S before b_1 or after b_m, we have the next lemma.

▶ **Lemma 8.** *For any position u in string S s.t. $1 \leq u \leq b_1$ or $b_m < u \leq n$, $|f^{-1}(u)| \leq 1$.*

Proof. Assume on the contrary that $|f^{-1}(u)| = 2$ for some $1 \leq u \leq b_1$. By Lemma 6, there exists $[x, y] \in f^{-1}(u)$ such that $[x, y] \in \mathcal{RS}_S$. By the definitions of f and f^{-1}, we have $y = u$. Also, by the definition of $\mathcal{RS}_S$, there exists a position $e < y$ in S such that $[x, e] \in \mathcal{M}_S$. Now we have $x \leq e < y = u \leq b_1$, however, this contradicts that b_1 is the beginning position of the first (leftmost) MUS in $\mathcal{M}_S$. Thus $|f^{-1}(u)| \leq 1$ for any $1 \leq u \leq b_1$.

Assume on the contrary that $|f^{-1}(u)| = 2$ for some $b_m < u \leq n$. By Lemma 6, there exists $[x', y'] \in f^{-1}(u)$ such that $[x', y'] \in \mathcal{MS}_S \cup \mathcal{LS}_S$. By the definition of f and f^{-1}, we have $x' = u$. There are two cases to consider:

- If $[x', y'] \in \mathcal{MS}_S$, then $[x', y'] \in \mathcal{M}_S$. Thus $x' = u > b_m$ is the beginning position of a MUS in $\mathcal{M}_S$, however, this contradicts that b_m is the beginning position of the last (rightmost) MUS in $\mathcal{M}_S$.
- If $[x', y'] \in \mathcal{LS}_S$, then by the definition of $\mathcal{LS}_S$ there exists a position $b > x'$ such that $[b, y'] \in \mathcal{M}_S$. Now we have $b > x' = u > b_m$, however, this contradicts that b_m is the beginning position of the last (rightmost) MUS in $\mathcal{M}_S$.

Consequently, $|f^{-1}(u)| \leq 1$ for any $b_m < u \leq n$. ◀

▶ **Lemma 9.** *For any non-empty string S, let $U = \{u : |f^{-1}(u)| = 2\}$. Then, $|U| \leq |\mathcal{M}_S| - 1$.*

Proof. Let $n = |S|$ and $m = |\mathcal{M}_S|$. Recall that for any $1 \leq i \leq m$, $[b_i, e_i]$ denotes the ith element of $\mathcal{M}_S$.

Let $B = \{b_i : 1 \leq i \leq m - 1\}$. We define function $g : U \to B$ as $g(u) = \max\{b < u : b \in B\}$. By the definition of U and Lemma 8, any position $u \in U$ satisfies $b_1 < u \leq b_m$. Therefore, $g(u)$ is well-defined for any position $u \in U$, and $g(u)$ returns the predecessor of u in the set B. It is clear that $|B| = m - 1$. Thus, if g is an injection, then we immediately obtain the claimed bound $|U| \leq |B| = m - 1$.

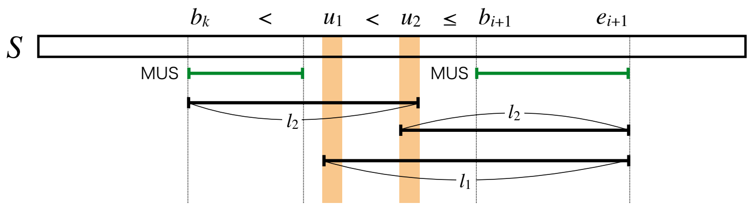


Figure 4 Illustration for Lemma 9. The two intervals show two MUSs $[b_k, e_k]$, $[b_{i+1}, e_{i+1}] \in \mathcal{M}_S$, where $b_k \leq b_i$. Both $[b_k, u_2]$ and $[u_2, b_{i+1}]$ are SUSs for position u_2, and $[u_1, e_{i+1}]$ is a SUS for position u_1. Since $u_1 < u_2$, it holds that $l_1 > l_2$, where l_1 and l_2 are the lengths of SUSs for positions u_1 and u_2, respectively. Then, the interval $[b_k, u_2]$ of length l_2 contains position u_1 and $S[b_k..u_2]$ is a unique substring of S. However, this contradicts that l_1 is the length of each SUS for position u_1.

In what follows, we show that g is indeed an injection. Assume on the contrary that g is not an injection. Let u_1 and u_2 be elements in U such that $u_1 < u_2$ and $g(u_1) = g(u_2)$. Let $b_i \in B$ such that $b_i = g(u_1) = g(u_2)$. Then, by the definition of g, we have $b_i < u_1 < u_2 \leq b_{i+1}$. See Figure 4 for illustration.

Let l_1 and l_2 be the lengths of the SUSs for positions u_1 and u_2, respectively. Since $|f^{-1}(u_2)| = 2$, it follows from Corollary 7 that there exists $b_k \in B$ such that $b_k \leq b_i$ and $\mathsf{SUS}_S(u_2) = \{[b_k, u_2], [u_2, e_{i+1}]\}$. This implies $l_2 = u_2 - b_k + 1 = e_{i+1} - u_2 + 1$. On the other hand, since $|f^{-1}(u_1)| = 2$, it follows from Corollary 7 that $[u_1, e_{i+1}] \in \mathsf{SUS}_S(u_1)$, which implies $l_1 = e_{i+1} - u_1 + 1$. Since $u_1 < u_2$, we have $l_1 > l_2$.

Now focus on a SUS $[b_k, u_2]$ for position u_2. Since $b_k \leq b_i < u_1 < u_2$, $[b_k, u_2]$ contains u_1. However, $[b_k, u_2]$ is a SUS for position u_2 and is of length $l_2 < l_1$. This contradicts that $[u_1, e_{i+1}]$ of length l_1 is each SUS for position u_1. Hence g is an injection. ◀

We are ready to prove the main result of this subsection, Theorem 3.

Proof. Let $n = |S|$, $m = |\mathcal{M}_S|$, $U = \{u : |f^{-1}(u)| = 2\}$, and $V = \{1, \cdots, n\} \setminus U$. It is clear that $|U| + |V| = n$. By Lemma 5, $V = \{u : |f^{-1}(u)| \leq 1\}$. Also, by Lemma 9, $|U| \leq m - 1$. Recall that $|\mathcal{PS}_S| = \sum_{u=1}^{n} |f^{-1}(u)|$. Putting all together, we obtain $|\mathcal{PS}_S| = \sum_{u=1}^{n} |f^{-1}(u)| \leq |V| + 2|U| = n + |U| \leq n + m - 1$. ◀

3.3 Matching upper and lower bounds

We are ready to show the main result of this paper.

▶ **Theorem 10.** *For any non-empty string S, $|\mathcal{PS}_S| \leq (3|S| - 1)/2$. This bound is tight, namely, for any odd $n \geq 5$ there exists a string T of length n s.t. $|\mathcal{PS}_T| = (3n - 1)/2$.*

Proof. By Theorem 2, we have $|\mathcal{M}_S| \leq 2|S| - |\mathcal{PS}_S|$. Also, by Theorem 3, we have $|\mathcal{PS}_S| - |S| + 1 \leq |\mathcal{M}_S|$. Thus $|\mathcal{PS}_S| - |S| + 1 \leq 2|S| - |\mathcal{PS}_S|$, which immediately leads to the claimed bound $|\mathcal{PS}_S| \leq (3|S| - 1)/2$.

We show that the above upperbound is indeed tight. For any odd number $n = 2k - 1 \geq 5$, consider string $T = a_1 x a_2 x \cdots a_{k-1} x a_k$, where $a_1, \ldots, a_k, x \in \Sigma$, $a_i \neq a_j$ for all $1 \leq i \neq j \leq k$, and $x \neq a_i$ for all $1 \leq i \leq k$. For any $1 \leq i \leq k$, $T[2i - 1] = a_i$ is a unique substring of T, and thus $[2i - 1, 2i - 1] \in \mathsf{SUS}_T(2i - 1)$. Also, for any $1 \leq i \leq k - 1$, $T[2i] = x$ is a repeating substring of T while $T[2i - 1..2i] = a_i x$ and $T[2i..2i + 1] = x a_{i+1}$ are unique substrings of T. This implies that $[2i - 1, 2i], [2i, 2i + 1] \in \mathsf{SUS}_T(2i)$. Hence, we have $|\mathcal{PS}_T| = k + 2(k - 1) = 3k - 2 = 3(n + 1)/2 - 2 = (3n - 1)/2$. ◀

3.4 Lower bound for fixed-size alphabet

The lowerbound of Theorem 10 is due to a series of strings over an alphabet of unbounded size. In this subsection, we fix the alphabet size σ and present a series of strings that contain many point SUSs.

▶ **Theorem 11.** *Let $n \geq 2$ and $2 \leq \sigma \leq (n+3)/2$. There exists a string T of length n over an alphabet of size σ such that $|\mathcal{PS}_T| = n + \sigma - 2$.*

Proof. Let $\Sigma = \{a_1, \cdots, a_{\sigma-1}, x\}$ and $T = a_1 x a_2 x \cdots a_{\sigma-1} x^{n-2\sigma+3}$. For any $1 \leq i \leq \sigma - 1$, $T[2i-1] = a_i$ is a unique substring of T, and thus $[2i-1, 2i-1] \in \mathsf{SUS}_T(2i-1)$. For any $1 \leq j \leq \sigma - 2$, $T[2j] = x$ is a repeating substring of T while $T[2j-1..2j] = a_j x$ and $T[2j..2j+1] = x a_{j+1}$ are unique substrings of T. This implies that $[2j-1, 2j], [2j, 2j+1] \in \mathsf{SUS}_T(2j)$. For any $2\sigma - 2 \leq k \leq n-1$, $T[2\sigma-2..k] = x^{k-2\sigma+3}$ is a repeating substring of T while $T[2\sigma-1..k] = a_{\sigma-1} x^{k-2\sigma+3}$ is a unique substrings of T. This implies that $[2\sigma-1, k] \in \mathsf{SUS}_T(k)$. Also, $T[2\sigma-1..n] = x^{n-2\sigma+2}$ is a repeating substring of T and $T[2\sigma-2..n] = x^{n-2\sigma+3}$ is a unique substring of T, and thus $[2\sigma-2..n] \in \mathsf{SUS}_T(n)$. Summing up all the point SUSs above, we obtain $|\mathcal{PS}_T| = \sigma - 1 + 2(\sigma-2) + n - 2\sigma + 2 + 1 = n + \sigma - 2$. ◀

4 Bounds on the number of interval SUSs

In this section, we show almost tight bounds for the maximum number of non-trivial interval SUSs $\mathcal{IS}_S$ of a string S. The following upper bound for $|\mathcal{IS}_S|$ can be obtained in an analogous way to Theorem 2.

▶ **Lemma 12.** *For any non-empty string S, $|\mathcal{IS}_S| \leq 2|S| - |\mathcal{M}_S|$.*

We also have the following lower bound for $|\mathcal{IS}_S|$.

▶ **Lemma 13.** *For any $\varepsilon > 0$, there exists a string T of length n such that $|\mathcal{IS}_T| > (2-\varepsilon)n$.*

Proof. Let $x = \lceil 3/(2\varepsilon) \rceil$, $T = c_1 a^x c_2 a^x c_3$ and $n = |T| = 2x + 3$. Clearly, c_1, c_2 and c_3 are MUSs of T and are in $\mathcal{IS}_T$. For all $2 \leq i \leq x+1$, $T[1..i]$ and $T[i..x+2]$ are unique substrings of T, and $T[2..i]$ and $T[i..x+1]$ are repeating substrings of T. This implies $T[1..i] \in \mathsf{SUS}_S([2, i])$ and $T[i..x+2] \in \mathsf{SUS}_S([i, x+1])$. Similarly, for all $x+3 \leq j \leq 2x+2$, $T[x+2..j] \in \mathsf{SUS}_S([x+3, j])$ and $T[j..2x+3] \in \mathsf{SUS}_S([j, 2x+2])$. Then, we have $|\mathcal{IS}_T| = 4x + 3$. Hence, $|\mathcal{IS}_T| - (2-\varepsilon)n = 4x + 3 - (2-\varepsilon)(2x+3) = 2\varepsilon x + 3\varepsilon - 3 = 2\varepsilon\lceil 3/(2\varepsilon) \rceil + 3\varepsilon - 3 \geq 3\varepsilon > 0$. ◀

As is shown in the following theorem, the number of non-trivial interval SUSs contained in the string T of Lemma 13 "almost coincides" with the upper bound of Lemma. Namely:

▶ **Theorem 14.** *For any $\varepsilon > 0$, there is a string T such that $(2|T| - |\mathcal{M}_T|) - (2-\varepsilon)|T| \leq 5\varepsilon$.*

Proof. For any $\varepsilon > 0$, consider the string T of Lemma 13. We remark that T contains 3 MUSs, namely, $|\mathcal{M}_T| = 3$. Hence, we obtain $(2|T| - |\mathcal{M}_T|) - (2-\varepsilon)|T| = \varepsilon|T| - |\mathcal{M}_T| = \varepsilon|T| - 3 = \varepsilon(2\lceil 3/(2\varepsilon) \rceil + 3) - 3 = 2\varepsilon\lceil 3/(2\varepsilon) \rceil + 3\varepsilon - 3 \leq 2\varepsilon(3/(2\varepsilon) + 1) + 3\varepsilon - 3 = 5\varepsilon \to 0\,(\varepsilon \to 0)$. ◀

5 Conclusions and open questions

In this paper, we presented matching upper and lower bounds for the maximum number of SUSs for the point SUS problem. Namely, we proved that any string of length n can contain at most $(3n-1)/2$ SUSs for the point SUS problem, and showed that this bound is tight by giving a string of length n containing $(3n-1)/2$ SUSs. For a fixed alphabet size σ, we also presented a string of length n containing $n + \sigma - 2$ SUSs. Moreover, we showed that any string of length n which contains m MUSs can have at most $2n - m$ non-trivial interval SUSs, and that for any $\varepsilon > 0$ there is a string of length n which contains $(2 - \varepsilon)n$ non-trivial interval SUSs.

An interesting future work is to show a non-trivial upper bound of the maximum number of point SUSs for a fixed alphabet size σ. We conjecture that the tight upper bound matches our lower bound $n + \sigma - 2$. Another future work is to close the small gap between the upper and lower bounds on the maximum number of non-trivial interval SUSs shown in Theorem 14.

References

1 Wing-Kai Hon, Sharma V. Thankachan, and Bojian Xu. An in-place framework for exact and approximate shortest unique substring queries. In Khaled M. Elbassioni and Kazuhisa Makino, editors, *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC 2015)*, volume 9472 of *LNCS*, pages 755–767. Springer, 2015. doi:10.1007/978-3-662-48971-0_63.

2 Xiaocheng Hu, Jian Pei, and Yufei Tao. Shortest unique queries on strings. In Edleno Silva de Moura and Maxime Crochemore, editors, *Proceedings of the 21st International Symposium on String Processing and Information Retrieval (SPIRE 2014)*, volume 8799 of *LNCS*, pages 161–172. Springer, 2014. doi:10.1007/978-3-319-11918-2_16.

3 Atalay Mert Ileri, M. Oguzhan Külekci, and Bojian Xu. Shortest unique substring query revisited. In Alexander S. Kulikov, Sergei O. Kuznetsov, and Pavel A. Pevzner, editors, *Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM 2014)*, volume 8486 of *LNCS*, pages 172–181. Springer, 2014. doi:10.1007/978-3-319-07566-2_18.

4 Takuya Mieno, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Shortest unique substring queries on run-length encoded strings. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *LIPIcs*, pages 69:1–69:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.MFCS.2016.69.

5 Jian Pei, Wush Chi-Hsuan Wu, and Mi-Yen Yeh. On shortest unique substring queries. In Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE 2013)*, pages 937–948. IEEE Computer Society, 2013. doi:10.1109/ICDE.2013.6544887.

6 Kazuya Tsuruta, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Shortest unique substrings queries in optimal time. In Viliam Geffert, Bart Preneel, Branislav Rovan, Julius Stuller, and A Min Tjoa, editors, *Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2014)*, volume 8327 of *LNCS*, pages 503–513. Springer, 2014. doi:10.1007/978-3-319-04298-5_44.

7 Bojian Xu. On stabbing queries for generalized longest repeat. In Jun Huan, Satoru Miyano, Amarda Shehu, Xiaohua Tony Hu, Bin Ma, Sanguthevar Rajasekaran, Vijay K. Gombar, Matthieu-P. Schapranow, Illhoi Yoo, Jiayu Zhou, Brian Chen, Vinay Pai, and Brian G. Pierce, editors, *Proceedings of the 2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2015)*, pages 523–530. IEEE Computer Society, 2015. doi:10.1109/BIBM.2015.7359738.

Can We Recover the Cover?

Amihood Amir[1], Avivit Levy[2], Moshe Lewenstein[3], Ronit Lubin[4], and Benny Porat[5]

1 **Bar-Ilan University, Ramat Gan, Israel; and Johns Hopkins University, Baltimore, MD, USA**
`amir@cs.biu.ac.il`
2 **Shenkar College, Ramat Gan, Israel**
`avivitlevy@shenkar.ac.il`
3 **Bar-Ilan University, Ramat Gan, Israel**
`moshe.lewenstein@gmail.com`
4 **Bar-Ilan University, Ramat Gan, Israel**
`ronit.moldovan@gmail.com`
5 **Bar-Ilan University, Ramat Gan, Israel**
`bennyporat@gmail.com`

Abstract

Data analysis typically involves *error recovery* and *detection of regularities* as two different key tasks. In this paper we show that there are data types for which these two tasks can be powerfully combined. A common notion of regularity in strings is that of *a cover*. Data describing measures of a natural coverable phenomenon may be corrupted by errors caused by the measurement process, or by the inexact features of the phenomenon itself. Due to this reason, different variants of approximate covers have been introduced, some of which are $\mathcal{NP}$-hard to compute. In this paper we assume that the Hamming distance metric measures the amount of corruption experienced, and study the problem of recovering the correct cover from data corrupted by mismatch errors, formally defined as the *cover recovery problem (CRP)*. We show that for the Hamming distance metric, coverability is a powerful property allowing *detecting* the original cover and *correcting* the data, under suitable conditions.

We also study a relaxation of another problem, which is called the *approximate cover problem (ACP)*. Since the ACP is proved to be $\mathcal{NP}$-hard [5], we study a relaxation, which we call *the candidate-relaxation of the ACP*, and show it has a polynomial time complexity. As a result, we get that the ACP also has a polynomial time complexity in many practical situations. An important application of our ACP relaxation study is also a polynomial time algorithm for the cover recovery problem (CRP).

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics, G.4 Mathematical Software, I.5.2 Design Methodology

Keywords and phrases periodicity, quasi-periodicity, cover, approximate cover, data recovery

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.25

1 Introduction

Data analysis typically involves error recovery and detection of regularities as two different key tasks. In this paper we show that there are data types for which these two tasks can be powerfully combined. A classical tool for handling data recovery is through the use of error correcting codes. Error correcting codes are an invaluable method of adding redundancy to data so that the initial data can be recovered even after the introduction of a bounded number of errors. Errors in raw natural data with no prior knowledge of its structure are usually considered beyond the feasible scope of recovery. Nonetheless, it was recently [4] shown, that data regularity, even if its structure is unknown a-priori, can serve as an aid to error recovery.

Regularities in strings arise in various areas of science, including coding and automata theory, formal language theory, combinatorics, molecular biology and many others. A typical form of regularity is *periodicity*, meaning that a “long” string T can be represented as a concatenation of copies of a “short” string P, possibly ending in a prefix of P. Periodicity has been extensively studied in Computer Science over the years (see [28]).

1.1 Regularities and Data Recovery

Recently, it was shown [4] that periodicity can serve as an aid to error recovery. It was proven that if no more than $\frac{n}{(1+\epsilon)p}$ mismatch errors are introduced to a periodic string of length n having period of length p then, even if p is not known a-priori, it is possible to recover $O(\log n)$ possible candidates, one of which is *guaranteed* to be the original period. This surprising result was further reinforced by discovering that a similar result holds not just for mismatch error corruptions bounded by the Hamming distance, but for any errors bounded by a pseudo local metric (e.g. the swap or interchange metrics). An interesting additional result was that even under some non-pseudo local metrics, such as the edit distance, periodicity can still allow recovery of $O(\log n)$ candidate periods [4, 2]. However, these candidate periods are distinguished in that none are cyclic rotations of each other. In other words, if we take one representative of all candidates that are cyclic rotations of each other, we end up with the small number of candidates. It was unknown whether there are other regularities in natural phenomena that allow recovery of the original string. Identifying such a type of regularity is the first topic of this paper.

In particular, for many phenomena, it is desirable to broaden the definition of periodicity and study wider classes of repetitive patterns in strings. One common such notion is that of a *cover*, defined as follows.

▶ **Definition 1** (Cover)**.** A length m substring C of a string T of length n, is said to be a *cover* of T, if $n > m$ and every letter of T lies within some occurrence of C.

Note that the string C is both a prefix and a suffix of the string T. For example, consider the string $T = abaababaaba$. Clearly, T is “almost” periodic with period aba, however, as it is not completely periodic, the algorithms that exploit repetitions cannot be applied to it. On the other hand, the string $C = aba$ is a cover of T, which allows applying to T cover-based algorithms. We study error correction feasibility for coverable phenomena.

1.2 Related Work

We review related regularity types and other approaches to handle errors in regularities. Quasi-periodicity was introduced by Ehrenfeucht in 1990 (according to [7]). The earliest paper in which it was studied is by Apostolico, Farach and Iliopoulos [9], which defined the *quasi-period* of a string to be the length of its shortest cover and presented an algorithm for computing the quasi-period of a given string in $O(n)$ time and space. The new notion attracted immediately several groups of researchers (e.g. [10], [29, 30], [27], [11]). An overview on the first decade of the research on covers can be found in the surveys [7, 20, 32].

While covers are a significant generalization of the notion of periods as formalizing regularities in strings, they are still restrictive, in the sense that it remains unlikely that an arbitrary string has a cover shorter than the word itself. Due to this reason, different variants of quasi-periodicity have been introduced. These include *seeds* [19], *maximal quasi-periodic substring* [8], the notion of *k-covers* [21], *λ-cover* [33], *enhanced covers* [16], *partial cover* [23]. Since the notion of a seed is necessary to our study and presentation of results, we give its formal definition here.

▶ **Definition 2** (Seed)**.** A length m substring C of a string T of length n, is said to be a *seed* of T, if $n > m$ and there exists a superstring T' of T such that C is a cover of T'.

Note that the first and last occurrence of the seed C in T may be incomplete. Other recently explored directions include the inverse problem for cover arrays [14], extensions to strings in which not all letters are uniquely defined, such as *indeterminate strings* [6] or *weighted sequences* [34]. Some of the related problems are $\mathcal{NP}$-hard (see e.g., [6, 12, 23]).

In applications such as molecular biology and computer-assisted music analysis, finding exact repetitions and covers is not always sufficient. A more appropriate notion is that of *approximate repetitions*, where errors are allowed (see, e.g., [13, 15]). This notion was first studied in 1993 by Landau and Schmidt [25, 26] who concentrated on approximate tandem repeats. Note that, the natural definition of an approximate repetition is not clear. One possible definition is that the distance between any two adjacent repeats is small. Another possibility is that all repeats lie at a small distance from a single "original". Such a definition of *approximate seeds* is studied in [12, 18, 17]. Indeed, all these definitions along with other ones were proposed and studied (see [3, 24, 31]). Yet another possibility is that all repeats must be equal, but we allow a fixed total number of mismatches. The possibility presented in [3] is a global one, assuming that an original unknown string is a sequence of repeats without errors, but the process of sequence creation or transmission incurs errors to the sequence of repeats, and, thus, the examined input string is not a sequence of repeats. Therefore, a (smallest) repeat generating a string with the minimum total number of mismatches with the input string is sought. Extension of this definition approach to approximate covers is another topic of this paper.

1.3 Our Results

In this paper we show that *coverability* is also a tool that allows error correction. We formally define the *Cover Recovery Problem (CRP)* and characterize the feasibility of its solution. In particular, we show:

▶ **Theorem 3.** *Let S be a string coverable by a cover C of length c, and let $\varepsilon > 0$. Assume that at most $\frac{n}{(2+\varepsilon)c}$ mismatch errors were introduced to S resulting in a string S'. Then there exist $O(\log n)$ possible primitive substrings of S', one of which is guaranteed to be C or a seed of C.*

In addition, extending the approach of [3] to the notion of covers, [5] define the *approximate cover problem (ACP)*, in which we are given a text that is a sequence of some cover repetitions with possible mismatch errors. Since the ACP is proved to be $\mathcal{NP}$-hard [5], we study a relaxation of this problem. In our relaxation, which we call *the candidate relaxation of the ACP*, a candidate cover is also given, and we seek to align it with the given text (this alignment is called *a tiling*) such that the number of mismatches is minimized. This scenario is quite realistic in the case where a cover is sought for a string where the errors are distributed in a manner that at least one occurrence of the cover appears in the string without errors. We examine this relaxation and show it has polynomial time complexity. As a result, we get that the ACP also has polynomial time complexity in many practical situations. This ACP relaxation study enables also an efficient algorithm for recovering the candidate covers in CRP.

Paper Contributions. The main contributions of this paper are:

- Proving that recovery of raw data from errors is possible not only for periodic phenomena but also for the less rigid coverable phenomena.
- Demonstrating that efficient recovery is feasible even when the underlying problem of computing an approximate cover is $\mathcal{NP}$-hard. This is in line with the previous result of [4] that show efficient recovery for the interchange metric, which is $\mathcal{NP}$-hard to compute.
- Formalizing the candidate relaxation of the ACP and showing it is polynomial time computable. This study served both to give a solution to the CRP and to suggest an efficient solution for the ACP in many practical situations.

The paper is organized as follows. In Section 2, we give formal definitions and basic lemmas. In Section 3, we study the cover recovery problem (CRP) and characterize the extent to which the cover of the unknown uncorrupted original string can be recovered given the possibly corrupted by mismatch errors input string. In Section 4, we study the candidate relaxation of the ACP with its application to the ACP itself and, more importantly, to the CRP. We conclude with some open problems in Section 5.

2 Preliminaries

In this section we give the needed formal definitions and basic lemmas.

▶ **Definition 4** (Tiling). Let T be a string over alphabet Σ such that the string C over alphabet Σ is a cover of T. Then, the sorted list of indices representing the start positions of occurrences of the cover C in the text T is called the *tiling* of C in T.

In this paper we have a text T which may have been introduced to errors and, therefore, is not coverable. However, we would like to refer to a retained tiling of an unknown string C in T although C does not cover T because of mismatch positions. The following definition makes a distinction between a list of indices that may be assumed to be a tiling of the text before mismatch errors occurred and a list of indices that cannot be such a tiling.

▶ **Definition 5** (A Valid Tiling). Let T be an n-length string over alphabet Σ and let L be a sorted list of indices $L \subset \{1, ..., n\}$. Let $m = n + 1 - L_{last}$, where L_{last} is the last index in L. Then, L is called *a valid tiling* of T, if $i_1 = 1$ and for every $i_k, i_{k+1} \in L$, it holds that $i_{k+1} - i_k \leq m$.

▶ **Notation 1.** *Let C be an m length string over alphabet Σ. Denote by $S(C)$ a string of length n, $n > m$, such that C is a cover of $S(C)$.*

Note that $S(C)$ is not uniquely defined even for a fixed $n > m$, since every different valid tiling of the m-length string C generates a different n-length string $S(C)$. A unique version can be obtained if a valid tiling L is also given.

▶ **Notation 2.** *Let T be an n-length string over alphabet Σ and let L be a valid tiling of T. Let $m = n + 1 - L_{last}$, where L_{last} is the last index in the tiling L. For any m-length string C', let $S_L(C')$ be the n-length string obtained using C' as a cover and L as the tiling as follows: $S_L(C')$ begins with a copy of C' and for each index i in L a new copy of C' is concatenated starting from index i of $S_L(C')$ (running over a suffix of the last copy of C' if the difference between i and the previous index in L is less than m).*

▶ **Definition 6.** Let T be a string of length n over alphabet Σ. Let H be the Hamming distance. The *distance of T from being covered* is:

$$dist = \min_{C \in \Sigma^*, |C|<n, S(C) \in \Sigma^n} H(S(C), T).$$

We will also refer to $dist$ as *the number of errors in T*.

▶ **Definition 7.** Let T be an n-long string over alphabet Σ. An m-long string C over Σ, $m \in \mathbb{N}$, $m < n$, is called *an m-length approximate cover of T*, if for every string C' of length m over Σ, $\min_{S(C') \in \Sigma^n} H(S(C'), T) \geq \min_{S(C) \in \Sigma^n} H(S(C), T)$, where H is the Hamming distance of the given strings.
We refer to $\min_{S(C) \in \Sigma^n} H(S(C), T)$ as the *number of errors of an m-length approximate cover of T*.

▶ **Definition 8** (Approximate Cover). Let T be a string of length n over alphabet Σ. A string C over alphabet Σ is called an *approximate cover of T* if:
1. C is an m-length approximate cover of T for some $m \in \mathbb{N}$, $m < n$, for which

$$\min_{S(C) \in \Sigma^n} H(S(C), T) = dist.$$

2. for every m'-length approximate cover of T, C', s.t. $\min_{S(C') \in \Sigma^n} H(S(C'), T) = dist$, it holds that: $m' \geq m$.

Primitivity. By definition, an approximate cover C should be *primitive*, i.e., it cannot be covered by a string other than itself (otherwise, T has a cover with a smaller length). Note that a periodic string can be covered by a smaller string (not necessarily the period), and therefore, is not primitive.

▶ **Definition 9.** The *Approximate Cover Problem* (ACP) is the following:
INPUT: String T of length n over alphabet Σ.
OUTPUT: An approximate cover of T, C, and the number of errors in T.

The goal of the following definition and lemmas is Lemma 15, which is a crucial tool for the efficiency of the candidate relaxation algorithm.

▶ **Definition 10** (String Mask). Given a string C of length m, the mask M of C is a boolean array of length m, such that $M[i] = 1$ if and only if the suffix $C[i..m]$ is equal to the prefix $C[1..m-i+1]$.

▶ **Lemma 11.** *Let C be a string of length m and let M be its mask. Let i, j be indices such that $1 \leq i < j \leq m$ and $M[i] = M[j] = 1$, then the substring $C[i..m]$ has a period of length $j - i$.*

▶ **Lemma 12.** *Let C be a primitive string of length m and let M be its mask. Let i be the smallest index such that $1 < i \leq m$ and $M[i] = 1$, then $i > \lfloor \frac{m}{2} \rfloor + 1$.*

▶ **Lemma 13.** *Let C be a string of length m and let M be its mask. Let i, j be indices such that $M[i] = M[j] = 1$, $j - i = g > 0$. Let k be the minimal index such that $j < k \leq m$ and $M[k] = 1$. Then, $k = j + g$ or $k \geq j + \lfloor \frac{g}{2} \rfloor$.*

▶ **Lemma 14.** *Let C be a string of length m and let M be its mask. Let i, j, k, ℓ be indices such that $i < j$, $k < \ell$, $M[i] = M[j] = M[k] = M[\ell] = 1$ and $j - i = \ell - k$ then $C[i..j-1] = C[k..\ell-1]$.*

▶ **Lemma 15.** *Let C be a primitive string of length m and let M be its mask. Let I_M be the sorted list of indices i such that $1 \leq i \leq m$ and $M[i] = 1$. Let $S_C = \{C[i_k..i_{k+1} - 1] \mid i_k, i_{k+1}$ are adjacent indices in $I_M\} \cup \{C[i_{last}..m] \mid i_{last} = \max_{i_k \in I_M} i_k\}$ be a set of substrings of C. Then, $|S_C| = O(\log m)$.*

3 Characterization of the Cover Recovery Problem Approximation

In this section we study the Cover Recovery Problem (CRP) and characterize the extent to which the cover of the original unknown uncorrupted original string can be recovered given the possibly corrupted by mismatch errors input string. The term *approximation* here refer to the ability to give a relatively small size set of candidates that *includes* the *exact* cover of the original string or a seed of it. We begin with a formal definition of the CRP problem.

▶ **Definition 16** (The Cover Recovery Problem).
INPUT: An $\varepsilon > 0$ and a string S' of length n over alphabet Σ, which is a string S covered by the primitive cover C possibly corrupted by at most $\frac{n}{(2+\varepsilon)c}$ mismatch errors, where c is the length of C.
OUTPUT: A small size set O of candidate strings such that $C \in O$.

First, we show the bounds on the number of errors that still guarantees a small-size set O of candidates. We then prove a bound on the size of this set O. In Section 4 we then conclude how this set can be identified, and thus the original uncorrupted string can be *approximately* recovered. Some more formal definitions and lemmas are needed. We start with the definitions of *alignment* and *neighbourhood* that we use to prove the bound on the number of errors that still enable a recovery.

▶ Remark. Throughout this section we use c to denote a cover length and C the cover string, i.e., $c = |C|$.

▶ **Definition 17.** Let $S = S[1], \ldots, S[s]$ and $T = T[1], \ldots, T[t]$ be strings, and let $1 \leq i \leq |T|$. The *alignment of S with T in location i* is the comparison of $S[j]$ and $T[i + j - 1], \forall j = 1, \ldots, \min(s, t - i + 1)$. In other words, we place S above T such that the first location of S is aligned with the i-th location of T.

▶ **Definition 18.** Let $C = C[1], \ldots, C[c]$ be a primitive cover, and let $1 \leq i \leq c$. We call i a *neighbouring index* of C if $\forall j,\ j = 1, \ldots, c - i$, we have $C[i + j] = C[j]$. For any neighbouring index i, denote by $C \circ_i C$ the string composed of the prefix of length i of C concatenated by C. We call $C \circ_i C$ *the neighbourhood of C at index i*. In particular, if $i = c$ then $C \circ_i C$ is C^2, the concatenation of C with itself.

If we are interested in a neighbourhood of C where the location is not important, we will denote it by $C \circ C$.

Lemma 19 is the basic building block in our error bound proof.

▶ **Lemma 19.** *Let C be a primitive cover and $C \circ_i C$ be a neighbourhood of C at location i. Then for every $j \neq i$, $1 < j \leq c$, the alignment of C with $C \circ_i C$ in location j has at least one mismatch.*

Proof. Because C is a primitive cover, then $i > c/2$, by Lemma 12. If $1 < j \leq c/2$ then an exact alignment leads to non-primitivity of C, contradiction. However, if there is an exact alignment for $c/2 < j \neq i$, then $|j - i| < c/2$ and thus we again have a contradiction to the primitivity of C. Therefore, there must be at least one mismatch in an alignment at any index $j \neq i$. ◀

We make use of following lemma for proving the upper bound on the number of candidates in our output set.

▶ **Lemma 20.** *Let S and C be two primitive strings such that C is a seed of S. Then there is at most one string S' with the following properties:*

1. *S' is covered by C.*
2. *S is a substring of S'*
3. *S' is the shortest string with properties 1 and 2 above.*

Proof. Assume there are two such strings, S' and S''. Since they are both shortest possible superstrings of S (i.e., strings containing S as a substring), then S matches each of them in their first occurrence of C. If $S' \neq S''$ then there must be at least one index i in S where C starts in S' but not in S''. However, then by Lemma 19 there must be at least one mismatch in the alignment of at least one of them with S, contradiction to the fact that S is a substring of both of them. ◀

▶ **Lemma 21.** *Let $n \in \mathbb{N}$ and let S_1, S_2 be two n-long coverable strings with C_1 and C_2 the covers of S_1 and S_2 respectively, where $c_1 \geq c_2$ and C_2 is not a seed of C_1. Then*

$$H(S_1, S_2) \geq \frac{n}{c_1}.$$

We are now ready to prove our approximation bound for the CRP. Lemma 22 is needed for proving our characterization theorem.

▶ **Lemma 22.** *Let $\varepsilon > 0$ be a constant, S an n-long string, and C_1, C_2 are c_1 and c_2-length approximate seeds of S with at most $\frac{n}{(2+\varepsilon)\cdot c_1}$, $\frac{n}{(2+\varepsilon)\cdot c_2}$ errors respectively (w.l.o.g. assume that $c_1 \geq c_2$), where C_2 is not a seed of C_1. Then,*

$$c_1 \geq (1+\varepsilon)\cdot c_2$$

Proof. Let S_1 be the n-long string such that C_1 is its seed and S_2 be the n-long string such that C_2 is its seed. We are given that $H(S_1, S) \leq \frac{n}{(2+\varepsilon)\cdot c_1}$ and $H(S_2, S) \leq \frac{n}{(2+\varepsilon)\cdot c_2}$. Therefore,

$$\frac{n}{(2+\varepsilon)\cdot c_1} + \frac{n}{(2+\varepsilon)\cdot c_2} \geq H(S_1, S) + H(S_2, S).$$

By triangle inequality we have,

$$H(S_1, S) + H(S_2, S) \geq H(S_1, S_2).$$

By Lemma 21,

$$H(S_1, S_2) \geq \frac{n}{c_1}.$$

Therefore,

$$\frac{n}{(2+\varepsilon)\cdot c_1} + \frac{n}{(2+\varepsilon)\cdot c_2} \geq \frac{n}{c_1}$$

from which we get,

$$c_2 + c_1 \geq (2+\varepsilon)c_2$$

or,

$$c_1 \geq (1+\varepsilon)c_2.$$ ◀

We conclude with our characterization theorem, which is a more accurate version of Theorem 3.

▶ **Theorem 23.** *Let S be an n-long string. Then, there are at most $\log_{1+\varepsilon} n + 1$ different c-length approximate covers C of S with at most $\frac{n}{(2+\varepsilon)\cdot c}$ errors such that none is a seed of another.*

Proof. First, note that there cannot be two such different c-length approximate covers unless one is a seed of the other, because then, by Lemma 22, we get $c \geq (1+\varepsilon)c$, contradiction. Thus, such different c-length approximate covers must have different length. Now, let $1 \leq l_1 < l_2 < \ldots < l_{t-1} < l_t \leq n$ be the different lengths of c-length approximate covers of S. By Lemma 22,

$$(1+\varepsilon)^{t-1} \leq (1+\varepsilon)^{t-1} \cdot l_1 < (1+\varepsilon)^{t-2} \cdot l_2 < \ldots < (1+\varepsilon)^2 \cdot l_{t-2} < (1+\varepsilon) \cdot l_{t-1} < l_t \leq n$$

Therefore, $t - 1 \leq \log_{1+\varepsilon} n$. ◀

▶ **Example 24.** We now show an example where a string has many substrings that all cover the given string with two errors. However, all these substrings have a single shortest 2-error seed. Consider the string $S = aaaaaaaaa(baaaa)^k baaaaaaaaa$. Then, all the following primitive strings cover S with two errors: *aaaabaaaa*, *aaaabaaa*, *aaaabaa*, *aaaaba*, *aaabaaaa*, *aabaaaa*, *abaaaa*. They all have either *abaaaa* or *aaaaba* as a seed. Note that there are 2 such shortest 2-error covers, however, each is a seed of the other.

4 The Candidate Relaxation of the ACP

In this section we study the following relaxation of the approximate cover problem:

▶ **Definition 25** (The Candidate Relaxation of the ACP).
INPUT: String T of length n over alphabet Σ, and a candidate cover C of length m over alphabet Σ.
OUTPUT: $\min_{S(C)\in\Sigma^n} H(S(C),T)$, i.e., the minimum number of errors in any valid tiling of C in T.

▶ Remark. If $k = \min_{S(C)\in\Sigma^n} H(S(C),T)$, we use the term k-error cover for the given C.

Note that, since a candidate cover must be primitive, we may assume that this is indeed the case. A linear-time verification is possible using the algorithm of [9]. We describe a dynamic programming algorithm for this problem, which uses the well-known Knuth-Morris-Pratt [22] and Abrahamson-Kosaraju [1] algorithms. Our algorithm consists of a preparation phase, and a dynamic programming phase. We denote by m^* the number of set bits in the mask M of the given candidate C.

4.1 The preparation phase

The preparation phase is composed of the following three stages:

1. **Computing the mask of C.** This computation can be performed efficiently using the KMP algorithm. We compute the "failure automaton" for C. Denote the states of the automaton by $s_0, s_1, s_2, \ldots, s_m$. We consider the final state s_m of the automaton, and follow the sequence of fail links that start from it. Assume that this sequence is s_m, s_{i_1}, s_{i_2}, etc. The first link in the sequence means that C_1, the longest proper prefix of C that is equal to the corresponding suffix, is of length i_1. The second link means that C_2, the

longest proper prefix of C_1 that is equal to the corresponding suffix of C_1, is of length i_2. However, C_2 is also the second longest prefix of C that is equal to the corresponding suffix of C. By continuing in this process, we obtain the sequence $C_1, C_2, \ldots$ of *all* prefixes of C that are equal to the corresponding suffixes. Hence, the corresponding sequence of lengths $i_1, i_2, \ldots$ gives the (decreasing) sequence of indices $j_\ell = m - i_\ell + 1$, for which $M[j_\ell] = 1$, where M is the mask of C.

2. **Dividing C into disjoint substrings.** We divide C into substrings according to the indices i for which $M[i] = 1$. Specifically, if the (increasing) sequence of indices i for which $M[i] = 1$ is $i_1, i_2, \ldots i_{m^*}$ where $1 = i_1 < i_2 < \ldots < i_{m^*}$, then the substrings we consider are all substrings of C of the form $s_j = C[i_j..i_{j+1} - 1]$, for $1 \leq j \leq m^* - 1$, along with the suffix $s_{m^*} = C[i_{m^*}..m]$.
3. **Computing the Hamming distance from substrings of T to the strings s_j.** For each string s_j, $1 \leq j \leq m^*$, we compute its Hamming distance to all substrings of T simultaneously using the Abrahamson-Kosaraju algorithm. Since for many values of j, s_j is equal to s_{j-1} (actually, by Lemma 15, the sequence $s_1, s_2, \ldots, s_{m^*}$ contains only $O(\log m)$ distinct elements), we first check whether $s_j = s_{j-1}$ and apply the Abrahamson-Kosaraju algorithm only in the rare cases of inequality. The array of Hamming distances returned by the Abrahamson-Kosaraju algorithm is denoted below by $Hamming(s_j, T)$.

4.2 The dynamic programming phase

When the preparation phase is done, we are ready to compute the minimal k such that C is a k-error cover of T. This computation is performed in a dynamic fashion. Namely, we go over all suffixes of T in an increasing order, and for each suffix $T[i..n]$, we compute the minimal $k(T[i..n])$ such that C is a $k(T[i..n])$-cover of $T[i..n]$, utilizing the computations performed for the previous suffixes. The values $k(T[i..n])$ are stored in an array MIN, where $MIN[i] = k(T[i..n])$. In the beginning of the algorithm, all values of MIN are initialized to ∞. The output of the algorithm is $MIN[1]$.

As a cover must be a suffix of the covered string, we have $MIN[i] = \infty$ for all $i > n-m+1$, meaning that there does not exist a string of length $n - i + 1$ that can be covered by C. For the same reason, $MIN[n - m + 1] = H(C, T[n - m + 1..n])$, as there is a unique way to cover a string of length m by C. Since any two overlapping occurrences of C in a tiling that covers the suffix $T[i..n]$ must differ by a value j such that $M[j + 1] = 1$, and since $|s_1| = \min(\{j : 1 < j \leq m, M[j + 1] = 1\})$, it is impossible to cover a string of length $m + j$, $1 \leq j < |s_1|$, by copies of C. Thus, $MIN[i] = \infty$ for all $n - m - |s_1| + 1 < i < n - m + 1$. The following steps are performed for all $i \leq n - m - |s_1|$, in a decreasing order.

For each such i, we go over all possible strings of length $n - i + 1$, $S_{L_i}(C)$ that cover $T[i..n]$ by C with k-errors (resulted from different tiling L_i for which its first index is aligned with index i in the text). As each such tiling must start with a copy of C, and as the second occurrence of C in this tiling must differ from the initial one either by m or by a value j such that $M[j + 1] = 1$, we can compute the minimal number of error in any such tiled strings $S_{L_{ij}}(C)$ (for which the first occurrence of C is aligned with index i in T and the second occurrence of C is index j) as $Error(S_{L_{ij}}(C)) = H(C[1..j], T[i..i + j - 1]) + MIN[i + j]$ (note that by the structure of the algorithm, $MIN[i + j]$ is already known at this stage.) The value $MIN[i]$ is given by:

$$MIN[i] = \min_{j \in \{j : M[j+1]=1\} \cup \{m\}} Error(S_{L_{ij}}(C)).$$

Naively, we can go over all m^* possible values of j, compute $Error(S_{L_{ij}}(C))$ for each of them, and find out the minimum. For the sake of efficiency, we compute these values incrementally, by advancing the starting point of the second occurrence of C in the covering by $|s_j|$ every time. Formally, this is performed as follows.

We define a counter j that corresponds to the initial shift of the second occurrence of C in the tiling relative to the position i in T. j is initialized to 0. Then, for $\ell = 1, 2, \dots, m^*$, we advance j by $|s_\ell|$ and check whether $H(C[1..j], T[i..i+j-1]) + MIN[i+j]$ for $j = \sum_{r=1}^{\ell} |s_r|$ is lower than the previously best value of $Error$. If the answer is positive, the temporary value of $MIN[i]$ is replaced by $H(C[1..j], T[i..i+j-1]) + MIN[i+j]$.

In order to compute the values $H(C[1..j], T[i..i+j-1])$ efficiently, we observe that for $j = \sum_{r=1}^{\ell} |s_r|$, we have

$$\begin{aligned} H(C[1..j], T[i..i+j-1]) &= H(s_1, T[i..i+|s_1|-1]) \\ &+ H(s_2, T[i+|s_1|..i+|s_1|+|s_2|-1]) + \dots \\ &+ H(s_\ell, T[i+\sum_{r=1}^{\ell-1}|s_r|..i+\sum_{r=1}^{\ell-1}|s_r|]) \end{aligned}$$

Hence, we compute $H(C[1..j], T[i..i+j-1])$ incrementally by keeping a counter err, initializing it to 0, and advancing it by $H(s_\ell, T[i+\sum_{r=1}^{\ell-1}|s_r|..i+\sum_{r=1}^{\ell-1}|s_r|])$ when j is advanced by $|s_\ell|$. Finally, in order to skip unnecessary operations, for each ℓ we check whether $i+j+|s_\ell| \leq n-m+1$, as otherwise, an occurrence of C clearly cannot start at position $i+j$.

After going over $\ell = 1, 2, \dots, m^*$, we fix the last temporary value $MIN[i]$ to be its final value, and proceed to $i-1$. As mentioned before, $MIN[1]$ is the output of the algorithm. A pseudo-code of the algorithm is presented in Figure 1.

The correctness of the Candidate Relaxation Dynamic Programming algorithm is given in Lemma 26. The complexity of the algorithm is given in Lemma 27.

▶ **Lemma 26.** *Let T be a length-n string and let C be a length-m cover. Let MIN be the final array obtained by the dynamic programming algorithm described above with input T and C. Then for any $1 \leq i \leq n$, $MIN[i]$ is equal to the minimal k such that C is a k-error cover of $T[i..n]$.*

Proof. The proof is by an inverse induction on i. The induction basis is the cases $i > n-m-|s_1|+1$, for which $MIN[i]$ was calculated explicitly above and is easily seen to be equal to their final value computed by the algorithm.

Assume that the claim holds for all $i > i_0$, and consider the case $i = i_0$. Let $S_{L_{i_0}}(C)$ be the tiled string of $T[i_0..n]$ by copies of C starting from index i_0, for which the minimal number of errors $k(T[i_0..n])$ is attained. The tiling $S_{L_{i_0}}(C)$ must start with a copy of C, and the second occurrence of C in $S_{L_{i_0}}(C)$ must differ from the initial one either by m or by a value j such that $M[j+1] = 1$. As the total error of $S_{L_{i_0}}(C)$ is $k(T[i_0..n])$, we have

$$k(T[i_0..n]) \geq H(C[1..j], T[i..i_0+j-1]) + k(T[i_0+j..n]).$$

On the other hand, by the structure of our algorithm, its outputs satisfy

$$MIN[i] \leq H(C[1..j], T[i..i_0+j-1]) + MIN[i_0+j] = H(C[1..j], T[i..i_0+j-1]) + k(T[i_0+j..n]),$$

where the equality holds by the induction assumption. Hence, $MIN[i_0] \leq k(T[i_0..n])$. Finally, since $MIN[i]$ is obtained in the algorithm by computing the error of a concrete cover (that can be traced inductively), it is clear that $MIN[i] \geq k(T[i_0..n])$. This completes the proof. ◀

```
The Candidate Relaxation Dynamic Programming Algorithm
Input: A string $T$ of length $n$, and a candidate cover $C$ of length $m$
1  find the mask $M$ of $C$ using the KMP algorithm
2  $start \leftarrow 1$
3  for $i \leftarrow 2$ to $m$ do
4          if $M[i] = 1$ then
5                  $s \leftarrow s \cup C[start..i-1]$
6                  $start \leftarrow i$
7  $s \leftarrow s \cup C[start..m]$
8  for each substring $s_i$ do
9          if $|s_i| = |s_{i-1}|$ then
10                 $Hamming(s_i, T) \leftarrow Hamming(s_{i-1}, T)$
11         else
12                 $Hamming(s_i, T) \leftarrow Abrahamson-Kosaraju(s_i, T)$
13 for $i \leftarrow 1$ to $n$ do
14         $MIN[i] \leftarrow \infty$
15 $MIN[n-m+1] \leftarrow H(C, T[n-m+1..n])$
16 for $i \leftarrow n-m+1-|s_1|$ to 1 by -1 do
17         $j \leftarrow 0$
18         $err \leftarrow 0$
19         for each substring $s_\ell$ do
20                 if $j + |s_\ell| \leq n - m$ then
21                         $err \leftarrow err + Hamming(s_\ell, T[i+j])$
22                         if $MIN[i] > err + MIN[i+j+|s_\ell|]$ then
23                                 $MIN[i] \leftarrow err + MIN[i+j+|s_\ell|]$
24                         $j \leftarrow j + |s_\ell|$
Output:
25 $MIN[1]$
```

Figure 1 The dynamic programming algorithm for the candidate relaxation of the ACP.

▶ **Lemma 27.** *Let T be a text of length n and C a candidate cover of length m. Then, the time complexity of the Candidate Relaxation Dynamic Programming algorithm on T and C is $O(n \cdot m^* + n\sqrt{m}\log m)$, where m^* is the number of set bits in the mask M of C.*

Proof. First, we analyze the preparation phase of the algorithm. As explained above in the description of the algorithm, computing the mask M of C can be done by running the KMP algorithm for C, which requires $O(m)$ operations. Dividing C into disjoint substrings given the mask M of C can clearly be done in $O(m)$ operations. Computing the Hamming distance from substrings of T to the strings s_j can be performed by applying the Abrahamson-Kosaraju algorithm once for each of the substrings s_j. As by Lemma 15, the number of distinct substrings s_j is $O(\log m)$, the Abrahamson-Kosaraju algorithm is applied only $O(\log m)$ times, while for the other values of j (whose total number is bounded from above by m) we perform only a simple "copy" operation. The complexity of each application of the Abrahamson-Kosaraju algorithm is $O(n\sqrt{m}\log m)$, and hence, the total complexity of this step is $O(\log m \cdot n\sqrt{m}\log m)$.

A refinement of the analysis of this computation shows that the complexity is actually $O(n\sqrt{m}\log m)$. Note that the Abrahamson-Kosaraju algorithm is applied for distinct strings of the form s_j. Consider the lengths of these strings. By Lemma 15, if we denote $|s_k| = g_k$ and let h_k denote the distance from the end of s_k to the end of C, we have that whenever

$s_{k+1} \neq s_k$, either $g_{k+1} \leq g_k/2$ or $h_{k+1} \leq 3h_k/4$. Moreover, as the latter condition arises only in the case $h_k \leq 2g_k$ (see the proof of Lemma 15), it follows that the sequence of lengths $g_1 > g_2 > \dots$ of strings on which the Abrahamson-Kosaraju algorithm is applied satisfies $g_{k+4} < g_k/4$. Since $g_1 \leq m$, the total complexity of this step is at most $O(n\sqrt{m \log m})$.

We now analyze the dynamic programming phase. The main loop of the dynamic programming is performed for all $1 \leq i \leq n - m - |s_1|$, i.e., $O(n)$ times. For each i, we go over the m^* strings s_j, and for each of them, we perform a few simple operations (i.e., table lookups and comparisons). Hence, the time complexity of this phase is $O(n \cdot m^*)$.

Therefore, the total time complexity of the algorithm is $O(n \cdot m^* + n\sqrt{m \log m})$. ◀

This completes the proof of Theorem 28.

▶ **Theorem 28.** *Given a text T of length n a candidate cover C of length m over alphabet Σ. Then, the candidate relaxation of the approximate cover problem of T can be solved in $O(n \cdot m^* + n\sqrt{m \log m})$ time, where m^* is the number of set bits in the mask M of C.*

Theorem 28 has the following useful applications to the ACP (Corollary 29) and CRP (Corollary 30).

▶ **Corollary 29.** *Let T be a text of length n over alphabet Σ. Denote by $\gamma(T)$ the maximum of $m^* + \sqrt{m \log m}$ over all primitive substrings C of T with length $m < n$, where m^* is the number of set bits in the mask M of C. Assume that the error distribution guarantees that at least one occurrence of an approximate cover of the text is without errors. Then, the approximate cover problem of T can be solved in $O(n^3 \cdot \gamma(T))$ time.*

Proof. The condition implies that C is a substring of T. Take each of the $O(n^2)$ primitive substrings of T of length less than n as a candidate cover in the algorithm and run the dynamic programming algorithm of Figure 1. The corollary then follows from Theorem 28. ◀

▶ **Corollary 30.** *Let S be a n-long string and $\varepsilon > 0$. Denote by $\gamma(S)$ the maximum of $m^* + \sqrt{m \log m}$ over all primitive substrings C of S with length $m < n$, where m^* is the number of set bits in the mask M of C. Then, a set of at most $\log_{1+\varepsilon} n$ different m-length approximate covers C of S such that none is a seed of another, each with at most $\frac{n}{(2+\varepsilon)\cdot m}$ errors, can be constructed in $O(n^3 \cdot \gamma(S))$ time.*

Proof. Use the same algorithm as in the proof of Corollary 29 but retain as candidates in the output set only m-length approximate covers C of S, for which the candidate relaxation algorithm finds at most $\frac{n}{(2+\varepsilon)\cdot m}$ errors. From this set retain only candidates that do not have a shorter or same length candidates as seeds. ◀

5 Open Problems

In this paper we initiated the study of the CRP as well as a new relaxation of the ACP. Some interesting questions and open problems are:

- Since the ACP is proved to be $\mathcal{NP}$-hard, it is interesting to find other polynomial time relaxations of the ACP, besides the candidate relaxation studied in this paper. Such a study will broaden our understanding as well as suggest practical solutions.
- In this paper we considered the Hamming distance as a metric in the definition of approximate cover. Other string metrics can be considered as well. It is interesting to see if and how the complexity of the problem changes with the use of other string metrics.

References

1 Karl R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987. doi:10.1137/0216067.

2 Amihood Amir, Mika Amit, Gad M. Landau, and Dina Sokol. Period recovery over the Hamming and edit distances. In Evangelos Kranakis, Gonzalo Navarro, and Edgar Chávez, editors, *Proceedings of the 12th Latin American Symposium on Theoretical Informatics (LATIN 2016)*, volume 9644 of *LNCS*, pages 55–67. Springer, 2016. doi:10.1007/978-3-662-49529-2_5.

3 Amihood Amir, Estrella Eisenberg, and Avivit Levy. Approximate periodicity. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Proceedings of the 21st International Symposium on Algorithms and Computation (ISAAC 2010)*, volume 6506 of *LNCS*, pages 25–36. Springer, 2010. doi:10.1007/978-3-642-17517-6_5.

4 Amihood Amir, Estrella Eisenberg, Avivit Levy, Ely Porat, and Natalie Shapira. Cycle detection and correction. *ACM Trans. Algorithms*, 9(1):13:1–13:20, 2012. doi:10.1145/2390176.2390189.

5 Amihood Amir, Avivit Levy, Ronit Lubin, and Ely Porat. Approximate cover of strings. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *LIPIcs*, pages 26:1–26:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.26.

6 Pavlos Antoniou, Maxime Crochemore, Costas S. Iliopoulos, Inuka Jayasekera, and Gad M. Landau. Conservative string covering of indeterminate strings. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference (PSC 2008)*, pages 108–115. Czech Technical University in Prague, 2008. URL: http://www.stringology.org/event/2008/p10.html.

7 Alberto Apostolico and Dany Breslauer. Of periods, quasiperiods, repetitions and covers. In Jan Mycielski, Grzegorz Rozenberg, and Arto Salomaa, editors, *Structures in Logic and Computer Science: A Selection of Essays in Honor of Andrzej Ehrenfeucht*, volume 1261 of *LNCS*, pages 236–248. Springer, 1997. doi:10.1007/3-540-63246-8_14.

8 Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theor. Comput. Sci.*, 119(2):247–265, 1993. doi:10.1016/0304-3975(93)90159-Q.

9 Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Inf. Process. Lett.*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.

10 Dany Breslauer. An on-line string superprimitivity test. *Inf. Process. Lett.*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.

11 Dany Breslauer. Testing string superprimitivity in parallel. *Inf. Process. Lett.*, 49(5):235–241, 1994. doi:10.1016/0020-0190(94)90060-4.

12 Manolis Christodoulakis, Costas S. Iliopoulos, Kunsoo Park, and Jeong Seop Sim. Approximate seeds of strings. *J. Autom. Lang. Comb.*, 10(5/6):609–626, 2005.

13 Tim Crawford, Costas S. Iliopoulos, and Rajeev Raman. String-matching techniques for musical similarity and melodic recognition. In Walter B. Hewlett and Eleanor S. Field, editors, *Melodic Similarity: Concepts, Procedures, and Applications*, volume 11 of *Computing in Musicology*, pages 73–100. MIT Press, Cambridge, Massachusetts, 1998.

14 Maxime Crochemore, Costas S. Iliopoulos, Solon P. Pissis, and German Tischler. Cover array string reconstruction. In Amihood Amir and Laxmi Parida, editors, *Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM 2010)*, volume 6129 of *LNCS*, pages 251–259. Springer, 2010. doi:10.1007/978-3-642-13509-5_23.

15 Maxime Crochemore, Costas S. Iliopoulos, and Hiafeng Yu. Algorithms for computing evolutionary chains in molecular and musical sequences. In Costas S. Iliopoulos, editor, *Proceedings of the 9th Australian Workshop on Combinatorial Algorithms (AWOCA 1998)*, pages 172–184, France, 1998. URL: `https://hal-upec-upem.archives-ouvertes.fr/hal-00619988/file/9807-EC.pdf`.

16 Tomás Flouri, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, William F. Smyth, and Wojciech Tyczyński. Enhanced string covering. *Theor. Comput. Sci.*, 506:102–114, 2013. `doi:10.1016/j.tcs.2013.08.013`.

17 Ondřej Guth and Bořivoj Melichar. Using finite automata approach for searching approximate seeds of strings. In Xu Huang, Sio-Iong Ao, and Oscar Castillo, editors, *Intelligent Automation and Computer Engineering*, volume 52 of *Lecture Notes in Electrical Engineering*, pages 347–360. Springer Netherlands, 2010. `doi:10.1007/978-90-481-3517-2_27`.

18 Ondřej Guth, Bořivoj Melichar, and Miroslav Balík. Searching all approximate covers and their distance using finite automata. In Peter Vojtáš, editor, *Proceedings of the Conference on Theory and Practice of Information Technologies (ITAT 2008)*, volume 414 of *CEUR Workshop Proceedings*, pages 21–26, 2009. URL: `http://ceur-ws.org/Vol-414/paper4.pdf`.

19 Costas S. Iliopoulos, Dennis W. G. Moore, and Kunsoo Park. Covering a string. *Algorithmica*, 16(3):288–297, 1996. `doi:10.1007/BF01955677`.

20 Costas S. Iliopoulos and Laurent Mouchard. Quasiperiodicity and string covering. *Theor. Comput. Sci.*, 218(1):205–216, 1999. `doi:10.1016/S0304-3975(98)00260-6`.

21 Costas S. Iliopoulos and William F. Smyth. An on-line algorithm of computing a minimum set of k-covers of a string. In Costas S. Iliopoulos, editor, *Proceedings of the 9th Australian Workshop on Combinatorial Algorithms (AWOCA 1998)*, pages 97–106, 1998.

22 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. `doi:10.1137/0206024`.

23 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Fast algorithm for partial covers in words. *Algorithmica*, 73(1):217–233, 2015. `doi:10.1007/s00453-014-9915-3`.

24 Roman M. Kolpakov and Gregory Kucherov. Finding approximate repetitions under Hamming distance. *Theor. Comput. Sci.*, 1(303):135–156, 2003. `doi:10.1016/S0304-3975(02)00448-6`.

25 Gad M. Landau and Jeanette P. Schmidt. An algorithm for approximate tandem repeats. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM 1993)*, volume 684 of *LNCS*, pages 120–133. Springer, 1993. `doi:10.1007/BFb0029801`.

26 Gad M. Landau, Jeanette P. Schmidt, and Dina Sokol. An algorithm for approximate tandem repeats. *J. Comput. Biol.*, 8(1):1–18, 2001. `doi:10.1089/106652701300099038`.

27 Yin Li and William F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002. `doi:10.1007/s00453-001-0062-2`.

28 M. Lothaire, editor. *Combinatorics on words*. Cambridge Mathematical Library. Cambridge University Press, 1997. `doi:10.1017/CBO9780511566097`.

29 Dennis Moore and William F. Smyth. An optimal algorithm to compute all the covers of a string. *Inf. Process. Lett.*, 50(5):239–246, 1994. `doi:10.1016/0020-0190(94)00045-X`.

30 Dennis Moore and William F. Smyth. A correction to "An optimal algorithm to compute all the covers of a string". *Inf. Process. Lett.*, 54(2):101–103, 1995. `doi:10.1016/0020-0190(94)00235-Q`.

31 Jeong Seop Sim, Costas S. Iliopoulos, Kunsoo Park, and William F. Smyth. Approximate periods of strings. *Theor. Comput. Sci.*, 262(1):557–568, 2001. `doi:10.1016/S0304-3975(00)00365-0`.

32 William F. Smyth. Repetitive perhaps, but certainly not boring. *Theor. Comput. Sci.*, 249(2):343–355, 2000. `doi:10.1016/S0304-3975(00)00067-0`.

33 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. Algorithms for computing the lambda-regularities in strings. *Fundam. Inform.*, 84(1):33–49, 2008. URL: `http://content.iospress.com/articles/fundamenta-informaticae/fi84-1-04`.

34 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. Varieties of regularities in weighted sequences. In Bo Chen, editor, *Proceedings of the 6th International Conference on Algorithmic Aspects in Information and Management (AAIM 2010)*, volume 6124 of *LNCS*, pages 271–280. Springer, 2010. `doi:10.1007/978-3-642-14355-7_28`.

Approximate Cover of Strings

Amihood Amir[1], Avivit Levy[2], Ronit Lubin[3], and Ely Porat[4]

1 **Bar-Ilan University, Ramat Gan, Israel; and**
Johns Hopkins University, Baltimore, MD, USA
`amir@cs.biu.ac.il`
2 **Shenkar College, Ramat Gan, Israel**
`avivitlevy@shenkar.ac.il`
3 **Bar-Ilan University, Ramat Gan, Israel**
`ronit.moldovan@gmail.com`
4 **Bar-Ilan University, Ramat Gan, Israel**
`porately@cs.biu.ac.il`

Abstract

Regularities in strings arise in various areas of science, including coding and automata theory, formal language theory, combinatorics, molecular biology and many others. A common notion to describe regularity in a string T is a *cover*, which is a string C for which every letter of T lies within some occurrence of C. The alignment of the cover repetitions in the given text is called *a tiling*. In many applications finding exact repetitions is not sufficient, due to the presence of errors. In this paper, we use a new approach for handling errors in coverable phenomena and define the *approximate cover problem (ACP)*, in which we are given a text that is a sequence of some cover repetitions with possible mismatch errors, and we seek a string that covers the text with the minimum number of errors. We first show that the ACP is $\mathcal{NP}$-hard, by studying the *cover-size relaxation of the ACP*, in which the requested size of the approximate cover is also given with the input string. We show this relaxation is already $\mathcal{NP}$-hard. We also study another two relaxations of the ACP, which we call the *partial-tiling relaxation of the ACP* and the *full-tiling relaxation of the ACP*, in which a tiling of the requested cover is also given with the input string. A given full tiling retains all the occurrences of the cover before the errors, while in a partial tiling there can be additional occurrences of the cover that are not marked by the tiling. We show that the partial-tiling relaxation has a polynomial time complexity and give experimental evidence that the full-tiling also has polynomial time complexity. The study of these relaxations, besides shedding another light on the complexity of the ACP, also involves a deep understanding of the properties of covers, yielding some key lemmas and observations that may be helpful for a future study of regularities in the presence of errors.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics, G.4 Mathematical Software, I.5.2 Design Methodology

Keywords and phrases periodicity, quasi-periodicity, cover, approximate cover

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.26

1 Introduction

Regularities in strings arise in various areas of science, including coding and automata theory, formal language theory, combinatorics, molecular biology and many others. A typical form of regularity is *periodicity*, meaning that a "long" string T can be represented as a concatenation of copies of a "short" string P, possibly ending in a prefix of P. Periodicity has been extensively studied in Computer Science over the years (see [26]).

For many phenomena, it is desirable to broaden the definition of periodicity and study wider classes of repetitive patterns in strings. One common such notion is that of a *cover*, defined as follows.

▶ **Definition 1** (Cover). A length m substring C of a string T of length n, is said to be a *cover* of T, if $n > m$ and every letter of T lies within some occurrence of C.

Note that by the definition of cover, the string C is both a prefix and a suffix of the string T. For example, consider the string $T = abaababaaba$. Clearly, T is "almost" periodic with period aba, however, as it is not completely periodic, the algorithms that exploit repetitions cannot be applied to it. On the other hand, the string $C = aba$ is a cover of T, which allows applying to T cover-based algorithms. In this paper we study coverable phenomena in the presence of errors.

There are related regularity types and several approaches to handle errors in regularities. Quasi-periodicity was introduced by Ehrenfeucht in 1990 (according to [5]). The earliest paper in which it was studied is by Apostolico, Farach and Iliopoulos [7], which defined the *quasi-period* of a string to be the length of its shortest cover and presented an algorithm for computing the quasi-period of a given string in $O(n)$ time and space. The new notion attracted immediately several groups of researchers (e.g. [8], [27, 28], [25], [9]). An overview on the first decade of the research on covers can be found in the surveys [5, 19, 30].

While covers are a significant generalization of the notion of periods as formalizing regularities in strings, they are still restrictive, in the sense that it remains unlikely that an arbitrary string has a cover shorter than the word itself. Due to this reason, different variants of quasi-periodicity have been introduced. These include *seeds* [18], *maximal quasi-periodic substring* [6], the notion of *k-covers* [20], *λ-cover* [31], *enhanced covers* [15], *partial cover* [21]. Other recently explored directions include the inverse problem for cover arrays [13], extensions to strings in which not all letters are uniquely defined, such as *indeterminate strings* [4] or *weighted sequences* [32]. Some of the related problems are $\mathcal{NP}$-hard (see e.g., [4, 10, 21]).

In applications such as molecular biology and computer-assisted music analysis, finding exact repetitions and covers is not always sufficient. A more appropriate notion is that of *approximate repetitions*, where errors are allowed (see, e.g., [12, 14]). This notion was first studied in 1993 by Landau and Schmidt [23, 24] who concentrated on approximate tandem repeats. Note that, the natural definition of an approximate repetition is not clear. One possible definition is that the distance between any two adjacent repeats is small. Another possibility is that all repeats lie at a small distance from a single "original". Such a definition of *approximate seeds* is studied in [10, 17, 16]. Indeed, all these definitions along with other ones were proposed and studied (see [1, 22, 29]). Yet another possibility is that all repeats must be equal, but we allow a fixed total number of mismatches. The possibility presented in [1] is a global one, assuming that an original unknown string is a sequence of repeats without errors, but the process of sequence creation or transmission incurs errors to the sequence of repeats, and, thus, the examined input string is not a sequence of repeats. Therefore, a (smallest) repeat generating a string with the minimum total number of mismatches with the input string is sought. Extension of this definition approach to approximate covers is the topic of this paper.

1.1 Our Results

In this paper we extend the approach of [1] to the notion of covers and define the *approximate cover problem (ACP)*, in which we are given a text that is a sequence of some cover repetitions with possible mismatch errors, and we seek a string that covers the text with the minimum number of errors. The alignment of the cover repetitions in the given text is called *a tiling*. We prove that the ACP is $\mathcal{NP}$-hard by studying a relaxation of this problem, which we call *the cover-size relaxation of the ACP*. In this relaxation the requested size of the approximate cover is also given with the input string. We prove that this relaxation is already $\mathcal{NP}$-hard, thus proving the $\mathcal{NP}$-hardness of ACP.

We also study another two relaxations of the problem, which we call *the partial-tiling relaxation of the ACP* and *the full-tiling relaxation of the ACP*. In this relaxations a tiling of the requested cover is also given, and we seek a string such that when using the given tiling to align it with the given text, the number of mismatches is minimized. The full tiling retains all the occurrences of the cover before the errors, while in the partial tiling there can be additional occurrences of the cover that are not marked by the tiling. We examine these relaxations and show the partial-tiling has polynomial time complexity and give experimental evidence that the full-tiling also has polynomial time complexity. The study of these relaxations, besides shedding another light on the complexity of the ACP, also involves a deep understanding of the properties of covers and seeds, yielding some key lemmas and observations (such as [2]) that may be helpful for a future study of regularities in the presence of errors.

Paper Contributions. The main contributions of this paper are:

- Proving that the ACP is $\mathcal{NP}$-hard.
- Formalizing the partial-tiling relaxation of the ACP and proving it is polynomial time computable.
- Formalizing the full-tiling relaxation of the ACP and suggesting a polynomial time algorithm for its computation, while giving an experimental evidence for the correctness of this algorithm.

The paper is organized as follows. In Section 2, we give formal definitions. In Section 3, we study the cover-size relaxation of the ACP and prove the $\mathcal{NP}$-hardness of the ACP. In Section 4, we study the partial-tiling relaxation of the ACP and show it is polynomial-time computable. In Section 5, we study the full-tiling relaxation of the ACP, suggest a polynomial-time algorithm for this problem and experimentally test its correctness. We conclude with some open problems in Section 6.

2 Preliminaries

In this section we give the needed formal definitions.

▶ **Definition 2** (Tiling). Let T be a string over alphabet Σ such that the string C over alphabet Σ is a cover of T. Then, the sorted list of indices representing the start positions of occurrences of the cover C in the text T is called the *tiling* of C in T.

In this paper we have a text T which may have been introduced to errors and, therefore, is not coverable. However, we would like to refer to a retained tiling of an unknown string C in T although C does not cover T because of mismatch positions. The following definition makes a distinction between a list of indices that may be assumed to be a tiling of the text before mismatch errors occurred and a list of indices that cannot be such a tiling.

▶ **Definition 3** (A Valid Tiling). Let T be an n-length string over alphabet Σ and let L be a sorted list of indices $L \subset \{1, ..., n\}$. Let $m = n + 1 - L_{last}$, where L_{last} is the last index in L. Then, L is called *a valid tiling* of T, if $i_1 = 1$ and for every $i_k, i_{k+1} \in L$, it holds that $i_{k+1} - i_k \leq m$.

▶ **Notation 1.** *Let C be an m length string over alphabet Σ. Denote by $S(C)$ a string of length n, $n > m$, such that C is a cover of $S(C)$.*

Note that $S(C)$ is not uniquely defined even for a fixed $n > m$, since every different valid tiling of the m-length string C generates a different n-length string $S(C)$. A unique version can be obtained if a valid tiling L is also given.

▶ **Notation 2.** *Let T be an n-length string over alphabet Σ and let L be a valid tiling of T. Let $m = n + 1 - L_{last}$, where L_{last} is the last index in the tiling L. For any m-length string C', let $S_L(C')$ be the n-length string obtained using C' as a cover and L as the tiling as follows: $S_L(C')$ begins with a copy of C' and for each index i in L a new copy of C' is concatenated starting from index i of $S_L(C')$ (maybe running over a suffix of the last copy of C').*

▶ **Definition 4.** Let T be a string of length n over alphabet Σ. Let H be the Hamming distance. The *distance of T from being covered* is:

$$dist = \min_{C \in \Sigma^*, |C| < n, S(C) \in \Sigma^n} H(S(C), T).$$

We will also refer to *dist* as *the number of errors in T*.

▶ **Definition 5.** Let T be an n-long string over alphabet Σ. An m-long string C over Σ, $m \in \mathbb{N}$, $m < n$, is called *an m-length approximate cover of T*, if for every string C' of length m over Σ, $\min_{S(C') \in \Sigma^n} H(S(C'), T) \geq \min_{S(C) \in \Sigma^n} H(S(C), T)$, where H is the hamming distance of the given strings.
We refer to $\min_{S(C) \in \Sigma^n} H(S(C), T)$ as the *number of errors of an m-length approximate cover of T*.

▶ **Definition 6** (Approximate Cover)**.** Let T be a string of length n over alphabet Σ. A string C over alphabet Σ is called an *approximate cover of T* if:

1. C is an m-length approximate cover of T for some $m \in \mathbb{N}$, $m < n$, for which

$$\min_{S(C) \in \Sigma^n} H(S(C), T) = dist.$$

2. for every m'-length approximate cover of T, C', s.t. $\min_{S(C') \in \Sigma^n} H(S(C'), T) = dist$, it holds that: $m' \geq m$.

Primitivity. By definition, an approximate cover C should be *primitive*, i.e., it cannot be covered by a string other than itself (otherwise, T has a cover with a smaller length). Note that a periodic string can be covered by a smaller string (not necessarily the period), and therefore, is not primitive.

▶ **Definition 7.** The *Approximate Cover Problem* (ACP) is the following:
INPUT: String T of length n over alphabet Σ.
OUTPUT: An approximate cover of T, C, and the number of errors in T.

3 $\mathcal{NP}$-Hardness of the ACP

In this section we prove the $\mathcal{NP}$-hardness of the ACP. To this end, we study a variant of the problem where m, the length of a requested approximate cover, is also given together with the input string T, and we are requested to find a string C of length m that is an

m-length approximate cover of T, i.e., C covers T with the minimum number of errors over all strings of length m. We call this problem *the cover-size relaxation of the ACP*. Clearly, if the cover-size relaxation of the ACP is already $\mathcal{NP}$-hard, then so is the ACP.

Our hardness proof uses a reduction from the 3-SAT problem, in which the input is a logical formula φ on N variables in 3-CNF (each clause has exactly three literals), and we need to decide whether φ is satisfiable or not. The $\mathcal{NP}$-hardness of 3-SAT is well-known (see e.g. [11]).

3.1 The Reduction from 3-SAT

Given a 3-CNF formula φ on N variables, $x_1, \ldots, x_N$, with ℓ clauses. Assume without loss of generality that the literals in each clause are sorted by the index of their variables. We need to define a text T of length n over an alphabet Σ and to specify the size m of the requested approximate cover. We will then show that φ is satisfiable if and only if T has an m-approximate cover with at most some specified number of errors to be defined.

We begin by defining the alphabet Σ to include all the variables and their negation together with 4 additional dummy variables: $x_0, x_{-1}, x_{N+1}, x_{N+2}$ and also a special padding character p. Formally,

$$\Sigma = \{x_i, \bar{x_i} | i \in [1..N]\} \cup \{x_{-1}, x_0, x_{N+1}, x_{N+2}, p\}.$$

The definition of the text T has two parts: a header and a body, where the body of T is defined according to the clauses of the given logical formula φ, and the header preceding this body imposes a structure on an m-approximate cover for T.

The definition of the body of T follows directly from the formula φ. For each clause $C_j = L_1^j \vee L_2^j \vee L_3^j$ of φ, $1 \le j \le \ell$, we add to the body of T the substring $L_1^j L_2^j L_3^j$, preceded and followed by a padding of $2N + 14$ occurrences of the character p. The role of this padding is to avoid overlaps between occurrences of an approximate cover covering substrings originating from different clauses. The header is composed of $\ell(N+3)$ copies of the following string: $B = p \ldots p x_{N+2} x_{N+1} \bar{x_N} \ldots \bar{x_1} x_0 x_{-1} p \ldots p x_{N+2} x_{N+1} x_N \ldots x_1 x_0 x_{-1} p \ldots p$, where each padding contains $N + 7$ occurrences of p.

We define the size of the requested approximate cover m to be $3N + 18$. Note that the size of T and m as well as their construction are polynomial in N and ℓ. Lemma 8 assures the correctness of the reduction.

▶ **Lemma 8.** *φ is satisfiable if and only if T has an m-approximate cover with at most $\ell(N + 3)(N + 1)$ errors.*

We have, therefore, proven Theorem 9.

▶ **Theorem 9.** *ACP is $\mathcal{NP}$-hard.*

4 The Partial-Tiling Relaxation of the ACP

In this section we study another relaxation of the approximate cover problem: the partial-tiling relaxation, in which we are given a retained tiling of the cover before the errors has occurred together with the input string itself. In order to formally define the relaxation we need Definitions 10 and 11.

▶ **Definition 10.** Let T be an n-length string over alphabet Σ and let L be a valid tiling of T. Let $m = n + 1 - L_{last}$, where L_{last} is the last index in the tiling L. Then, an

L-approximate cover of T is a primitive string C such that for every string C' of length m over Σ, $H(S_L(C'),T) \geq H(S_L(C),T)$, where H is the hamming distance of the given strings. $\min_{C\in\Sigma^m} H(S_L(C),T)$ is the number of errors of an L approximate cover of T.

▶ **Definition 11.** Let T be an n-length string over alphabet Σ. Let L be a valid tiling of T and let L' be a valid tiling of T such that $L \subseteq L'$. Let $m' = n + 1 - L'_{last}$, where L'_{last} is the last index in the tiling L'. Then, a *partial L-approximate cover of* T is a primitive string C of length m' such that for every string C' of length m' over Σ, $H(S_{L'}(C'),T) \geq H(S_{L'}(C),T)$, where H is the hamming distance of the given strings.
$\min_{C\in\Sigma^{m'}} H(S_{L'}(C),T)$ is the number of errors of a partial L-approximate cover of T.

▶ **Definition 12** (The Partial-Tiling Relaxation of the ACP).
INPUT: String T of length n over alphabet Σ, and a valid tiling L of T.
OUTPUT: A partial L-approximate cover C of T.

We describe an algorithm for the partial-tiling relaxation of the approximate cover problem in two parts. We first describe the mandatory part of the algorithm, which we call the Histogram Greedy Algorithm. This algorithm does the main work in finding an approximate cover subject to the tiling L. It returns a candidate for the final L approximate cover to be output. This candidate is legal if it is primitive and illegal, otherwise. We then describe the second part, which we call the Partial-Tiling Primitivity Coercion. In this part, the legality of the candidate is checked, and if needed, the candidate is corrected in order to coerce the primitivity requirement.

4.1 The Histogram Greedy Algorithm

This part of the algorithm performs the following steps given the text T and the valid tiling L:

1. Find m, the length of an approximate cover subject to the tiling L, by computing the difference between $n+1$, and the last index in the tiling L, L_{last}, which indicates the last occurrence of the cover in T.
2. Compute the m-length mask M of an approximate cover, by initializing M to zeroes, setting $M[1] = 1$, then reading the tiling L from beginning to end and for each $i_k, i_{k+1} \in L$ setting $M[i_{k+1} - i_k] = 1$.
3. Compute the m-long string V_C of variables from an auxiliary alphabet

   $$\Sigma_V = \{v_1, v_2, \ldots, v_m\}.$$

   First, we initialize the m-long string V_C to $v_1v_2\ldots v_m$. Then, we read the mask M from end to beginning, and for every j such that $M[j] = 1$, we update the string V_C by equalizing the substrings $V_C[1..m-j+1]$ and $V_C[j..m]$. In the equalization process, when we obtain an equation $v_k = v_\ell$ for $k < \ell$, we replace both letters by v_k. The resulting string V_C represents C in the following sense: for any pair of indices $1 \leq i < j \leq m$, if $V_C[i] = V_C[j]$ then $C[i] = C[j]$. However, it can be that $V_C[i] \neq V_C[j]$, while $C[i] = C[j]$. In other words, V_C carries the information on equalities imposed by the mask M between indices of C.
4. Compute the n-long string V_T of variables from the auxiliary alphabet Σ_V, which is a string covered by V_C according to the tiling L of T. V_C is computed using the tiling L and V_C as follows: it begins with a copy of V_C and for each index i in L a new copy of V_C is concatenated starting from index i of V_T (maybe running over a suffix of the last copy of V_C).

5. Compute the histogram $Hist_{V_C,\Sigma}$ using the alignment of T with V_T and counting for each variable $V \in V_C$ and each $\sigma \in \Sigma$, the number of indices i in T, V_T for which $V_T[i] = V$ and $T[i] = \sigma$.
6. Compute an L approximate cover candidate C greedily according to the histogram $Hist_{V_C,\Sigma}$, as follows: for every index $1 \leq i \leq m$, set $C[i] = \sigma_0$, where $Hist_{V_C,\Sigma}[V_C[i], \sigma_0] = \max_{\sigma\in\Sigma} Hist_{V_C,\Sigma}[V_C[i], \sigma]$, i.e., for each index in C we choose the alphabet symbol that minimizes the number of mismatch errors between $S_L(C)$ and T in the relevant indices according to the tiling L.

The algorithm outputs the m-length string C from its last step and the histogram table $Hist_{V_C,\Sigma}$.

Lemma 13 describes a property of the output C returned by the Histogram Greedy algorithm, and immediately follows from the greedy criterion used in step 6 of the algorithm. Lemma 14 describes the algorithm time complexity.

▶ **Lemma 13.** *Let C be the output of the Histogram Greedy algorithm. Then,*

$$H(T, S_L(C)) = \min_{C'\in\Sigma^m} H(T, S_L(C')).$$

▶ **Lemma 14.** *The time complexity of the Histogram Greedy algorithm is: $O(|\Sigma| \cdot m + n)$.*

Despite Lemma 13, the output C of the Histogram Greedy algorithm might not be an L approximate cover of T, because it might not be primitive, as the following example shows.

Example: Assume that $V_C = XYZWXY$ and $\Sigma = \{a, b\}$ and that the histogram $Hist_{V_C,\Sigma}$ computed by the algorithm is the following:

$V_C \diagdown \Sigma$	a	b
X	4	1
Y	2	3
Z	2	1
W	0	3

Then, the Histogram Greedy algorithm chooses: $X = a$, $Y = b$, $Z = a$, $W = b$, and outputs $C = ababab$, which cannot be considered a legal cover since it is not primitive, i.e., C itself can be covered by the shorter string ab. However, the partial L-approximate cover can have a tiling L', such that $L \subseteq L'$, which exactly is the case with ab. Therefore, ab should be returned as the partial L-approximate cover of T. The Partial-Tiling Primitivity Coercion algorithm described in Subsection 4.2 is responsible for checking the legality of the output string received from the Histogram Greedy algorithm and returning a partial L-approximate cover.

Note, that the input tiling L requires an m-length string as an output. Therefore, the (primitive) 2-length approximate cover ab is precluded as an L-approximate cover. Assuming that the input tiling L is the retained tiling of the cover of the original text before the errors occurred, such a case means that, though ab is a string covering T subject to a partial tiling L with the least number of errors, it does not cover T with L as a full tiling. In this sense, L is an evidence that the original cover is of larger length than ab and that more errors actually happened. Section 5 is devoted to finding an L-approximate cover.

4.2 The Partial-Tiling Primitivity Coercion Algorithm

This part of the algorithm gets as input the string C returned by the Histogram Greedy algorithm and performs the following steps:

1. Check the primitivity of C (using the linear-time algorithm of [7]). If C is primitive, return C.
2. Else, return the primitive cover C' of C (found using the linear-time algorithm of [7] in the first step).

The time complexity of the Partial-Tiling Primitivity Coercion algorithm is immediate from the linear-time complexity of the algorithm in [7]. Thus, we get:

▶ **Lemma 15.** *The time complexity of the Partial-Tiling Primitivity Coercion algorithm is $O(m)$.*

Theorem 16 follows.

▶ **Theorem 16.** *Given a text T of length n over alphabet Σ and a valid tiling L. Let L_{last} be the last index in L. Then, the partial-tiling relaxation of the approximate cover problem of T can be solved in $O(|\Sigma| \cdot m + n)$ time, where $m = n + 1 - L_{last}$.*

5 The Full-Tiling Relaxation of the ACP

In this section we study another relaxation of the approximate cover problem: the full-tiling relaxation, in which we are given a retained tiling of the cover before the errors have occurred together with the input string itself. Unlike the situation in the problem of the previous section, this tiling is assumed to be exact. Therefore, the algorithm cannot return as cover a string that in order to cover T must have repetitions that are not marked in the tiling L. The formal definition of the problem is as follows.

▶ **Definition 17** (The Full-Tiling Relaxation of the ACP).
INPUT: String T of length n over alphabet Σ, and a valid tiling L of T.
OUTPUT: An L-approximate cover C of T.

In order to impose the requirement of the definition of an L-approximate cover of T to be a primitive string such that all its repetitions to cover T (with minimum number of errors) are marked in the tiling L, we need a different primitivity coercion algorithm than the one described in the previous section. This algorithm is described in Subsection 5.1. Unfortunately, proving the correctness of this algorithm requires a deep understanding of the properties of coverability in the presence of mismatch errors. Although we are making progress in proving this needed background (see, for example [2]), a lack in the complete understanding of the phenomenon prevents us from proving the correctness formally. Hence, in Subsection 5.2, we resort to experimental evidence of the correctness.

5.1 The Full-Tiling Primitivity Coercion Algorithm

This part of the algorithm gets as input the string C returned by the Histogram Greedy algorithm (Subsection 4.1) and performs the following steps:

1. Check the primitivity of C (using the linear-time algorithm of [7]). If C is primitive, return C.
2. Else, find $V_k \in V_C$ such that if the assignment of V_k is changed from the symbol with the largest value in the row of V_k in $Hist_{V_C,\Sigma}$ to the symbol with the second largest value in this row, thus obtaining a new m-length candidate string C', such that the difference $H(S_L(C'), T) - H(S_L(C), T)$ is minimized and where C' is primitive.

Lemma 18 below describes the time complexity of the Full-Tiling Primitivity Coercion algorithm and immediately follows from the linear-time complexity of the algorithm [7] we use in the first step and the description of the second step.

▶ **Lemma 18.** *The time complexity of the Full-Tiling Primitivity Coercion algorithm is* $O(|\Sigma| \cdot m)$.

Remark: Note that we can use a different algorithm that instead of checking the change of single variables to the second best assignment and choosing the one that gives primitivity with the least number of errors (as our algorithm does), checks the changing to the second best assignment of all subsets of variables and chooses the set that gives primitivity with the least number of errors. This algorithm is obviously correct , i.e., assures primitivity with the least number of errors, however, it has an exponential-time complexity. On the other hand, our algorithm is assured to have polynomial-time complexity, so a proof of its correctness will assure the polynomial-time complexity of the full-tiling relaxation of the ACP.

5.2 Experimental Tests of the Full-Tiling Relaxation Algorithm

Experiment were designed to test the full-tiling relaxation algorithm, which is composed of the algorithms of Subsections 4.1 and 5.1. In particular, we also wanted to experimentally test how many times the full-tiling primitivity coercion is necessary. Note that, due to the result of [3], this algorithm is only of interest to test under a rather high error rate, in which there is an error in every occurrence of the approximate cover of the text, otherwise, the dynamic programming algorithm solving the candidate-relaxation of the ACP is applicable, where trying every substring of T as a candidate cover [3]. In order to comprehensively test the algorithm, the inputs for the tests were classified according to the following criteria:

cover size: A cover C of size m is constructed, where m is small (less than 10), medium (10-100) or large (100-400). Covers of size more than 400 were not created due to space limitations.

alphabet size: The alphabet size was chosen to be either small (at most $\sqrt{m}$) or large (more than $\sqrt{m}$).

tiling style: Given a cover C and its mask M, a tiling L for the text $S_L(C)$ is constructed where the decision of the next index in L is made according to the following styles: random – an equal priority is given to every set bit in M, left priority – a decreasing priority is given to the set bits in M, right priority – an increasing priority is given to the set bits in M.

error rate: The input string T is constructed from $S_L(C)$ by inserting mismatch errors according the following error rates: medium (in every m characters at least one error), high (in every m characters at least $\sqrt{m}$ errors).

error style: The mismatching character is determined according to the following style: random (replacing by a uniformly at random choice of another character from the alphabet) or priority (replacing by another character with priority to the first character in the alphabet, and if the first character is to be replaced then by a uniformly at random chosen different character).

These criteria guarantee that the inputs created for testing the algorithm all have a coverable original string, that its valid tiling is retained. This original string is then introduced with a sufficiently high error rate to produce the current string together with the valid tiling as inputs for the tiling relaxation algorithm. Therefore, all the tested inputs have an L approximate cover and our tiling relaxation algorithm is indeed applicable for them. Moreover, the above criteria for input generation also aim at neutralizing the effect of the cover size, the alphabet size, the tiling style, the error rate or the error style on the validity of the hypothesis, by exhaustively using all reasonable alternatives.

A total of 372000 texts T were constructed as described above and served as inputs (together with the tiling L) to the full-tiling relaxation algorithm. The results are given in Tables 1 and 2 (see Appendix). The column “Percent of Inputs” describes how many of the input texts had each row’s characteristics. Numbers are rounded to two digits after decimal point. The column “Identical” describes in how many of the input texts the Histogram Greedy algorithm of Subsection 4.1 returned the original cover C of the text $S_L(C)$ built prior to the error insertion process. The column “Primitive” describes in how many of the input texts the Histogram Greedy algorithm of Subsection 4.1 returned a primitive cover and there was no need to proceed with the second phase of the Full-Tiling Primitivity Coercion algorithm of Subsection 5.1. The column “Non-Primitive” describes in how many of the input texts the Histogram Greedy algorithm of Subsection 4.1 returned a non-primitive string and, therefore, the second phase of the Full-Tiling Primitivity Coercion algorithm of Subsection 5.1 was performed. This latter case happened in 8912 texts, which are about 2% of the texts.

Experiments Conclusion: Primitivity coercion was necessary in 2% of the total tested inputs. In a 100% of the tests the returned string after the Full-Tiling Primitivity Coercion algorithm was indeed an L-approximate cover of the input string.

6 Open Problems

In this paper we initiated the study of the approximate cover problem using a new approach. We proved that the some relaxations (the cover size relaxation) of the approximate cover problem are $\mathcal{NP}$-hard, thus proving that the ACP is $\mathcal{NP}$-hard, while other relaxations (the partial-tiling relaxation and the full-tiling relaxation) are polynomial-time computable. Some interesting questions and open problems are:

- Our $\mathcal{NP}$-hardness proof uses unbounded-size alphabet. Is the ACP still $\mathcal{NP}$-hard for finite alphabet?
- It is interesting to define other relaxations of the ACP and to study their complexity in order to have a deeper understanding of the ACP.
- In this paper we only experimentally checked the correctness of our full-tiling relaxation algorithm. We would like to have a formal proof of its correctness.
- In this paper we considered the Hamming distance as a metric in the definition of approximate cover. Other string metrics can be considered as well. It is interesting to see if and how the complexity of the problem changes with the use of other string metrics.

References

1 Amihood Amir, Estrella Eisenberg, and Avivit Levy. Approximate periodicity. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Proceedings of the 21st International Symposium on Algorithms and Computation (ISAAC 2010)*, volume 6506 of *LNCS*, pages 25–36. Springer, 2010. doi:10.1007/978-3-642-17517-6_5.

2 Amihood Amir, Costas S. Iliopoulos, and Jakub Radoszewski. Two strings at Hamming distance 1 cannot be both quasiperiodic, 2017. arXiv:1703.00195.

3 Amihood Amir, Avivit Levy, Moshe Lewenstein, Ronit Lubin, and Benny Porat. Can we recover the cover? In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter,

editors, *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *LIPIcs*, pages 25:1–25:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.25.

4 Pavlos Antoniou, Maxime Crochemore, Costas S. Iliopoulos, Inuka Jayasekera, and Gad M. Landau. Conservative string covering of indeterminate strings. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference (PSC 2008)*, pages 108–115. Czech Technical University in Prague, 2008. URL: http://www.stringology.org/event/2008/p10.html.

5 Alberto Apostolico and Dany Breslauer. Of periods, quasiperiods, repetitions and covers. In Jan Mycielski, Grzegorz Rozenberg, and Arto Salomaa, editors, *Structures in Logic and Computer Science: A Selection of Essays in Honor of Andrzej Ehrenfeucht*, volume 1261 of *LNCS*, pages 236–248. Springer, 1997. doi:10.1007/3-540-63246-8_14.

6 Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theor. Comput. Sci.*, 119(2):247–265, 1993. doi:10.1016/0304-3975(93)90159-Q.

7 Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Inf. Process. Lett.*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.

8 Dany Breslauer. An on-line string superprimitivity test. *Inf. Process. Lett.*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.

9 Dany Breslauer. Testing string superprimitivity in parallel. *Inf. Process. Lett.*, 49(5):235–241, 1994. doi:10.1016/0020-0190(94)90060-4.

10 Manolis Christodoulakis, Costas S. Iliopoulos, Kunsoo Park, and Jeong Seop Sim. Approximate seeds of strings. *J. Autom. Lang. Comb.*, 10(5/6):609–626, 2005.

11 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*, chapter NP-Completeness, pages 966–1021. The MIT Press, 2001.

12 Tim Crawford, Costas S. Iliopoulos, and Rajeev Raman. String-matching techniques for musical similarity and melodic recognition. In Walter B. Hewlett and Eleanor S. Field, editors, *Melodic Similarity: Concepts, Procedures, and Applications*, volume 11 of *Computing in Musicology*, pages 73–100. MIT Press, Cambridge, Massachusetts, 1998.

13 Maxime Crochemore, Costas S. Iliopoulos, Solon P. Pissis, and German Tischler. Cover array string reconstruction. In Amihood Amir and Laxmi Parida, editors, *Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM 2010)*, volume 6129 of *LNCS*, pages 251–259. Springer, 2010. doi:10.1007/978-3-642-13509-5_23.

14 Maxime Crochemore, Costas S. Iliopoulos, and Hiafeng Yu. Algorithms for computing evolutionary chains in molecular and musical sequences. In Costas S. Iliopoulos, editor, *Proceedings of the 9th Australian Workshop on Combinatorial Algorithms (AWOCA 1998)*, pages 172–184, France, 1998. URL: https://hal-upec-upem.archives-ouvertes.fr/hal-00619988/.

15 Tomás Flouri, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, William F. Smyth, and Wojciech Tyczyński. Enhanced string covering. *Theor. Comput. Sci.*, 506:102–114, 2013. doi:10.1016/j.tcs.2013.08.013.

16 Ondřej Guth and Bořivoj Melichar. Using finite automata approach for searching approximate seeds of strings. In Xu Huang, Sio-Iong Ao, and Oscar Castillo, editors, *Intelligent Automation and Computer Engineering*, volume 52 of *Lecture Notes in Electrical Engineering*, pages 347–360. Springer Netherlands, 2010. doi:10.1007/978-90-481-3517-2_27.

17 Ondřej Guth, Bořivoj Melichar, and Miroslav Balík. Searching all approximate covers and their distance using finite automata. In Peter Vojtáš, editor, *Proceedings of the Conference on Theory and Practice of Information Technologies (ITAT 2008)*, volume 414 of *CEUR*

Workshop Proceedings, pages 21–26, 2009. URL: http://ceur-ws.org/Vol-414/paper4.pdf.

18 Costas S. Iliopoulos, Dennis W. G. Moore, and Kunsoo Park. Covering a string. *Algorithmica*, 16(3):288–297, 1996. doi:10.1007/BF01955677.

19 Costas S. Iliopoulos and Laurent Mouchard. Quasiperiodicity and string covering. *Theor. Comput. Sci.*, 218(1):205–216, 1999. doi:10.1016/S0304-3975(98)00260-6.

20 Costas S. Iliopoulos and William F. Smyth. An on-line algorithm of computing a minimum set of k-covers of a string. In Costas S. Iliopoulos, editor, *Proceedings of the 9th Australian Workshop on Combinatorial Algorithms (AWOCA 1998)*, 1998.

21 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Fast algorithm for partial covers in words. *Algorithmica*, 73(1):217–233, 2015. doi:10.1007/s00453-014-9915-3.

22 Roman M. Kolpakov and Gregory Kucherov. Finding approximate repetitions under Hamming distance. *Theor. Comput. Sci.*, 1(303):135–156, 2003. doi:10.1016/S0304-3975(02)00448-6.

23 Gad M. Landau and Jeanette P. Schmidt. An algorithm for approximate tandem repeats. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM 1993)*, volume 684 of *LNCS*, pages 120–133. Springer, 1993. doi:10.1007/BFb0029801.

24 Gad M. Landau, Jeanette P. Schmidt, and Dina Sokol. An algorithm for approximate tandem repeats. *J. Comput. Biol.*, 8(1):1–18, 2001. doi:10.1089/106652701300099038.

25 Yin Li and William F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002. doi:10.1007/s00453-001-0062-2.

26 M. Lothaire, editor. *Combinatorics on words*. Cambridge Mathematical Library. Cambridge University Press, 1997. doi:10.1017/CBO9780511566097.

27 Dennis Moore and William F. Smyth. An optimal algorithm to compute all the covers of a string. *Inf. Process. Lett.*, 50(5):239–246, 1994. doi:10.1016/0020-0190(94)00045-X.

28 Dennis Moore and William F. Smyth. A correction to "An optimal algorithm to compute all the covers of a string". *Inf. Process. Lett.*, 54(2):101–103, 1995. doi:10.1016/0020-0190(94)00235-Q.

29 Jeong Seop Sim, Costas S. Iliopoulos, Kunsoo Park, and William F. Smyth. Approximate periods of strings. *Theor. Comput. Sci.*, 262(1):557–568, 2001. doi:10.1016/S0304-3975(00)00365-0.

30 William F. Smyth. Repetitive perhaps, but certainly not boring. *Theor. Comput. Sci.*, 249(2):343–355, 2000. doi:10.1016/S0304-3975(00)00067-0.

31 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. Algorithms for computing the lambda-regularities in strings. *Fundam. Inform.*, 84(1):33–49, 2008. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi84-1-04.

32 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. Varieties of regularities in weighted sequences. In Bo Chen, editor, *Proceedings of the 6th International Conference on Algorithmic Aspects in Information and Management (AAIM 2010)*, volume 6124 of *LNCS*, pages 271–280. Springer, 2010. doi:10.1007/978-3-642-14355-7_28.

Table 1 Experimental Tests of the Full-Tiling Relaxation Algorithm for Small Alphabets.

Cover Size	Tiling Style	Error Rate	Error Style	Percent of Inputs	Identical	Primitive	Non-Primitive
small	left	medium	random	1.57	80.89	13.04	6.07
small	left	medium	priority	1.57	80.55	31.51	5.95
small	random	medium	random	1.57	78.80	15.28	5.93
small	random	medium	priority	1.57	78.30	15.46	6.24
small	right	medium	random	1.57	76.20	17.47	6.32
small	right	medium	priority	1.57	76.13	17.82	6.05
small	left	high	random	1.57	5.55	77.78	16.67
small	left	high	priority	1.57	1.87	81.39	16.74
small	random	high	random	1.57	5.24	78.57	16.19
small	random	high	priority	1.57	1.68	82.18	16.13
small	right	high	random	1.57	4.74	80.41	14.85
small	right	high	priority	1.57	1.27	84.28	14.45
medium	left	medium	random	3.22	100	0	0
medium	left	medium	priority	3.22	100	0	0
medium	random	medium	random	3.22	100	0	0
medium	random	medium	priority	3.22	100	0	0
medium	right	medium	random	3.22	100	0	0
medium	right	medium	priority	3.22	100	0	0
medium	left	high	random	3.22	89.80	10.17	0.03
medium	left	high	priority	3.22	87.87	12.09	0.03
medium	random	high	random	3.22	89.39	10.59	0.02
medium	random	high	priority	3.22	87.42	12.54	0.05
medium	right	high	random	3.22	89.07	10.90	0.33
medium	right	high	priority	3.22	86.63	13.35	0.03
large	left	medium	random	0.81	100	0	0
large	left	medium	priority	0.81	100	0	0
large	random	medium	random	0.81	100	0	0
large	random	medium	priority	0.81	100	0	0
large	right	medium	random	0.81	100	0	0
large	right	medium	priority	0.81	100	0	0
large	left	high	random	0.81	100	0	0
large	left	high	priority	0.81	100	0	0
large	random	high	random	0.81	100	0	0
large	random	high	priority	0.81	100	0	0
large	right	high	random	0.81	100	0	0
large	right	high	priority	0.81	100	0	0

Table 2 Experimental Tests of the Full-Tiling Relaxation Algorithm for Large Alphabets.

Cover Size	Tiling Style	Error Rate	Error Style	Percent of Inputs	Identical	Primitive	Non-Primitive
small	left	medium	random	0.59	89.45	9.59	0.96
small	left	medium	priority	0.59	76.51	21.93	1.56
small	random	medium	random	0.59	87.57	11.24	1.19
small	random	medium	priority	0.59	76.15	22.39	1.47
small	right	medium	random	0.59	86.56	12.89	0.55
small	right	medium	priority	0.59	75.51	23.40	1.10
small	left	high	random	0.59	25.55	70.78	3.67
small	left	high	priority	0.59	1.84	84.45	13.72
small	random	high	random	0.59	24.82	70.96	4.22
small	random	high	priority	0.59	2.06	86.15	11.79
small	right	high	random	0.59	23.76	72.75	3.49
small	right	high	priority	0.59	1.88	85.32	12.80
medium	left	medium	random	1.62	100	0	0
medium	left	medium	priority	1.62	100	0	0
medium	random	medium	random	1.62	100	0	0
medium	random	medium	priority	1.62	100	0	0
medium	right	medium	random	1.62	100	0	0
medium	right	medium	priority	1.62	100	0	0
medium	left	high	random	1.62	99.77	0.23	0
medium	left	high	priority	1.62	85.11	14.89	0
medium	random	high	random	1.62	99.75	0.25	0
medium	random	high	priority	1.62	84.19	15.81	0
medium	right	high	random	1.62	99.90	0.10	0
medium	right	high	priority	1.62	84.11	15.90	0
large	left	medium	random	0.54	100	0	0
large	left	medium	priority	0.54	100	0	0
large	random	medium	random	0.54	100	0	0
large	random	medium	priority	0.54	100	0	0
large	right	medium	random	0.54	100	0	0
large	right	medium	priority	0.54	100	0	0
large	left	high	random	0.54	100	0	0
large	left	high	priority	0.54	100	0	0
large	random	high	random	0.54	100	0	0
large	random	high	priority	0.54	100	0	0
large	right	high	random	0.54	100	0	0
large	right	high	priority	0.54	100	0	0

Beyond Adjacency Maximization: Scaffold Filling for New String Distances[*]

Laurent Bulteau[1], **Guillaume Fertin**[2], **and Christian Komusiewicz**[3]

1 **LIGM UMR 8049, Université Paris-Est, Marne-la-Vallée, France**
`laurent.bulteau@u-pem.fr`
2 **LS2N UMR CNRS 6004, Université de Nantes, Nantes, France**
`guillaume.fertin@univ-nantes.fr`
3 **Friedrich-Schiller-Universität Jena, Jena, Germany**
`christian.komusiewicz@uni-jena.de`

Abstract

In Genomic Scaffold Filling, one aims at polishing *in silico* a draft genome, called scaffold. The scaffold is given in the form of an ordered set of gene sequences, called contigs. This is done by confronting the scaffold to an already complete reference genome from a close species. More precisely, given a scaffold $\mathcal{S}$, a reference genome G and a score function $f()$ between two genomes, the aim is to complete $\mathcal{S}$ by adding the missing genes from G so that the obtained complete genome $\mathcal{S}^*$ optimizes $f(\mathcal{S}^*, G)$. In this paper, we extend a model of Jiang et al. [CPM 2016] (i) by allowing the insertions of strings instead of single characters (i.e., some groups of genes may be forced to be inserted together) and (ii) by considering two alternative score functions: the first generalizes the notion of common adjacencies by maximizing the number of common k-mers between $\mathcal{S}^*$ and G (k-Mer Scaffold Filling), the second aims at minimizing the number of breakpoints between $\mathcal{S}^*$ and G (Min-Breakpoint Scaffold Filling). We study these problems from the parameterized complexity point of view, providing fixed-parameter (FPT) algorithms for both problems. In particular, we show that k-Mer Scaffold Filling is FPT wrt. parameter ℓ, the number of additional k-mers realized by the completion of $\mathcal{S}$—this answers an open question of Jiang et al. [CPM 2016]. We also show that Min-Breakpoint Scaffold Filling is FPT wrt. a parameter combining the number of missing genes, the number of gene repetitions and the target distance.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics

Keywords and phrases computational biology, strings, FPT algorithms, kernelization

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.27

1 Introduction

The recent development and continuous improvement of NGS technologies has increased our ability to produce, rapidly and inexpensively, a first draft of any genome. However, the cost of polishing such drafts to obtain a complete genome has not decreased at the same rate, thus many species are left with a genome in its *scaffold* form: a scaffold may be represented as a sequence of *contigs* (each being a contiguous sequence of genes), separated by unknown

[*] Christian Komusiewicz gratefully acknowledges support from the DFG, project MAGZ (KO 3669/4-1). A visit from Laurent Bulteau and Christian Komusiewicz which initiated this study was funded by PEPS INS2I from CNRS and Internationalisation des Laboratoires – Action 1 from Université de Nantes.

gaps, sometimes with an indication on the length of the gap. It is thus natural to ask for methods that reconstruct the complete original genome starting from its scaffold form. This is usually done with the help of a reference genome G, that is, the complete genome of a close-enough species, in the following way: turn the scaffold $\mathcal{S}$ into a complete genome $\mathcal{S}^*$ by adding, in between the contigs of $\mathcal{S}$, genes that are present in G but not in $\mathcal{S}$, in such a way that some predefined score function between $\mathcal{S}^*$ and G is optimized. The score function is usually defined so as to follow the parsimony principle: when $\mathcal{S}^*$ and G are as close as possible, the score is optimized.

Formally, a genome G is a string built on some alphabet Σ (each character in the alphabet representing a gene or gene family), and a scaffold $\mathcal{S}$ is defined as sequence $(C_1, \ldots, C_m)$ of contigs, where each C_i, $1 \leq i \leq m$, is itself a string over Σ. For a string S of length n, we let $c(S)$ be the multiset of characters it contains, and $a(S) := \{S[i, i+1] \mid i \in [n-1]\}$ be the *multiset of adjacencies* in S. By extension, if $\mathcal{S}$ is a scaffold, $c(\mathcal{S})$ (resp. $a(\mathcal{S})$) denotes the multiset of characters (resp. adjacencies) contained in the contigs of $\mathcal{S}$. For two strings S and T, let $a(S,T) := a(S) \cap a(T)$ denote the *multiset of common adjacencies* in S and T. For a scaffold $\mathcal{S}$ and a multiset $\mathcal{T}$ of strings, we use $\mathcal{S} + \mathcal{T}$ to denote the set of strings that can be obtained from $\mathcal{S}$ by inserting the strings of $\mathcal{T}$ in between the contigs of $\mathcal{S}$. The One-Sided-Scaffold-Filling problem, introduced in [13], was the first serious attempt at modeling scaffolds as a sequence of contigs with repeats.

One-Sided-Scaffold-Filling
Input: A complete genome G and a scaffold $\mathcal{S}=(C_1, \ldots, C_m)$ over alphabet Σ, and a multiset $\mathcal{T} = c(G) - c(\mathcal{S})$ of characters.
Task: Find $\mathcal{S}^* \in \mathcal{S} + \mathcal{T}$ s.t. $|a(\mathcal{S}^*, G)|$ is maximized.

Note that Jiang et al. [13] also considered the variant in which only a subset $\mathcal{T}' \subseteq \mathcal{T}$ of the letters of $\mathcal{T}$ needs to be inserted. In this paper, we study two alternative problems.

k-Mer Scaffold Filling. The first one generalizes both of the problems considered by Jiang et al. [13] in several ways. First, we do not constrain the multiset of letters to insert to be $c(G) - c(\mathcal{S})$. Instead, the set $\mathcal{T}$ could contain letters in higher or lower multiplicity than in $c(G) - c(\mathcal{S})$. This is helpful if it is known, for example, that some genes occur in higher multiplicity in the desired genome than in G. Second, we allow that $\mathcal{T}$ contains strings instead of only letters. This allows to incorporate knowledge about the gene order that is not present in the tuple of scaffold contigs. For example, one may now deal with contigs whose position relative to the other contigs is not known. Third, we allow that the number of strings to insert can be prespecified as an input constraint. More precisely, in our variant the input contains two numbers t_1, t_2 and we search for a solution that inserts at least t_1 and at most t_2 strings from $\mathcal{T}$. This way, one can guarantee for example that the size of the resulting genome lies within some predetermined range. If we want all of $\mathcal{T}$ to be inserted in $\mathcal{S}$, it suffices to set $t_1 := |\mathcal{T}| =: t_2$. The second variant of Jiang et al. [13] in which an arbitrary subset of $\mathcal{T}$ may be inserted is obtained by setting $t_1 := 1$ and $t_2 := |\mathcal{T}|$.

Finally, as similarity measurement we do not restrict ourselves to maximizing the number of common adjacencies. Instead, we maximize, for a predetermined parameter k, the number of common k-mers (the term *k-mer* is usually used for DNA strings; we thus extend its use here in the context of gene sequences): indeed, as illustrated in Figure 1, a higher value of k tends to increase the accuracy of the result.

For a string S of length n and a positive integer k, let $a_k(S) := \{S[i, i+k] \mid i \in [n-k]\}$ denote the *multiset of k-mers* in S. For two strings S and T, $a_k(S,T) := a_k(S) \cap a_k(T)$

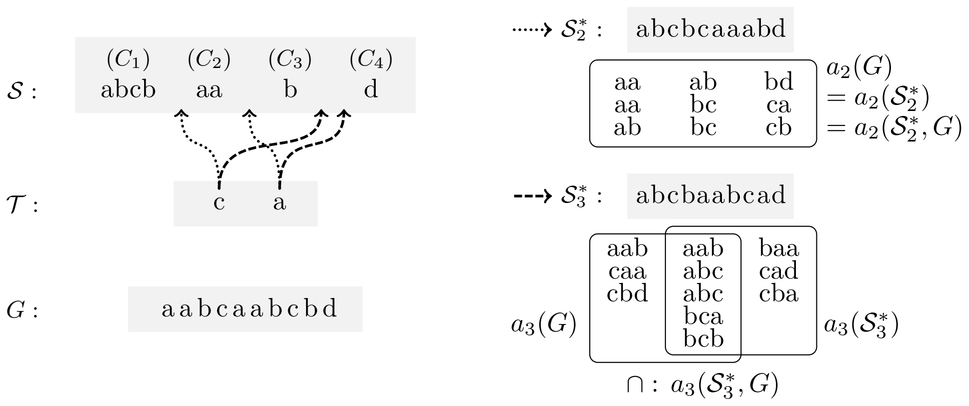


Figure 1 Left: an example instance of k-Mer-SF, for $k = 2$ and $k = 3$, with scaffold $\mathcal{S}$ containing 4 contigs, $\mathcal{T}$ containing 2 length-1 strings to be inserted, $t_1 = t_2 = 2$, and a reference genome G. An optimal solution for $k = 2$ (resp. $k = 3$) inserts the strings from $\mathcal{T}$ as indicated by dotted (resp. dashed) arcs to create $\mathcal{S}_2^*$ (resp. $\mathcal{S}_3^*$). Top-right: the 2-mers of $\mathcal{S}_2^*$ and G: note that the maximum number of common 2-mers is reached, although the strings $\mathcal{S}_2^*$ and G are quite different. Bottom-right: the 3-mers of $\mathcal{S}_3^*$ and G (there are 5 common 3-mers). Note that neither solution is optimal for both values of k, since $|a_2(\mathcal{S}_3^*, G)| = 7 < 9$, and $|a_3(\mathcal{S}_2^*, G)| = 4 < 5$. In this example, $\mathcal{S}_3^*$ should be more relevant than $\mathcal{S}_2^*$, since the former can be obtained from G with a single transposition event (by swapping factors aabca and abcb).

denotes the *multiset of common k-mers* in S and T. Note that counting common adjacencies is the special case $k = 2$. The problem we are interested in is thus defined as follows (see Figure 1 for an illustration with $k = 2$ and $k = 3$).

k-Mer Scaffold Filling (k-Mer-SF)
Input: A complete genome G and a scaffold $\mathcal{S}$ of contigs $(C_1, \ldots, C_m)$ over alphabet Σ, a multiset $\mathcal{T}$ of strings over Σ and integers, t_1, t_2 s.t. $t_1 \leq t_2 \leq |\mathcal{T}|$.
Task: Find $\mathcal{T}' \subseteq \mathcal{T}$, $t_1 \leq |\mathcal{T}'| \leq t_2$, and $\mathcal{S}^* \in \mathcal{S} + \mathcal{T}'$ s.t. $|a_k(\mathcal{S}^*, G)|$ is maximized.

Min-Breakpoint Scaffold Filling. The second problem we study here differs from One-Sided-Scaffold-Filling in two ways: first, just as k-Mer-SF, we allow that $\mathcal{T}$ contains strings instead of only letters. Second, instead of maximizing the number of common adjacencies, we aim at minimizing the number of breakpoints. We need additional definitions: given two strings S and T such that $c(S) = c(T)$, thus of same length n, and a bijection $\pi : [n] \to [n]$ such that $S[i] = T[\pi(i)]$, the *breakpoint distance wrt.* π between S and T is $|\{i \mid \pi(i+1) \neq \pi(i), 1 \leq i < n\}|$. The *string breakpoint distance* between S and T, denoted $b(S, T)$, is the minimum over all bijections π of the breakpoint distance of S and T wrt. π. We are now ready to define our second problem, which is also illustrated in Figure 2.

Min-Breakpoint Scaffold Filling (Min-Bkpt-SF)
Input: A complete genome G and a scaffold $\mathcal{S}$ over alphabet Σ and a multiset $\mathcal{T}$ of strings over Σ.
Task: Find $\mathcal{S}^* \in \mathcal{S} + \mathcal{T}$ s.t. $b(\mathcal{S}^*, G)$ is minimized.

Note that there exists attempts at defining breakpoints in strings in the context of scaffold filling [15, 16]. In that case, just as in permutations, breakpoints are defined as the dual of common adjacencies. This differs from the present definition: in our case, there exist strings

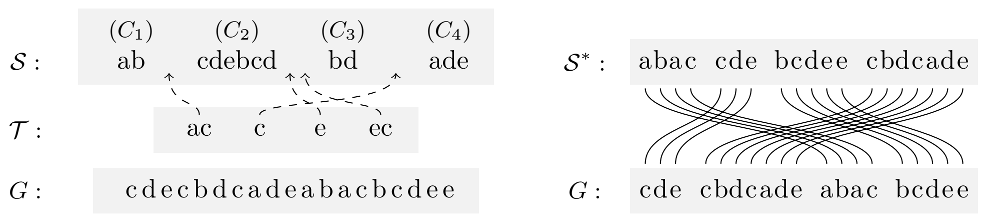


Figure 2 Left: an example instance of MIN-BKPT-SF, with scaffold $\mathcal{S}$ containing 4 contigs, $\mathcal{T}$ containing 4 strings of length 1 and 2 to be inserted, and a reference genome G. An optimal solution inserts the strings from $\mathcal{T}$ as indicated by dashed arcs to create $\mathcal{S}^*$. Right: the resulting string $\mathcal{S}^*$ is at breakpoint distance 3 from G: the letters of $\mathcal{S}$ and G can be matched together to form 4 common blocks (hence $4 - 1 = 3$ breakpoints).

S and T of length n such that $a(S,T) + b(S,T) = n + k$ with $k = \Omega(n)$. Moreover, our definition of breakpoints is of particular interest when S and T are very close: for instance, the number of common adjacencies cannot discriminate the case $S = T$ from $S \neq T$ (take e.g. $S =$aabbab and $T =$abaabb), whereas our definition of breakpoints does, and even allows to estimate how S differs from T.

Finally, note that following the motivation in genomic scaffolding, we demand that no insertions are made before C_1 or after C_m. This variant is more general than the one where insertions are allowed everywhere since we can simply add two additional contigs which have letters not occurring in G, one in the beginning and one in the end. Then, there is always an optimal solution that does not insert before the first or after the last contig.

k-MER-SF and MIN-BKPT-SF are studied here from the parameterized (or multivariate) complexity point of view [5, 8, 10]. The main parameters that will be used in the following are: k, the length of the k-mers; $\ell := a_k(\mathcal{S}^*, G) - a_k(\mathcal{S}, G)$, the number of additional common k-mers brought by completion; d, the duplication number, that is, the maximum number of times a letter appears in G; m, the number of contigs in $\mathcal{S}$; t_2, the upper bound on number of letters to insert; λ, an upper bound on the length of the strings in $\mathcal{T}$; $b = b(\mathcal{S}^*, G)$, the sought breakpoint distance.

Related work and our contribution. Genomic Scaffold Filling (GSF) has been introduced by Muñoz et al. [20] in 2010. The problem has initially been defined for permutations, i.e. genomes are modeled as duplication-free sequences. Under this setting, GSF is polynomial-time solvable for the DCJ distance [20] and for maximizing the number of adjacencies [16, 19] (or, equivalently, minimizing the breakpoint distance). This method has been validated through simulations and the comparison of two plant genomes [20].

When scaffolds are modeled as strings (thus allowing gene repetitions), it becomes harder to compute relevant parsimony measures, hence almost all works are concerned with maximizing the number of common adjacencies. In many cases with this model, the "contig" constraint has been lifted, so that a scaffold has been modeled as a simple string, and insertion can be done between any pair of consecutive letters. For GSF, Jiang et al. [17, 16] showed the problem to be NP-hard, and from then on several approximation algorithms have been given – the best to date achieves a ratio of 1.2 [14]. Bulteau et al. [3] showed that GSF is FPT in the sought number of adjacencies ℓ.

More recently, as in this paper, scaffolds were considered to be a sequence $(C_1, \ldots, C_m)$ of contigs. Jiang et al. [13] considered GSF under this model, again with the maximum adjacency measure. They proved the problem to be NP-hard even if only two contigs are given, gave a 2-approximation for the problem, and showed the problem to be FPT wrt. the combined parameter ℓ, the number of sought common adjacencies, and d, the duplication number. A short survey of the most recent results concerning GSF under the maximum adjacency setting can be found in [21]. In particular, the following question was raised [13, 21]: what is the FPT status of the problem when the parameter is the number of adjacencies?

In this paper, we study k-Mer-SF and Min-Bkpt-SF from a parameterized complexity point of view. In particular, we show that k-Mer-SF is W[1]-hard wrt. parameter t_2, and FPT wrt. parameter ℓ, thereby answering the above open question positively. We also provide a polynomial kernel for the parameter $\ell + m$ for the case where $t_2 = |\mathcal{T}|$, $\lambda = 1$ and $k = 2$, which corresponds to earlier definitions of the GSF problem. Concerning Min-Bkpt-SF, we provide hardness results even in some restricted cases (e.g. when $b = 0$ and $m = 2$), and we provide several FPT results wrt. combinations of some of the input parameters.

Preliminaries. For a string S, we use $S[i]$ to denote the letter at position i and $S[i, j]$ to denote the substring starting at position i and ending at position j; if $i > j$, then $S[i, j]$ is defined as the empty string. We use $S_k[i] := S[i, i + k - 1]$ to denote the length-k substring of S starting at position i. For two strings S and T we denote the concatenation of S and T by $S \circ T$. For a multiset or tuple of strings S, we use $||S||$ to denote the sum of the lengths of the strings contained in S. We use $[n] := \{1, \ldots, n\}$ to denote the numbers from 1 through n. For a multiset X over a universe U and an element u of U, let $m(X, u)$ denote the multiplicity of u in X. If $m(X, u) \geq 1$, then we write $x \in X$, if $m(X, u) = 0$, then we write $u \notin X$. We extend the definition of functions in a natural way to work with multiset domains. That is, a function $f : X \to Y$ with X a multiset is defined as a function $f_S : S_X \to Y$ where $S_X := \{(x, i) : x \in X, i \in [m(X, x)]\}$ is a set containing, for each $x \in X$, $m(X, x)$ many different elements corresponding to x. We write $f(x) := \{f_S((x, i)) : i \in [m(X, x)]\}$ to denote the set of images of x. Throughout the paper, $n := |G|$ will denote the length of the input genome G both in k-Mer-SF and Min-Bkpt-SF.

For the relevant definitions of parameterized complexity theory, refer to [8, 10].

2 The Relation between k-Mer Scaffold Filling and Partial Set Cover

In the following, we describe a reduction from the following variant of Set Cover to k-Mer-SF.

Partial Set Cover
Input: A family $\mathcal{F} = \{F_1, \ldots, F_m\}$ of subsets of a universe $U = \{u_1, \ldots, u_n\}$ and integers κ and τ.
Task: Find a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ of size at most κ such that $|\bigcup_{F_i \in \mathcal{F}'} F_i| \geq \tau$.

On the positive side, Partial Set Cover can be solved in $(2e)^\tau \cdot |U| \cdot |F|$ time [2]. For the parameter κ, however, Partial Set Cover is W[1]-hard [12].

▶ **Lemma 1.** *For each $k \geq 2$, there is a polynomial-time reduction from* Partial Set Cover *to* k-Mer-SF *such that $t_2 = \kappa$ and $|G| = \mathcal{O}(n)$.*

Proof. Given an instance of Partial Set Cover, construct an instance of k-Mer-SF for $k = 2$ as follows. For each u_i, introduce two letters a_i and b_i and introduce a further letter x. Now let G be the string $a_1b_1xa_2b_2x \cdots xa_nb_n$. Observe that each element in U corresponds to exactly one adjacency in G.

Now, for each $F_i \in \mathcal{F}$, construct a string T_i and add it to $\mathcal{T}$. The string T_i contains the substring $a_j b_j$ for each $u_j \in F_i$. More precisely, if $F_i = \{u_i^1, \ldots, u_i^q\}$, then $T_i = a_i^1 b_i^1 \cdots a_i^q b_i^q$. Now add two contigs $C_1 = C_2 = y$ to the scaffold $\mathcal{S}$ and set $t_1 = 0$ and $t_2 = \kappa$. This concludes the construction of the k-MER-SF instance. It remains to show the correctness of the reduction, that is,

$(\mathcal{F}, \kappa, \tau)$ is a yes-instance of PARTIAL SET COVER $\Leftrightarrow$ there is a solution $\mathcal{S}^*$ of $(G, \mathcal{S}, \mathcal{T}, 0, \kappa)$ for k-MER-SF such that $|a_k(\mathcal{S}^*, G)| \geq \tau$.

($\Rightarrow$) Let $\mathcal{F}'$ be a solution of PARTIAL SET COVER. Then, for each $F_i \in \mathcal{F}'$, insert the string T_i in $\mathcal{S}$ (in arbitrary order) and denote the resulting string by $\mathcal{S}^*$. Since the scaffold $\mathcal{S}$ contains no letters from G, all adjacencies of G are missing in $\mathcal{S}$. Let U' denote the elements that are covered by $\mathcal{F}'$. For each $u_j \in U'$, there is some F_i containing u_j and thus some T_i containing the adjacency $a_j b_j$ which is an adjacency of G. Thus, $\mathcal{S}^*$ contains at least $|U'|$ many adjacencies.

($\Leftarrow$) Let $\mathcal{S}^*$ be a solution such that $|a_k(\mathcal{S}^*, G)| \geq \tau$. Observe that every common adjacency of $\mathcal{S}^*$ and G is of the form $a_j b_j$, since all other adjacencies of G contain the letter x. Since each adjacency is of the form $a_j b_j$, there are thus at least τ distinct indices j such that $\mathcal{S}^*$ contains the adjacency $a_j b_j$. Moreover, every such adjacency is contained in some $T_i \in \mathcal{T}'$. By construction of $\mathcal{T}$, u_j is contained in the set F_i. Therefore, the set $\mathcal{F}' := \{F_i \mid T_i \in \mathcal{T}'\}$ covers at least τ elements of U. Since $\mathcal{T}' \leq \kappa$, there are thus at most κ sets in $\mathcal{F}$ that cover at least τ elements of U.

To obtain the reduction for arbitrary $k > 2$, one may adapt the construction by representing each u_i by a string of length k. ◀

Lemma 1 directly implies the following hardness result for the parameter t_2 that bounds the number of strings to insert.

▶ **Corollary 2.** *For each $k \geq 2$, k-MER-SF is W[1]-hard with respect to the parameter t_2.*

Next, observe that PARTIAL SET COVER is a special case of SET COVER. This implies that PARTIAL SET COVER does not admit a polynomial kernel with respect to $|U| + \kappa$ unless coNP $\subseteq$ NP/poly [9]. Together with Lemma 1 and the facts that the decision version of k-MER-SF is contained in NP and that SET COVER is NP-complete, we thus obtain the following.

▶ **Corollary 3.** *For each $k \geq 2$, k-MER-SF does not admit a kernel with respect to $|G|+\lambda+t_2$ unless coNP $\subseteq$ NP/poly.*

3 A Fixed-Parameter Algorithm for k-Mer Scaffold Filling

We now show how to solve k-MER-SF in $2^{\mathcal{O}(\ell)} \cdot n^{\mathcal{O}(1)}$ time. Let $p_k(\mathcal{S}, G) := a_k(G) \setminus a_k(\mathcal{S})$ denote the multiset of k-mers that is in G but not in the scaffold $\mathcal{S}$. We call these the *potential* common k-mers. Also, for a solution $\mathcal{S}^*$ we will call the common k-mers of $\mathcal{S}^*$ and G that are not k-mers of $\mathcal{S}$ the *realized k-mers.* The algorithm that we describe is based on a combination of dynamic programming and color-coding [1]. It has running time $\mathcal{O}(n^2 \cdot m \cdot k^3 \cdot \ell \cdot |\mathcal{T}| \cdot 8.16^{\ell} \cdot 5.44^{t_2})$. Thus, it is a fixed-parameter algorithm for the combined parameter $\ell + t_2$. As the following reduction rules show, this also gives a fixed-parameter algorithm for the parameter $k + \ell$.

▶ **Reduction Rule 1.** *If $t_1 > k \cdot \ell + 1$, then set $t_1 = k \cdot \ell + 1$. If $t_2 > k \cdot \ell + 1$, then set $t_2 = k \cdot \ell + 1$.*

Proof of Correctness. First, consider the change of t_1. The reduction rule decreases the value of the lower bound t_1 for $|\mathcal{T}'|$. Thus, every feasible solution for the original instance is a feasible solution for the reduced instance that realizes the same number of common k-mers. To show correctness, we must thus only show that for every feasible solution of the reduced instance, there is a feasible solution of the original instance that realizes the same number of common k-mers. To this end, let $\mathcal{T}^*$ be an optimal feasible solution of the new instance and let $\mathcal{S}^*$ denote the resulting string. Let $K := a_k(\mathcal{S}^*, G) \setminus (a_k(\mathcal{S}) \cup a_k(G))$ denote the multiset of potential common k-mers that are realized by $\mathcal{S}^*$. By definition of ℓ, we have $\ell = |K|$. Consider an injective mapping from K to the k-mers in $\mathcal{S}^*$ that contain at least one position from an inserted string $T \in \mathcal{T}^*$. Observe that the total number of positions in these k-mers of $\mathcal{S}^*$ is at most $k \cdot \ell$. By pigeonhole principle, there is thus at least one string $T \in \mathcal{T}^*$ such that none of the k-mers containing T is an image of the mapping. Now obtain a solution for the original instance by adding $t_2 - (k \cdot \ell + 1)$ strings from $\mathcal{T} \setminus \mathcal{T}^*$ directly after T. This does not affect any k-mers that are images of the mapping. Thus, the number of realized k-mers does not decrease, and the decrease of t_1 is correct.

Next, consider the case that the rule increases the value of the upper bound t_2 for $|\mathcal{T}'|$. Thus, every feasible solution for the reduced instance is a feasible solution for the original instance that realizes the same number of common k-mers. To show correctness, we must thus show that for every feasible solution of the original instance, there is a feasible solution of the reduced instance that realizes the same number of common k-mers. To this end, let $\mathcal{T}'$ be an optimal feasible solution of the original instance and assume that $\mathcal{T}'$ is the smallest among all optimal solutions. If $|\mathcal{T}'| \leq k \cdot \ell + 1$, then $\mathcal{T}'$ gives also a feasible solution for the reduced instance. Otherwise, as in the proof for the reduction of t_1, there is at least one string $T \in \mathcal{T}'$ such that none of the k-mers containing T is an image of the mapping from the realized k-mers. Now obtain a solution for the original instance by removing T. This does not affect any k-mers that are images of the mapping. Thus, the number of realized k-mers does not decrease. Moreover, since $t_1 \leq k \cdot \ell + 1$, $|\mathcal{T}' \setminus \{T\}| \geq t_1$. Thus, $|\mathcal{T}' \setminus \{T\}|$ is a feasible solution for the original instance which contradicts the assumption that $\mathcal{T}'$ is a smallest such solution. ◀

Color Coding. Somewhat deviating from the standard color-coding, we use two random colorings α and β. Here, $\alpha : p_k(\mathcal{S}, G) \to [\ell]$ is a coloring of the potential common k-mers and $\beta : \mathcal{T} \to [t_2]$ is a coloring of the strings that may be inserted. The idea is that there is a significant chance for the two random colorings that all of the realized common k-mers and inserted strings have different colors. Under this assumption, we can use dynamic programming on $\mathcal{S}$ to reconstruct a solution that realizes ℓ of the potential common k-mers.

Dynamic Programming. In the dynamic programming routine, we gradually find partial solutions of increasing size, inserting strings from $\mathcal{T}$ into the scaffold in a left-to-right manner. That is, we first insert between the first and second contig, then between the second and third and so on. We use the coloring to avoid inserting some string of $\mathcal{T}$ twice. We fill a five-dimensional table $Q[i, j, \kappa, A, B]$ with 0/1-entries corresponding to partial solutions. In this table:

- the index $i \in [m]$ corresponds to the set of contigs that precede the last character that was inserted,
- the indices $j \in [|G|]$ and $\kappa \in \{0, \dots, k\}$ are used to identify the longest common suffix between the partial solution and G,

- the color sets $A \subseteq [\ell]$ and $B \subseteq [t_2]$ denote the colors of the k-mers that were realized and the colors of the strings that were inserted.

We define the meaning of table Q as follows. A table entry $Q[i, j, \kappa, A, B] = 1$ if and only if there is a set $p' \subseteq p_k(\mathcal{S}, G)$ and a set $\mathcal{T}' \subseteq \mathcal{T}$ such that

1. $\alpha(p') = A$ and $|p'| = |A|$,
2. $\beta(\mathcal{T}') = B$ and $|\mathcal{T}'| = |B|$,
3. there is a string $\mathcal{S}^* \in \mathcal{S}_i + \mathcal{T}'$ such that $a(\mathcal{S}^*, G) \setminus a(\mathcal{S}, G) \subseteq p'$ and $G_\kappa[j]$ is the longest common substring of G that is a suffix of $\mathcal{S}^*$, among all substrings of length at most k of G.

Here, $\mathcal{S}_i := (C_1, \ldots, C_i)$ denotes the scaffold consisting of the first i contigs in the same order as in $\mathcal{S}$ and $G_\kappa[j]$ denotes the length-κ substring of G starting at position j. Before we describe the recurrence in detail, consider the following. When extending a partial solution $\mathcal{S}^*$, we have two choices: either add a string from $\mathcal{T}$ at the end of $\mathcal{S}^*$ or add the next contig, that is, add C_i if the last contig in $\mathcal{S}^*$ is C_{i-1}. The resulting string contains additional k-mers as substrings and some of these k-mers may be potential common k-mers with G. Clearly, to determine the set of additional k-mers it suffices to know the length-k suffix of $\mathcal{S}^*$ and the string that we add. The number of different length-k suffixes, however, is $|\Sigma|^k$. Therefore, storing these in a dynamic programming table would incur a substantial overhead for both running time and space consumption. To be more efficient, we make use of the following fact.

▶ **Fact 1.** *Let S, T and G be strings such that the longest substring of S that is a suffix of G has length at most κ, and let $S' = (S \circ T)[i, j]$ be a substring of $S \circ T$ such that $i \le |S|$ and $j > |S|$. If S' is a substring of G, then $i \ge |S| - \kappa$.*

Hence, the set of additional common k-mers of $\mathcal{S}^* \circ T$ and G are completely determined by the combination of T and the longest suffix of $\mathcal{S}^*$ that is also a substring of G. Finally, the additional realized k-mers are those that are not yet realized by $\mathcal{S}^*$. For our dynamic programming table, we are thus interested in the k-mers that have a certain color set. To determine the possible contribution of adding a string T we use a table $P[j, \kappa, A, T]$. An entry $P[j, \kappa, A, T]$ of P has value 1 if there is a surjective mapping

$$\psi : a_k(G_\kappa[j] \circ T, G) \to A$$

from the multiset of common k-mers of $G_\kappa[j] \circ T$ and G to the color set $A \subseteq [\ell]$ such that $\psi(x) \subseteq \alpha(x)$ for all $x \in a_k(G_\kappa[j] \circ T, G)$. Otherwise, the entry has value 0. Informally, the table P tells us whether adding T to a partial solution helps to realize potential k-mers with the colors of A. If we add a contig C_i, then we may count only those common k-mers that are not completely contained in C_i. Accordingly, the entries $P[j, \kappa, A, C_i]$ have value 1 if there is a surjective mapping

$$\psi : (a_k(G_\kappa[j] \circ C_{i+1}, G) \setminus a_k(C_i)) \to A$$

such that $\psi(x) \subseteq \alpha(x)$ for all $x \in a_k(G_\kappa[j] \circ T, G) \setminus a_k(C_i)$.

▶ **Lemma 4.** *The value of each entry of P can be computed in $\mathcal{O}(k \cdot |T| + |T| \cdot |A|^2)$ time if the multiset of k-mers of G is stored in a trie.*

Proof. First, in $\mathcal{O}(k + |T|)$ time, compute the string $T' = G_\kappa[j] \circ T$. Then compute the multiset $a_k(T', G)$ of common k-mers of T' and G. This can be done in $\mathcal{O}(k \cdot |T|)$ time: the number of k-mers in T' is at most $|T|$ and for each k-mer, we may use the trie to check whether it is in G and to count the number of k-mers of T' that are equivalent. The multiset of common k-mers is then given by determining for each k-mer, the minimum of the multiplicities in T' and in G.

Now, we can determine whether there is a mapping ψ by computing a maximum matching in the graph that is defined by the restriction of α to $a_k(T', G)$, that is, the bipartite graph constructed as follows: For each k-mer K of $a_k(T', G)$ we introduce $m(a_k(T', G), K)$ vertices corresponding to K; this gives one part $V_{T'}$ of the bipartition. The other part of the bipartition is given by A. We draw an edge between $v \in V_{T'}$ and $u \in A$ if $v \in \alpha(K_v)$, where K_v is the k-mer corresponding to v. Then we compute a maximum matching in this graph. Since this matching has at most $|A|$ edges and since the graph has size $\mathcal{O}(|T| \cdot |A|)$, this can be done in time $\mathcal{O}(|T| \cdot |A|^2)$. If every vertex of A is contained in a matching edge, then the table entry is set to 1, otherwise it is 0. ◀

With the table P at hand, we use the following recurrence to fill Q. Informally, the table entry has value 1 if some string T or the next contig C_i can be used, together with a previous partial solution, to realize common k-mers of the desired colors and if the resulting partial solution has a suffix as specified by the values of j and κ.

$$Q[i, j, \kappa, A, B] = \begin{cases} 1 & \exists j', \kappa', A' \subseteq A, T \in \mathcal{T}: \\ & \quad Q[i, j', \kappa', A', B \setminus \beta(T)] = 1 \wedge \\ & \quad P[j', \kappa', A \setminus A', T] = 1 \wedge \\ & \quad G_\kappa[j] = \operatorname{len}(G, G_{\kappa'}[j'] \circ T) \\ 1 & \exists j', \kappa', A': \\ & \quad Q[i-1, j', \kappa', A', B] = 1 \wedge \\ & \quad P[j', \kappa', A \setminus A', C_i] = 1 \wedge \\ & \quad G_\kappa[j] = \operatorname{len}(G, G_{\kappa'}[j'] \circ C_i) \\ 0 & \text{otherwise.} \end{cases}$$

Here, for two strings S and T we use $\operatorname{len}(S, T)$ to denote the longest substring of S that is a suffix of length at most k of T.

▶ **Theorem 5.** *k-MER-SF can be solved in $\mathcal{O}(n^2 \cdot m \cdot k^2 \cdot |\mathcal{T}| \cdot 8.16^\ell \cdot 5.44^{t_2})$ time.*

Proof. The overall number of entries in P is $\mathcal{O}(n \cdot k \cdot |\mathcal{T}| \cdot 2^{t_2})$. Each entry of P can be computed in $\mathcal{O}(k \cdot |T| + |T| \cdot |A|^2)$ time by Lemma 4. This gives an overall running time of $\mathcal{O}(n \cdot k^2 \cdot ||\mathcal{T}|| \cdot 2^{t_2} \cdot (t_2)^2)$ for filling P. The overall number of entries in Q is $\mathcal{O}(m \cdot n \cdot k \cdot 2^\ell \cdot 2^{t_2})$. For each entry $Q[i, j, \kappa, A, B]$, we consider $\mathcal{O}(n \cdot k \cdot 2^{|A|} \cdot |\mathcal{T}|)$ cases in the recurrence. The first two conditions of each case can be determined in $\mathcal{O}(1)$ time, the third condition can be computed in $\mathcal{O}(1)$ time after a preprocessing in which we compute $\operatorname{len}(G, G_\kappa[j] \circ T)$ for each combination of T, κ and j. This can be done in $\mathcal{O}(n^2 \cdot k \cdot |\mathcal{T}|)$ time overall. Thus, the total running time for filling Q including preprocessing is $\mathcal{O}(n^2 \cdot m \cdot k^2 \cdot |\mathcal{T}| \cdot 3^\ell \cdot 2^{t_2})$. After Q is filled, we can determine whether there is a solution realizing ℓ potential common k-mers by considering $Q[m, j, \kappa, [\ell], B]$ for all B with $t_1 \leq |B| \leq t_2$. The overall running time of the algorithm now follows from the number of trials that are necessary to obtain a constant false-negative error probability, as shown by Alon et al. [1], these are exactly $e^{\ell + t_2}$ many. ◀

▶ **Corollary 6.** *k-MER-SF can be solved in $\mathcal{O}(n^2 \cdot m \cdot k^2 \cdot |\mathcal{T}| \cdot 8.16^\ell \cdot 5.44^{k\ell})$ time.*

4 A Polynomial Kernel for a Special Case

As an additional result, we obtain a polynomial problem kernel for the special case when $t_2 \geq 2\ell + 1$, $\lambda = 1$ and $k = 2$ and the parameter is the combination of ℓ and m. Observe that $t_2 \geq 2\ell + 1$ essentially means that there are no upper-bound constraints on the solution size. Thus, our kernel also works for the natural case that $t_2 = |\mathcal{T}|$. The details are given in Appendix. Moreover, observe that even though the problem setting is very restricted compared to the general k-Mer Scaffold Filling, it contains the GSF problem of Jiang et al. [13] as a special case.

▶ **Theorem 7.** *For $k = 2$, $\lambda = 1$ and $t_2 \geq 2\ell + 1$, k-Mer-SF admits a problem kernel of size $\mathcal{O}(\ell^3 \cdot (\ell + m)^2)$ that can be computed in polynomial time.*

5 Minimizing the Number of Breakpoints

In this section, we consider the Min-Bkpt-SF problem. Another formulation of the string breakpoint distance between S and T is via minimum common string partitions [7]. Intuitively, the breakpoint distance is b if the strings S and T can each be partitioned into $b + 1$ factors, so that both partitions use the same multiset of factors. For example, strings $aabcda$ and $bcaada$ have a breakpoint distance of 2 since they can both be partitioned into the size-three factor set $\{aa, bc, da\}$ (see also Figure 2 for another example). This distance is NP-hard to compute [11], however, several FPT algorithms can be used. We will make use of the following two:

- An FPT algorithm for the parameter combining b, the breakpoint distance, and d, the number of duplications of any letter [4];
- An FPT algorithm for parameter b alone [6] (which is mainly of theoretical interest, as the exponential running time on b is rather prohibitive).

We first give two NP-hardness results, each one using a different approach giving different constraints on the values of the parameters. We then introduce two FPT algorithms using $|\mathcal{T}|$ as a parameter (as well as the breakpoint distance b, and either the number of contigs m or the duplication number d). Without parameter $|\mathcal{T}|$, we show that the problem is in XP for parameter b when all strings in $\mathcal{T}$ have length 1.

NP-hardness for $|\mathcal{T}| = 0$, $m = 1$, and either $d = 2$ or $|\Sigma| = 2$. The first hardness result below directly follows from the fact that the string breakpoint distance is hard to compute. Hence, any parameterized algorithm needs to put some restriction on the target distance b.

▶ **Theorem 8.** Min-Bkpt-SF *is NP-hard with $|\mathcal{T}| = 0$, $m = 1$ even when either $d = 2$ or $|\Sigma| = 2$.*

Proof. With an empty set $\mathcal{T}$ and a single contig C_1, Min-Bkpt-SF comes down to computing the breakpoint distance between two strings. It is NP-hard even in special cases of binary alphabet [18], as well as when any letter occurs at most twice [11]. ◀

NP-hardness for $m = 2$ and $b = 0$. When $b = 0$, we look for a way of inserting strings of $\mathcal{T}$ in $\mathcal{S}$ in order to obtain exactly G. This problem turns out to be NP-hard, hence, again, any parameterized algorithm needs not only to put a bound on the the target distance b, but also some restriction on the set of missing strings.

▶ **Theorem 9.** Min-Bkpt-SF *is NP-hard even when* $m = 2$ *and* $b = 0$.

Proof. We propose a reduction from Unary Bin Packing:

Input: A list of n integers $(x_1, x_2, \ldots, x_n)$, given in unary, integers B and k such that, $kB = \sum x_i$.
Task: Find a partition $(P_1, \ldots, P_k)$ of $[n]$ such that, for all $j \in [k]$, $\sum_{i \in P_j} x_i = B$.

We reduce to Min-Bkpt-SF as follows: let $G = (10^B)^k 1$, $\mathcal{S}$ consist of $m = 2$ contigs C_1 and C_2 with $C_1 = C_2 = 1$, let $\mathcal{T}$ contain $k-1$ strings 1 and strings 0^{x_i} for all $i \in [n]$. Finally, set $b = 0$. Consider $\mathcal{S}^* \in \mathcal{S} + \mathcal{T}$; $\mathcal{S}^*$ yields a partition of $[n]$ into k subsets P_j, where $i \in P_j$ if string 0^{x_i} from $\mathcal{T}$ is inserted between the $(j-1)$th and jth string 1 of $\mathcal{T}$ (or before the first/after the last for $j = 1$ and $j = k$ respectively). Then $b(\mathcal{S}^*, G) = 0$ if and only if each P_j, $j \in [k]$, is such that $\sum_{i \in P_j} x_i = B$. Conversely, any such partition of $[n]$ yields a string in $\mathcal{S} + \mathcal{T}$ at distance 0 from G. Hence, the instance $(\mathcal{S}, \mathcal{T}, G, b = 0)$ of Min-Bkpt-SF is a yes-instance if and only if the original Unary Bin Packing instance is a yes-instance. ◀

An FPT Algorithm for the parameter $(|\mathcal{T}|, m, b)$. We now present an algorithm for the case that the parameter combines the number τ of strings in $\mathcal{T}$, the number of contigs m, and the breakpoint distance b.

▶ **Theorem 10.** Min-Bkpt-SF *is FPT for parameters* $|\mathcal{T}|$, b *and* m.

Proof. If we consider parameters $|\mathcal{T}|$ and m, then together they allow the exhaustive enumeration of all possible strings in $\mathcal{S} + \mathcal{T}$: First, compute the $|\mathcal{T}|!$ possible arrangements of strings in $\mathcal{T}$, then split the resulting string into at most m blocks without breaking substrings corresponding to the strings in $\mathcal{T}$ (this creates at most $\binom{|\mathcal{T}|}{m} \leq 2^{|\mathcal{T}|}$ branches), then consider all choices to insert them between the contigs of $\mathcal{S}$ (this creates at most 2^m branches).

Once a candidate $\mathcal{S}^*$ is known, it remains to compute the breakpoint distance with G in time $f(b)\mathrm{n}^{\mathcal{O}(1)}$ [6]. Overall, this gives an FPT algorithm for parameters $|\mathcal{T}|$, b and m. ◀

An FPT Algorithm for the parameter $(|\mathcal{T}|, d, b)$. The next algorithm is more efficient if the number of duplications d is small and avoids the dependency on the contig number m.

▶ **Theorem 11.** Min-Bkpt-SF *can be solved in time* $\mathcal{O}((4|\mathcal{T}|d^2)^{|\mathcal{T}|} d^{2b} bn^2)$.

Proof. Given an optimal solution $\mathcal{S}^*$, we call T-factor a maximal factor of $\mathcal{S}^*$ containing strings from $\mathcal{T}$ (a T-factor is the concatenation of all substrings inserted between two contigs). A T-factor is *left-joined* (resp. *right-joined*) if there is no breakpoint to its left, i.e., between the last letter of the previous contig and its own first letter (resp., to its right). A T-factor is *stand-alone* if it is neither left- nor right-joined. We can assume wlog. that there is a single stand-alone T-factor, as any two such factors can be inserted in the same gap between two contigs, so that they are merged into one without increasing the distance.

The first step of our algorithm is to guess the T-factors (using $|\mathcal{T}|^{|\mathcal{T}|}$ branches). They will be denoted $f_1, \ldots, f_h$. For each T-factor, we guess whether it is left-joined, and whether it is right-joined (using $4^{|\mathcal{T}|}$ branches). We first deal with the single stand-alone T-factor, if any: guess the correct gap where it should be inserted (among $m \leq n$ choices), insert it there and merge it with the surrounding two contigs. Consider now a T-factor f_j, assume that it is left-joined (otherwise it is necessarily right-joined, then processed symmetrically). Let u be the first letter of f_j. Guess the position i such that $G[i]$ is matched to u (among d options). Let u' be the letter at position $i-1$ in G. Since there is no breakpoint before u, f_j

must be inserted after a contig ending with u'. There are at most d such contigs, so we can enumerate all options. There are at most d^2 branches for each T-factor, and $d^{2|\mathcal{T}|}$ branches overall. The branching above allows to enumerate all candidate strings for a solution, it remains to check whether any of them has breakpoint distance b to G. We compute this distance for each candidate, using an FPT algorithm [4] with running time $\mathcal{O}(d^{2b}bn)$. The overall running time is $\mathcal{O}((4|\mathcal{T}|d^2)^{|\mathcal{T}|}d^{2b}bn^2)$. ◀

An XP Algorithm for the parameter b when $\lambda = 1$

▶ **Theorem 12.** Min-Bkpt-SF *can be solved in time $\mathcal{O}(n^{b+1}(b+1)!)$ when all strings in $\mathcal{T}$ have length 1.*

Proof. The algorithm runs as follows: first enumerate all possible positions of the breakpoints in G, using $|G|^b$ branches. The optimal string $\mathcal{S}^*$ can be guessed by trying all possible rearrangements of the $b+1$ factors separated by breakpoints, using $(b+1)!$ branches.

It remains to check that $\mathcal{S}^* \in \mathcal{S} + \mathcal{T}$, i.e., some filling of $\mathcal{S}$ gives $\mathcal{S}^*$. This task is NP-hard in the general case (see Theorem 9) however, it is straightforwardly achieved in linear time when all strings in T have length 1: it suffices to check that all contigs in $\mathcal{S}$ are factors of $\mathcal{S}^*$ in the correct order, and that $\mathcal{T}$ contains exactly the multi-set of missing letters. ◀

6 Conclusion

For k-Mer-SF, the most interesting direction seems to be to extend the problem kernelization to more general cases by allowing either $k > 2$ (that is, considering k-mer distance for general k), $\lambda > 1$ (that is, allowing the insertion of strings or letters), or considering the case where the number of scaffold contigs is unbounded. For Min-Bkpt-SF it remains open whether the problem is FPT for other parameters than $|\mathcal{T}|$, for example m, b or d. We conjecture that the FPT algorithm for MCSP with parameters b and d [4] can be extended for our problem with this combination of parameters. Hopefully some new techniques might reduce the complexity to achieve fixed-parameter tractability for $b + d$ or $b + \ell$ only.

Finally, from a broader point of view, the problems that we consider here are fundamental on strings. Indeed, they belong to a larger family of problems that can be described as follows: Given a string G and a partial string S, complete the partial string S such that it is as close as possible to G. Investigating this type of problems more systematically could be a rewarding topic.

References

1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995. doi:10.1145/210332.210337.

2 Markus Bläser. Computing small partial coverings. *Inf. Process. Lett.*, 85(6):327–331, 2003. doi:10.1016/S0020-0190(02)00434-9.

3 Laurent Bulteau, Anna Paola Carrieri, and Riccardo Dondi. Fixed-parameter algorithms for scaffold filling. *Theor. Comput. Sci.*, 568:72–83, 2015. doi:10.1016/j.tcs.2014.12.005.

4 Laurent Bulteau, Guillaume Fertin, Christian Komusiewicz, and Irena Rusu. A fixed-parameter algorithm for minimum common string partition with few duplications. In Aaron E. Darling and Jens Stoye, editors, *Proceedings of the 13th International Workshop on Algorithms in Bioinformatics (WABI 2013)*, volume 8126 of *LNCS*, pages 244–258. Springer, 2013. doi:10.1007/978-3-642-40453-5_19.

5 Laurent Bulteau, Falk Hüffner, Christian Komusiewicz, and Rolf Niedermeier. Multivariate algorithmics for NP-hard string problems. *Bull. EATCS*, 114, 2014. URL: `http://eatcs.org/beatcs/index.php/beatcs/article/view/310`.

6 Laurent Bulteau and Christian Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In Chandra Chekuri, editor, *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, pages 102–121. SIAM, 2014. `doi:10.1137/1.9781611973402.8`.

7 Xin Chen, Jie Zheng, Zheng Fu, Peng Nan, Yang Zhong, Stefano Lonardi, and Tao Jiang. Assignment of orthologous genes via genome rearrangement. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 2(4):302–315, October 2005. `doi:10.1109/TCBB.2005.48`.

8 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. `doi:10.1007/978-3-319-21275-3`.

9 Michael Dom, Daniel Lokshtanov, and Saket Saurabh. Kernelization lower bounds through colors and IDs. *ACM Trans. Algorithms*, 11(2):13:1–13:20, 2014. `doi:10.1145/2650261`.

10 Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013. `doi:10.1007/978-1-4471-5559-1`.

11 Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. *Electron. J. Comb.*, 12, 2005. URL: `http://www.combinatorics.org/Volume_12/Abstracts/v12i1r50.html`.

12 Jiong Guo, Rolf Niedermeier, and Sebastian Wernicke. Parameterized complexity of vertex cover variants. *Theory Comput. Syst.*, 41(3):501–520, 2007. `doi:10.1007/s00224-007-1309-3`.

13 Haitao Jiang, Chenglin Fan, Boting Yang, Farong Zhong, Daming Zhu, and Binhai Zhu. Genomic scaffold filling revisited. In Roberto Grossi and Moshe Lewenstein, editors, *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *LIPIcs*, pages 15:1–15:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.CPM.2016.15`.

14 Haitao Jiang, Jingjing Ma, Junfeng Luan, and Daming Zhu. Approximation and nonapproximability for the one-sided scaffold filling problem. In Dachuan Xu, Donglei Du, and Ding-Zhu Du, editors, *Proceedings of the 21st International Conference on Computing and Combinatorics (COCOON 2015)*, volume 9198 of *LNCS*, pages 251–263. Springer, 2015. `doi:10.1007/978-3-319-21398-9_20`.

15 Haitao Jiang, Chunfang Zheng, David Sankoff, and Binhai Zhu. Scaffold filling under the breakpoint distance. In Eric Tannier, editor, *Proceedings of the International Workshop on Comparative Genomics (RECOMB-CG 2010)*, volume 6398 of *LNCS*, pages 83–92. Springer, 2010. `doi:10.1007/978-3-642-16181-0_8`.

16 Haitao Jiang, Chunfang Zheng, David Sankoff, and Binhai Zhu. Scaffold filling under the breakpoint and related distances. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 9(4):1220–1229, 2012. `doi:10.1109/TCBB.2012.57`.

17 Haitao Jiang, Farong Zhong, and Binhai Zhu. Filling scaffolds with gene repetitions: Maximizing the number of adjacencies. In Raffaele Giancarlo and Giovanni Manzini, editors, *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*, volume 6661 of *LNCS*, pages 55–64. Springer, 2011. `doi:10.1007/978-3-642-21458-5_7`.

18 Haitao Jiang, Binhai Zhu, Daming Zhu, and Hong Zhu. Minimum common string partition revisited. *J. Comb. Optim.*, 23(4):519–527, 2012. `doi:10.1007/s10878-010-9370-2`.

19 Nan Liu, Peng Zou, and Binhai Zhu. A polynomial time solution for permutation scaffold filling. In T.-H. Hubert Chan, Minming Li, and Lusheng Wang, editors, *Proceedings of the 10th International Conference on Combinatorial Optimization and Applica-*

tions (COCOA 2016), volume 10043 of *LNCS*, pages 782–789. Springer, 2016. doi:10.1007/978-3-319-48749-6_60.

20 Adriana Muñoz, Chunfang Zheng, Qian Zhu, Victor A. Albert, Steve Rounsley, and David Sankoff. Scaffold filling, contig fusion and comparative gene order inference. *BMC Bioinformatics*, 11:304, 2010. doi:10.1186/1471-2105-11-304.

21 Binhai Zhu. Genomic scaffold filling: A progress report. In Daming Zhu and Sergey Bereg, editors, *Proceedings of the 10th International Workshop on Frontiers in Algorithmics (FAW 2016)*, volume 9711 of *LNCS*, pages 8–16. Springer, 2016. doi:10.1007/978-3-319-39817-4_2.

A Kernelization Algorithm from Section 4

In this section, we present an kernelization algorithm for the parameter ℓ and the special case when $t_2 =\geq 2\ell+1$, $\lambda = 1$ and $k = 2$ and m is a constant.

To obtain a problem kernel, we need to reduce the size of three objects: the genome G, the scaffold $\mathcal{S}$, and the multiset $\mathcal{T}$ of letters that we may add. We will achieve this in two steps: First, we will reduce the number of copies of letters in $\mathcal{T}$ and of copies of adjacencies and the size of $\mathcal{S}$. After this reduction step, we observe that it only remains to reduce the number of different letters in the instance. Consequently, this is what we reduce in the remainder of the algorithm.

Throughout the description of the algorithm, let x denote a letter that does not occur in $\mathcal{S}$ and not in $\mathcal{T}$ and let y denote a letter that does not occur in G.

The first rule for the kernel removes superfluous copies of letters from $\mathcal{T}$. It is obviously correct, since no solution can add such letters.

▶ **Reduction Rule 2.** *If $\mathcal{T}$ contains a letter b more than t_2 times, remove a copy of b from $\mathcal{T}$.*

The next rule, Rule 3 aims at separating the adjacencies in G and keeping only the potential adjacencies in G. This will be useful to reduce the size of G but also to reduce the size of $\mathcal{S}$, which is done by Rule 4. Recall that the potential common k-mers of an instance are the k-mers which are contained in G not in $\mathcal{S}$. For $k = 2$, we may speak of *potential common adjacencies*. Also, for a solution $\mathcal{S}^*$ we will call the common adjacencies of $\mathcal{S}^*$ and G that are not adjacencies of $\mathcal{S}$ the *realized adjacencies*.

▶ **Reduction Rule 3.** *Let x be a letter that is not contained in any contig of $\mathcal{S}$ and not contained in any string of $\mathcal{T}$, and let $p(\mathcal{S}, G) := \{P_1, \ldots, P_q\} = a(G) \setminus a(\mathcal{S})$ denote the potential common adjacencies of G and $\mathcal{S}$. Replace G by $P_1 \circ x \circ P_2 \circ \cdots \circ P_{q-1} \circ x \circ P_q$.*

▶ **Reduction Rule 4.** *Let y be a letter not contained in G. Replace every contig C_i of length at least 2 by $C_i[1] \circ y \circ C_i[|C_i|]$.*

▶ **Lemma 13.** *Let $I = (G, \mathcal{S}, \mathcal{T}, t_1, t_2)$ be an instance of k-MER-SF, and let $I' = (G', \mathcal{S}', \mathcal{T}', t'_1, t'_2)$ be the instance obtained from I by applying Rules 3 and 4 to I. Then, I and I' are equivalent.*

Proof. A solution for I realizing at least ℓ adjacencies, directly gives a solution for I' realizing at least ℓ adjacencies: Every realized adjacency for I is from $p(\mathcal{S}, G) = a(G) \setminus a(\mathcal{S})$ and thus contained in $a(G')$. Moreover, it is not contained in $a(\mathcal{S}')$ and thus it is a potential common adjacency in I'. Now since $\mathcal{T} = \mathcal{T}'$ and since, for any solution of I, all realized adjacencies contain either only letters from $\mathcal{T}$ or only the first or the last letter from some contig C_i, they can be realized in I' as well by inserting the letters in the same order and between the same contigs as in I. The converse direction follows from the same arguments. ◀

Observe that after application of Rule 4, the scaffold has size $\mathcal{O}(m)$.

After separating the potential adjacencies in G with the help of Rule 3, we may now speak of *removing a copy of a potential adjacency bc from* G, which means to replace

$$G = P_1 \circ y \circ \cdots \circ P_{i-1} \circ y \circ bc \circ y \circ P_{i+1} \cdots \circ P_q$$

by

$$P_1 \circ y \circ \cdots \circ P_{i-1} \circ y \circ P_{i+1} \cdots \circ P_q$$

for some arbitrary $P_i = bc$.

▶ **Reduction Rule 5.** *If there is a potential adjacency bc that occurs more than ℓ times in G, then remove a copy of this adjacency from G.*

Proof of Correctness. Let I denote the original instance and let I' denote the instance obtained by the application of the rule. We need to show only that if I has a solution, then so does I', as the other direction is trivial. Thus, assume that I has a solution realizing at least ℓ adjacencies. Choose an arbitrary multiset P of ℓ realized adjacencies and observe that there is at least one copy of the adjacency bc in G that is not in P. Thus, removing this adjacency from G gives a new genome G^* such that $a(G^*) \setminus a(\mathcal{S}) \geq |P| \geq \ell$ since the adjacencies of P are contained in $a(G^*)$. Since the multiset of potential common adjacencies of G^* and of the genome in I' are the same and since the scaffold $\mathcal{S}$ and the set $\mathcal{T}$ are not changed by the rule, I' has a solution realizing at least ℓ common adjacencies. ◀

As the following lemma shows, we have already achieved the goal of the first step, that is, we have reduced the number of copies of all letters in I.

▶ **Lemma 14.** *Let I be an instance that is reduced with respect to Rules 1–5, and let c denote the number of different letters occurring in G, $\mathcal{S}$, and $\mathcal{T}$. Then, $|G| \leq 3\ell c^2$ and $||\mathcal{T}|| \leq c \cdot (2\ell + 1)$.*

Proof. First, we bound the size of G. If the overall number of letters is c, then there are at most c^2 different adjacencies in G. Moreover, by the construction of G, every third adjacency does not contain x. Thus, if $|G| > 3\ell \cdot c^2$, then there is some adjacency that does not contain the letter x and that occurs more than ℓ times. This contradicts the assumption that the instance is reduced with respect to Rule 5.

The bound on the total length of $\mathcal{T}$ follows from the fact that $\mathcal{T}$ contains at most c many different letters, each of which occurs at most $2\ell + 1$ times since the instance is reduced with respect to Rule 1 and 2. ◀

According to Lemma 14, to obtain a polynomial problem kernel it is sufficient to reduce the number of letters in the instance to $\ell^{\mathcal{O}(1)}$. Consequently, this is our aim in the second step of the kernelization algorithm.

First, we remove those letters from G and $\mathcal{T}$ which are useless in the sense that they occur in no adjacencies which can become common adjacencies of a solution $\mathcal{S}^*$ and G.

▶ **Definition 15.** We call an adjacency bc *realizable* if bc occurs in G, and

- $b \in \mathcal{T}$ and $c \in \mathcal{T}$, or
- $b = C_i[|C_i|]$ and $c \in \mathcal{T}$ for some contig C_i, or
- $b = C_i[|C_i|]$ and $c = C_{i+1}[1]$ for some contig C_i, or
- $b \in \mathcal{T}$ and $c = C_i[1]$ for some contig C_i.

We now remove those letters that do not occur in realizable adjacencies.

▶ **Reduction Rule 6.** *If $|\mathcal{T}| > t_1$ and $\mathcal{T}$ contains a letter b that occurs in no realizable adjacency, then remove a copy of b from $\mathcal{T}$.*

If G contains an adjacency bc that does not contain x and that cannot be realized, then remove bc from G.

The correctness of the rule above follows in a straightforward manner from the fact that we will never insert a letter that is removed by the rule or realize an adjacency that is removed by the rule.

For the final step of the kernelization, we build an auxiliary letter-adjacency graph $H = (V, E)$ as follows. For each letter in $\mathcal{T}$, G, and $\mathcal{S}$, add one vertex to H. Make two vertices b and c adjacent in this graph if the adjacency bc or the adjacency cb is realizable. Observe that after application of Rule 6, every vertex in H except x,y, and possibly the $2m - 2$ vertices corresponding to letters of contigs, has at least one neighbor. Thus, our aim in the following is to reduce the number of vertices in H that have at least one neighbor.

▶ **Reduction Rule 7.** *Let M be a maximum matching in G. If $|M| \geq \ell + 1$, then answer "yes".*

Proof of Correctness. We show how to construct a solution for the k-Mer-SF instance. Let $\{\{b_1, c_1\}, \{b_2, c_2\}, \dots, \{b_{\ell+1}, c_{\ell+1}\}\}$ be a set of $\ell + 1$ edges contained in M and assume without loss of generality that $b_i c_i$ is a realizable adjacency for each i.

First, assume that for all $i \in [\ell + 1]$ either $b_i \in \mathcal{T}$ or $c_i \in \mathcal{T}$. For each i, do the following. If $b_i \in \mathcal{T}$ and $c_i \in \mathcal{T}$, add $b_i c_i$ between C_1 and C_2. If $b_i \in \mathcal{T}$ and $c_i = C_j[1]$ for some C_j, then add b_i directly in front of C_j. If $b_i = C_j[|C_j|]$ and $c_i \in \mathcal{T}$, then add c_i directly after C_j. The set of inserted letters realizes at least $\ell+1$ adjacencies, since the $\ell+1$ pairs $\{b_i, c_i\}$ are disjoint and since for each we realize one adjacency. To obtain a feasible solution, insert $t_1 - (\ell + 1)$ further letters at an arbitrarily chosen position. This breaks at most one adjacency thus giving a solution that realizes at least ℓ adjacencies.

If for some i, we have $b_i = C_j[|C_j|]$ and $c_i = C_{j+1}[1]$, then choose an arbitrary such i and add all adjacencies $b_q c_q$ with $b_q \in \mathcal{T}$ and $c_q \in \mathcal{T}$ between C_j and C_{j+1}. Add $t_1 - (\ell + 1)$ further letters right before C_{j+1}. Insert all other letters as described above. The number of realized adjacencies is at least ℓ. All adjacencies of M except the adjacency $b_i c_i$ are realized: if at least one letter in the adjacency is from $\mathcal{T}$, then they are realized because this letter is inserted in the right position. If neither b_j nor c_j are from T, then the adjacency is realized because no letters are inserted between the consecutive contigs that contain b_j and c_j. ◀

Now let $V(M)$ denote the endpoints of the matching. We show that if $|V \setminus V(M)| > (2\ell + m) \cdot |V(M)|$, then we can safely remove some adjacency from G.

To apply the next rule, we build two bipartite graphs H^1 and H^2. In both graphs, the vertex parts are $B := V(M)$ and $C := (V \setminus V(M))$. In H^1, we add an edge between $b \in B$ and $c \in C$ when bc is a realizable adjacency. In H^2, we add an edge between $b \in B$ and $c \in C$ when cb is a realizable adjacency.

▶ **Reduction Rule 8.**

- *If there is a vertex $b \in B$ of degree at least $2\ell + m + 1$ in H^1, then remove the adjacency bc from G, where c is an arbitrary neighbor of b.*
- *If there is a vertex $b \in B$ of degree at least $2\ell + m + 1$ in H^2, then remove the adjacency cb from G, where c is an arbitrary neighbor of b.*

Proof of Correctness. We show the correctness for the first part of the rule, the second part is symmetric. Consider an instance before application of the rule and assume it has a solution realizing at least ℓ adjacencies. If none of these adjacencies is bc, the adjacency removed from G by the rule, then this solution directly implies a solution for the reduced instance. Otherwise, fix an arbitrary minimal set P of positions containing a letter of the ℓ many realized adjacencies. Observe that $|P| \leq 2\ell$ and there are at most ℓ pairs of consecutive contigs that have nonempty intersection with these positions. Now, consider the adjacency bc that is contained in the solution but not contained in G. There are at least $m+1$ letters d that are adjacent to b in H^1 such that not all copies of d are contained in P. Of these, at most m are letters from contings. Thus, there is a d such that $d \in \mathcal{T}$ and not all copies of d are contained in P. Therefore, inserting d behind b destroys the adjacency bc, but instead creates the adjacency bd. This adjacency is also contained in G and not realized by any position of P. This restores the number of realized adjacencies to ℓ. ◀

▶ **Theorem 7.** *For $k = 2$, $\lambda = 1$ and $t_2 \geq 2\ell + 1$, k-MER-SF admits a problem kernel of size $\mathcal{O}(\ell^3 \cdot (\ell + m)^2)$ that can be computed in polynomial time.*

Proof. Consider an instance that is reduced with respect to all presented reduction rules. By Lemma 14, our claim follows if we show that the number of letters in I is $\mathcal{O}(\ell \cdot (\ell + m))$. This can be seen by considering the graph H: the graph H has $\mathcal{O}(m)$ vertices that have no neighbors. The number of further vertices is $\mathcal{O}(\ell \cdot (\ell + m))$: After applying Rule 7, the size of the matching M is $\mathcal{O}(\ell)$. Any vertex in H that is incident with at least one edge and not an endpoint of M is adjacent to a vertex of M either in H^1 or in H^2. After applying Rule 8, the number of these vertices is at most $|V(M)| \cdot 2 \cdot (2\ell + m + 1) = \mathcal{O}(\ell \cdot (\ell + m))$. This gives the bound on the number of vertices in G and thus on the instance size.

The running time follows from the fact that all reduction rules can be clearly performed in polynomial time. ◀

On the Weighted Quartet Consensus Problem*

Manuel Lafond[1] and Celine Scornavacca[2]

1 **Department of Mathematics and Statistics, University of Ottawa, Ottawa, Canada**
mlafond2@uottawa.ca

2 **Institut des Sciences de l'Evolution, Université Montpellier, CNRS, IRD, EPHE, Montpellier, France**
celine.scornavacca@umontpellier.fr

Abstract

In phylogenetics, the *consensus* problem consists in summarizing a set of phylogenetic trees that all classify the same set of species into a single tree. Several definitions of consensus exist in the literature; in this paper we focus on the Weighted Quartet Consensus problem, a problem with unknown complexity status so far. Here we prove that the Weighted Quartet Consensus problem is NP-hard and we give a 1/2-factor approximation for this problem. During the process, we propose a derandomization procedure of a previously known *randomized* 1/3-factor approximation. We also investigate the fixed-parameter tractability of this problem.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics, G.2.2 Graph Theory

Keywords and phrases phylogenetic tree, consensus tree, quartets, complexity, fixed-parameter tractability

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.28

1 Introduction

Phylogenetics is the branch of biology that studies evolutionary relationships among different species. These relationships are represented via *phylogenetic trees* – acyclic connected graphs with leaves labeled by species – which are reconstructed from molecular and morphological data [12]. One fundamental problem in phylogenetics is to summarize the information contained in a set of unrooted trees $\mathcal{T}$ classifying the same set of species into a single tree T. The set $\mathcal{T}$ can consist of optimal trees for a single data set, of trees issued from a bootstrap analysis of a unique data set, or even of trees issued from the analysis of different data sets. Several consensus methods have been proposed in the past, the probably most known are the strict consensus [23, 18] and the majority-rule consensus [17, 3]. For a survey, see [7].

In this paper we focus on the Weighted Quartet Consensus (WQC) problem [19], also called the Median Tree with Respect to Quartet Distance problem [2] and Quartet Consensus problem in [16]. Roughly speaking, this problem consists in finding a consensus tree maximizing the weights of the 4-leaf trees – *quartets* – it displays, where the weight of a quartet is defined as its frequency in the set of input trees (for a more formal definition, see next section).

More general versions of this problem, where the input trees are allowed to have missing species or where the weight of a quartet is not defined w.r.t. a set of trees, are known to be

* This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Mitacs Globalink Campus France program.

NP-hard [24] (and in fact, even Max-SNP-Hard), but the complexity of the WQC problem has remained open so far. This problem has been conjectured to be NP-hard [2, 19], and heuristics have recently been implemented and widely used, for instance ASTRAL [20], which is a practical implementation of Bryant and Steel's work from[8] (in fact, we show that the ASTRAL algorithm is a 2-approximation for the minimization version of WQC). So far, the best known approximation algorithm for the WQC problem consists in choosing a random tree as a solution [16]. This tree is expected to contain a third of the quartets from the input trees, and so it is a randomized factor 1/3 approximation. In [2], the *minimization* version of the problem is studied, where the objective is to find a median tree T minimizing the sum of quartet *distances* between T and the input trees (the quartet distance between two trees T_1 and T_2 is defined as the number of quartets in T_1 that are not in T_2). A 2-approximation algorithm is given, based on the fact that the quartet distance is a metric [9, 2].

A related problem that has received more attention is the *Complete Maximum Quartet Compatibility* problem (CMQC) (see [5, 4, 16, 14, 25, 26, 10, 21, 22]). In the CMQC problem, we are given, for each set S of four species, exactly one quartet on S, and the objective is to find a tree containing a maximum number of quartets from the input. This can be seen as a version of WQC in which each set of four species has one quartet of weight 1, and the others have weight 0. The CMQC and WQC are however fundamentally different. Although one could apply an algorithm for CMQC to WQC (by keeping only the most frequent quartet on each set of four taxa), maximizing the most-frequent quartets may enforce the presence of many low-frequency quartets. A better solution may prefer more of the middle-frequency quartets, and we give an example of this phenomenon. It was shown in [16] that the CMQC problem admits a polynomial time approximation scheme (PTAS), but it can only be extended to WQC intances on a constant number of trees. Also, CMQC was shown in [14, 10] to be fixed-parameter tractable w.r.t. the parameter "number of quartets to reject from the input".

The main result of this paper is to establish the NP-hardness of the WQC problem. In Section 2, we introduce preliminary notions, and in Section 3 we propose a reduction from the NP-hard Cyclic Ordering problem to WQC. It can be shown that this hardness result transfers to the *rooted* setting, in which case we want to optimize *triplets* (3-leaf rooted trees) rather than quartets. In Section 4, we discuss how being in a consensus setting, i.e. having weights based on a set of input trees on the same leaf set rather than arbitrary weights, does not necessarily make the problem easier, as one could expect: We list some structural properties that, surprisingly, are not satisfied by optimal solutions of a WQC instance. Nevertheless, in Section 5 we devise a factor 1/2 approximation algorithm for WQC running in time $O(k^2n^2 + kn^4 + n^5)$, where k is the number of trees and n the number of species (the best known randomized algorithm in the non-consensus setting is a factor 1/3 one). As a matter of fact, our algorithm includes a derandomization of this procedure, which had never been done before. Finally in Section 6, we show that the known FPT algorithms for the CMQC problem can be extended to the consensus setting. This yields an FPT algorithm that is efficient on instances in which there is not too much ambiguity, i.e. when few competing quartets on the same 4 species appear with the same frequency. We then conclude with some remarks and open problems related to the quartet consensus problem.

2 Preliminaries

An *unrooted phylogenetic tree* T consists of vertices connected by edges, in which every pair of nodes is connected by exactly one path and no vertex is of degree two. The *leaves* of a tree

T, denoted by $L(T)$ are the set of vertices of degree one. Each leaf is associated to a label; the set of labels associated to the leaves of a tree T is denoted by $\mathcal{L}(T)$. We suppose that there is a bijection between $L(T)$ and $\mathcal{L}(T)$. Due to this bijectivity, we will refer to leaves and labels interchangeably. We denote $|\mathcal{L}(T)|$ as $|T|$. In the following, we will often omit the word "phylogenetic" and, unless otherwise stated, all trees are leaf-labeled. A *binary* unrooted tree has only vertices with degree three and vertices with degree one. A rooted (binary) phylogenetic tree is defined in the same way, except that it has exactly one node of degree two called the *root*, denoted by $r(T)$. Note that sometimes in the literature, rooted trees are seen as directed and such that all arcs are oriented away from the root. Unless stated otherwise, all trees in this paper are unrooted.

Given an unrooted phylogenetic tree T and a subset $Y \subseteq \mathcal{L}(T)$, we denote by $T|Y$ the tree obtained from the minimal subgraph of T connecting Y when contracting degree-2 vertices. A *quadset* is a set of four labels. For a quadset $\{a,b,c,d\}$, there are three non-isomorphic[1] unrooted binary trees, called *quartets*, which are denoted respectively by $ab|cd$, $ac|bd$, and $ad|bc$, depending on how the central edge splits the four labels. We say that an unrooted tree T *displays* the quartet $ab|cd$ if $T|\{a,b,c,d\}$ is $ab|cd$. We denote the set of quartets that an unrooted tree T displays by $Q(T)$. Note that if T is binary, then $|Q(T)| = \binom{|\mathcal{L}(T)|}{4}$. A set of quartets Q on a set L is said to be *complete* if for each quadset $\{a,b,c,d\} \subseteq L$, there is in Q *exactly* one quartet among $ab|cd$, $ac|bd$, and $bc|ad$.

We are now ready to state our optimization problem. The Weighted Quartet Consensus problem asks for a tree that has as many quartets as possible in common with a given set of trees on the same set of labels $\mathcal{X}$:

Weighted Quartet Consensus (WQC) problem
Input: a set of unrooted trees $\mathcal{T} = \{T_1, \ldots, T_k\}$ such that $\mathcal{L}(T_1) = \ldots = \mathcal{L}(T_k) = \mathcal{X}$.
Output: a binary unrooted tree M with $\mathcal{L}(M) = \mathcal{X}$ that maximizes $\sum_{T \in \mathcal{T}} |Q(M) \cap Q(T)|$.

The problem is weighted as each quartet on $\mathcal{X}$ is weighted by frequency in $\mathcal{T}$, see below.

In this paper we will focus on the particular case where the input trees are all binary. In fact, proving the problem NP-hard for this restricted case implies NP-hardness of the general problem. Note however that relaxing the requirement of the output M to be binary leads to a different problem, as one needs to define how unresolved quartets in M are weighted.

In the remainder of the article, we will sometimes consider multi-sets of quartets, that are sets in which the same quartet can appear multiple times. Denote by $f_{\mathcal{Q}}(q)$ the number of times that a quartet q appears in a multi-set $\mathcal{Q}$ (we may write $f(q)$ if $\mathcal{Q}$ is unambiguous). We say that a tree T *contains* k quartets of $\mathcal{Q}$ if there are distinct quartets $q_1, \ldots, q_p \in Q(T)$ such that $\sum_{i=1}^{p} f(q_i) = k$. The Weighted Quartet Consensus problem can be rephrased as follows: given trees $T_1, \ldots, T_k$, finding a tree M that contains a maximum number of quartets from $Q(T_1) \uplus Q(T_2) \uplus \ldots \uplus Q(T_k)$, where $\uplus$ denotes multiset union. We will implicitly work with the decision version of WQC, i.e. for a given integer q, is there a consensus tree M containing at least q quartets from $Q(T_1) \uplus Q(T_2) \uplus \ldots \uplus Q(T_k)$?

Given a quadset $\{a,b,c,d\}$, we say that $ab|cd$ is *dominant* (w.r.t. f) if $f(ab|cd) \geq f(ac|bd)$ and $f(ab|cd) \geq f(ad|bc)$. We say that $ab|cd$ is *strictly dominant* if both inequalities are strict.

[1] Isomorphism preserving labels.

3 NP-hardness of the Weighted Quartet Consensus problem

In this section, we present a reduction from the Cyclic ordering problem. This problem has been used in phylogenetics before in [15] in the context of inferring rooted binary trees from rooted triplets that are not required to originate from a set of trees on the same leaf set.

But beforehand, we need some additional notation. A *caterpillar* is an unrooted binary tree obtained by taking a path $P = p_1 p_2 \dots p_r$, then adding a leaf ℓ_i adjacent to p_i for each $1 \leq i \leq r$, then adding another leaf ℓ'_1 adjacent to p_1 and a leaf ℓ'_r adjacent to p_r. The two leaves ℓ'_1 and ℓ'_r inserted last are called the *ends* of the caterpillar. An *augmented caterpillar* T is a binary tree obtained by taking a caterpillar, then replacing each leaf by a binary rooted tree (replacing a leaf ℓ by a tree T' means replacing ℓ by $r(T')$). If T_1, T_2 are the two trees replacing the ends of the caterpillar, then T is called a (T_1, T_2)-augmented caterpillar. Note that every binary tree is a (T_1, T_2)-augmented caterpillar for some T_1, T_2. Let T be a caterpillar with leaves $(\ell_1, \ell_2, \dots, \ell_k)$ taken in the order in which we encounter them on the path between the two ends l_1 and l_k (more precisely, traverse the $\ell_1 - \ell_k$ path, and each time an internal node is encountered, append its adjacent leaves to the sequence), and let $T_1, \dots, T_k$ be rooted binary trees. We denote by $(T_1|T_2|\dots|T_k)$ the (T_1, T_k)-augmented caterpillar obtained by replacing ℓ_i by $r(T_i)$, $1 \leq i \leq k$. This notation gives rise to a natural ordering of its subtrees, and we say that $T_i < T_j$ if $i < j$ (i.e. T_i appears before T_j in the given ordering of the caterpillar subtrees). Note that by reversing such an ordering, we obtain the same unrooted tree. However, in the proofs the given ordering will be important. Also, since T_1, T_2, and T_{k-1}, T_k are interchangeable, we will simply say that these two pairs are incomparable. If each T_i consists of a single leaf ℓ_i for $2 \leq i \leq k-1$, then we may write $(T_1|\ell_2|\dots|\ell_{k-1}|T_k)$, and $\ell_i < \ell_j$ in T to indicate that ℓ_i appears before ℓ_j in the ordering.

We are now ready to describe the Cyclic Ordering problem. Let $L = (s_1, \dots, s_n)$ be a linear ordering of a set S of at least 3 elements, and let (a, b, c) be an ordered triple, with $a, b, c \in S$. We say that L *satisfies* (a, b, c) if one of the following holds in L: $a < b < c, b < c < a$ or $c < a < b$. If $\mathcal{C}$ is a set of ordered triples we say that L satisfies $\mathcal{C}$ if it satisfies every element of $\mathcal{C}$. Intuitively speaking, L satisfies (a, b, c) when, by turning L into a directed cyclic order by attaching s_n to s_1, one can go from a to b, then to c and then to a. This leads to the following problem definition:

Cyclic Ordering problem
Input: A set S of n elements and a set $\mathcal{C}$ of m ordered triples (a, b, c) of distinct members of S.
Question: Does there exist a linear ordering $L = (s_1, \dots, s_n)$ of S that satisfies $\mathcal{C}$?

The Cyclic Ordering problem is NP-hard [13]. In the rest of this section, we let S and $\mathcal{C}$ be the input set and triples, respectively, of a Cyclic Ordering problem instance. We denote $n = |S|$ and $m = |\mathcal{C}|$. We shall use the following simple yet useful characterization.

▶ **Lemma 1.** *A linear ordering L of S satisfies $\mathcal{C}$ if and only if for each $(a, b, c) \in \mathcal{C}$, exactly two of the following relations hold in L: $a < b, b < c, c < a$.*

Proof. ($\Rightarrow$): let L be a linear ordering satisfying $\mathcal{C}$, and let $(a, b, c) \in \mathcal{C}$. Thus in L, one of $a < b < c, b < c < a$ or $c < a < b$ holds. It is straightforward to verify that in each case, exactly two of $a < b, b < c, c < a$ hold.
($\Leftarrow$): suppose that L does not satisfy $\mathcal{C}$. Then there is some $(a, b, c) \in \mathcal{C}$ such that one of $a < c < b, b < a < c$ or $c < b < a$ does not hold. Again, one can easily verify that each of these cases satisfies only one of $a < b, b < c$ and $c < a$. ◀

Now, from S and $\mathcal{C}$ we construct a set of unrooted binary trees $\mathcal{T}$ on the same set of labels (we will omit the straightforward verification that this construction can be carried out in polynomial time). First let W and Z be two rooted binary trees each on $(nm)^{100}$ leaves (the topology is arbitrary, and the 100 exponent could be optimized). Our trees are defined on the leaf set $\mathcal{X} = S \cup \mathcal{L}(W) \cup \mathcal{L}(Z)$ (note that $S, \mathcal{L}(W), \mathcal{L}(Z)$ are disjoint). Let $C \in \mathcal{C}$ with $C = (a, b, c)$. We construct 6 trees from C, that is, 3 pairs of trees:

- The "$a < b$" trees: let $(s_1, \dots, s_{n-2})$ be an arbitrary ordering of $S \setminus \{a, b\}$. Then we build the trees $T_C(ab) = (W|a|b|s_1|s_2|\dots|s_{n-2}|Z)$ and $\overleftarrow{T}_C(ab) = (W|s_{n-2}|s_{n-3}|\dots|s_1|a|b|Z)$.
- The "$b < c$" trees: let $(\hat{s}_1, \dots, \hat{s}_{n-2})$ be an arbitrary ordering of $S \setminus \{b, c\}$. Then we build the trees $T_C(bc) = (W|b|c|\hat{s}_1|\hat{s}_2|\dots|\hat{s}_{n-2}|Z)$ and $\overleftarrow{T}_C(bc) = (W|\hat{s}_{n-2}|\hat{s}_{n-3}|\dots|\hat{s}_1|b|c|Z)$.
- The "$c < a$" trees: let $(\bar{s}_1, \dots, \bar{s}_{n-2})$ be an arbitrary ordering of $S \setminus \{c, a\}$. Then we build the trees $T_C(ca) = (W|c|a|\bar{s}_1|\bar{s}_2|\dots|\bar{s}_{n-2}|Z)$ and $\overleftarrow{T}_C(ca) = (W|\bar{s}_{n-2}|\bar{s}_{n-3}|\dots|\bar{s}_1|c|a|Z)$.

Denote by $\mathcal{T}(C)$ the set of 6 constructed trees for $C \in \mathcal{C}$. In this section, the input for our Weighted Quartet Consensus instance constructed from S and $\mathcal{C}$ is $\mathcal{T} = \bigcup_{C \in \mathcal{C}} \mathcal{T}(C)$. Note that $|\mathcal{T}| = 6m$.

Observe that each tree of $\mathcal{T}(C)$ is a (W, Z)-augmented caterpillar. Moreover, note that the majority of ordered pairs are "balanced" in the pairs of constructed trees: Let $a, b \in S$ and $x, y \in S \setminus \{a, b\}$, and let $\{T_C(ab), \overleftarrow{T}_C(ab)\}$ be an "$a < b$" tree-pair. Then we have $x < y$ in $T_C(ab)$ if and only if $y < x$ in $\overleftarrow{T}_C(ab)$. Similarly for any $x \in S \setminus \{a, b\}$, $a < x, b < x$ in $T_C(ab)$ but $x < a, x < b$ in $\overleftarrow{T}_C(ab)$. Only $a < b$ holds in both trees.

Let $T \in \mathcal{T}$, and let $B(T)$ denote the set of quartets of T that have at least two members of $\mathcal{L}(W)$, or at least two members of $\mathcal{L}(Z)$. Thus $B(T)$ consists in all the quartets of the form $w_1w_2|xy$ and $z_1z_2|xy$ of T, where $w_1, w_2 \in \mathcal{L}(W), z_1, z_2 \in \mathcal{L}(Z)$ and $x, y \in \mathcal{X}$ (note that no quartet of $B(T)$ has the form $w_1x|yw_2$ for $x, y \notin \mathcal{L}(W)$, nor the form $z_1x|yz_2$ for $x, y \notin \mathcal{L}(Z)$). Note that for any tree $T' \in \mathcal{T}$, $B(T) = B(T')$. Let $K := 6m|B(T)|$ be the total number of such quartets in $\mathcal{T}$, i.e. K is the size of $\biguplus_{T \in \mathcal{T}} B(T)$. We observe the following:

▶ Remark. Any (W, Z)-augmented caterpillar on $\mathcal{X}$ contains the K quartets $\biguplus_{T \in \mathcal{T}} B(T)$.

Now, denote $\hat{O} := 3m|W||Z| \left(\binom{n-2}{2} + 2(n-2)\right)$. Let $T \in \mathcal{T}$ and suppose that T is an "$a < b$" tree, for some $a, b \in S$. For $w \in \mathcal{L}(W)$ and $z \in \mathcal{L}(Z)$, $x, y \in S$, a quartet $wx|yz$ displayed by T is called an *out-quartet* of T if $\{x, y\} \neq \{a, b\}$, and an *in-quartet* of T if $x = a$ and $y = b$ (note that $x = b$ and $y = a$ is not possible, by construction). Let $out(T)$ and $in(T)$ denote the set of out-quartets and in-quartets, respectively, of T. Note that each tree T has $|W||Z|$ in-quartets and $|W||Z| \left(\binom{n-2}{2} + 2(n-2)\right)$ out-quartets (because there are $\binom{n-2}{2} + 2(n-2)$ ways to choose $\{x, y\} \neq \{a, b\}$). Thus $\hat{O}$ is half the total number of out-quartets.

▶ **Lemma 2.** *Any weighted quartet consensus tree M for $\mathcal{T}$ contains at most $\hat{O}$ quartets from $\biguplus_{T \in \mathcal{T}} out(T)$. Moreover, if M is a (W, Z)-augmented caterpillar $(W|s_1|\dots|s_n|Z)$, where $S = \{s_1, \dots, s_n\}$, then M contains exactly $\hat{O}$ quartets from $\biguplus_{T \in \mathcal{T}} out(T)$.*

Proof. Let $w \in \mathcal{L}(W)$ and $z \in \mathcal{L}(Z)$. Let $\{T_C(ab), \overleftarrow{T}_C(ab)\}$ be an "$a < b$" tree-pair of $\mathcal{T}$, for some $a, b \in S$, and let $x, y \in S$ such that $\{x, y\} \neq \{a, b\}$. Because $x < y$ in $T_C(ab)$ if and only if $y < x$ in $\overleftarrow{T}_C(ab)$, we get that the out-quartet $wx|yz$ is in $T_C(ab)$ if and only if $wy|xz$ is in $\overleftarrow{T}_C(ab)$. Since M can only contain one of the two quartets, it follows that M can contain at most half of the quartets from $out(T_C(ab)) \uplus out(\overleftarrow{T}_C(ab))$. Thus M contains at most half the quartets from $\biguplus_{T \in \mathcal{T}} out(T)$, which is $3m|W||Z| \left(\binom{n-2}{2} + 2(n-2)\right) = \hat{O}$. As for the second assertion, if $M = (W|s_1|\dots|s_n|Z)$ then M contains one of $wx|yz$ or $wy|xz$ for each $x, y \in S$. Thus if M does not contain the out-quartet $wx|yz$ from $T_C(ab)$, then it contains the out-quartet $wy|xz$ from $\overleftarrow{T}_C(ab)$. We deduce that M contains at least half the quartets from $out(T_C(ab)) \uplus out(\overleftarrow{T}_C(ab))$, and thus half the quartets from $\biguplus_{T \in \mathcal{T}} out(T)$. ◀

What follows is a key Lemma. The proof is not so straightforward and can be found in Appendix B.1.

▶ **Lemma 3.** *Any optimal consensus tree for $\mathcal{T}$ is a (W, Z)-augmented caterpillar.*

We finally arrive at our main result.

▶ **Theorem 4.** *The* Weighted Quartet Consensus *problem is NP-hard.*

Proof. We show that there exists a linear ordering of S satisfying $\mathcal{C}$ if and only if there exists a weighted quartet consensus tree M for $\mathcal{T}$ that contains at least $K + \hat{O} + 4m|W||Z|$ quartets from $\biguplus_{T \in \mathcal{T}} Q(T)$. For the rest of the proof, we let $w \in \mathcal{L}(W)$ and $z \in \mathcal{L}(Z)$ be arbitrary leaves of W and Z, respectively.

($\Rightarrow$): let $L = (s_1, s_2, \ldots, s_n)$ be a linear ordering of S satisfying $\mathcal{C}$. Then we claim that the weighted quartet consensus tree $M = (W|s_1|s_2|\ldots|s_n|Z)$ contains the desired number of quartets. Since M is a (W, Z)-augmented caterpillar, M contains K quartets of $\mathcal{T}$ that have two or more elements from $\mathcal{L}(W)$, or two or more elements from $\mathcal{L}(Z)$, see remark on page 5. Moreover by Lemma 2, M contains $\hat{O}$ quartets from $\biguplus_{T \in \mathcal{T}} out(T)$. As for the in-quartets, let $(a, b, c) \in \mathcal{C}$ and let $\mathcal{T}((a, b, c))$ be the set of 6 trees corresponding to (a, b, c). By Lemma 1, L satisfies two of the relations $a < b, b < c, c < a$. This implies that M has exactly two of the following quartets: $wa|bz, wb|cz, wc|az$. Since, for every $w \in \mathcal{L}(W)$ and $z \in \mathcal{L}(Z)$, each of these three quartets appears as an in-quartet in exactly two trees of $\mathcal{T}((a, b, c))$ (e.g. $wa|bz$ is an in-quartet of $T_{(a,b,c)}(ab)$ and $\overleftarrow{T}_{(a,b,c)}(ab)$), it follows that M contains $4|W||Z|$ quartets of $\biguplus_{T \in \mathcal{T}((a,b,c))} in(T)$. As this holds for every $(a, b, c) \in C$, M contains $4m|W||Z|$ quartets of $\biguplus_{T \in \mathcal{T}} in(T)$. Summing up, we get that M has at least $K + \hat{O} + 4m|W||Z|$ quartets from $\mathcal{T}$.

($\Leftarrow$): suppose that no linear ordering of S satisfies $\mathcal{C}$. Let M be an optimal consensus tree for $\mathcal{T}$. By Lemma 3, we may assume that M is a (W, Z)-augmented caterpillar. We bound the number of quartets of $\mathcal{T}$ that can be contained in M.

First, by Lemma 3, M contains K quartets of $\mathcal{T}$ that have at least two elements of $\mathcal{L}(W)$ or at least two elements of $\mathcal{L}(Z)$. As for the quartets with one or zero elements from $\mathcal{L}(W) \cup \mathcal{L}(Z)$, in any tree $T \in \mathcal{T}$ there are at most $(|W| + |Z|)n^3$ quartets of the form $wa|bc$ or $za|bc$ with $a, b, c \in S$, and at most n^4 quartets of the form $ab|cd$ with $a, b, c, d \in S$. Thus M contains at most $6m((|W| + |Z|)n^3 + n^4) < (|W| + |Z|)mn^5$ quartets of $\mathcal{T}$ that are of the form $wa|bc, za|bc$ or $ab|cd$ with $a, b, c \in S$ (the inequality holds because $n \geq 3$ and $|W| = |Z| = (nm)^{100}$). Also, by Lemma 2, M contains at most $\hat{O}$ quartets from $\biguplus_{T \in \mathcal{T}} out(T)$. It remains to count the in-quartets.

Let $(a, b, c) \in \mathcal{C}$. The following in-quartets appear, each twice, in $\mathcal{T}((a, b, c))$: $wa|bz$, $wb|cz$, $wc|az$. It is easy to check that these three quartets are incompatible, i.e. no tree can contain all three of them, and hence M can have at most two of them. We deduce that there must be at least two trees $T, \overleftarrow{T}$ of $\mathcal{T}((a, b, c))$ such that M contains no quartet from $in(T) \uplus in(\overleftarrow{T})$. Therefore M contains at most $4|W||Z|$ quartets from $\biguplus_{T \in \mathcal{T}((a,b,c))} in(T)$, and thus at most $4m|W||Z|$ quartets from $\biguplus_{T \in \mathcal{T}} in(T)$ assuming that the $4|W||Z|$ bound is attained for every $(a, b, c) \in \mathcal{C}$. We will however show that there must be some $(a, b, c) \in \mathcal{C}$ such that M contains only $2|W||Z|$ of the quartets in $\biguplus_{T \in \mathcal{T}((a,b,c))} in(T)$.

Now, since M is a (W, Z)-augmented caterpillar, we write $M = (W|T_1|T_2|\ldots|T_k|Z)$. For some $a \in S$, let $T(a)$ be the tree of $\{T_1, \ldots, T_k\}$ that contains a as a leaf. Then a quartet $wa|bz$ is in $Q(M)$ if and only if $T(a) < T(b)$. Let L be a linear ordering of S such that $T(a) < T(b) \Rightarrow a < b$ in L. Since no linear ordering of S can satisfy $\mathcal{C}$, by Lemma 1 there must be some $(a, b, c) \in \mathcal{C}$ such that only one of $a < b, b < c, c < a$ holds in L. This also means that at most one of $T(a) < T(b), T(b) < T(c), T(c) < T(a)$ holds (because $\neg(a < b) \Rightarrow \neg(T(a) < T(b))$). Thus M has at most one of the $wa|bz, wb|cz, wc|az$ quartets. It follows that there are at least $2|W||Z|$ quartets from $\biguplus_{T \in \mathcal{T}((a,b,c))} in(T)$ that M does not contain. Therefore M contains at most $4m|W||Z| - 2|W||Z|$ quartets of $\biguplus_{T \in \mathcal{T}} in(T)$.

In total, the number of quartets that M contains from the input is bounded by $K + \hat{O} + (|W| + |Z|)mn^5 + (4m - 2)|W||Z| < K + \hat{O} + 4m|W||Z|$, by our choice of $|W|$ and $|Z|$. ◀

The implications of these results for the Weighted Triplet Consensus (WTC) problem are presented in Appendix A. The same techniques can be used to show that WTC is NP-hard.

4 The (non)-structure of WQC

In the rest of this paper, we aim at designing algorithms building on the fact that the weight of each quartet is not arbitrary, and is rather based on a set of input trees on the same leaf set. When designing optimized algorithms for a problem, understanding the relationship between the input and the optimal solution(s) can be of great help. In phylogenetics, several problems are harder in the supertree setting, i.e. when the input trees do not all contain the same species, than in the consensus setting as in the WQC problem. An example is the problem of finding an unrooted phylogenetic tree containing as minors a set of unrooted phylogenetic trees – the compatibility problem – which is NP-hard in the supertree setting [24] and polynomially solvable in the consensus setting [1]. Despite the NP-hardness of WQC, there may exist some properties inherent to the consensus setting that are useful for devising efficient FPT algorithm, or for establishing lower bounds on the value of an optimal solution in order to develop approximation algorithms.

In attempt to establish useful properties of the weights of quartets in the consensus setting, we initially conjectured that the following relationships between the input trees and the solution(s) hold. Despite being seemingly reasonable, we prove all these conjectures false.

1. let D be the set of strictly dominant quartets of the input multiset $\mathcal{Q}$, i.e. the quartets $ab|cd$ such that $f(ab|cd) > f(ac|bd)$ and $f(ab|cd) > f(ad|bc)$. Then there is a constant $\alpha > 0$ such that there exists an optimal solution containing at least $\alpha|D|$ quartets of D;
2. if a quartet $ab|cd$ has a higher weight than the sum of the other quartets on the same quadset, i.e. $f(ab|cd) > f(ac|bd) + f(ad|bc)$, then some optimal solution contains $ab|cd$;
3. more generally, there exists $\beta > 0$ such that if a quartet $ab|cd$ is in a fraction β of the input trees, then $ab|cd$ must be in some optimal solution. In particular, if $ab|cd$ is in *every* input tree, then there is some optimal solution that contains $ab|cd$;
4. if a quartet $ab|cd$ is in no input tree, then no optimal solution contains $ab|cd$.
5. call $ab|cd$ a *strictly least-frequent quartet* if $f(ab|cd) < f(ac|bd)$ and $f(ab|cd) < f(ad|bc)$. Suppose that there exists a tree T^* on leaf set $\mathcal{X}$ that contains no strictly least-frequent quartet, and choose such a T^* that contains a maximum number of quartets from the input. Then T^* is an optimal solution for WQC.

Unfortunately, we answered negatively to all conjectures, see Appendix B.2.

5 Approximability of WQC

In this section, we show that WQC admits a factor 1/2 approximation algorithm that runs in polynomial time. Hereafter, the input set of trees is $\mathcal{T} = \{T_1, \ldots, T_k\}$ and we denote

$\mathcal{Q} = Q(T_1) \uplus \ldots \uplus Q(T_k)$. We say that a minimization (resp. maximization) problem P can be approximated within a factor $\alpha > 1$ (resp. $\beta < 1$) if there is an algorithm that, for every instance I of P, runs in polynomial time and outputs a solution of value $APP(I)$ such that $APP(I) \leq \alpha OPT(I)$ (resp. $APP(I) \geq \beta OPT(I)$), where $OPT(I)$ is the optimal value of I.

As mentioned before, the Complete Maximum Quartet Compatibility (CMQC) problem admits a PTAS, though it can only be applied to the WQC problem when the number of input trees is constant. There does not seem to exist an easy extension of the PTAS algorithm for the case of an unbounded number of trees, which makes WQC seem "harder" than CMQC. Nevertheless, we give a simple factor 1/2 approximation algorithm, which is better than the (randomized) factor 1/3 approximation, the best known so far, for the general Maximum Quartet Consistency problem in which the given quartet set is not necessarily complete. We borrow ideas from [9] to show that this can be achieved by taking the best solution from either a 1/3 approximation to WQC, or a factor 2 approximation to WMQI, the minimization version of WQC (see below). For two unrooted binary trees T_1, T_2 on leaf set $\mathcal{X}$, denote $d_Q(T_1, T_2) = |Q(T_1) \setminus Q(T_2)|$. The WMQI problem is defined as follows:

Weighted Minimum Quartet Inconsistency (WMQI) problem
Input: a set of unrooted trees $\mathcal{T} = \{T_1, \ldots, T_k\}$ such that $\mathcal{L}(T_1) = \ldots = \mathcal{L}(T_k) = \mathcal{X}$.
Output: a tree M with $\mathcal{L}(M) = \mathcal{X}$ that minimizes $\sum_{T \in \mathcal{T}} d_Q(M, T)$.

Note that the WMQI problem is equivalent to finding a minimum (in the multiset sense) number of quartets to discard from $\mathcal{Q}$ so that it is compatible.

It is not hard to show that d_Q is a metric. In particular, d_Q satisfies the triangle inequality, i.e. for any 3 trees T_1, T_2, T_3 on the same leaf set, $d_Q(T_1, T_3) \leq d_Q(T_1, T_2) + d_Q(T_2, T_3)$. This leads to a factor 2 approximation algorithm for WMQI obtained by simply returning the best tree from the input. Intuitively, the input tree that is the closest to the others cannot be too far from the best solution, which is a median tree in the metric space. See [2] for details.

▶ **Theorem 5** ([2]). *The following is a factor 2 approximation algorithm for WMQI: output the tree $T \in \mathcal{T}$ that minimizes $\sum_{T_i \in \mathcal{T}} d_Q(T, T_i)$.*

In [2], the authors explain how to compute $d_Q(T_1, T_2)$ in time $O(n^2)$. Therefore the factor 2 approximation can be implemented to run in time $O(k^2 n^2)$, by simply computing d_Q between every pair of trees.

Theorem 5 has a direct implication on the approximation guarantees of the ASTRAL algorithm in [20], an implementation of the work from Bryant and Steel [8]. This algorithm finds, in polynomial time, an optimal solution M for a restricted version of WMQI where every bipartition of M is also a bipartition in at least one of the input trees. The solution T returned by the algorithm of Theorem 5 above trivially satisfies this condition. Thus, M is at least as good as T, implying the following.

▶ **Corollary 6.** *The ASTRAL algorithm is a factor 2 approximation for WMQI.*

We do not know whether the factor 2 is tight for the ASTRAL algorithm - we conjecture that ASTRAL can actually achieve a better approximation ratio. As shown in the rest of this section, this would have interesting applications for the approximability of WQC.

Indeed, both WQC and WMQI share the same set of optimal solutions, but the two problems are not necessarily identical in terms of approximability. We show however that WMQI can be used to approximate WQC. As stated earlier, there is a trivial factor 1/3 randomized approximation for WQC: output a random tree T. Each quartet of $\mathcal{Q}$ has a 1/3 chance of being contained by T, and so the expected number of quartets of $\mathcal{Q}$ contained by

T is $|\mathcal{Q}|/3 = k\binom{n}{4}/3$ (here $|\mathcal{Q}|$ denotes the multiset cardinality). Call this the "random-tree-algorithm". For the sake of having a *deterministic* algorithm, we show the following:

▶ **Lemma 7.** *The "random-tree-algorithm" can be derandomized, i.e. there is a deterministic algorithm that, in time $O(kn^4 + n^5)$, finds a tree containing at least $|\mathcal{Q}|/3$ quartets from $\mathcal{Q}$.*

Proof. We derandomize the factor 1/3 algorithm using the standard method of conditional expectation. For the simplicity of exposition, we will construct a rooted tree T in a top-down manner (T can be unrooted after the construction). Call a rooted tree T *internally binary* if the only nodes of T that have more than two children have only leaves as children. We start with a fully unresolved internally binary tree T on leaf set $\mathcal{X}$ (i.e. T consists of a root whose n children are in bijection with $\mathcal{X}$). We then iteratively split each unresolved node v of T into two subtrees so as to maximize the expected number of quartets that T contains. We stop when T is a binary tree.

To describe the algorithm more precisely, suppose that T is an internally binary tree on leaf set $\mathcal{X}$, and let v be a node of T with more than 2 children, say $\{v_1, \ldots, v_m\} \subseteq \mathcal{X}$ (if no such v exists, then T is binary and we can stop). We split v by first removing $\{v_1, \ldots, v_m\}$ from T, adding two children x and y to v, and reinserting $v_1, \ldots, v_m$ one after another, each as either a child of x or a child of y. We describe how this choice is made. Suppose that for $i \geq 1$, $v_1, \ldots, v_{i-1}$ have been reinserted, resulting in the tree T_{i-1}, and that we need to process v_i. Denote by $T_{i,x}$ (resp. $T_{i,y}$) the tree obtained by inserting v_i as a child of x (resp. of y) in T_{i-1}. We then define a random binary tree $T'_{i,x}$ from $T_{i,x}$ as follows: for each $v' \in \{v_{i+1}, \ldots, v_m\}$, reinsert v' as a child of x with probability 1/2, or as a child of y with probability 1/2. Then, replace each non-binary node w with children $\mathcal{X}'$ by a rooted binary tree on leaf set $\mathcal{X}'$ chosen uniformly at random. We define the random binary tree $T'_{i,y}$ from $T_{i,y}$ using the same process.

For a random tree T' obtained by the above process and for $q \in \mathcal{Q}$, let $I(q, T')$ be an indicator variable for whether $q \in Q(T')$. That is, $I(q, T') = 1$ if $q \in Q(T')$, and $I(q, T') = 0$ otherwise. Let $I(T') = \sum_{q \in \mathcal{Q}} I(q, T') f_{\mathcal{Q}}(q)$ [2]. We seek

$$\max_{T' \in \{T'_{i,x}, T'_{i,y}\}} \mathbb{E}\left[I(T')\right] = \max_{T' \in \{T'_{i,x}, T'_{i,y}\}} \mathbb{E}\left[\sum_{q \in \mathcal{Q}} I(q, T') f_{\mathcal{Q}}(q)\right]$$
$$= \max_{T' \in \{T'_{i,x}, T'_{i,y}\}} \sum_{q \in \mathcal{Q}} \Pr\left[q \in Q(T')\right] f_{\mathcal{Q}}(q)\,.$$

If $T'_{i,x}$ attains this maximum, we insert v_i below x, and otherwise we insert v_i below y. After every child v_i of v has been inserted, we process the next non-binary node. This concludes the algorithm description (we shall detail how to compute $\Pr[q \in Q(T')]$ below).

If T is an internally binary tree, by a slight abuse of notation define $\mathbb{E}\left[I(T)\right] = \mathbb{E}\left[I(T')\right]$, where T' is the random binary tree obtained by replacing each non-binary node of T on leaf set $\mathcal{X}'$ by a random binary tree on leaf set $\mathcal{X}'$.

▶ **Claim 1.** *Let T be an internally binary tree, and suppose that $\mathbb{E}[I(T)] \geq |\mathcal{Q}|/3$. Let v be a non-binary node of T, and let T_v be the tree obtained after splitting v using the above algorithm. Then $\mathbb{E}[I(T_v)] \geq |\mathcal{Q}|/3$.*

[2] Observe that here, $q \in \mathcal{Q}$ means that there exists at least one occurrence of q in the multiset $\mathcal{Q}$, and so each quartet present in $\mathcal{Q}$ is considered once in the summation, independently of $f_{\mathcal{Q}}(q)$.

Let $\{v_1, \ldots, v_m\}$ be the children of v. To prove the claim, we use induction on the number of processed children of v to show that after each insertion of a child v_i, the obtained tree $T_i \in \{T_{i,x}, T_{i,y}\}$ satisfies $\mathbb{E}[I(T_i')] \geq |\mathcal{Q}|/3$, where $T_i' \in \{T_{i,x}', T_{i,y}'\}$ is the random tree corresponding to T_i obtained from the above process (i.e. reinserting $v_{i+1}, \ldots, v_m$ randomly under x or y, and resolving non-binary nodes randomly). This proves the statement since $T_m = T_v$ (and thus $\mathbb{E}\left[I(T_v)\right] = \mathbb{E}\left[I(T_m)\right] = \mathbb{E}[I(T_m')] \geq |\mathcal{Q}|/3)$. As a base case, if $i = 1$ it is easy to see that $T_{1,x}'$ and $T_{1,y}'$ are identical, and that $\mathbb{E}[I(T_{1,x}')] = \mathbb{E}[I(T_{1,y}')] = \mathbb{E}[I(T)] \geq |\mathcal{Q}|/3$. For $i > 1$, let T_{i-1} be the tree obtained after inserting v_{i-1}, and suppose without loss of generality that $T_{i-1} = T_{i-1,x}$. Because, in $T_{i-1,x}'$, we insert v_i below x or y each with probability $\frac{1}{2}$, we have

$$\begin{aligned}\mathbb{E}\left[I(T_{i-1,x}')\right] &= \frac{1}{2}\mathbb{E}\left[I(T_{i-1,x}')|v_i \text{ is a child of } x\right] + \frac{1}{2}\mathbb{E}\left[I(T_{i-1,x}')|v_i \text{ is a child of } y\right] \\ &= \frac{1}{2}\left(\mathbb{E}\left[I(T_{i,x}')\right] + \mathbb{E}\left[I(T_{i,y}')\right]\right) .\end{aligned}$$

By induction, we also have $\mathbb{E}[I(T_{i-1,x}')] \geq |\mathcal{Q}|/3$. Combined with the above equality, we obtain $\frac{1}{2}\left(\mathbb{E}\left[I(T_{i,x}')\right] + \mathbb{E}\left[I(T_{i,y}')\right]\right) \geq |\mathcal{Q}|/3$. This implies that one of $\mathbb{E}[I(T_{i,x}')]$ or $\mathbb{E}[I(T_{i,y}')]$ must be at least $|\mathcal{Q}|/3$. ◀

Since the fully unresolved tree T from which we start satisfies $\mathbb{E}[I(T)] \geq |\mathcal{Q}|/3$, Claim 1 shows that the algorithm does terminate with a tree containing at least $|\mathcal{Q}|/3$ quartets from $\mathcal{Q}$. It remains to be show how to compute, when reinserting a node v_i, the expectations for $T_{i,x}'$ and $T_{i,y}'$.

In fact, it suffices to be able to compute, for a given quartet $q = ab|cd$, the probability $\Pr[q \in Q(T')]$ for $T' \in \{T_{i,x}', T_{i,y}'\}$. Moreover, if $\Pr[q \in Q(T_{i,x}')] = \Pr[q \in Q(T_{i,y}')]$, then this probability does not contribute to determining which scenario maximizes expectation, and in this case we do not need to consider q. In particular, if none of a, b, c, d is equal to v_i, then $\Pr[q \in Q(T_{i,x}')] = \Pr[q \in Q(T_{i,y}')]$. Therefore, it is enough to consider only quartets in which v_i is included. We will assume that $v_i = a$. Moreover, we may assume that two or three of $\{b, c, d\}$ are children of v in T (recall that v is the parent of v_i in T), because otherwise the probability that $ab|cd$ is in T' is unaffected by whether a is a child of x or a child of y.

There are still multiple cases depending on which of b, c and d are children of v, and which have been reinserted or have not, but this probability can be easily found algorithmically. Let $U = \{b, c, d\} \cap \{v_{i+1}, \ldots, v_m\}$, i.e. the leaves in $\{b, c, d\}$ that have not been reinserted yet in T'. We obtain new trees $S_1', \ldots, S_h'$ by reinserting, in T', the members of U below x or y in every possible way – there are only $2^{|U|} \leq 8$ possibilities, so $h \leq 8$. Then, for $1 \leq j \leq h$ denote by $S_j'|_q$ the tree S_j' restricted to $\{a, b, c, d\}$ (i.e. obtained by removing every leaf not in $\{a, b, c, d\}$, then contracting degree 2 vertices). Note that $S_j'|_q$ may be non-binary. We get $\Pr[q \in Q(T')] = \sum_{j=1}^{h} \frac{1}{h} \Pr[q \in Q(S_j'|_q)]$. This is because every leaf in $v_{i+1}, \ldots, v_m$ other than b, c, d is reinserted independently from the choice for b, c, d, and every non-binary node remaining after the reinsertions is resolved uniformly. The probability $\Pr[q \in Q(S_j'|_q)]$ is straightforward to compute, as only a constant number of cases can occur since $S_j'|_q$ has only 4 leaves. We omit the details.

Time complexity: we must first preprocess the input in order to compute $f_{\mathcal{Q}}(q)$ for each quartet q. This takes time $O(kn^4)$. As for the computation of $\Pr[q \in Q(T')]$, assume that the lowest common ancestor (lca) of two leaves can be found in constant time. This can be achieved naively by simply storing the lca for each pair of leaves in a table of size $O(n^2)$, and updating the table in time $O(n)$ each time a decision on some v_i is made (this does not hinder the total time complexity of the algorithm, though there are more clever ways to handle dynamic tree *lca* queries [11]). Then the restrictions $S'_1|_q, \ldots, S'_h|_q$ can be computed in constant time. It is then straightforward to see that, by the above process, $\Pr[q \in Q(T')]$ can be computed in constant time. Each time a node v_i needs to be reinserted, this probability must be computed for the $O(n^3)$ quartets containing v_i. There are $n-1$ splits to be performed, and each split requires inserting $O(n)$ nodes. Thus the "binarization" process takes total time $O(n^5)$, and altogether the derandomization takes time $O(kn^4 + n^5)$. ◀

The above leads to a (deterministic) 1/3- approximation. This can be used to show the following. The proof is similar to that of [9, Theorem 2] and is relegated to Appendix B.3.

▶ **Theorem 8.** *If WMQI can be approximated within a factor α, then WQC can be approximated within a factor $\beta = \alpha/(3\alpha - 2)$.*

Combined with Theorem 5 and letting $\alpha = 2$ in Theorem 8 we get the following.

▶ **Corollary 9.** *WQC can be approximated within a factor 1/2 in time $O(k^2n^2 + kn^4 + n^5)$.*

6 Fixed-parameter tractability of WQC

In this section we describe how, based on previous results on the minimum quartet incompatibility problem on complete sets, WQC can be solved in time $O(4^{d'+k'_2+k'_3}n + n^4)$. Here k'_2 and k'_3 are the number of quadsets that have 2 and 3 dominant quartets, respectively, and d' is the number of strictly dominant quartets that we are allowed to reject. The algorithm makes direct use of the Gramm-Niedermeyer algorithm [14], henceforth called the GN algorithm.

The GN algorithm solves the following problem: given a *complete set* of quartets Q, find, if it exists, a complete and compatible set of quartets Q' such that at most d quartets of Q' are not in the input set Q (i.e. $|Q' \setminus Q| \leq d$). This is accomplished by repeatedly applying the following theorem:

▶ **Theorem 10** ([14])**.** *Let Q be a complete set of quartets. Then Q is compatible if and only if for each set of five taxa $\{a, b, c, d, e\} \subseteq \mathcal{X}$, $ab|cd \in Q$ implies $ab|ce \in Q$ or $ae|cd \in Q$.*

The idea behind the GN algorithm is as follows: find a set of five taxa $\{a, b, c, d, e\}$ that does not satisfy the condition of Theorem 10, then correct the situation by branching into the four possible choices:

1. remove $ab|cd$ from Q and add $ac|bd$ to Q;
2. remove $ab|cd$ from Q and add $ad|bc$ to Q;
3. remove $\{ac|be, ae|bc\} \cap Q$ from Q and add $ab|ce$ to Q;
4. remove $\{ac|de, ad|ce\} \cap Q$ from Q and add $ae|cd$ to Q.

The quartets added to Q will not be questioned in the following branchings. With some optimization, this leads to a $O(4^d n + n^4)$ FPT algorithm.

In [14], the authors also note that this algorithm can be extended to sets of quartets Q that contain ambiguous quadsets, i.e. sets $\{a, b, c, d\}$ for which 2 or 3 of the possible quartets on $\{a, b, c, d\}$ are in Q. Suppose there are k_2 and k_3, respectively, quadsets that have 2 and 3 quartets in Q. The modified algorithm then, in a first phase, branches into the $2^{k_2}3^{k_3}$ ways of choosing one quartet per such quadset, thereby obtaining a complete set of quartets for each possibility. The GN algorithm is thus applied to the so-obtained complete sets. This yields a $O(2^{k_2} \cdot 3^{k_3} \cdot 4^d n + n^4)$ algorithm.

It is not hard to see that this gives an FPT algorithm for WQC, where the parameter k_2 (resp. k_3) is the number of quadsets such that 2 (resp. 3) possible quartets appear in the input trees, and d is the number of quartets $ab|cd$ that appear in every input tree, and that we are allowed to discharge. Note however that, in the consensus setting, there is no reason to believe that k_2 and k_3 are low - in we fact we believe that $k_2 + k_3$ typically takes values in $\Theta(n^4)$. One reason is that even the slightest amount of noise on a quadset makes it included in the count of either k_2 or k_3 (e.g. if $k - 1$ trees agree on $ab|cd$ and only one contains $ac|bd$).

The GN algorithm can, however, be used on a more interesting set of parameters. Define k'_2 (resp. k'_3) as the number of quadsets that have exactly 2 (resp. 3) dominant quartets, and let d' be the number of strictly dominant quartets that we are allowed to discharge. It is reasonable to believe that, if each tree of the input is close to the true tree T^*, most "true" quartets will appear as strictly dominant in the input, and there should not be too many ambiguous quadsets. There is a very simple algorithm achieving time $O(4^{d'+k'_2+k'_3}n + n^4)$. Construct a complete set of quartets Q as follows: for each quadset $\{a, b, c, d\}$, choose a dominant quartet on $\{a, b, c, d\}$ and add it to Q (if multiple choices are possible, choose arbitrarily). Then, run the GN algorithm on Q with the following modification: each time a quartet q is removed from Q and replaced by another quartet q', decrement either d', k'_2 or k'_3, depending on whether q belongs to a quadset with $1, 2$ or 3 dominant quartets. It follows that if there exists a complete and compatible set of quartets Q' such that at most d' strictly dominant quartets are rejected, then the modified algorithm will find it. It should be noted however that finding such a set Q' does not guarantee that the corresponding tree is an optimal solution. Indeed, since quartets are weighted, two solutions Q' and Q'' may both reject only d' strictly dominant quartets, yet one has higher weight than the other. However, the correctness of the algorithm follows from the fact that the GN algorithm finds the set of *every* solution discarding at most d' dominant quartets - and thus it suffices to pick the solution from this set that has optimal weight.

We finally mention that the FPT algorithms published in [10] are improved versions of the GN algorithm, can also return every solution and thus can be modified in the same manner. These yield FPT algorithms that can solve WQC in time $O(3.0446^{d'+k'_2+k'_3}n + n^4)$ and $O(2.0162^{d'+k'_2+k'_3}n^3 + n^5)$.

7 Conclusion

In this paper, we have shown that the WQC problem is NP-hard, answering a question of [19] and [2]. In the latter, the authors also propose a variant of the problem in which the output tree T is not required to be binary. In this case, one needs to assign a cost p to the unresolved quartets. Our reduction can be extended to show that hardness holds for high enough p, but the complexity of the general case remains open. We have also shown that WQC can be approximated within a factor 1/2. One open question is whether the problem admits a PTAS as the related CMQC problem. The fixed-parameter tractability aspects of WQC also deserve further investigation. This would require identifying some structural properties that are present in the consensus setting and that can be used for designing practical exact algorithms. But as we have shown, this might not be an easy task, as many properties which seem reasonable for the consensus setting do not hold.

References

1 Alfred V. Aho, Yehoshua Sagiv, Thomas G. Szymanski, and Jeffrey D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM J. Comput.*, 10(3):405–421, 1981. doi:10.1137/0210030.

2 Mukul S. Bansal, Jianrong Dong, and David Fernández-Baca. Comparing and aggregating partially resolved trees. *Theor. Comput. Sci.*, 412(48):6634–6652, 2011. doi:10.1016/j.tcs.2011.08.027.

3 Jean-Pierre Barthélemy and Fred R. McMorris. The median procedure for n-trees. *J. Classif.*, 3(2):329–334, 1986. doi:10.1007/BF01894194.

4 Vincent Berry, David Bryant, Tao Jiang, Paul Kearney, Ming Li, Todd Wareham, and Haoyong Zhang. A practical algorithm for recovering the best supported edges of an evolutionary tree (extended abstract). In David B. Shmoys, editor, *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 287–296. ACM/SIAM, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338265.

5 Vincent Berry, Tao Jiang, Paul Kearney, Ming Li, and Todd Wareham. Quartet cleaning: Improved algorithms and simulations. In Jaroslav Nešetřil, editor, *Proceedings of the 7th Annual European Symposium on Algorithms (ESA 1999)*, volume 1643 of *LNCS*, pages 313–324. Springer Berlin Heidelberg, 1999. doi:10.1007/3-540-48481-7_28.

6 David Bryant. *Building trees, hunting for trees, and comparing trees.* PhD thesis, University of Canterbury, New Zealand, 1997. URL: http://hdl.handle.net/10092/7914.

7 David Bryant. A classification of consensus methods for phylogenetics. In Melvin F. Janowitz, François-Joseph Lapointe, Fred R. McMorris, Boris Mirkin, and Fred S. Roberts, editors, *Proceedings of DIMACS Working Group Meetings on Bioconsensus*, volume 61 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 163–184. Americal Mathematical Society, 2003. doi:10.1090/dimacs/061/11.

8 David Bryant and Mike Steel. Constructing optimal trees from quartets. *J. Algorithms*, 38(1):237–259, 2001. doi:10.1006/jagm.2000.1133.

9 Jaroslaw Byrka, Sylvain Guillemot, and Jesper Jansson. New results on optimizing rooted triplets consistency. *Discrete Appl. Math.*, 158(11):1136–1147, 2010. doi:10.1016/j.dam.2010.03.004.

10 Maw-Shang Chang, Chuang-Chieh Lin, and Peter Rossmanith. New fixed-parameter algorithms for the minimum quartet inconsistency problem. *Theory Comput. Syst.*, 47(2):342–367, 2010. doi:10.1007/s00224-009-9165-y.

11 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi:10.1137/S0097539700370539.

12 Joseph Felsenstein. *Inferring phylogenies.* Sinauer Associates Sunderland, 2004.

13 Zvi Galil and Nimrod Megiddo. Cyclic ordering is NP-complete. *Theor. Comput. Sci.*, 5(2):179–182, 1977. doi:10.1016/0304-3975(77)90005-6.

14 Jens Gramm and Rolf Niedermeier. Minimum quartet inconsistency is fixed parameter tractable. In Amihood Amir, editor, *Proceedings of the 12th Annual Symposium of Combinatorial Pattern Matching (CPM 2001)*, volume 2089 of *LNCS*, pages 241–256. Springer Berlin Heidelberg, 2001. doi:10.1007/3-540-48194-X_23.

15 Jesper Jansson. On the complexity of inferring rooted evolutionary trees. *Electron. Notes Discrete Math.*, 7:50–53, 2001. doi:10.1016/S1571-0653(04)00222-7.

16 Tao Jiang, Paul Kearney, and Ming Li. A polynomial time approximation scheme for inferring evolutionary trees from quartet topologies and its application. *SIAM J. Comput.*, 30(6):1942–1961, 2001. doi:10.1137/S0097539799361683.

17 Timothy Margush and Fred R. McMorris. Consensus n-trees. *Bull. Math. Biol.*, 43(2):239–244, 1981. doi:10.1007/BF02459446.

18 Fred R. McMorris, David B. Meronk, and Dean A. Neumann. A view of some consensus methods for trees. In Joseph Felsenstein, editor, *Proceedings of the NATO Advanced Study Institute on Numerical Taxonomy*, volume 1 of *NATO Advanced Science Institutes Series, Series G: Ecological Sciences*, pages 122–126. Springer, 1983. doi:10.1007/978-3-642-69024-2_18.

19 Siavash Mirarab. *Novel scalable approaches for multiple sequence alignment and phylogenomic reconstruction*. PhD thesis, University of Texas at Austin, 2015. URL: `http://hdl.handle.net/2152/31377`.

20 Siavash Mirarab, Rezwana Reaz, Md. Shamsuzzoha Bayzid, Théo Zimmermann, M. Shel Swenson, and Tandy Warnow. ASTRAL: genome-scale coalescent-based species tree estimation. *Bioinformatics*, 30(17):i541–i548, 2014. `doi:10.1093/bioinformatics/btu462`.

21 António Morgado and Joao Marques-Silva. A pseudo-boolean solution to the maximum quartet consistency problem, 2008. `arXiv:0805.0202`.

22 António Morgado and Joao Marques-Silva. Combinatorial optimization solutions for the maximum quartet consistency problem. *Fundam. Inform.*, 102(3-4):363–389, 2010. `doi:10.3233/FI-2010-311`.

23 Robert R. Sokal and F. James Rohlf. Taxonomic congruence in the leptopodomorpha re-examined. *Syst. Zool.*, 30(3):309–325, 1981. `doi:10.2307/2413252`.

24 Michael Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *J. Classif.*, 9(1):91–116, 1992. `doi:10.1007/BF02618470`.

25 Gang Wu, Jia-Huai You, and Guohui Lin. A lookahead branch-and-bound algorithm for the maximum quartet consistency problem. In Rita Casadio and Gene Myers, editors, *Proceedings of the 5th International Workshop on Algorithms in Bioinformatics (WABI 2005)*, volume 3692 of *LNCS*, pages 65–76. Springer, Springer, 2005. `doi:10.1007/11557067_6`.

26 Gang Wu, Jia-Huai You, and Guohui Lin. Quartet-based phylogeny reconstruction with answer set programming. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 4(1):139–152, 2007. `doi:10.1109/TCBB.2007.1008`.

A Implications for the Weighted Triplet Consensus problem

For each set of three labels $\{a, b, c\}$, there are three non-isomorphic[3] rooted binary trees called *triplets*. They are denoted by $ab|c$, $ac|b$ and $bc|a$, depending on the leaf having the root as father (c, b and a respectively). We say that a tree T induces or displays the triplet $ab|c$ if $T|\{a, b, c\} = ab|c$. For a rooted tree R, denote by $tr(R)$ the set of triplets of R.

When the consensus is sought for rooted trees, the objective is to find a rooted tree M that induces a maximum number of triplets contained in the input trees. The Weighted Triplet Consensus (WTC) is defined as follows.

Weighted Triplet Consensus (WTC) problem
Input: a set of rooted trees $\mathcal{R} = \{R_1, \ldots, R_k\}$ such that $\mathcal{L}(R_1) = \ldots = \mathcal{L}(R_k) = \mathcal{X}$.
Output: a binary rooted tree M with $\mathcal{L}(M) = \mathcal{X}$ that maximizes $\sum_{R \in \mathcal{R}} |tr(M) \cap tr(R)|$.

As in the unrooted problem, other versions of WTC where the input trees may have missing species or where the weight of a triplet is not defined w.r.t. a set of trees, are known to be NP-hard [6]. The WTC problem is conjectured to be NP-hard in [2] (we note that a more general version where the output can be non-binary is also conjectured NP-hard).

We give the main idea behind the proof of the hardness of WTC. Let $\mathcal{T} = \bigcup_{C \in \mathcal{C}} \mathcal{T}(C)$ be the set of unrooted trees constructed in the reduction above. For a tree $T \in \mathcal{T}$, let e be the edge separating Z from the rest of the tree (i.e. by removing e from T, one connected component is exactly Z). Obtain a rooted tree R from T by rooting T at e, that is subdivide e, thereby creating a degree 2 vertex which is the root of R. The set of rooted trees $\mathcal{R}$ is obtained by applying this rooting to every $T \in \mathcal{T}$ (the Z subtree could be removed but we keep it here to make the correspondence easier to see).

[3] Isomorphism preserving labels and the root node.

Similarly as above, it can be shown that since every input tree is a rooted (W, Z)-caterpillar, then any solution must also have this form. This implies in turn that there exists a linear ordering of S satisfying $\mathcal{C}$ if and only if there is a solution M to WTC containing every triplet from the input on 2 or 3 members of $\mathcal{L}(W)$, every triplet containing at least one member of $\mathcal{L}(Z)$, plus at least $4m|W| + 3m|W|\left(\binom{n-2}{2} + 2(n-2)\right)$ triplets of the form $wa|b$ with $a, b \in S$. This is obtained by defining the notions of *in-triplets* and *out-triplets* analogously as in the previous section, but with respect to W only. That is, in a "$a < b$" tree, for $a, b, c, d \in S, w \in \mathcal{L}(W)$ and $\{c, d\} \neq \{a, b\}$, $wa|b$ would be an in-triplet, whereas $wc|d$ or $wd|c$ would be out-triplets. One can argue that for a cyclic triple $(a, b, c) \in \mathcal{C}$ and the set of trees $\mathcal{T}((a, b, c))$, an optimal consensus tree can contain $4|W|$ of the $6|W|$ possible in-triplets, plus at most half of the $6m|W|\left(\binom{n-2}{2} + 2(n-2)\right)$ possible out-triplets. The arguments are essentially the same as the ones given in the hardness proof of WQC, and so we omit the details.

▶ **Theorem 11.** *The* Weighted Triplet Consensus *problem is NP-hard.*

B Deferred proofs

B.1 Proof of Lemma 3

Despite the Lemma 3 statement being quite intuitive, it requires a surprising amount of care. We start by a simple proposition that will be needed.

▶ **Proposition 12.** *Let X, Y be two non-empty sets such that $Y \not\subseteq X$. Then $|X| \cdot |Y \setminus X| \geq |Y| - 1$.*

Proof. Suppose first that $X \cap Y = \emptyset$. Then clearly $|X||Y \setminus X| = |X||Y| \geq |Y| - 1$. Suppose otherwise that $X \cap Y \neq \emptyset$, and denote $X' = X \cap Y$. Then $|Y \setminus X| = |Y| - |X'|$ and since $Y \not\subseteq X$, we must have $|Y| \geq |X'| + 1$. We also have $|X||Y \setminus X| = |X|(|Y| - |X'|) \geq |X'|(|Y| - |X'|)$; we claim the latter term to be at least $|Y| - 1$. Let us assume for contradiction that $|X'|(|Y| - |X'|) < |Y| - 1$. If $|X'| = 1$, this is clearly impossible, so assume $|X'| > 1$. Then we get $|X'||Y| - |Y| < |X'|^2 - 1$ leading to $|Y| < \frac{|X'|^2 - 1}{|X'| - 1} = |X'| + 1$, contradicting $|Y| \geq |X'| + 1$. ◀

Before proceeding, we must introduce the notion of a rooted subtree of a binary unrooted tree T. Note that by removing an edge $e = \{u, v\}$ of T, we obtain two disjoint rooted subtrees T_1 and T_2, respectively rooted at u and v. Call T' a *rooted subtree* of T if T' is a rooted tree that can be obtained by removing an edge of T. For $X \subset \mathcal{L}(T)$, a *rooted subtree for* X is a rooted subtree T' of T such that $X \subseteq \mathcal{L}(T')$. We denote by $T[X]$ the rooted subtree for X that contains a minimum number of leaves (if there are multiple choices, choose $T[X]$ arbitrarily among the possible choices). Note that $T[X]$ may contain leaves other than X.

We now prove that any optimal solution to $\mathcal{T}$ as constructed in our reduction must be a (W, Z)-augmented caterpillar. Suppose that M is an optimal solution for $\mathcal{T}$, and that M is not a (W, Z)-augmented caterpillar. Denote $M_W = M[\mathcal{L}(W)]$ and $M_Z = M[\mathcal{L}(Z)]$. If M is a (W', Z')-augmented caterpillar $(W'|T_1| \dots |T_k|Z')$ for some trees W', Z' with $\mathcal{L}(W') = \mathcal{L}(W)$ and $\mathcal{L}(Z') = \mathcal{L}(Z)$, it is not hard to see that $M' = (W|T_1| \dots |T_k|Z)$ is a better solution than M, a contradiction. Thus, M is not such a caterpillar, and this implies that either $\mathcal{L}(M_W) \neq \mathcal{L}(W)$ or $\mathcal{L}(M_Z) \neq \mathcal{L}(Z)$ (or both). That is, the rooted subtrees containing $\mathcal{L}(W)$ and/or $\mathcal{L}(Z)$ have "outsider" leaves. Suppose first that $\mathcal{L}(M_W) \neq \mathcal{L}(W)$ holds. Then there exists a node x with children x_l and x_r in M_W such that all leaves X_l below x_l are in $\mathcal{L}(W)$ with $\mathcal{L}(W) \not\subseteq X_l$ (otherwise $M_W = M[\mathcal{L}(W)]$ would be chosen incorrectly), and no leaf X_r

below x_r belongs to $\mathcal{L}(W)$ (this can be seen by observing that the minimal node x of M_W having leaves both in W and not in W has this property).

We claim that $\mathcal{L}(Z) \nsubseteq X_r$. Suppose otherwise that $\mathcal{L}(Z) \subseteq X_r$. Then $|X_r| \geq |Z|$ and so $|M_W| \geq |W| + |Z|$. However in M, by removing the xx_r edge we obtain two rooted trees, one of which is a rooted subtree for $\mathcal{L}(W)$. Moreover, this subtree has at most $|W| + |S| < |W| + |Z|$ leaves, which contradicts the minimality of $M_W = M[\mathcal{L}(W)]$. We deduce that $\mathcal{L}(Z)$ is not a subset of X_r.

Now, observe that M contains the quartet $w_1 y | w_2 z$ for each $w_1 \in X_l, y \in X_r, w_2 \in \mathcal{L}(W) \setminus X_l, z \in \mathcal{L}(Z) \setminus X_r$. There are at least $|X_l||X_r|(|\mathcal{L}(W) \setminus X_l|)(|\mathcal{L}(Z) \setminus X_r|) \geq (|W|-1)(|Z|-1)$ such quartets (the inequality is obtained by applying Proposition 12 to $|X_l| \cdot |\mathcal{L}(W) \setminus X_l|$ and $|X_r| \cdot |\mathcal{L}(Z) \setminus Z|$). Moreover, each input tree of $\mathcal{T}$ contains the quartet $w_1 w_2 | yz$ instead, and hence in total in $\mathcal{T}$ there are at least $6m(|W|-1)(|Z|-1)$ quartets of the form $w_1 w_2 | yz$ that M does not contain. In the same manner, if the case $\mathcal{L}(M_Z) \neq \mathcal{L}(Z)$ holds, then there are at least $6m(|W|-1)(|Z|-1)$ quartets of the form $z_1 z_2 | yw$ that M does not contain, where here $z_1, z_2 \in \mathcal{L}(Z), y \notin \mathcal{L}(Z), w \in \mathcal{L}(W)$.

Now, let $\rho(M)$ be the number of quartets that M contains from $\biguplus_{T \in \mathcal{T}} Q(T)$ that have the form $wx|yz$, where $w \in \mathcal{L}(W), z \in \mathcal{L}(Z), x, y \in S$. Formally,

$$\rho(M) = \sum_{\substack{wx|yz \in Q(M) \\ x,y \in S \\ w \in \mathcal{L}(W) \\ z \in \mathcal{L}(Z)}} f(wx|yz)$$

where $f(wx|yz)$ denotes the number of trees of $\mathcal{T}$ that contain the $wx|yz$ quartet. For a given $u \in \mathcal{L}(W) \cup \mathcal{L}(Z)$, let $\rho(M, u)$ denote the number of quartets counted in $\rho(M)$ that contain u. Formally, if $w \in \mathcal{L}(W)$, we have

$$\rho(M, w) = \sum_{\substack{wx|yz \in Q(M) \\ x,y \in S \\ z \in \mathcal{L}(Z)}} f(wx|yz) \,.$$

The definition of $\rho(M, z)$ is the same for $z \in \mathcal{L}(Z)$, except that z gets fixed instead of w in the summation. Notice that $\rho(M) = \sum_{w \in \mathcal{L}(W)} \rho(M, w) = \sum_{z \in \mathcal{L}(Z)} \rho(M, z)$. Let $w^* = \arg\max_{w \in \mathcal{L}(W)} \{\rho(M, w)\}$. We obtain an alternative solution M' from M in the following manner: remove all leaves of $\mathcal{L}(W) \setminus \{w^*\}$ from M, delete the degree 2 nodes, and replace w^* by the W tree. Note that if $w^* x | yz$ is a quartet of M, then $wx|yz$ is a quartet of M' for all $w \in \mathcal{L}(W)$, and so $\rho(M', w) \geq \rho(M, w)$ for all such w by the choice of w^*. Consequently, $\rho(M') \geq \rho(M)$. We repeat the same operation on M' for the Z tree and obtain our final tree M^*. That is, we find $z^* = \arg\max_{z \in \mathcal{L}(Z)} \{\rho(M', z)\}$, and replace z^* by the Z tree. As above, we obtain $\rho(M^*) \geq \rho(M')$. Since M^* has W and Z as rooted subtrees, it follows that M^* is a (W, Z)-augmented caterpillar.

We argue that M^* contains more quartets from the input trees than M. First observe that the quartets on which M and M^* differ must contain a member of $\mathcal{L}(W) \cup \mathcal{L}(Z)$, since only these leaves switched position. The tree M^* contains every quartet of $\biguplus_{T \in \mathcal{T}} Q(T)$ that have at least two members of $\mathcal{L}(W)$, or two members of $\mathcal{L}(Z)$. This includes the aforementioned (at least) $6m(|W|-1)(|Z|-1)$ quartets of the form $w_1 w_2 | yz$ or $z_1 z_2 | yw$ that M does not contain. As for the quartets that contain one member of $\mathcal{L}(W)$ and one member of $\mathcal{L}(Z)$, M^* contains at least as many such quartets as M since in $\biguplus_{T \in \mathcal{T}} Q(T)$, these quartets are all of the form $wx|yz$, and we have $\rho(M^*) \geq \rho(M)$. Finally, each tree of $\mathcal{T}$ has at most $(|W| + |Z|)n^3$ quartets that have exactly one member of $\mathcal{L}(W) \cup \mathcal{L}(Z)$. Thus

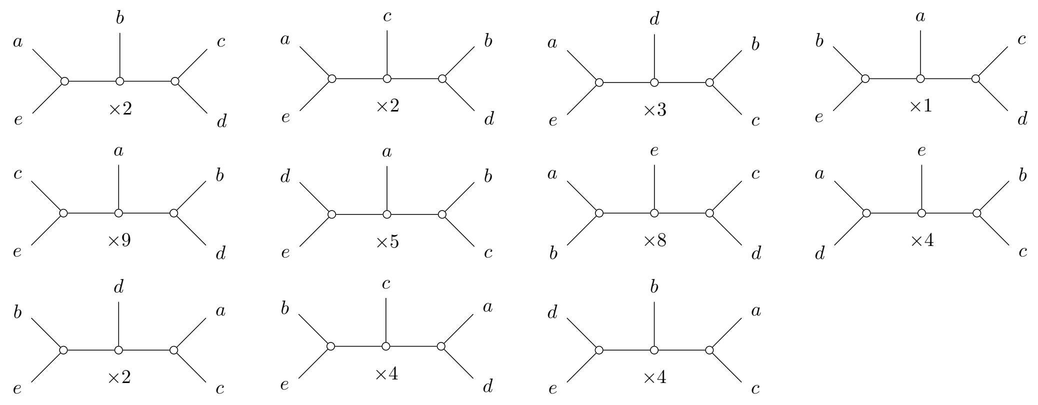


Figure 1 An instance of WQC such that the optimal solution (the third tree on the first row) contains no strictly dominant quartet. The numbers correspond to the number of times that each tree appears in the input.

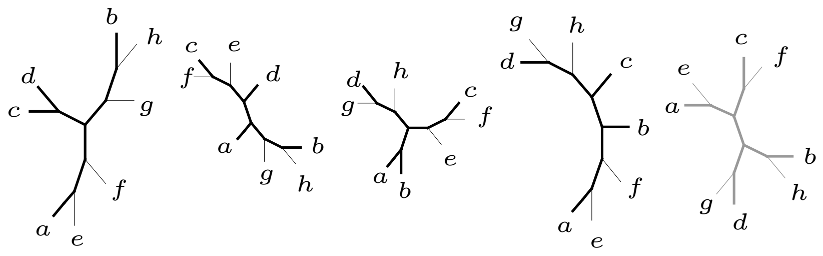


Figure 2 The first four trees form an instance of WQC in which every tree contains $ab|cd$. The rightmost tree is the unique optimal solution to the WQC instance (every possible solution was verified computationally).

at most $6m(|W| + |Z|)n^3$ quartets of this type are contained by M and not contained by M^*, but since this is smaller than $6m(|W| - 1)(|Z| - 1)$ for our choice of $|W|$ and $|Z|$, M^* contains more quartets from the input than M.

B.2 Proofs of Section 4

Conjecture 1 is disproved by Theorem 13, and Conjecture 3 by Theorem 14, which implies that Conjectures 2 and 4 are also false; finally Conjecture 5 is disproved by Theorem 15.

▶ **Theorem 13.** *There exists an instance of WQC such that every optimal solution contains none of the strictly dominant quartets.*

Figure 1 shows an instance of WQC demonstrating Theorem 13. In this instance, for every quadset S, there is a strictly dominant quartet appearing 17 times, whereas the second-most and third-most quartets appear in 16 and 11 trees, respectively. For example, $f(ac|bd) = 17, f(ad|bc) = 16$ and $f(ab|cd) = 11$. One can check that the best tree is the third one on the top row (the $ae|bc$ with d grafted on the middle branch). Call this tree T^*. For every quadset S, T^* contains the second-most frequent quartet on S. The reason why T^* is optimal is that, in the particular instance of Figure 1, any other tree T that contains a strictly dominant quartet for some quadset S must also contain a least frequent quartet on some other quadset S'. Hence, as there are 5 quadsets, T contains at most $4 \cdot 17 + 11 = 79$ quartets from the input, whereas T^* contains $5 \cdot 16 = 80$. Note that this example consists of trees on only 5 leaves. We do not know if such instances exist for any $n > 5$ leaves.

▶ **Theorem 14.** *There exists an instance of WQC such that there is a quartet q that appears in every input tree, but q is not a quartet of any optimal solution.*

Figure 2 shows an instance of WQC proving Theorem 14. Each input tree contains the $ab|cd$ quartet, whereas the optimal solution, which is unique, does not. The rightmost tree contains 180 quartets from the input multiset $\mathcal{Q}$, whereas any other tree has at most 176.

Finally, we note that the main interest behind Conjecture 5 is the following: if it holds, in cases where the set F of strictly least-frequent quartets is complete we could tell in polynomial time – using results of [8] – whether there is a tree T^* that contains no quartet from F. Conjecture 5 could then lead to interesting approximations or FPT algorithms. However, least-frequent quartets cannot be excluded automatically.

▶ **Theorem 15.** *There exists an instance of WQC such that every optimal solution contains a strictly least-frequent quartet, even if there exists a tree T^* with no such quartet.*

The instance corresponding to Theorem 15 is obtained from the instance shown in Figure 1, by removing all occurrences of the third tree on the top row (i.e. this tree now appears 0 times instead of 3 times). The second-most frequent quartets now appear 13 times each, and so the tree T^* that contains all these quartets has a total weight of $5 \cdot 13 = 65$. However, there are trees with a total weight of 75, which are optimal (for instance, the tree of cardinality 9 in the figure). Each such tree contains a strictly dominant quartet, and as mentioned before, also a strictly least-frequent quartet.

B.3 Proof of Theorem 8

Let $N := k\binom{n}{4}$, i.e. the total number of quartets in $\mathcal{Q}$, let p be the maximum number of quartets that can be preserved from $\mathcal{Q}$ for compatibility, and let d be the minimum number of quartets to discard from $\mathcal{Q}$ in order to attain compatibility (here p and d refer to multiset cardinalities). Note that $d = N - p$. We show that taking the best tree between the one obtained from the factor α algorithm for WMQI and the one obtained from the "random-tree-algorithm" achieves a factor β for WQC. Suppose first that $p \leq N/(3\beta)$. By Lemma 7, the "random-tree-algorithm" yields a tree containing at least $|\mathcal{Q}|/3 = N/3$ quartets from $\mathcal{Q}$, and since $N/3 = \beta N/(3\beta) \geq \beta p$, it yields a solution to WQC within a factor β from optimal. Thus we may assume that $p > N/(3\beta) = N(3\alpha - 2)/(3\alpha)$. Since we have an α approximation for WMQI, we may obtain a solution discarding at most $\alpha d = \alpha(N - p)$ quartets. This solution preserves at least $N - (\alpha(N - p)) = \alpha p + (1 - \alpha)N$ quartets from $\mathcal{Q}$. We claim that this attains a factor β approximation. Suppose instead that $\alpha p + (1 - \alpha)N < \beta p$. Then $p < (\alpha - 1)N/(\alpha - \beta)$ which, with a little work, yields $p < N(3\alpha - 2)/(3\alpha)$, contradicting our assumption on p. Thus, the WMQI approximation preserves at least βp quartets.

Optimal Omnitig Listing for Safe and Complete Contig Assembly[*]

Massimo Cairo[†1], **Paul Medvedev**[2], **Nidia Obscura Acosta**[3], **Romeo Rizzi**[‡4], **and Alexandru I. Tomescu**[§5]

1 **University of Trento, Trento, Italy**
`massimo.cairo@unitn.it`

2 **The Pennsylvania State University, State College, PA, USA**
`pashadag@cse.psu.edu`

3 **Helsinki Institute for Information Technology HIIT, Department of Computer Science, University of Helsinki, Helsinki, Finland**
`obscura.nidia@helsinki.fi`

4 **Department of Computer Science, University of Verona, Verona, Italy**
`romeo.rizzi@univr.it`

5 **Helsinki Institute for Information Technology HIIT, Department of Computer Science, University of Helsinki, Helsinki, Finland**
`alexandru.tomescu@helsinki.fi`

Abstract

Genome assembly is the problem of reconstructing a genome sequence from a set of reads from a sequencing experiment. Typical formulations of the assembly problem admit in practice many genomic reconstructions, and actual genome assemblers usually output *contigs*, namely substrings that are promised to occur in the genome. To bridge the theory and practice, Tomescu and Medvedev [RECOMB 2016] reformulated contig assembly as finding all substrings common to all genomic reconstructions. They also gave a characterization of those walks (*omnitigs*) that are common to all closed edge-covering walks of a (directed) graph, a typical notion of genomic reconstruction. An algorithm for listing all maximal omnitigs was also proposed, by launching an exhaustive visit from every edge.

In this paper, we prove new insights about the structure of omnitigs and solve several open questions about them. We combine these to achieve an $O(nm)$-time algorithm for outputting all the maximal omnitigs of a graph (with n nodes and m edges). This is also optimal, as we show families of graphs whose total omnitig length is $\Omega(nm)$. We implement this algorithm and show that it is 9-12 times faster in practice than the one of Tomescu and Medvedev [RECOMB 2016].

1998 ACM Subject Classification G.2.2 Graph Theory, E.1 Data Structures, J.3 Life and Medical Sciences

Keywords and phrases genome assembly, graph algorithm, edge-covering walk, strong bridge

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.29

[*] We acknowledge partial support from the Academy of Finland grant 274977 to A.I.T. and grant 284598 (CoECGR) to N.O.A., from the European Union's Horizon 2020 Marie Skłodowska-Curie grant agreement No 690941 to M.C. and A.I.T., and from NSF awards DBI-1356529, CCF-1439057, IIS-1453527, and IIS-1421908 to P.M.

[†] First author contribution.

[‡] Shared last author contribution.

[§] Shared last author contribution.

1 Introduction

Genome assembly is the problem of reconstructing a genome sequence from a set of reads from a sequencing experiment. It is one of the oldest problems in bioinformatics, but many challenges remain. For example, assemblers for novel sequence technologies such as Oxford nanopore are still only in development. Assembly of heterogeneous tumor data is also a challenge. Many of these challenges can be met by building on top of existing assembly algorithms. However, recent directions to improve the theoretical underpinning of assembly have the potential to improve assembly across a wide breadth of scenarios.

Genome graphs have been the basis of most assembly algorithms. There is the *edge-centric de Bruijn graph* [2, 15], where every k-mer (string of length k) of the reads becomes a node and every $(k+1)$-mer of the reads becomes an edge, or the *node-centric de Bruijn graph*, where the nodes are the same but the edges are $(k-1)$-overlaps between nodes. In a *string graph*, every read becomes a node and large enough non-transitive overlaps between reads are represented as edges [11, 16]. In a recent paper [17], these graphs were unified under the "genome graph" model. Theoretical formulations of the assembly problem define what a *genome reconstruction* is: typically, this is a walk in a genome graph, subject to some constraints. For example, a genome reconstruction could be a closed (i.e., circular) walk covering every edge of the genome graph exactly once [14, 8, 13] (to be called *edge-covering* in the ongoing), or a closed Eulerian walk [9, 10, 12, 5].

However, algorithms to find an entire genome reconstruction are rarely implemented in practice, because there is usually more than one valid genome reconstruction. When assemblers have no way to distinguish different reconstructions, they instead output *contigs*, which are stretches of DNA that are assumed to be in the genome. To bridge theory and practice, Tomescu and Medvedev proposed in [17] an alternative formulation of the contig assembly problem. A string is considered *safe* if it is guaranteed to occur in every valid genome reconstruction. A contig assembly algorithm should ideally be safe (i.e., only outputting safe strings) and *complete* (i.e., every safe string should be output by the algorithm).

Previous work. The notion of a safe and complete algorithm embodies several previous results. Contig assembly was first approached by finding *unitigs* [6], namely those paths whose internal nodes have in- and out-degree one. Later, some generalizations of unitigs have been considered. For example, [15] considered paths whose internal nodes have out-degree one, with no restriction on their in-degree; [10, 4, 7] considered the unitigs of a genome graph simplified with the so-called *Y-to-V transformation* (we further discuss this at the end of Section 4). Although no underlying notion of genomic reconstruction was explicit in these studies, it can be shown that the resulting paths are safe for closed edge-covering walks. However, as [17] notices, such approaches do not find all the safe strings. Other studies have indeed given safe and complete algorithms for some reconstruction notions. Nagarajan and Pop [12] attribute to [18] the characterization of the walks common to all closed Eulerian walks. For edge-weighted genome graphs, [12] claims that a simple algorithm exists for finding all those walks common to all *shortest* closed edge-covering walks.

Tomescu and Medvedev [17] considered the genomic reconstruction notion of a closed edge-covering walk. This model is strictly more general than the above two ones, and thus safe strings for it are also safe for them. Moreover, it is also more realistic, because the Eulerian notion assumes that all positions in the genome are sequenced exactly the same number of times, while the minimality criterion from other notion may over-collapse repeated regions. However, it still assumes that the reads are error-free, single-stranded, come from a circular genome, and every position in the genome appears in some read.

In [17] a characterization of those walks common to all such genomic reconstructions was given. These walks were called *omnitigs* (see Definition 1), and an algorithm for finding all maximal omnitigs was presented. We refer to [17] for further details on the practical merits of omnitigs, as opposed to e.g., unitigs. The asymptotic running time of this algorithm was not fully analyzed in [17] except to say it was polynomial time. However, it is based on launching an exhaustive visit from every edge of the graph, and extending all such possible walks as long as they are omnitigs. Its running time remained several orders of magnitude slower than finding unitigs, and improving it was recognized as an important open problem.

Contributions and approach of this paper. The main result of this paper is an algorithm (Algorithm 3) running in time $O(nm)$ for outputting all maximal omnitigs of a graph (m is the number of edges, n is the number of nodes, and in this paper all graphs are directed). This algorithm is also *optimal*, in the sense that there are families of graphs for which the total length of their omnitigs is $\Omega(nm)$ (see e.g., Figure 4).

This algorithm is based on three insights.

1. A structural result connecting branches of a graph (i.e., edges whose source node has out-degree at least two) with left-maximal omnitigs (Theorem 8). In particular, there can be only one left-maximal omnitig ending with a given branch, and the structure of such omnitigs is almost fully characterizable. This also implies that the number of maximal omnitigs is at most m and their individual lengths are bounded by $3n - 1$. We also give families of graphs that achieve these upper bounds, showing that they are tight. Previously, only an upper bound of nm was known on the number of maximal omnitigs and an upper bound of nm on their lengths [17]. This is encouraging also from a practical point of view, because the popular (maximal) unitigs have the same tight asymptotic bounds on their number and individual length (but not on their total length, which is m).
2. A partial order between branches, based on whether or not they are connected by "simple" omnitigs (Definition 13), which we prove to be acyclic. This allows us to reuse computation when recursively computing the left-maximal omnitig ending with a given branch.
3. A connection between omnitigs and strong bridges of a graph (i.e., those edges whose removal disrupts strong connectivity [3]). In particular, omnitigs that do *not* start with a strong bridge are easy to find (Lemma 17). Since there are at most $O(n)$ strong bridges in a graph, this implies that also the number of hard cases is $O(n)$, and not $O(m)$.

We also implement the new algorithm, and show in Section 5 that it is 9-12 times faster in practice than the one of [17]. Finally, at the end of Section 4 we demonstrate that the Y-to-V transformation, used as pre-processing step in the implementation of [17] to simplify the input, can result in shorter maximal omnitigs. This transformation is a well-known method (e.g. [10, 4, 7]) for reducing the genome graph, maintaining the property that its unitigs spell safe strings.

2 Background and notation

In this paper, a *graph* is a tuple $G = (V, E, s, t)$, where V is a finite set of *nodes*, E is a finite set of *edges*, and $s, t \colon E \to V$ assign to each edge $e \in E$ its *source node* $s(e)$ and its *destination node* $t(e)$. Parallel edges and self-loops are allowed. We say that an edge e goes *from* $s(e)$ *to* $t(e)$. The *reverse graph* of G is defined as $G^R = (V, E, t, s)$.

A *walk* on G is a sequence $w = (v_0, e_1, v_1, e_2, \dots, v_{\ell-1}, e_\ell, v_\ell)$, $\ell \geq 0$, where $v_0, v_1, \dots, v_\ell \in V$ are nodes and each e_i is an edge from v_{i-1} to v_i. We say that w goes *from* $s(w) = v_0$ *to* $t(w) = v_\ell$ and has *length* $|w| = \ell$. A walk w is called *empty* if $|w| = 0$, and *non-empty*

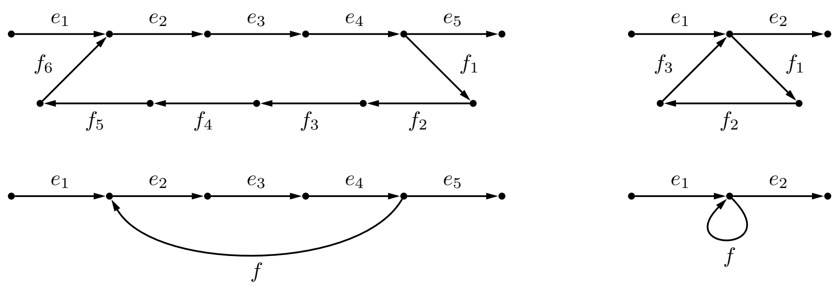


Figure 1 Examples of walks $e_1 \cdots e_\ell$ which are not omnitigs, due to the existence of a path p satisfying the conditions of Definition 1. In the first row, $p = f_1 \cdots f_{|p|}$ with $|p| > 1$. In the second row, $p = f$. In the left column, p is a non-empty open path. In the right column, p is a closed path.

otherwise. (There exists exactly one empty walk $\epsilon_v = (v)$ for every node $v \in V$, and $s(\epsilon_v) = t(\epsilon_v) = v$.) A walk w is called *closed* if it is non-empty and $s(w) = t(w)$, otherwise it is *open*. A *path* is a walk whose nodes $v_0, v_1, \ldots, v_\ell$ are all distinct, except that $v_\ell = v_0$ is allowed (in which case we have either a closed or an empty path). A graph is *strongly connected* if there is a path (or, equivalently, a walk) from any node to any other node.

In the rest of this paper, a strongly connected graph $G = (V, E, s, t)$ is given, with $|V| = n$ and $|E| = m \geq n$. We adopt the following conventions. Letters u, v denote nodes, letters e, f, g, h denote edges, which are identified with the corresponding length-1 walks, letters p, q, r denote paths, and letters w, x, y, z denote generic walks (each letter possibly with subscripts or superscripts). Juxtaposition ww' denotes the concatenation of walks w and w', where $t(w) = s(w')$ is implicitly assumed. We start from the following definition of omnitigs offered in [17].

▶ **Definition 1** (Omnitig). A non-empty walk $w = e_1 \cdots e_\ell$ is an *omnitig* if, for every $1 \leq i < j \leq \ell$, there is no non-empty path from $s(e_j)$ to $t(e_i)$, with first edge different from e_j, and last edge different from e_i.

The main result from [17] is that those walks that are sub-walks of all closed edge-covering walks of a strongly connected graph are precisely its omnitigs. Clearly every edge is an omnitig and any proper subwalk of an omnitig is an omnitig. Figure 1 illustrates examples of walks that are not omnitigs. An omnitig w is *right-maximal* (resp., *left-maximal*) if there is no walk we (resp., ew) which is an omnitig. An omnitig is *maximal* if it is both left- and right-maximal. We note that in [17] two types of omnitigs were considered, depending on the genome model used. Here, we use omnitigs to refer the edge-centric omnitigs from [17].

3 Structure of maximal omnitigs

In this section we prove some structural properties of maximal omnitigs. To better understand the ways in which omnitigs might possibly overlap, we propose the notion of *branch* and *univocal walk*. A node u is called *branching* if its out-degree is more than one. In this case, any edge e with $s(e) = u$ is called a *branch*, and any two distinct edges $e \neq e'$ with $s(e) = s(e') = u$ are called *siblings*. The set of all branches is denoted by $B \subseteq E$. An edge is called an *R-branch* if it is a branch in G^R. A walk is called *univocal* if none of its edges is a branch and *R-univocal* if none of its edges is an R-branch.

We start by showing some facts about branches and univocal walks.

▶ **Lemma 2.** *If G contains at least a branch, then every univocal walk is an open path.*

Proof. A minimal counterexample is a univocal closed path p. Since every path from $s(p)$ is a prefix of p, and G is strongly connected, then p contains every node in the graph, and there are no branches. ◀

▶ **Lemma 3.** *If w is an omnitig and q is a univocal path from $t(w)$, then wq is an omnitig.*

Proof. Let p be a path certifying that wq is not an omnitig by Definition 1. If $s(p)$ is a node of q, then a whole suffix of q is a prefix of p, since q is univocal; in this way, the property that the first edge of p differs from e_j would be contradicted. Therefore $s(p)$ is a node of w, but then p is a path actually certifying that w is not an omnitig, again a contradiction. ◀

▶ **Lemma 4.** *Every left-maximal omnitig contains a branch.*

Proof. Let w be a counterexample, i.e., a left-maximal omnitig which is univocal. Let e be any edge with $t(e) = s(w)$ (at least one exists since G is strongly connected). The edge e is an omnitig, and thus by Lemma 3 ew is an omnitig, violating the left-maximality of w. ◀

The crucial observation underlying our algorithm is that any omnitig containing a branch can be extended in an unique way to the left to obtain a left-maximal omnitig. This is expressed in Theorem 8 below. To prove Theorem 8, we need the following lemmas.

▶ **Lemma 5.** *Let fqe be an omnitig where q is an open path and e is a branch. Take any sibling e' of e and a closed path $e'p$ starting with e'. Then, fq is a suffix of $e'p$.*

Proof. Let fqe be a minimal counterexample. Then, fqe and qe are both omnitigs, and by minimality q is a suffix of $e'p$, whereas fq is not. Since q is an open path, then $q \neq e'p$, so q is actually a suffix of p. Thus we can regard $e'p$ as obtained by concatenating its suffix q to its remaining prefix r, i.e., $e'p = rq$. Here, r is a non-empty path and fulfills all conditions stated in Definition 1: it starts with $e' \neq e$, and ends with an edge $f' \neq f$ (otherwise fq would be a suffix of $rq = e'p$). This shows that fqe is not an omnitig: a contradiction. ◀

▶ **Lemma 6.** *Let $e'pe$ be a walk where e and e' are siblings and $e'p$ is a closed path. Then, $e'pe$ is an omnitig iff p is univocal and e' is the only sibling of e.*

Proof. ($\Longleftarrow$) The only path satisfying Definition 1 must start with e', and hence be a prefix of $e'p$. ($\Longrightarrow$). First we show that e' is the only sibling of e. Let e'' be any sibling of e, and take any closed path $e''p'$. Then, $e'p$ is a suffix of $e''p'$ by Lemma 5. Being both closed paths, we have $e'p = e''p'$ and in particular $e'' = e'$.

We now prove that p is univocal. Assume not, and write $p = qfr$ where f is any branch. Let f' be a sibling of f, and $f'p'$ a closed path. Clearly, $s(f') = s(f) \neq s(e)$, hence f' does not appear in the closed path $e'p = e'qfr$. Let q' be the shortest prefix of p' where $t(q')$ is a node of p. Observe that q' exists since $t(p') = s(f') = s(f)$ is a node of p. Moreover, the last edge of q', if any, does not appear in $e'p$. Notice that $t(q')$ is either a node of q or a node of r. If $t(q')$ is a node of q, then the path $f'q'$ shows that $e'qf$ is not an omnitig. Otherwise, if $t(q')$ is a node of r, then the path $e'qf'q'$ shows that fre is not an omnitig. In either case $e'pe = e'qfre$ is not an omnitig: a contradiction. ◀

▶ **Lemma 7.** *There is no omnitig of the form $fqrqe$ where qr is a closed path, r is non-empty, e is a branch, and f is an R-branch.*

Proof. Assume for a contradiction that $fqrqe$ is an omnitig violating the claim of the lemma. Let e' be the first edge of r. We will prove that $e' \neq e$. Write $r = e'r'$ and observe that $r'q$ is an open path, so $e'r'qe$ satisfies the hypothesis of Lemma 5. Let $e'' \neq e$ be a sibling of e,

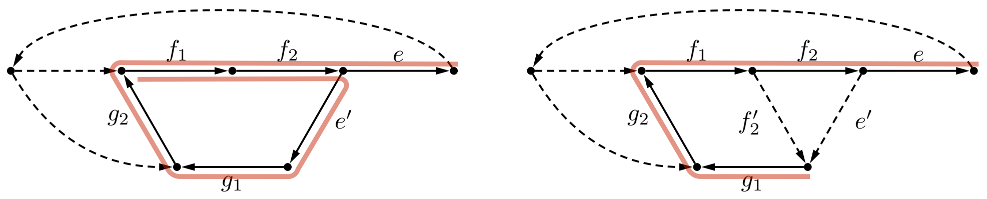


Figure 2 Example of graphs where the two cases of Theorem 8 occur, for $p = g_1g_2f_1f_2$ and $p' = f_1f_2$. In the first case (left), p is univocal and the left-maximal omnitig is $we = p'e'pe = f_1f_2e'g_1g_2f_1f_2e$. In the second case (right) p is not univocal due to the edge f_2', and the left-maximal omnitig is $we = g_1g_2f_1f_2e$. Omnitigs we are shown in red and have solid edges.

and $e''p$ a closed path. Then, by Lemma 5, $e'r'q$ is a suffix of $e''p$. In fact, since both $e'r'q$ and $e''p$ are closed paths, then $e'r'q = e''p$ and $e' = e'' \neq e$, as claimed.

The very same argument applies on the reverse graph, since the notion of omnitig is symmetric, as well as the statement of the lemma. Therefore, also the last edge f' of r is distinct from f. Now, r is a non-empty path with first edge $e' \neq e$ and last edge $f' \neq f$. Hence, r satisfies the conditions of Definition 1, showing that $fqrqe$ is not an omnitig. ◀

▶ **Theorem 8.** *There exists a unique left-maximal omnitig we, ending with a given branch e. Moreover, for any sibling e' of e and a closed path $e'p$, either:*

- *$we = p'e'pe$, where p' is the longest R-univocal path to $s(e)$,* or
- *we is a suffix of pe,*

where the first case occurs iff e' is the only sibling of e and p is univocal.

Proof. Consider any omnitig we. We show that we is either a suffix of pe or of the form $we = p''e'pe$, where p'' is an R-univocal path. This suffices to show that there is a unique left-maximal omnitig we, and that one of the two cases occurs.

If w is an open path then we is a suffix of pe by Lemma 5. Otherwise, take the shortest suffix $e''p$ of w which is not an open path. Since p is an open path ($e''p$ is the shortest suffix of w which is not), then $e'' = e'$ by Lemma 5.

Hence, a minimal counterexample for our claim is an omnitig of the form $we = fqe'pe$ where q is R-univocal (hence an open path by Lemma 2 applied to the reversed graph) and f is an R-branch. Since $t(q) = t(p)$ and q is R-univocal, then q is a suffix of $e'p$. In fact, q is a suffix of p, since it is open. Hence, we can write $e'p = rq$, where r is non empty, and $we = fqrqe$, violating Lemma 7.

Finally, the conditions in which the first case occurs are stated in Lemma 6, noticing that $p'e'pe$ is an omnitig iff $e'pe$ is an omnitig, by Lemma 3 applied in the reverse graph. ◀

▶ **Corollary 9.** *There are at most m maximal omnitigs.*

Proof. Any maximal omnitig has a branch by Lemma 4; hence it has the form $w = w'er$, where e is its last branch and r is univocal. By Theorem 8, w' is uniquely determined by e, and, by Lemma 3, r is the longest univocal path from $t(e)$, also uniquely determined by e. In conclusion, every omnitig has a last branch and every branch is the last branch of at most one maximal omnitig. ◀

▶ **Corollary 10.** *Every maximal omnitig traverses any node at most three times, and thus has length at most $3n - 1$.*

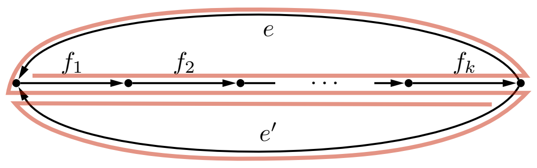


Figure 3 A family of graphs parametrized by $k \geq 0$ where the bound given in Corollary 10 is tight. Let $p = f_1 \cdots f_k$. The maximal omnitigs are $pepe'p$ and $pe'pep$: both traverse each node exactly three times; $pepe'p$ is marked in red.

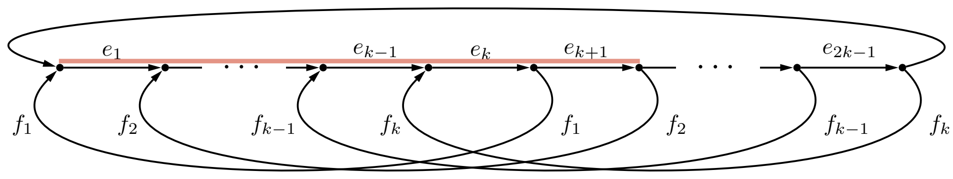


Figure 4 A family of sparse graphs G_k parametrized by $k \geq 1$ where there are $\Theta(k)$ nodes and edges, and the total length of maximal omnitigs is $\Theta(k^2)$. This shows that the bound given in Corollary 11 is tight, in the sparse case. Indeed, the walk $w_i = e_i e_{i+1} \cdots e_{i+k} e_{i+k}$ is a maximal omnitig, for $1 \leq i \leq k-1$, and has length $k+1$; walk w_1 is marked in red.

Proof. Any maximal omnitig has the form $w = w'er$ where e is its last branch. By Theorem 8, either w' is an open path, or $w = p'e'per$ where p', p, r are univocal, and hence open paths by Lemma 2. Consider that open paths visit each node at most once. ◀

▶ **Corollary 11.** *The total length of maximal omnitigs is $O(nm)$.*

In a complete graph with node set V, $|V| \geq 3$, and edge set $V \times V$ every single edge is a maximal omnitig, hence the bound given in Corollary 9 is tight. Figures 3 and 4 demonstrate graph families showing that the bounds of Corollary 10 and Corollary 11 are also tight. That is, they contain maximal omnitigs of length $3n - 1$, and the total length of the maximal omnitigs can be $\Omega(nm)$.

4 The algorithm

We start by considering a procedure LongestSuffix that takes an omnitig w' and an edge e with $s(e) = t(w')$, and computes the longest suffix of $w = w'e$ that is still an omnitig. A pseudo-code for such a procedure is shown in Algorithm 1, and it is an adaptation of the ideas given in [17].

▶ **Lemma 12.** *The function* LongestSuffix *can be implemented in $O(m)$.*

The strategy of our algorithm is to first pick a branch e, since by Lemma 4 every maximal omnitig contains one, and then construct the only left-maximal omnitig ending with e, according to Theorem 8. To this end, we may need to compute the longest suffix of $e'p$ which is an omnitig; however, this could require quadratic time to output a single left-maximal omnitig. Instead, we show that it is possible to recycle the computational effort among different branches, in order to pay linear time per-branch. We introduce the following notion of order between branches.

▶ **Definition 13.** For any two distinct non-sibling branches $e, f \in B$, write $f \prec e$ if there exists an omnitig fpe where p is univocal.

Algorithm 1: Function LongestSuffix.

```
1 Function LongestSuffix($w$)
      Input   : A non-empty walk $w = w'e$ where $w'$ is an omnitig and $e$ is a branch.
      Returns : The longest suffix of $w$ which is an omnitig.
2     Denote $w = w'e = f_1 \cdots f_\ell e$.
3     Compute the set $S_e \subseteq V$ of nodes reachable from $s(e)$ without using $e$.
4     Let $\hat{\imath}$ be the largest index $i \in \{1, \ldots, \ell\}$ such that there exists an edge $g \notin \{e, f_i\}$
        with $s(g) \in S_e$ and $t(g) = t(f_i)$, taking $\hat{\imath} = 0$ if no such index exists.
5     return $f_{\hat{\imath}+1} \cdots f_\ell e$
```

▶ **Lemma 14.** *For any $e \in B$ there is at most one $f \in B$ such that $f \prec e$.*

Proof. Take a sibling e' of e and a closed path $e'p$. Let f be the last branch on $e'p$ (it exists since its first edge e' is a branch) and let fq be the suffix of $e'p$ starting with f, where q is univocal. Assume $\tilde{f} \prec e$ and let $\tilde{f}\tilde{q}e$ be an omnitig with $\tilde{q}$ univocal. By Lemma 2, $\tilde{q}$ is an open path, and by Lemma 5, $\tilde{f}\tilde{q}e$ is a suffix of $e'pe$, thus $\tilde{f} = f$ and $\tilde{q} = q$. ◀

Our algorithm for computing the left-maximal omnitig ending with a given branch e works as follows. We first check whether the first case of Theorem 8 occurs, by verifying the condition provided therein. If not, then we consider the suffix fq of $e'p$ defined as in the proof of Lemma 14. We have two cases.

- fqe is not an omnitig. Then, an invocation of LongestSuffix(fqe) yields the only left-maximal omnitig ending with e.
- fqe is an omnitig. Then, $s(f) \neq s(e)$ since fq is open, thus $f \prec e$. In this case, we apply the procedure recursively to the branch f, obtaining an omnitig w''. Then, the left-maximal omnitig ending with e must be a suffix of $w''qe$, and can be obtained as LongestSuffix($w''qe$).

Lemma 15 is crucial in showing that the recursion is well-founded. As we will show later, thanks to memoization, this recursive application allows to reuse the computational effort and leads to a faster worst-case running time.

▶ **Lemma 15.** *The relation $\prec$ is acyclic.*

To achieve the claimed $O(nm)$ running time, we need a further improvement. We recall the definition of strong bridge in a strongly connected graph [3].

▶ **Definition 16.** An edge e is a *strong bridge* if, by removing e, the graph is no longer strongly connected. Equivalently, there is a pair of nodes u, v, such that every path from u to v contains e.

The lemma below states that omnitigs containing non-strong-bridges have a simpler structure.

▶ **Lemma 17.** *If fq is an omnitig and an open path, and f is* not *a strong bridge, then q is univocal.*

Proof. A minimal counterexample is an omnitig fqe, where fqe is an open path and e is a branch. Fix a sibling e' of e, and take a closed path $e'p$ such that p does not contain f, which exists since f is not a strong bridge. By Theorem 8, fq is a suffix of p: a contradiction since p does not contain f. ◀

Algorithm 2: Computing the only left-maximal omnitig ending with a branch e.

```
1  Function OmnitigEndingWith($e$)
       Input   : A branch $e$.
       Returns : The only left-maximal omnitig $we$.
2      Let $e'$ be any sibling of $e$ and $e'p$ be any closed path starting with $e'$.
3      Let $f$ be the last branch of $e'p$ (possibly $f = e'$) and $fq$ the suffix of $e'p$ starting
         with $f$.
4      Let $p'$ be the longest $R$-univocal path to $s(e)$.
5      if $e$ has only one sibling $e'$ and $p$ is univocal then return $p'e'pe$
6      if $e$ is not a strong bridge then return $p'e$
7      $w' \leftarrow$ LongestSuffix($fqe$)
8      if $w' \neq fqe$ then return $w'$
9      $w'' \leftarrow$ OmnitigEndingWith($f$)          ▷ OmnitigEndingWith is memoized
10     return LongestSuffix($w''qe$)
```

Algorithm 3: Computing all the maximal omnitigs.

```
1  $W \leftarrow \emptyset$
2  for $e \in B$ do
3      $w \leftarrow$ OmnitigEndingWith($e$)
4      Let $p$ be the longest univocal path from $t(e)$.
5      $W \leftarrow W \cup \{wp\}$
6  end
7  Remove from $W$ the non-right-maximal walks.
8  return $W$
```

It is known that there are at most $2n - 2 = O(n)$ strong bridges in a given graph, and they can be computed in $O(m)$ time [3, 1]. The observation of Lemma 17 allows to handle those branches e which are *not* strong bridges is a special way, and apply the full algorithm only on the $O(n)$ strong bridges. The procedure just described is illustrated in Algorithm 2.

▶ **Lemma 18.** *The function* OmnitigEndingWith *in Algorithm 2 is correct.*

The full algorithm (Algorithm 3) amounts to computing, for each branch $e \in B$, the left-maximal omnitig ending with e, and then appending the longest possible univocal suffix.

▶ **Theorem 19.** *Algorithm 3 is correct and can be implemented to run in time $O(nm)$.*

Proof. It is clear from Lemma 18 and Lemma 3 that Algorithm 3 terminates and returns a set W containing only left-maximal omnitigs. For correctness, we only need to show that, after the for-loop, W contains all the maximal omnitigs. Consider any maximal omnitig w. By Lemma 4, w contains a branch. Let e be the last branch of w, and write $w = w'ep$ where p is univocal. By Lemma 3, $w'e$ is left-maximal (otherwise also $w = w'ep$ is not left-maximal), and p is the longest univocal path from $t(e)$, (otherwise $w = w'ep$ is not right-maximal). By Lemma 18, in the iteration of the for-loop, relative to the branch $e \in B$, the call OmnitigEndingWith(e) returns $w'e$, and $w'ep$ is added to W.

To prove our bound on the running time, we observe that, when the function OmnitigEndingWith returns before line 7, then it takes $O(n)$ time only. Indeed, the length of

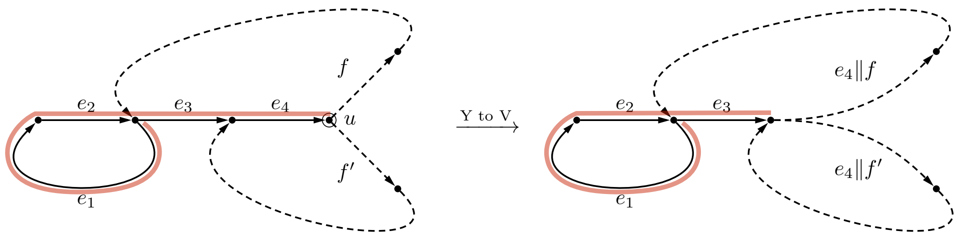


Figure 5 Left: the walk $e_1e_2e_3e_4$ is a maximal omnitig. Right: after applying the Y-to-V reduction to node u, only the omnitig $e_1e_2e_3$ is maximal, and e_3e_4 does not appear in any omnitig.

the open paths p and p' is $O(n)$. Moreover, when the condition at line 5 occurs, then the path p is univocal, and its construction can be performed in $O(n)$ time, without running a full visit of the graph. These executions of OmnitigEndingWith account for an overall running time $O(nm)$, due to memoization, since there are $O(m)$ branches.

The execution continues after line 7 only $O(n)$ times, since the number of strong bridges is $O(n)$. In this case, the running time is dominated by the calls to LongestSuffix, which take $O(m)$ time each by Lemma 12. Again, due to memoization, the overall running time is $O(nm)$. The set of strong bridges is computed once at the beginning, in linear time.

It remains to show how to implement line 7 in time $O(nm)$. First, the total length of the walks in W is $O(nm)$, because to each of the $O(m)$ walks returned by OmnitigEndingWith, each of length $O(n)$ (by Corollary 10), we append a path, thus having length $O(n)$. One way to remove the non-right-maximal omnitigs from W is to regard each walk in W as a string over the alphabet E, construct a trie containing them, in time $O(nm)$, and remove those ending in an internal node. ◀

Finally, we would like to remark on the Y-to-V reduction. Let v be a node that has exactly one in-neighbor u and more than one out-neighbors $w_1, \ldots, w_d$. The *Y-to-V reduction applied to v* removes v and its incident edges and adds an edge from u to w_i, for all $1 \leq i \leq d$. The Y-to-V reduction was suggested as a pre-processing step to the omnitig algorithm in [17] to improve the running time. However, this reduction can destroy some omnitigs, see Figure 5.

5 Experimental results

We implemented Algorithm 3 using the code base of [17].[1] We focused our experiments on measuring the running time improvements, since the practical merits of omnitigs for genome assembly were discussed in [17]. The algorithms were run on a machine with Intel Xeon 2.10GHz CPUs. Because the Y-to-V transformation is not omnitig-preserving, we disabled it from the code of [17]. We circularized three reference sequences of human chromosomes 2, 10, and 14. Each had a length of 243, 136 and 107 million nucleotides, respectively. We built the edge-centric de Bruijn graph for each, using $k = 55$. This is a typical genome graph on which contig assembly is performed.

As shown in Table 1, our algorithm was 9–12 times faster on a single thread, suggesting that our theoretical improvements indeed translate into faster running times. For the largest dataset, our algorithm took just over 2 hours, while [17] took over 22 hours. We also observe, as expected, that the running time depends on the size of the graph and the number of omnitigs, and not on their length.

[1] Available at `https://github.com/alexandrutomescu/complete-contigs`.

Table 1 Wall-clock running time comparison between the omnitig algorithm of [17] and our Algorithm 3.For fairness of comparison, the algorithms were run on a single thread, though we note that [17] supports parallelization.

	# nodes	# edges	time by [17]	time by Algorithm 3	# omnitigs	avg len (bp)
chr2	696,209	887,295	1,342 min	138 min	304,760	838
chr10	369,448	467,517	433 min	36 min	158,396	887
chr14	223,694	283,798	137 min	11 min	96,434	968

6 Conclusion

Apart from its application to genome assembly, the problem addressed in this paper is a fundamental graph theoretical one. It also fits into a line of research for finding all partial solutions common to natural notions of walks in graphs, such as Eulerian walks [18] or shortest edge-covering walks [12]. We presented here an optimal $O(nm)$ algorithm for finding all maximal omnitigs and showed that it can be an order of magnitude faster than a previous one based on exhaustive visits. When applied to genome assembly, our algorithm remains significantly slower than finding unitigs. However, we believe that an embarrassingly parallel implementation is possible, and that it will improve running time by another order of magnitude in practice.

Acknowledgements. We thank Martin Milanič for pointing us to persistent solutions and blockers.

References

1 Donatella Firmani, Giuseppe F. Italiano, Luigi Laura, Alessio Orlandi, and Federico Santaroni. Computing strong articulation points and strong bridges in large scale graphs. In Ralf Klasing, editor, *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA 2012)*, volume 7276 of *LNCS*, pages 195–207, Berlin, Heidelberg, 2012. Springer-Verlag. `doi:10.1007/978-3-642-30850-5_18`.

2 Ramana M. Idury and Michael S. Waterman. A new algorithm for DNA sequence assembly. *J. Comput. Biol.*, 2(2):291–306, 1995. `doi:10.1089/cmb.1995.2.291`.

3 Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. Finding strong bridges and strong articulation points in linear time. *Theor. Comput. Sci.*, 447:74–84, August 2012. `doi:10.1016/j.tcs.2011.11.011`.

4 Benjamin Grant Jackson. *Parallel methods for short read assembly*. PhD thesis, Iowa State University, 2009. URL: `http://lib.dr.iastate.edu/etd/10704`.

5 Evgeny Kapun and Fedor Tsarev. De Bruijn superwalk with multiplicities problem is NP-hard. *BMC Bioinformatics*, 14(S-5):S7, 2013. `doi:10.1186/1471-2105-14-S5-S7`.

6 John D. Kececioglu and Eugene W. Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1/2):7–51, 1995. `doi:10.1007/BF01188580`.

7 Carl Kingsford, Michael C. Schatz, and Mihai Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics*, 11(1):21, 2010. `doi:10.1186/1471-2105-11-21`.

8 Yuri P. Lysov, Vladimir L. Florentiev, Alexandr A. Khorlin, Konstantin R. Khrapko, and Valentine V. Shik. Determination of the nucleotide sequence of dna using hybridization with oligonucleotides. A new method. *Dokl. Akad. Nauk SSSR*, 303(6):1508–1511, 1988. URL: `http://view.ncbi.nlm.nih.gov/pubmed/3250844`.

9 Paul Medvedev and Michael Brudno. Maximum likelihood genome assembly. *J. Comput. Biol.*, 16(8):1101–1116, 2009. doi:10.1089/cmb.2009.0047.

10 Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of models for sequence assembly. In Raffaele Giancarlo and Sridhar Hannenhalli, editors, *Proceedings of the 7th International Workshop on Algorithms in Bioinformatics (WABI 2007)*, volume 4645 of *LNCS*, pages 289–301. Springer, 2007. doi:10.1007/978-3-540-74126-8_27.

11 Gene Myers. Efficient local alignment discovery amongst noisy long reads. In Daniel G. Brown and Burkhard Morgenstern, editors, *Proceedings of the 14th International Workshop on Algorithms in Bioinformatics (WABI 2014)*, volume 8701 of *LNCS*, pages 52–67. Springer, 2014. doi:10.1007/978-3-662-44753-6_5.

12 Niranjan Nagarajan and Mihai Pop. Parametric complexity of sequence assembly: Theory and applications to next generation sequencing. *J. Comput. Biol.*, 16(7):897–908, 2009. doi:10.1089/cmb.2009.0005.

13 Giuseppe Narzisi, Bud Mishra, and Michael C. Schatz. On algorithmic complexity of biomolecular sequence assembly problem. In Adrian-Horia Dediu, Carlos Martín-Vide, and Bianca Truthe, editors, *Proceedings of the 1st International Conference on Algorithms for Computational Biology (AlCoB 2014)*, volume 8542 of *LNCS*, pages 183–195. Springer, 2014. doi:10.1007/978-3-319-07953-0_15.

14 Pavel A. Pevzner. L-Tuple DNA sequencing: computer analysis. *J. Biomol. Struct. Dyn.*, 7(1):63–73, August 1989. URL: http://www.tandfonline.com/doi/abs/10.1080/07391102.1989.10507752.

15 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. U.S.A.*, 98(17):9748–9753, 2001. doi:10.1073/PNAS.171285098.

16 Jared T. Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Res.*, 22(3):549–556, 2012. doi:10.1101/GR.126953.111.

17 Alexandru I. Tomescu and Paul Medvedev. Safe and complete contig assembly via omnitigs. In Mona Singh, editor, *Proceedings of the 20th Annual Conference on Research in Computational Molecular Biology (RECOMB 2016)*, volume 9649 of *LNCS*, pages 152–163. Springer, 2016. doi:10.1007/978-3-319-31957-5_11.

18 Michael S. Waterman. *Introduction to Computational Biology: Maps, Sequences and Genomes*, volume 1 of *Chapman & Hall/CRC Interdisciplinary Statistics*. CRC Press, 1995. URL: https://www.crcpress.com/9780412993916.

Dynamic Elias-Fano Representation[*]

Giulio Ermanno Pibiri[1] and Rossano Venturini[2]

1 **Computer Science Department, University of Pisa, Pisa, Italy**
giulio.pibiri@di.unipi.it

2 **Computer Science Department, University of Pisa, Pisa, Italy**
rossano.venturini@unipi.it

Abstract

We show that it is possible to store a dynamic ordered set $\mathcal{S}(n, u)$ of n integers drawn from a bounded universe of size u in space close to the information-theoretic lower bound and yet preserve the asymptotic time optimality of the operations. Our results leverage on the *Elias-Fano* representation of $\mathcal{S}(n, u)$ which takes $\mathsf{EF}(\mathcal{S}(n, u)) = n\lceil\log \frac{u}{n}\rceil + 2n$ bits of space and can be shown to be less than half a bit per element away from the information-theoretic minimum.

Considering a RAM model with memory words of $\Theta(\log u)$ bits, we focus on the case in which the integers of $\mathcal{S}$ are drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$. We represent $\mathcal{S}(n, u)$ with $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ bits of space and: 1. support static predecessor/successor queries in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$; 2. make $\mathcal{S}$ grow in an append-only fashion by spending $\mathcal{O}(1)$ per inserted element; 3. support random access in $\mathcal{O}(\log n/ \log\log n)$ worst-case, insertions/deletions in $\mathcal{O}(\log n/ \log\log n)$ amortized and predecessor/successor queries in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time. These time bounds are optimal.

1998 ACM Subject Classification E.1 Data Structures, E.4 Coding and Information Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases succinct data structures, integer sets, predecessor problem, Elias-Fano

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.30

1 Introduction

The problem we consider is the one of representing in compressed space a dynamic ordered set $\mathcal{S}$ of n integer keys, which is a fundamental textbook problem (see the introduction to parts III and V of [8]). In general, any self-balancing search tree data structure, e.g., AVL or Red-Black tree, solves the problem optimally in the comparison model, by implementing all operations in $\mathcal{O}(\log n)$ worst-case time and using linear space [8]. However, by exploiting the fact that the stored keys are integers drawn from a bounded universe of size u, the problem is known to admit more efficient solutions in terms of asymptotic time complexity while still retaining linear space [8, 23, 26, 30, 13, 14]. Classical examples include the van Emde Boas tree [26, 27, 28], x/y-fast trie [30] and the fusion tree [14], that was the first data structure able to surpass the information-theoretic lower bound, by exhibiting an optimal [13] amount of time per operation within a number of memory words proportional to the size of the input. Some efforts have been spent in trying to reduce the space requirements of the representation [16, 18, 25] but known compressed solutions do not closely match the information-theoretic lower bound of the underlying integer set.

[*] This work was partially supported by the EU H2020 Program under the scheme INFRAIA-1-2014-2015: *Research Infrastructures*, grant agreement #654024 *SoBigData: Social Mining & Big Data Ecosystem* and by the *Pegaso* Project, POR FSE 2014-2020.

In this paper we show that it is possible to *preserve the optimal bounds* for the operations under almost *optimal space requirements*. The key ingredient of our data structures is the *Elias-Fano* representation of monotone integer sequences [10, 11]. In particular, Elias-Fano encodes a monotone integer sequence $\mathcal{S}(n, u)$ in $\mathsf{EF}(\mathcal{S}(n, u)) = n\lceil \log \frac{u}{n} \rceil + 2n$ bits, which can be shown to be less than half a bit per element away from optimality [10], maintaining the capability of randomly access an integer in $\mathcal{O}(1)$ worst-case time. The query Predecessor, which, given an integer x, returns $\max\{y \in \mathcal{S} : y < x\}$, is possible as well over the compressed sequence in $\mathcal{O}(1 + \log \frac{u}{n})$ worst-case time. These properties make Elias-Fano extremely efficient on crucial practical applications, e.g., inverted indexes compression, just to mention the most noticeable one. Since inverted indexes can indeed be regarded as being a collection of sorted integer sequences, recent works [29, 20] have shown that Elias-Fano exhibits the best time/space trade-off thanks to its efficient search capabilities and strong theoretical guarantees. For this specific application, the operation that has to be supported efficiently is $\mathsf{Successor}(x) = \min\{y \in \mathcal{S} : y \geq x\}$, which is commonly called NextGEQ (Next Greater or EQual) [29, 20]. Throughout the paper we adopt the classical nomenclature and discuss $\mathsf{Predecessor}(x)$ as it is well known that the twin query $\mathsf{Successor}(x)$ is solved in a similar way.

The natural question is whether it is possible to extend the *static* Elias-Fano representation to *dynamic* scenarios, in which integers can also be inserted/deleted in/from $\mathcal{S}$. To this end, we consider the case in which the n integers of $\mathcal{S}$ are drawn from a *polynomial universe* of size $u = n^\gamma$, for any $\gamma = \Theta(1)$. This is the classical operational setting as considered by Fredman and Saks [13] (list representation problem) and let us concentrate on the typical case of practical interest. In order to characterize the asymptotic complexity of the data structures described in the paper and review the literature, we use a RAM model with word size $w = \Theta(\log u)$ bits. We also adopt the usual trans-dichotomous assumption [14], making w grow with n as needed. We maintain $\mathcal{S}(n, u)$ using $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ bits of space, hence introducing a *sublinear* space overhead with respect to its static Elias-Fano representation, and show how:

1. static predecessor/successor queries can be supported in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time (note that the first term of the bound, i.e., $\mathcal{O}(1 + \log \frac{u}{n})$, is optimal only for polynomial universes of size $u = n^\gamma$ with $1 \leq \gamma \leq 1 + \log\log n/\log n$);
2. to extend $\mathcal{S}$ in an append-only fashion, i.e., by assuming that integers are inserted in the data structure in sorted order, using a constant amount of work per integer;
3. to maintain $\mathcal{S}$ in a fully dynamic way, supporting random access in $\mathcal{O}(\log n/\log\log n)$ worst-case, insertions/deletions in $\mathcal{O}(\log n/\log\log n)$ amortized and predecessor/successor queries in $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time.

2 Related Work

We organize the discussion of the related work in three parts. The first part concerns the review of the results about the static predecessor problem. The second one explains in details the (static) Elias-Fano representation of monotone integer sequences because it forms the backbone of our solutions. The last part finally describes the results closest to our work for the maintenance of a dynamic integer set.

2.1 Static Predecessor Problem

We could solve the static predecessor problem in $\mathcal{O}(1)$ worst-case by storing all results to every possible query using *perfect hashing* [12] in $\mathcal{O}(u)$ words of space. In order to not trivialize the problem, assume we have a polynomial space budget, e.g., we deal with a data structure occupying $\mathcal{O}(n^{\mathcal{O}(1)})$ words.

Ajtai [1] proved the first $\omega(1)$ lower bound, claiming that $\forall w$, $\exists n$ that gives $\Omega(\sqrt{\log w})$ query time. Only ten years later Beame and Fich [3, 4] proved two strong bounds for any *cell-probe* data structure[1]. They proved that $\forall w$, $\exists n$ that gives $\Omega(\log w / \log\log w)$ query time and that $\forall n$, $\exists w$ that gives $\Omega(\sqrt{\log n / \log\log n})$ query time. They also gave a static data structure achieving $\mathcal{O}(\min\{\log w / \log\log w, \sqrt{\log n / \log\log n}\})$ which is, therefore, optimal. Building on a long line of research, Pătraşcu and Thorup [21, 22] finally proved the following *optimal* space-time trade-off for a *static* data structure taking $m = n2^a w$ bits of space, with $a = \log \frac{m}{n} - \log w$

$$\Theta\Big(\min\Big\{\log_w n, \log\frac{w-\log n}{a}, \frac{\log\frac{w}{a}}{\log(\frac{a}{\log n}\log\frac{w}{a})}, \frac{\log\frac{w}{a}}{\log(\log\frac{w}{a}/\log\frac{\log n}{a})}\Big\}\Big). \tag{1}$$

This lower bound holds for cell-probe, RAM, trans-dichotomous RAM, external memory and communication game models. The first branch of the trade-off indicates that, whenever we are in RAM or external memory with one integer fitting in one memory word, fusion trees are optimal, as these require $\mathcal{O}(\log_w n) = \mathcal{O}(\log n / \log w)$ query time. The second branch holds for *polynomial universes*, i.e., whenever $u = n^\gamma$, for any $\gamma = \Theta(1)$. In such case we have that $w = \Theta(\log u) = \gamma \log n$, therefore y-fast tries [30] and van Emde Boas trees [26, 27, 28] are optimal with query time $\mathcal{O}(\log\log u) = \mathcal{O}(\log\log n)$. Finally, the last two branches of the trade-off treat the case for *super-polynomial universes*. In particular, the third branch matches the lower bound by Beame and Fich [3, 4] that requires $n^{\mathcal{O}(1)}$ words of space; the fourth branch improves this space occupancy, showing that $n^{1+1/\exp(\log^{1-\epsilon}\log u)}$ words are sufficient, for any $\epsilon > 0$.

2.2 Static Elias-Fano Representation

The integer encoding we describe in this section was independently proposed by Peter Elias [10] and Robert Mario Fano [11], hence its name. Given a monotonically increasing sequence $\mathcal{S}(n, u)$ of n positive integers drawn from a universe of size u (i.e., $\mathcal{S}[i-1] \le \mathcal{S}[i]$, for any $1 \le i < n$, with $\mathcal{S}[n-1] \le u$), we write each $\mathcal{S}[i]$ in binary using $\lceil \log u \rceil$ bits. Each binary representation is then split into two parts: a *high* part consisting in the first $\lceil \log n \rceil$ most significant bits that we call *high bits* and a *low* part consisting in the remaining $\ell = \lfloor \log \frac{u}{n} \rfloor$ bits that we similarly call *low bits*. Let us call h_i and ℓ_i the values of high and low bits of $\mathcal{S}[i]$ respectively. The Elias-Fano representation of $\mathcal{S}$ is given by the encoding of the high and low parts. The array $L = [\ell_0, \ldots, \ell_{n-1}]$ is stored in fixed-width and represents the encoding of the low parts. Concerning the high bits, we represent them in *negated unary*[2] using a bit vector of $n + \lceil \frac{u}{2^\ell} \rceil \le 2n$ bits as follows. We start from a 0-valued bit vector H and set the bit in position $h_i + i$, for all $i \in [0, n)$. The effect is that now the k-th unary integer m of H indicates that m integers of $\mathcal{S}$ have high bits equal to k. Finally the Elias-Fano representation of $\mathcal{S}$ is given by the concatenation of H and L and overall takes

$$\mathsf{EF}(\mathcal{S}(n, u)) = n\Big\lceil \log\frac{u}{n}\Big\rceil + 2n \text{ bits.} \tag{2}$$

[1] In the cell-probe computational model, described by Yao [31], computation is for free given that we only take into account word reads. It is not a very realistic model of computation, but it is useful to prove lower bounds because it is a stronger model than RAM and trans-dichotomous RAM.

[2] The negated unary representation of an integer x, is the bitwise NOT of its unary representation $U(x)$. An example: $U(5) = 00001$ and $\mathsf{NOT}(U(5)) = 11110$.

While we can opt for an arbitrary split ranging from 0 to $\lceil \log u \rceil$ into high and low parts, it can be shown that the value $\ell = \lfloor \log \frac{u}{n} \rfloor$ minimizes the overall space occupancy of the encoding [10]. As the information theoretic lower bound for a monotone sequence of n elements drawn from a universe of size u is $\lceil \log \binom{u+n}{n} \rceil \approx n \log \frac{u+n}{n} + n \log e$ bits, it can be shown that less than half a bit is wasted per element by the space bound in (2) [10]. Since we set a bit for every $i \in [0, n)$ in H and each h_i is extracted in $\mathcal{O}(1)$ time from $\mathcal{S}[i]$, it follows that $\mathcal{S}$ gets encoded with Elias-Fano in $\Theta(n)$ time.

Despite the simplicity of the encoding, it is possible to randomly access an integer from a sequence encoded with Elias-Fano *without* decompressing it. We refer to this operation as $\mathsf{Access}(i)$ in the following, which returns the i-th (smallest) element of the sequence. The operation is supported using an auxiliary data structure that is built on bit vector H, able to efficiently answer $\mathsf{Select}_1(i)$ queries, that return the position in H of the i-th 1 bit. This auxiliary data structure is *succinct* in the sense that it is negligibly small compared to $\mathsf{EF}(\mathcal{S}(n, u))$, requiring only $o(n)$ additional bits [7, 29].

Using the Select_1 primitive, it is possible to implement $\mathsf{Access}(i)$, which returns $\mathcal{S}[i]$ for any $i \in [0, n)$, in $\mathcal{O}(1)$. We basically have to re-link together the high and low bits of an integer, previously split up during the encoding phase. The low bits ℓ_i are trivial to retrieve as we need to read the range of bits $[i\ell, (i+1)\ell)$ from L. Note that we also need to store the quantity ℓ: a global redundancy of $\mathcal{O}(\log u)$ bits is sufficient. The retrieval of the high bits deserve, instead, a bit more care. Since we write in negated unary how many integers share the same high part, we have a bit set for every integer of S and a zero for every distinct high part. Therefore, to retrieve the high bits of the i-th integer, we need to know how many zeros are present in $H[0, \mathsf{Select}_1(i))$. This quantity is evaluated on H in $\mathcal{O}(1)$ as $\mathsf{Rank}_0(\mathsf{Select}_1(i)) = \mathsf{Select}_1(i) - i$. Notice, therefore, that the succinct rank/select data structure does not have to support Rank. Finally, linking the high and low bits is as simple as: $\mathsf{Access}(i) = ((\mathsf{Select}_1(i) - i) \ll \ell) \vee \ell_i$, where $\ll$ is the left shift operator and $\vee$ is the bitwise OR.

The query $\mathsf{Successor}(x)$ is supported in $\mathcal{O}(1 + \log \frac{u}{n})$ time[3], as follows. Let h_x be the high bits of x, i.e., its first $\lceil \log n \rceil$ most significant bits. Then $p_1 = \mathsf{Select}_0(h_x) - h_x$ represents the number of integers in $\mathcal{S}$ whose high bits are less than h_x. On the other hand, $p_2 = \mathsf{Select}_0(h_x + 1) - h_x - 1$ gives us the position at which the elements having high bits greater than h_x start. These two preliminary operations take $\mathcal{O}(1)$. We can now determine the successor of x by binary searching in this interval which may contain up to u/n integers. The algorithm for $\mathsf{Predecessor}(x)$ runs in a similar way. In particular, it could be that $\mathsf{Predecessor}(x)$ lies before the interval $[p_1, p_2)$: in this case $\mathcal{S}[p_1 - 1]$ is the element to return.

2.3 Dynamic Problems

We now review the most important results concerning the maintenance of a *dynamic* set of integers/binary strings, following the chronological order of their proposal.

The van Emde Boas tree is a recursive data structure that maintains $\mathcal{S}$ in $\mathcal{O}(u)$ words of space and supports the operations: Search which tests whether a given integer is present or not in $\mathcal{S}$, $\mathsf{Insert}/\mathsf{Delete}$ and $\mathsf{Predecessor}/\mathsf{Successor}$ all in $\mathcal{O}(\log w)$ worst-case time [26, 27, 28]. Willard [30] improved the space bound to $\mathcal{O}(n)$ words by introducing the *y-fast trie* that supports Search and $\mathsf{Predecessor}/\mathsf{Successor}$ queries in $\mathcal{O}(\log w)$ worst-case time, $\mathsf{Insert}/\mathsf{Delete}$ in amortized $\mathcal{O}(\log w)$ time.

[3] We report the bound as $\mathcal{O}(1 + \log \frac{u}{n})$, instead of $\mathcal{O}(\log \frac{u}{n})$, to cope with the case $n = u$.

The work by Fredman and Saks [13] is useful to understand which lower bounds apply to the problem we consider in the paper. They described the list representation problem, i.e., how to maintain $\mathcal{S}$ under the triad of operations Access/Insert/Delete, and proved that it can be solved in $\Omega(\log n/\log\log n)$ amortized time per operation if $w \le \log^{\gamma} n$ for some γ. No space bound is posed on such problem. Their lower bound does not apply to dynamic predecessor queries and holds for the cell-probe computational model [31]. Extending the result to the dynamic predecessor problem, they proved that any cell-probe data structure representing $\mathcal{S}$ using $(\log u)^{\mathcal{O}(1)}$ bits per memory cell and $n^{\mathcal{O}(1)}$ worst-case time for insertions, requires $\Omega(\sqrt{\log n/\log\log n})$ worst-case query time. They also proved that on a RAM, the dynamic predecessor problem can be solved in $\mathcal{O}(\min\{\log\log n \cdot \log w/\log\log w,\ \sqrt{\log n/\log\log n}\})$, using $\mathcal{O}(n)$ words. This bound was matched by Andersson and Thorup [2] with the so-called *exponential search tree.* This data structure has an optimal bound of $\mathcal{O}(\sqrt{\log n/\log\log n})$ worst-case time for searching and updating $\mathcal{S}$, using polynomial space. Raman, Raman and Rao [24] also addressed the list representation problem[4] for arrays of length n by providing two solutions. Their first data structure supports Access in $\mathcal{O}(1)$ and Insert/Delete in $\mathcal{O}(n^{\epsilon})$ worst-case time for any fixed positive $\epsilon < 1$; the second data structure implements all the three operations in $\mathcal{O}(\log n/\log\log n)$ amortized time. Both data structures use $o(n)$ bits of redundancy and the time bounds are optimal.

Fredman and Willard [14] showed that dynamic predecessor queries can be answered in $\mathcal{O}(\log n/\log\log n)$ time by using the *fusion tree.* This data structure is a B-tree with branching factor $B = \Theta(\log n)$ that stores in each internal node a *fusion node*, a small data structure able of answering predecessor queries in $\mathcal{O}(1)$ for sets up to $w^{1/5}$ integers. Updating a fusion node takes, however, $\mathcal{O}(B^4)$ time. The overall space of the data structure is $\mathcal{O}(n)$ words. The work by Pătraşcu and Thorup [23] has recently shown that it is possible to "dynamize" the fusion node, by supporting Insert and Delete in $\mathcal{O}(1)$. As a result, they have proposed a data structure representing $\mathcal{S}$ in $\mathcal{O}(n)$ words and optimal $\mathcal{O}(\log n/\log w)$ running time for the operations Insert, Delete, Predecessor, Successor, Rank and Select.

We also mention a few additional results, that will be useful in the following. Bille *et al.* [5] recently combined the static solution of Demaine and Pătraşcu [9] with the one by Pătraşcu and Thorup [23] to support *dynamic prefix sums* over an array of size n in optimal $\mathcal{O}(\log n/\log(w/\delta))$ time per operation and linear space, where δ is the number of bits needed to encode the quantity that we sum to the elements of the array. Though not devised for integer sets, the *extended CRAM* (Compressed Random Access Memory) data structure described by Jansson, Sadakane and Sung [17] allows a string $\mathcal{S}$ of length n to be stored using its k-th order empirical entropy $nH_k(\mathcal{S})$ plus a redundancy of $\mathcal{O}(n\log\sigma(k\log\sigma + (k+1)\log\log n)/\log n)$ bits for every $0 \le k < \log_{\sigma} n$, where σ is the size of the alphabet, in such a way that Insert/Delete of characters and Access to any consecutive $\log_{\sigma} n$ bits are all supported in optimal $\mathcal{O}(\log n/\log\log n)$ worst-case time. We will exploit the part of this work dedicated to the memory management. Grossi *et al.* [15] improved the previous space bound by using $nH_k(\mathcal{S}) + \mathcal{O}(n\log\log n/\log_{\sigma} n)$ bits and maintaining the asymptotic optimality for all operations. The paper by Navarro and Nekrich [19] illustrates a data structure supporting Access, Rank/Select queries, as well as symbol insertions/deletions on $\mathcal{S}$ in optimal $\mathcal{O}(\log n/\log\log n)$ time and taking $nH_0(\mathcal{S}) + \mathcal{O}(n + \sigma(\log\sigma + \log^{1+\epsilon} n))$ bits of space. Of particular interest for our purposes, is the data structure described in Appendix A.1 concerning the organization of data in small blocks. The high-level idea is to maintain a

[4] In their paper [24], the authors refer to the list representation problem, as introduced by Fredman and Saks [13], as the *dynamic array* problem. Also, the operation Access is named Index.

tree of constant height with node degree $\log^{\delta} n$, for some $0 < \delta < 1$, and leaves containing $o(\log n)$ elements each. As each internal node can fit in one machine word, the tree supports basic search operations in $\mathcal{O}(1)$ time by using a small pre-computed table. In Section 5 we will make use of a similar data structure, in order to handle mini blocks of sorted integers, which avoids the use of pre-computed tables.

3 Static Predecessor Queries in Optimal Time

In this section we are interested in determining the optimal running time of Predecessor for the Elias-Fano space bound in (2). As mentioned in Section 1, our focus is on polynomial universes, i.e., $u = n^{\gamma}$ for any $\gamma = \Theta(1)$, for which the second branch of the time/space trade-off in (1) becomes optimal. The following theorem shows that adding $o(n)$ bits of redundancy to $\mathsf{EF}(\mathcal{S}(n,u))$ is enough to support Predecessor queries in optimal time.

▶ **Theorem 1.** *There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of n integers drawn from a polynomial universe of size $u = n^{\gamma}$, for any $\gamma = \Theta(1)$, that takes $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space and supports* Access *in $\mathcal{O}(1)$ worst-case and* Predecessor/Successor *queries in optimal $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time.*

We resort on the time/space trade-off (1) by Pătraşcu and Thorup [21, 22]. In our case, $a = \log(\lceil \log \frac{u}{n} \rceil + 2)$ and $w = \Theta(\log u) = \gamma \log n$. In such setting, the second term of the trade-off becomes $\log \frac{w - \log n}{a} = \log((\gamma - 1)\log n / \log(\lceil \log \frac{u}{n} \rceil + 2) = \mathcal{O}(\log\log n)$. This proves that y-fast tries and van Emde Boas trees are optimal for static Predecessor queries within the Elias-Fano space bound. However, such bound only depends on n, whereas the plain Elias-Fano bound for Predecessor of $\mathcal{O}(1 + \log \frac{u}{n})$, introduced in Subsection 2.2, depends on both n and u. On the other hand, the relation $u = n^{\gamma}$ relates the two parameter by means of the constant $\gamma = \Theta(1)$. It is clear that varying γ one of the two bounds becomes optimal. Indeed, comparing $1 + \log \frac{u}{n}$ with $\log\log n$, we have that $1 + \log \frac{u}{n} \leq \log\log n$ whenever $u \leq \frac{n}{2} \log n$, i.e., when $n^{\gamma - 1} \leq \frac{1}{2} \log n$. From this last condition we derive that the plain Elias-Fano is faster than van Emde Boas whenever $1 \leq \gamma \leq 1 + \frac{\log\log n}{\log n}$. In this case the static Elias-Fano representation does *not* need to be augmented. When, instead, $\gamma > 1 + \frac{\log\log n}{\log n}$, the query time $\mathcal{O}(\log\log n)$ is optimal and *exponentially better* than plain Elias-Fano. Therefore, $\mathcal{O}(\min\{1 + \log \frac{u}{n}, \log\log n\})$ is an accurate characterization of the Predecessor time bound.

We are left to describe a data structure matching the bound of $\mathcal{O}(\log\log n)$, within $o(n)$ bits of additional space. We divide $\mathcal{S}$ into $\lceil n / \log^2 u \rceil$ blocks of $\log^2 u$ integers each (the last block may contain less integers). We can solve Predecessor queries in a block in $\mathcal{O}(\log\log u) = \mathcal{O}(\log\log n)$ time by applying binary search. Now, we need a data structure on top of $\mathcal{S}$ that allows us to identify the proper block in $\mathcal{O}(\log\log n)$ time. Call the first element of a block its lower bound. We attach to $\mathcal{S}$ an y-fast trie storing the lower bounds of the blocks. More precisely, each leaf in the y-fast trie holds the lower bound of a block and its position in $\mathcal{S}$. The integers stored in the y-fast trie are $\lceil n / \log^2 u \rceil$, therefore its space is $\mathcal{O}(\frac{n}{\log^2 u} \log u) = o(n)$ bits. To identify the block where the predecessor of x lies in, we answer a partial Predecessor(x) query among the integers stored in the y-fast trie in $\mathcal{O}(\log\log n)$ worst-case time. The position p in $\mathcal{S}$ of the block's lower bound, associated to the identified partial answer, indicates that the search must continue in the block $\mathcal{S}[p, \min\{p + \log^2 u, n\})$.

Concluding this section, observe that the time bound for Predecessor queries is always at most $\mathcal{O}(\log\log n)$ except when $1 \leq \gamma \leq 1 + \frac{\log\log n}{\log n}$: in this case, the plain Elias-Fano representation beats the time bound of $\mathcal{O}(\log\log n)$. Therefore, in what follows we report the bound as $\mathcal{O}(\min\{1+\log\frac{u}{n}, \log\log n\})$ but discuss the case for $\gamma > 1 + \frac{\log\log n}{\log n}$.

4 Extensible Elias-Fano Representation

When the integers are inserted in sorted order, we obtain an efficient *extensible* representation as these can only be added at the end of the sequence by means of an Append operation. This is a scenario of practical interest as it is the operational setting of append-only inverted indexes, e.g., the one of Twitter [6].

▶ **Theorem 2.** *There exists a data structure representing an ordered set $\mathcal{S}(n, u)$ of n integers, drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes* $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ *bits of space and supports:* Append *in $\mathcal{O}(1)$ amortized,* Access *in $\mathcal{O}(1)$ worst-case and* Predecessor/Successor *queries in optimal $\mathcal{O}(\min\{1+\log\frac{u}{n}, \log\log n\})$ worst-case time.*

We maintain an array B of size m in which integers are appended uncompressed. This array acts as a buffer, which is periodically encoded with Elias-Fano in $\Theta(m)$ time and dumped, so that new integers can be successfully appended. Each compressed representation of the buffer is appended in an array of blocks encoded with Elias-Fano. More precisely, when B becomes full we encode with Elias-Fano its corresponding *differential* buffer, i.e., the buffer whose values are $B[i] - B[0]$, $0 \le i < m$. Each time the buffer is compressed, we append in another array C the pair $\langle \mathsf{base}, \mathsf{low\_bits} \rangle = \langle B[0], \lceil \log(B[m-1]/m) \rceil \rangle$, i.e., the buffer lower bound value and the number of bits needed to encode the average gap of the Elias-Fano representation of the buffer.

Apart from the space taken by the compressed blocks, the space of the data structure is given by the following contributions:

- $(m+1)\log u$ bits for the buffer B of uncompressed integers and its size;
- $\mathcal{O}(\lceil \frac{n}{m} \rceil \log n)$ bits for pointers to rank/select data structures, low and high bit arrays;
- $\mathcal{O}(\lceil \frac{n}{m} \rceil \log u)$ bits for the array C.

Summing up, the redundancy is $\mathcal{O}((m+1+\lceil \frac{n}{m} \rceil)\log u)$ bits. We use a buffer of size $m = \log^2 u$ and, as done in Section 3, we index the buffer lower bounds in an y-fast trie. More precisely, each leaf of the fast trie stores a buffer lower bound and the index of the compressed block to which the lower bound belongs to. The values stored in the y-fast trie are $\lceil n/\log^2 u \rceil$, thus requiring $o(n)$ bits of space. The redundancy $\mathcal{O}((m+1+\lceil \frac{n}{m} \rceil)\log u)$ becomes $o(n)$ bits for $n = \omega(\log^3 u)$, which is already satisfied by requiring that $\gamma = \Theta(1)$.

To take into account the space taken by the representation of the blocks, we use the property that splitting a block encoded with Elias-Fano into two sub-blocks *never* increases the cost of representation of the block. This is possible because each sub-block can be encoded with a universe *relative* to the sub-block, which is smaller than the original block universe, by subtracting to each integer the lower bound of the sub-block. The following property can be easily extended to work with an arbitrary number of splits.

▶ **Property 1.** *Consider a monotone sequence $\mathcal{S}$ of n integers. Let $\mathcal{S}[i, j)$ indicate the range of $\mathcal{S}$ delimited by endpoints i and j. Then for any i, k and j such that $0 \le i < k < j < n$, we have* $\mathsf{EF}(\mathcal{S}[i, k)) + \mathsf{EF}(\mathcal{S}[k, j)) \le \mathsf{EF}(\mathcal{S}[i, j))$.

Proof. Let m and u be respectively size and universe of the sub-sequence $\mathcal{S}[i, j)$, and, similarly, let m_1, m_2, u_1, u_2 be the sizes and universes of the two sub-sequences $\mathcal{S}[i, k)$ and $\mathcal{S}[k, j)$ respectively. We have that $m = m_1 + m_2$ and $u = u_1 + u_2$. From Subsection 2.2, we know that $\mathsf{EF}(\mathcal{S}[i, j))$ takes $m\phi + m + \lceil \frac{u}{2^\phi} \rceil$. Similarly $\mathsf{EF}(\mathcal{S}[i, k)) = m_1\phi_1 + m_1 + \lceil \frac{u_1}{2^{\phi_1}} \rceil$

and $\mathsf{EF}(\mathcal{S}[k,j]) = m_2\phi_2 + m_2 + \lceil \frac{u_2}{2^{\phi_2}} \rceil$. $\mathsf{EF}(\mathcal{S}[i,k])$ and $\mathsf{EF}(\mathcal{S}[k,j])$ are minimized by setting $\phi_1 = \lfloor \log \frac{u_1}{m_1} \rfloor$ and $\phi_2 = \lfloor \log \frac{u_2}{m_2} \rfloor$ respectively [10], therefore, by replacing ϕ_1 and ϕ_2 with ϕ, we have that $\mathsf{EF}(\mathcal{S}[i,k]) + \mathsf{EF}(\mathcal{S}[k,j]) \leq m_1\phi + m_2\phi + m_1 + m_2 + \lceil \frac{u_1}{2^\phi} \rceil + \lceil \frac{u_2}{2^\phi} \rceil = m\phi + m + \lceil \frac{u}{2^\phi} \rceil$ $= \mathsf{EF}(\mathcal{S}[i,j])$. ◀

The operations are supported as follows. Since we compress the buffer each time it fills up (by taking $\Theta(m)$ time), Append is performed in $\mathcal{O}(1)$ amortized time. Appending new integers in the buffer accumulates a credit of $\mathcal{O}(\log^2 u)$ which largely pays the amortized cost $\mathcal{O}(\log\log u)$ of inserting a buffer lower bound into the y-fast trie. To Access the i-th integer, we retrieve the element x in position $i - jm$ from the compressed block of index $j = \lfloor \frac{i}{m} \rfloor$. This is done in $\mathcal{O}(1)$ worst-case time, since we know how many low bits are required to perform the access by reading $C[j]$.low_bits. We finally return the integer $x + C[j]$.base. Predecessor queries are supported similarly as in the description of Theorem 1. Given the integer x, we first resolve a partial Predecessor(x) query in the y-fast trie to identify the index j of the compressed block in which the predecessor is located. Then we return $C[j]$.base + Predecessor($x - C[j]$.base) by binary searching the block of index j in $\mathcal{O}(\log\log u) = \mathcal{O}(\log\log n)$ worst-case time.

From Theorem 2, the following corollary easily follows.

▶ **Corollary 3.** *There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of $n = \omega(\log^2 u)$ integers drawn from a universe of size u that takes* $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ *bits of space and supports* Append *and* Access *operations in $\mathcal{O}(1)$ worst-case time.*

Without using the y-fast trie we are able to achieve a worst-case running time for the Append operation in Corollary 3 by using a classical de-amortization argument (note, however, that Predecessor queries are not supported in optimal time anymore). We maintain two buffers, B_1 and B_2, instead of one. When one is full we use the other to store the elements that must be appended. Suppose B_1 is full. For each of the successive m Append operations, we compress one element from B_1 and append the new integer in B_2. These two steps require $\mathcal{O}(1)$ worst-case time each.

5 Dynamic Elias-Fano Representation

In this section we describe how the static Elias-Fano representation can be turned into an efficient *dynamic* data structure, i.e., supporting Access, Insert, Delete, Minimum, Maximum, Predecessor and Successor in optimal time and taking $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space.

As already discussed in Subsection 2.3, Fredman and Saks [13] proved that $\mathcal{O}(\frac{\log n}{\log\log n})$ amortized time is optimal for any data structure maintaining a set of integers subject to Access, Insert and Delete (list representation problem). Their result holds when $w \leq \log^\gamma n$ for some γ, which covers the case of polynomial universes $u = n^\gamma$ since $\gamma \leq \log^{\gamma-1} n$, for any $\gamma \geq 1$ and $n \geq 2$. We operate, therefore, in the same setting as Theorems 1 and 2, considering integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$. In this setting, Pătraşcu and Thorup [21] showed that $\mathcal{O}(\log\log n)$ query time of y-fast tries and van Emde Boas trees is optimal for the dynamic predecessor problem too.

▶ **Theorem 4.** *There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of n integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes* $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ *bits of space and supports:* Access *in $\mathcal{O}(\log n/\log\log n)$ worst-case;* Insert/Delete *in $\mathcal{O}(\log n/\log\log n)$ amortized;* Minimum/Maximum *in $\mathcal{O}(1)$ and* Predecessor/Successor *queries in $\mathcal{O}(\min\{1 + \log\frac{u}{n}, \log\log n\})$ worst-case time. These time bounds are optimal.*

In what follows, we first describe the layout of the data structure and then analyze its space and time complexities.

5.1 Data Structure Description

We begin our description by showing how to handle a dynamic collection of mini blocks in succinct space, which is a key tool to obtain the full dynamic data structure. This result builds on an idea from [19], Appendix A.1.

5.1.1 Maintaining a Sorted Collection of Mini Blocks

Let $\mathcal{C}$ be a collection of $k = \mathcal{O}(\text{polylog}\, n)$ blocks of sorted integers, with the following properties. The blocks of $\mathcal{C}$ form a *total order*, i.e., $u_j \leq f_{j+1}$, for all $j = 1, \ldots, k-1$, where f_j and u_j indicate, respectively, the first and last element of the j-th block in the total order. Each block supports random access to its elements in constant time and is of size $\Theta(b) = \rho b$ with $\frac{1}{2} \leq \rho \leq 2$ and $b = \mathcal{O}(\text{polylog}\, n)$.

▶ **Lemma 5.** *The total order of the blocks of $\mathcal{C}$ can be maintained by using a data structure that takes $\mathcal{O}(\text{polylog}\, n \cdot \log\log n)$ bits of space and supports the following operations in $\mathcal{O}(\log\log n)$ worst-case time:* Search(x) *which returns a pointer to the block containing the integer x;* Access(i) *which returns the i-th integer of the total order;* Insert/Delete *of a block.*

Pointers of $\mathcal{O}(\log\log n)$ bits to blocks are stored in the leaves of a τ-ary tree $\mathcal{T}$, with $\tau = \Theta(\log^\sigma n)$ for some $0 < \sigma < 1$. Given that we have $\mathcal{O}(\text{polylog}\, n)$ leaves, the height of $\mathcal{T}$ is constant and equal to $\mathcal{O}(1/\sigma)$. $\mathcal{T}$ operates as a B-tree, in which internal nodes have $\Theta(\tau) = \rho\tau$ children.

Logically, we divide the information stored at each *internal node* into two levels of representation. For each of the two levels we store $\Theta(\tau)$ pairs, where the i-th pair maintains information about the sub-tree rooted in the i-th child. The pairs are stored following the order of the upper bounds of the blocks indexed in the sub-trees rooted in the node's children. In the lower level, each pair contains a pointer to the sub-tree rooted in the child and the size of such sub-tree. The $\Theta(\tau)$ children sizes are kept in prefix sums to enable binary search. In the upper level, each pair contains a pointer to the *right-most* block indexed in the sub-tree rooted in its child and the size of such sub-tree. Each *leaf* holds, of course, only the lower level of information. Each node uses $\mathcal{O}(\tau(\log\log n + \log \text{polylog}\, n)) = \mathcal{O}(\tau \log\log n) = o(\log n)$ bits, thus fitting in (less than) a machine word. The space taken by whole data structure is, therefore, $\mathcal{O}(\tau^{\mathcal{O}(1/\sigma)} \log\log n) = \mathcal{O}(\text{polylog}\, n \cdot \log\log n)$ bits.

We now detail how the operations are implemented. To support Search(x), i.e., determining the block where the integer x is comprised, we percolate $\mathcal{T}$, locating the correct child at each node in $\mathcal{O}(\log \tau) = \mathcal{O}(\log\log n)$ by binary searching on blocks' upper bounds. Specifically, if the upper bounds of the i-th block is needed for comparison for some $1 \leq i \leq \Theta(\tau)$, we access the block following the pointer (to the right-most block) of the i-th pair stored in the upper level of the node and we retrieve the upper bound in $\mathcal{O}(1)$, given that we also know the size of the block. When we have to insert/delete an integer, we identify the proper block of the total order in/from which the integer must be inserted/deleted in $\mathcal{O}(\log\log n)$ time (as described for the Search operation) and update the pairs along the path from the root in constant time, as these pairs fits in $o(\log n)$ bits overall. If a split or merge of a block happens, it is handled as usual and solved in a constant number of $\mathcal{O}(1)$-time operations. During an Access(i) query, we follow the proper root-to-leaf path in $\mathcal{T}$. The traversal of the data structure does not need to access the blocks directly, but instead uses their sizes to determine the correct child at each level. By binary searching the sizes, we traverse the data structure in $\mathcal{O}(\log\log n)$ time. During the traversal of the path we also compute the sum Δ of the sizes of the preceding blocks by summing to the current value of Δ, at each level, the value stored in the $(j-1)$-th pair of the lower level if the j-th child is traversed. Finally we retrieve the $(i-\Delta)$-th integer from the identified block in $\mathcal{O}(1)$, as the blocks of $\mathcal{C}$ support random access.

5.1.2 Full Data Structure Layout

Let ℓ be $\log n/\log\log n$ for the rest of the paper. We logically divide the sorted sequence $\mathcal{S}(n,u)$ into mini blocks of $\Theta(\ell) = \rho\ell$ integers each. We organize the dynamic layout into two levels.

Lower level. We group $\mathcal{O}(\log^2 n)$ consecutive mini blocks together and index such collection using the data structure $\mathcal{T}$ described in Lemma 5. We refer to this collection as a "block" and say that $\mathcal{T}$ stores a block of $\mathcal{O}(\log^2 n)$ mini blocks. The set $\{\mathcal{T}_j\}_{j=1}^{k'}$, with $k' = n/\mathcal{O}(\ell\log^2 n)$, of all such data structures forms the *lower level* of the dynamic layout. Each $\mathcal{T}_j$ also stores the lower bound f_j of its block and the number of low bits required by its Elias-Fano representation in $\Theta(\log u)$ bits, so that we can subtract f_j to all the integers belonging to the mini blocks of $\mathcal{T}_j$.

Upper level. The set $\{f_j\}_{j=1}^{k'}$ of all the lower bounds of the blocks are indexed using an y-fast trie. The sizes of the blocks are maintained, instead, using the dynamic prefix sums data structure described in [5], which is a B-tree in which each node stores a dynamic prefix sums data structure operating on a small set of integers in $\mathcal{O}(1)$ time. In particular we use the operation $\mathsf{Update}(i,\Delta)$ of $\mathcal{P}$ as implemented in [5], which sums to the i-th integer of the data structure the quantity Δ (that fits in δ bits) and runs in optimal $\mathcal{O}(\log n/\log(w/\delta))$ worst-case time. In our setting this operation is supported in $\mathcal{O}(\ell)$ given that $\delta = \Delta = 1$.

These two data structures, respectively named $\mathcal{Y}$ and $\mathcal{P}$ in the following, form the *upper level* of the dynamic layout. The j-th leaf of $\mathcal{Y}$ and $\mathcal{P}$ stores a $\mathcal{O}(\log n)$-bit pointer to the data structure $\mathcal{T}_j$ in the lower level.

To handle the memory allocation for the mini blocks, we employ a different technique to manage the high and low part of their Elias-Fano representation. Recall from Subsection 2.2 that, given a sequence $\mathcal{S}(n,u)$, the high part of $\mathsf{EF}(\mathcal{S}(n,u))$ consists in a bitvector of at most $2n$ bits, whereas the low part is given by a vector of n $\lceil\log\frac{u}{n}\rceil$-bit integers. In our case, the high part of each mini block requires at most $2\ell = \mathcal{O}(w)$ bits and is stored using the data structure of Theorem 6 from [17] that allows to address and allocate the high part of a mini block in $\mathcal{O}(1)$ worst-case time. The low part of a mini block is instead stored using the data structure of Corollary 3 from [24] that supports Access in $\mathcal{O}(1)$ and $\mathsf{Insert}/\mathsf{Delete}$ in $\mathcal{O}(\ell^\epsilon)$ worst-case time for any fixed positive $\epsilon < 1$.

5.2 Space Analysis

The space required by the introduced layout will be clearly given by the contribution of:

- the data structures $\mathcal{Y}$ and $\mathcal{P}$ used in the upper level and the data structures $\mathcal{T}$ of Lemma 5 used in the lower level;
- the cost of representation of the mini blocks encoded with Elias-Fano;
- the overhead given by the mini blocks memory management.

In the following we separately analyze each contribution.

The space taken by the data structures $\mathcal{Y}$ and $\mathcal{P}$ in the upper level is $\mathcal{O}(\frac{n}{\ell \log^2 n} \log u) = o(n)$ bits. All the data structures $\mathcal{T}$ of Lemma 5 require $\mathcal{O}(\frac{n}{\ell \log^2 n} \log^2 n \log\log n) = o(n)$ bits too.

We now analyze the space taken by the encoding of the mini blocks. Since the universe of representation of a mini block could be as large as the one of its comprising block, i.e., u, storing the lower bounds of the mini blocks in order to use reduced universes (as already done for the blocks), would require $\mathcal{O}(\frac{n}{\ell} \log u)$ bits, which is too much. In what follows we show that it is not necessary to re-map the mini blocks using Property 1, hence these are kept encoded with the universe relative to their comprising block, if we carefully set the number of bits required to represent each *low part* in the Elias-Fano space bound (2). As pointed out previously, each low part in the Elias-Fano representation of a sequence $\mathcal{S}(n, u)$ is encoded using $\lceil \log \frac{u}{n} \rceil$ bits, which is the number of bits needed to encode the average gap u/n of $\mathcal{S}$. The number of bits for the average gap of a block is therefore $\lceil m \rceil = \lceil \log \frac{u}{\ell \log^2 n} \rceil$.
The idea is to choose a number of bits $\lceil m' \rceil$ for the encoding of the average gap of the mini blocks such that $\lceil m' \rceil = \lceil m \rceil$ for a sufficiently long sequence of p insertions/deletions. After p insertions/deletions have been performed, we rebuild the mini blocks using $\lceil m \rceil$ bits for the average gap. In other words, we want to guarantee that encoding the mini blocks with $\lceil m' \rceil$ bits for the average gap, instead of $\lceil m \rceil$, does *not* introduce any extra space. Since m' lies in the interval $[l, r] = [\log \frac{u}{\ell \log^2 n + p}, \log \frac{u}{\ell \log^2 n - p}]$, m' must be chosen in order to satisfy $\lceil m \rceil - 1 < m' < \lceil m \rceil$, which indeed implies $\lceil m' \rceil = \lceil m \rceil$. Precisely, we satisfy this condition by fixing $m' = m \pm \theta$ with $\lceil m \rceil - l < \pm\theta < \lceil m \rceil - r + 1$. To derive this condition, we distinguish three possible cases.

1. $[l, r] \subset [\lceil m \rceil - 1, \lceil m \rceil)$. In this case the condition $\lceil m \rceil - 1 < m' < \lceil m \rceil$ is already satisfied. The other two cases are symmetric.
2. $\lceil l \rceil = \lceil m \rceil - 1$. In this case we set $m' = m + \theta$. To let $\lceil m \rceil - 1 < m' < \lceil m \rceil$ holds, θ must be at least $\lceil m \rceil - l$ and at most $\lceil m \rceil + 1 - r$.
3. $\lceil r \rceil = \lceil m \rceil + 1$. In this case we set $m' = m - \theta$. To let $\lceil m \rceil - 1 < m' < \lceil m \rceil$ holds, θ must be at least $r - \lceil m \rceil - 1$ and at most $l - \lceil m \rceil$.

Cases 2. and 3. together yield the condition $\lceil m \rceil - l < \pm\theta < \lceil m \rceil - r + 1$.
Finally, we have to determine the proper number p of insertions/deletions before triggering the rebuilding of the mini blocks in order to attain to optimal insert/delete amortized time $\mathcal{O}(\ell)$. As blocks are of size $\Theta(\ell \log^2 n)$, p is chosen to be $\mathcal{O}(\log^2 n)$.

The techniques used to manage the memory allocation for the mini blocks introduce an overall redundancy of $o(n)$ bits. Precisely, the data structure of Theorem 6 from [17] has an overhead of $\mathcal{O}(w^4 + \frac{n}{\log n} \log^2 w) = o(n)$ bits, while the one of Corollary 3 from [24] uses $o(n)$ bits by choosing a proper positive $\epsilon < 1$.

In conclusion, by the above discussion and the use of Property 1, the space taken by the mini blocks can be safely upper bounded by $\mathsf{EF}(\mathcal{S}(n, u))$ and the redundancy sums up to $o(n)$ bits, so that the whole data structure requires $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ bits of space.

5.3 Operations

In this subsection we describe how the operations of Theorem 4 are implemented. As stated before, ℓ is a short-hand for $\log n / \log\log n$.
To Access the i-th integer, we first resolve $\mathsf{Search}(i)$ on $\mathcal{P}$ in $\mathcal{O}(\ell)$: $\mathsf{Search}(i) = j$ indicates that the j-th block contains the i-th integer given that $\mathsf{Sum}(j-1) < i \leq \mathsf{Sum}(j)$, where $\mathsf{Sum}(j)$ equals the sum of the sizes of the first j blocks. We then follow the pointer stored in

the j-th leaf of $\mathcal{P}$, which points to the data structure $\mathcal{T}_j$. We finally Access the integer x of index $i - \mathsf{Sum}(j-1)$ from $\mathcal{T}_j$ in $\mathcal{O}(\log\log n)$ and return $x + f_j$. The overall complexity is, therefore, $\mathcal{O}(\ell)$. To $\mathsf{Insert}/\mathsf{Delete}$ an integer x, we perform the following steps: 1. identify the proper data structure $\mathcal{T}_j$ by resolving a partial $\mathsf{Successor}(x)$ query on $\mathcal{Y}$ in $\mathcal{O}(\log\log n)$ and following the pointer retrieved at the identified leaf of $\mathcal{Y}$; 2. identify the correct mini block by $\mathsf{Search}(x - f_j)$ in $\mathcal{T}_j$ in $\mathcal{O}(\log\log n)$; 3. $\mathsf{Insert}/\mathsf{Delete}$ $x - f_j$ in $\mathcal{T}_j$ by rebuilding the proper mini block in $\Theta(\ell)$; 4. update $\mathcal{P}$ in $\mathcal{O}(\ell)$. During the third step, split or merge of a mini block can happen and it is handled in $\mathcal{O}(\ell)$ worst-case time by the data structure $\mathcal{T}_j$; rebuilding of the mini blocks can happen as pointed out in the previous section and it is handled in $\mathcal{O}(\ell)$ amortized time. If split/merge of a block happens, the lower bound of the block is inserted/removed from $\mathcal{Y}$ in $\mathcal{O}(\log\log n)$ time. The overall complexity is, therefore, $\mathcal{O}(\ell)$ amortized. The query $\mathsf{Predecessor}(x)$ is supported as follows ($\mathsf{Successor}(x)$ runs in a similar way). We identify the proper data structure $\mathcal{T}_j$ in $\mathcal{O}(\log\log n)$ by answering a partial $\mathsf{Predecessor}(x)$ query on $\mathcal{Y}$ and following the pointer retrieved at the identified leaf of $\mathcal{Y}$. Then we identify the proper mini block by $\mathsf{Search}(x - f_j)$ in $\mathcal{T}_j$ in $\mathcal{O}(\log\log n)$ time. We finally return $f_j + \mathsf{Predecessor}(x - f_j)$ by binary searching on the identified mini block. The overall complexity is $\mathcal{O}(\log\log n)$ worst-case. The minimum and maximum elements of $\mathcal{S}$ are stored uncompressed using $\Theta(\log u)$ bits and returned when requested in $\mathcal{O}(1)$. Upon insertion/deletions these are updated as needed.

6 Conclusions

In this paper we have shown how the Elias-Fano representation of a monotone integer sequence $\mathcal{S}$ can be adapted to obtain optimal data structures in terms of query time and almost optimal in terms of space. In particular, when integers are drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, our data structures take the same asymptotic space of the plain, static, Elias-Fano representation, i.e., $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits and support: 1. static $\mathsf{Predecessor}/\mathsf{Successor}$ queries in optimal worst-case time $\mathcal{O}(\min\{1 + \log\frac{u}{n}, \log\log n\})$ (Section 3); 2. a $\mathcal{O}(1)$ worst-case amount of work for Append when integers are inserted in sorted order (Section 4); 3. Access in optimal $\mathcal{O}(\log n/\log\log n)$ worst-case time, $\mathsf{Insert}/\mathsf{Delete}$ in optimal $\mathcal{O}(\log n/\log\log n)$ amortized time, $\mathsf{Predecessor}/\mathsf{Successor}$ queries in optimal $\mathcal{O}(\min\{1 + \log\frac{u}{n}, \log\log n\})$ worst-case time (Section 5).

As a last note, we observe that the data structure described in Section 5 allows us to support all operations in time $\mathcal{O}(\log\log u)$ when *non*-polynomial universes are considered, i.e., when n and u are not necessarily related by means of the formula $u = n^\gamma$ for any $\gamma = \Theta(1)$. In this setting, the data structure of Lemma 5 will take $\mathcal{O}(\operatorname{polylog} u \cdot \log\log u)$ bits and operate in $\mathcal{O}(\log\log u)$ time. In order to guarantee an overall redundancy of $o(n)$ bits, we let mini blocks be of size $\Theta((\log\log u)^2)$ and group $\mathcal{O}(\log^2 u)$ consecutive mini blocks into a block. The high part of a mini block fits into one machine word, whereas we can insert/delete a low part in $\mathcal{O}((\log\log u)^{2\epsilon})$ for Corollary 3 of [24], which is $\mathcal{O}(\log\log u)$ as soon as $\epsilon < \frac{1}{2}$. Therefore, the following corollary matches the asymptotic time bounds of y-fast tries and van Emde Boas trees but in almost optimally compressed space.

▶ **Corollary 6.** *There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of n integers drawn from a universe of size u that takes* $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ *bits of space and supports:* Access *and* $\mathsf{Predecessor}/\mathsf{Successor}$ *queries in $\mathcal{O}(\log\log u)$ worst-case;* $\mathsf{Insert}/\mathsf{Delete}$ *in $\mathcal{O}(\log\log u)$ amortized and* $\mathsf{Minimum}/\mathsf{Maximum}$ *in $\mathcal{O}(1)$.*

References

1 Miklós Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica*, 8(3):235–247, 1988. `doi:10.1007/BF02126797`.

2 Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007. `doi:10.1145/1236457.1236460`.

3 Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem. In Jeffrey Scott Vitter, Lawrence L. Larmore, and Frank Thomson Leighton, editors, *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC 1999)*, pages 295–304. ACM, 1999. `doi:10.1145/301250.301323`.

4 Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002. `doi:10.1006/jcss.2002.1822`.

5 Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic relative compression, 2015. `arXiv:1504.07851`.

6 Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. Earlybird: Real-time search at Twitter. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE 2012)*, pages 1360–1369. IEEE Computer Society, 2012. `doi:10.1109/ICDE.2012.149`.

7 David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996. URL: `http://hdl.handle.net/10012/64`.

8 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. URL: `http://mitpress.mit.edu/books/introduction-algorithms`.

9 Erik D. Demaine and Mihai Pătraşcu. Tight bounds for the partial-sums problem. In J. Ian Munro, editor, *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 20–29. SIAM, 2004. URL: `http://dl.acm.org/citation.cfm?id=982792.982796`.

10 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974. `doi:10.1145/321812.321820`.

11 Robert Mario Fano. On the number of bits required to implement an associative memory. Technical Report Memorandum 61, Computer Structures Group, MIT, Cambridge, MA, 1971. URL: `http://csg.csail.mit.edu/pubs/memos/Memo-61/Memo-61.pdf`.

12 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst-case access time. *J. ACM*, 31(3):538–544, 1984. `doi:10.1145/828.1884`.

13 Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC 1989)*, pages 345–354. ACM, 1989. `doi:10.1145/73007.73040`.

14 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. `doi:10.1016/0022-0000(93)90040-4`.

15 Roberto Grossi, Rajeev Raman, Srinivasa Rao Satti, and Rossano Venturini. Dynamic compressed strings with random access. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP 2013)*, volume 7965 of *LNCS*, pages 504–515. Springer, 2013. `doi:10.1007/978-3-642-39206-1_43`.

16 Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.*, 387(3):313–331, 2007. `doi:10.1016/j.tcs.2007.07.042`.

17 Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. CRAM: Compressed random access memory. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *Proceedings of the 39th International Colloquium on Automata, Languages, and Programming (ICALP 2012)*, volume 7391 of *LNCS*, pages 510–521. Springer, 2012. doi:10.1007/978-3-642-31594-7_43.

18 Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theor. Comput. Sci.*, 387(3):332–347, 2007. doi:10.1016/j.tcs.2007.07.013.

19 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. In Sanjeev Khanna, editor, *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2013)*, pages 865–876. SIAM, 2013. doi:10.1137/1.9781611973105.62.

20 Giuseppe Ottaviano and Rossano Venturini. Partitioned Elias-Fano indexes. In Shlomo Geva, Andrew Trotman, Peter Bruza, Charles L. A. Clarke, and Kalervo Järvelin, editors, *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2014)*, pages 273–282. ACM, 2014. doi:10.1145/2600428.2609615.

21 Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In Jon M. Kleinberg, editor, *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC 2006)*, pages 232–240. ACM, 2006. doi:10.1145/1132516.1132551.

22 Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007)*, pages 555–564. SIAM, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283443.

23 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In Boaz Barak, editor, *Proceedings of the 55th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2014)*, pages 166–175. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.26.

24 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct dynamic data structures. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Proceedings of the 7th International Workshop on Algorithms and Data Structures (WADS 2001)*, volume 2125 of *LNCS*, pages 426–437. Springer, 2001. doi:10.1007/3-540-44634-6_39.

25 Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In Clifford Stein, editor, *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 1230–1239. SIAM, 2006. doi:10.1145/1109557.1109693.

26 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In Daniel J. Rosenkrantz, editor, *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS 1975)*, pages 75–84. IEEE Computer Society, 1975. doi:10.1109/SFCS.1975.26.

27 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977. doi:10.1016/0020-0190(77)90031-X.

28 Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977. doi:10.1007/BF01683268.

29 Sebastiano Vigna. Quasi-succinct indices. In Stefano Leonardi, Alessandro Panconesi, Paolo Ferragina, and Aristides Gionis, editors, *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM 2013)*, pages 83–92. ACM, 2013. doi:10.1145/2433396.2433409.

30 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

31 Andrew Chi-Chih Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, 1981. doi:10.1145/322261.322274.

Synergistic Solutions on MultiSets*

Jérémy Barbay[1], Carlos Ochoa[2], and Srinivasa Rao Satti[3]

1 **Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile**
jeremy@barbay.cl

2 **Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile**
cochoa@dcc.uchile.cl

3 **Department of Computer Science and Engineering, Seoul National University, Seoul, South Korea**
ssrao@cse.snu.ac.kr

Abstract

Karp et al. (1988) described Deferred Data Structures for Multisets as "lazy" data structures which partially sort data to support online rank and select queries, with the minimum amount of work in the worst case over instances of size n and number of queries q fixed. Barbay et al. (2016) refined this approach to take advantage of the gaps between the positions hit by the queries (i.e., the structure in the queries). We develop new techniques in order to further refine this approach and take advantage all at once of the structure (i.e., the multiplicities of the elements), some notions of local order (i.e., the number and sizes of runs) and global order (i.e., the number and positions of existing pivots) in the input; and of the structure and order in the sequence of queries. Our main result is a synergistic deferred data structure which outperforms all solutions in the comparison model that take advantage of only a subset of these features. As intermediate results, we describe two new synergistic sorting algorithms, which take advantage of some notions of structure and order (local and global) in the input, improving upon previous results which take advantage only of the structure (Munro and Spira 1979) or of the local order (Takaoka 1997) in the input; and one new multiselection algorithm which takes advantage of not only the order and structure in the input, but also of the structure in the queries.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems, H.3.3 Information Search and Retrieval

Keywords and phrases deferred data structure, multivariate analysis, quick sort, select

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.31

1 Introduction

Consider a *multiset* $\mathcal{M}$ of size n. The *multiplicity* of an element x of $\mathcal{M}$ is the number m_x of occurrences of x in $\mathcal{M}$. We call the distribution of the multiplicities of the elements in $\mathcal{M}$ the *input structure*. As early as 1976, Munro and Spira [19] described a variant of the algorithm `MergeSort` using counters, which optimally takes advantage of the input structure when sorting a multiset $\mathcal{M}$ of n elements. Munro and Spira measure the "difficulty" of the instance in terms of the "input structure" by the entropy function $\mathcal{H}(m_1, \ldots, m_\sigma) = \sum_{i=1}^{\sigma} \frac{m_i}{n} \log \frac{n}{m_i}$,

* C. Ochoa is supported by CONICYT-PCHA/Doctorado Nacional/2013-63130161 (Chile). J. Barbay is supported by the project Fondecyt Regular no 1170366 from Conicyt and the Millennium Nucleus RC130003 "Information and Coordination in Networks" with code P10-024-F.

where σ is the number of distinct elements in $\mathcal{M}$ and $m_1, \ldots, m_\sigma$ are the multiplicities of the σ distinct elements in $\mathcal{M}$ (such that $\sum_{i=1}^{\sigma} m_i = n$), respectively. The time complexity of the algorithm is within $O(n(1 + \mathcal{H}(m_1, \ldots, m_\sigma))) \subseteq O(n(1+\log\sigma)) \subseteq O(n \log n)$.

Any array $\mathcal{A}$ representing a multiset lists its element in some order, which we call the *input order* and denote by a tuple. Maximal sorted subblocks in $\mathcal{A}$ are a local form of input order and are called *runs* [16]. As early as 1973, Knuth [16] described a variant of the algorithm `MergeSort` using a prepossessing step taking linear time to detect *runs* in the array $\mathcal{A}$. Takaoka [20] described a new sorting algorithm that optimally takes advantage of the distribution of the sizes of the runs in the array $\mathcal{A}$, which yields a time complexity within $O(n(1 + \mathcal{H}(r_1, \ldots, r_\rho))) \subseteq O(n(1+\log\rho)) \subseteq O(n \log n)$, where ρ is the number of runs in $\mathcal{A}$ and $r_1, \ldots, r_\rho$ are the sizes of the ρ *runs* in $\mathcal{A}$ (such that $\sum_{i=1}^{\rho} r_i = n$), respectively.

Given an element x of a multiset $\mathcal{M}$ and an integer $j \in [1..n]$, the *rank* $\texttt{rank}(x)$ of x is the number of elements smaller than x in $\mathcal{M}$, and *selecting* the j-th element in $\mathcal{M}$ corresponds to computing the value $\texttt{select}(j)$ of the j-th smallest element (counted with multiplicity) in $\mathcal{M}$. Those operations are central to the navigation of the Burrows-Wheeler transform [17] of a text when searching for occurrences of a pattern in it. As early as 1961, Hoare [12] showed how to support `rank` and `select` queries in average linear time, a result later improved to worst case linear time by Blum et al. [7]. Twenty years later, Dobkin and Munro [10] described a MultiSelection algorithm that supports several `select` queries and whose running time is optimal in the worst case over all multisets of size n and all sets of q queries hitting positions in the multisets separated by *gaps* (differences between consecutive `select` queries in sorted order) of sizes $g_0, \ldots, g_q$. Karp et al. [15] further extended Dobkin and Munro's result [10] to the online context, where the multiple `rank` and `select` queries arrive one by one. They called their solution a Deferred Data Structure and describe it as "lazy", as it partially sorts data, performing the minimum amount of work necessary in the worst case over all instances for a fixed n and q. Barbay et al. [2] refined this result by taking advantage of the gaps between the positions hit by the queries (i.e., the *query structure*).

This suggests the following questions:

1. Is there a sorting algorithm for multisets which takes the best advantage of both its *input order* and its *input structure* in a synergistic way, so that it performs as good as previously known solutions on all instances, and much better on instances where it can take advantage of both at the same time?
2. Is there a multiselection algorithm and/or a deferred data structure for answering `rank` and `select` queries which takes the best advantage not only of both of those notions of easiness in the input, but also of notions of easiness in the queries, such as the *query structure* and the *query order*?

We answer both questions affirmatively: In the context of Sorting, this improves upon both algorithms from Munro and Spira [19] and Takaoka [20]. In the context of MultiSelection and Deferred Data Structure for `rank` and `select` on Multisets, this improves upon Barbay et al.'s results [2] by adding three new measures of difficulty (input order, input structure and query order) to the single one previously considered (query structure). Additionally, we correct the analysis of the `Sorted Set Union` algorithm by Demaine et al. [9] (Section 2.2), and we define a simple yet new notion of "global" input order (Section 2.4), not mentioned in previous surveys [11, 18] nor extensions [3].

We present our results incrementally, each building on the previous one, such that **the most complete and complex result is in Section 4**. In Section 2 we describe how to measure the interaction of the order (local and global) with the structure in the input, and two new synergistic Sorting algorithms based on distinct paradigms (i.e., merging vs splitting) which take advantage of both the input order and structure. We refine the second of those results in Section 3 with the analysis of a MultiSelection algorithm which takes advantage of not only the order and structure in the input, but also of the *query structure*, in the offline setting. In Section 4 we analyze an online Deferred Data Structure taking advantage of the order and structure in the input on one hand, and of the order and structure in the queries on the other hand. We conclude with a discussion of our results in Section 5.

2 Sorting Algorithms

We review in Section 2.1 the algorithms `MergeSort with Counters` described by Munro and Spira [19] and `Minimal MergeSort` described by Takaoka [20]: each takes advantage of distinct features in the input. In Sections 2.2 and 2.3, we describe two synergistic Sorting algorithms, which outperform both `MergeSort with Counters` and `Minimal MergeSort` by taking advantage of both the order (local and global) and the structure in the input, in a synergistic way.

2.1 Known Algorithms

The algorithm `MergeSort with Counters` described by Munro and Spira [19] is an adaptation of the traditional sorting algorithm `MergeSort` that optimally takes advantage of the input structure when sorting a multiset $\mathcal{M}$ of size n. The algorithm divides $\mathcal{M}$ into two parts of equal size, sorts both parts recursively, and then merges the two sorted lists. When two elements of same value v are found, one is discarded and a counter holding the number of occurrences of v is updated. Munro and Spira measure the "difficulty" of the instance in terms of the input structure by the entropy function $\mathcal{H}(m_1, \dots, m_\sigma) = \sum_{i=1}^{\sigma} \frac{m_i}{n} \log \frac{n}{m_i}$, where σ is the number of distinct elements in $\mathcal{M}$ and $m_1, \dots, m_\sigma$ are the multiplicities of the σ distinct elements in $\mathcal{M}$ (such that $\sum_{i=1}^{\sigma} m_i = n$), respectively. The time complexity of the algorithm is then within $O(n(1 + \mathcal{H}(m_1, \dots, m_\sigma))) \subseteq O(n(1+\log \sigma)) \subseteq O(n \log n)$.

The algorithm `Minimal MergeSort` described by Takaoka [20] optimally takes advantage of the local input order, as measured by the decomposition into runs when sorting an array $\mathcal{A}$ of size n. The main idea is to detect the runs first and then merge them pairwise. The runs are detected in linear time. Merging the two shortest runs at each step further reduces the number of comparisons, making the running time of the merging process adaptive to the entropy of the sequence formed by the sizes of the runs. If the array $\mathcal{A}$ is formed by ρ runs and $r_1, \dots, r_\rho$ are the sizes of the ρ runs (such that $\sum_{i=1}^{\rho} r_i = n$), then the algorithm sorts $\mathcal{A}$ in time within $O(n(1 + \mathcal{H}(r_1, \dots, r_\rho))) \subseteq O(n(1+\log \rho)) \subseteq O(n \log n)$.

The algorithms `MergeSort with Counters` and `Minimal MergeSort` are incomparable, in the sense that neither one performs always better than the other. Simple modifications and combinations of these algorithms do not take full advantage of both the local input order and the input structure (see the extended version [5] for detailed counter examples).

In the following sections we describe two sorting algorithms that take the best advantage of both the order (local and global) and structure in the input all at once when sorting a multiset. The first one is a straightforward application of previous results, while the second one prepares the ground for the MultiSelection algorithm (Section 3) and the Deferred Data Structures (Section 4), which take advantage of the order (local and global) and structure in the input and of the order and structure in the queries.

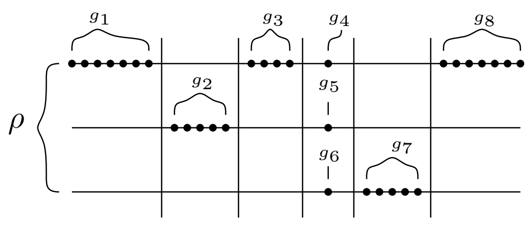


Figure 1 An instance of the Sorted Set Union problem with $\rho = 3$ sorted sets. In each sorted set $\mathcal{A}$, the entry $\mathcal{A}[i]$ is represented by a point of x-coordinate $\mathcal{A}[i]$. The sizes $(g_i)_{i\in[1..8]}$ of the blocks that form the sets are indicated. The sizes g_4, g_5 and g_6 are 1 because they correspond to elements of equal value and they determine the 4-th member of the partition π with value m_4 equals 3. The vertical bars separate the members of π.

2.2 "Kind-of-new" Sorting Algorithm `DLM Sort`

In 2000, Demaine et al. [9] described the algorithm `DLM Union`, an instance optimal algorithm that computes the union of ρ sorted sets. The algorithm scans the sets from left to right identifying *blocks* of consecutive elements in the sets that are also consecutive in the sorted union (see Figure 1 for a graphical representation of such a decomposition on a particular instance of the Sorted Set Union problem). In a minor way we refine their analysis as follows:

These blocks determine a partition π of the output into intervals such that any singleton corresponds to a value that has multiplicity greater than 1 in the input, and each other interval corresponds to a block as defined above. Each member i of π has a value m_i associated with it: if the member i of π is a block, then m_i is 1, otherwise, if the member i of π is a singleton corresponding to a value of multiplicity q, then m_i is q. If the instance is formed by δ blocks of sizes $g_1, \dots, g_\delta$ such that these blocks determine a partition π of size χ whose members have values $m_1, \dots, m_\chi$, we express the time complexity of `DLM Union` as within $\Theta(\sum_{i=1}^{\delta} \log g_i + \sum_{i=1}^{\chi} \log \binom{\rho}{m_i})$. This time complexity is within a constant factor of the complexity of any other algorithm computing the union of these sorted sets (i.e., the algorithm is instance optimal).

We adapt the `DLM Union` algorithm for sorting a multiset. The algorithm `DLM Sort` detects the runs first through a linear scan and then applies the algorithm `DLM Union`. After that, transforming the output of the union algorithm to yield the sorted multiset takes only linear time. The following corollary follows from our refined analysis above:

▶ **Corollary 1.** *Given a multiset $\mathcal{M}$ of size n formed by ρ runs and δ blocks of sizes $g_1, \dots, g_\delta$ such that these blocks determine a partition π of size χ of the output whose members have values $m_1, \dots, m_\chi$, the algorithm* `DLM Sort` *performs within $n + O(\sum_{i=1}^{\delta} \log g_i + \sum_{i=1}^{\chi} \log \binom{\rho}{m_i})$ data comparisons. This number of comparisons is optimal in the worst case over multisets of size n formed by ρ runs and δ blocks of sizes $g_1, \dots, g_\delta$ such that these blocks determine a partition π of size χ of the output whose members have values $m_1, \dots, m_\chi$.*

While the algorithm `DLM Sort` answers the Question 1 from Section 1, it does not yield a MultiSelection algorithm nor a Deferred Data Structure answering Question 2. In the following section we describe another sorting algorithm that also optimally takes advantage of the local order and structure in the input, but which is based on a distinct paradigm, more suitable to such extensions.

Algorithm 1 `Quick Synergy Sort`

```
Input: A multiset $\mathcal{M}$ of size $n$
Output: A sorted sequence of $\mathcal{M}$
1: Compute the $\rho$ runs of respective sizes $(r_i)_{i\in[1..\rho]}$ in $\mathcal{M}$ such that $\sum_{i=1}^{\rho} r_i = n$;
2: Compute the median $\mu$ of the middles of the runs, note $j \in [1..\rho]$ the run containing $\mu$;
3: Perform doubling searches for the value $\mu$ in all runs except the $j$-th, starting at both
   ends of the runs in parallel;
4: Find the maximum $\max_\ell$ (minimum $\min_r$) among the elements smaller (resp., greater)
   than $\mu$ in all runs except the $j$-th;
5: Perform doubling searches for the values $\max_\ell$ and $\min_r$ in the $j$-th run, starting at the
   position of $\mu$;
6: Recurse on the elements smaller than or equal to $\max_\ell$ and on the elements greater than
   or equal to $\min_r$.
```

2.3 New Sorting Algorithm `Quick Synergy Sort`

Given a multiset $\mathcal{M}$, the algorithm `Quick Synergy Sort` identifies the *runs* in linear time through a scanning process. It computes a pivot μ, which is the median of the set formed by the middle elements of each run, and partitions each *run* by μ. This partitioning process takes advantage of the fact that the elements in each *run* are already sorted. The insertion ranks of the pivots in the runs are identified by doubling searches [6]. It then recurses on the elements smaller than μ and on the elements greater than μ. (See Algorithm 1 for a more formal description).

▶ **Definition 2** (Median of the middles). Given a multiset $\mathcal{M}$ formed by runs, the "*median of the middles*" is the median element of the set formed by the middle elements of each run.

The number of data comparisons performed by the algorithm `Quick Synergy Sort` is asymptotically the same as the number of data comparisons performed by the algorithm `DLM Sort` described in the previous section. We divide the proof into two lemmas. We first bound the number of data comparisons performed by all the doubling searches of the algorithm `Quick Synergy Sort` (i.e., steps 3 and 5 of the Algorithm 1).

▶ **Lemma 3.** *Let $g_1, \dots, g_k$ be the sizes of the k blocks that form the r-th run. The overall number of data comparisons performed by the doubling searches of the algorithm `Quick Synergy Sort` to find the values of the medians of the middles in the r-th run is within $O(\sum_{i=1}^{k} \log g_i)$.*

Proof. Every time the algorithm finds the insertion rank of one of the medians of the middles in the r-th run, it partitions the run by a position separating two blocks. The doubling search steps can be represented as a tree. Each node of the tree corresponds to a step. Each internal node has two children, which correspond to the two subproblems into which the step partitions the run. The cost of the step is less than four times the logarithm of the size of the child subproblem with smaller size, because of the two doubling searches in parallel. The leaves of the tree correspond to the blocks themselves.

We prove that at each step the total cost is bounded by eight times the sum of the logarithms of the sizes of the leaf subproblems. This is done by induction over the number of steps. If the number of steps is zero then there is no cost. For the inductive step, if the number of steps increases by one, a new doubling search step is done and a leaf subproblem is partitioned into two new subproblems. At this step, a leaf of the tree is transformed into an

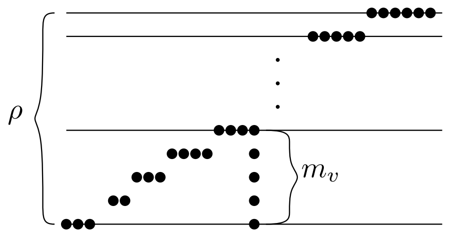


Figure 2 A multiset $\mathcal{M}$ formed by ρ runs. Each entry $\mathcal{M}[i]$ is represented by a point of x-coordinate $\mathcal{M}[i]$. There is an element of multiplicity m_v present in the last m_v runs and the rest of the runs are formed by only one block.

internal node and two new leaves are created. Let a and b such that $a \leq b$ be the sizes of the new leaves created. The cost of this step is less than $4 \log a$. The cost of all the steps then increases by $4 \lg a$, and hence the sum of the logarithms of the sizes of the leaves increases by $8(\lg a + \lg b) - 8 \lg(a + b)$. But if $a \geq 4$ and $b \geq a$, then $2 \lg(a + b) \leq \lg a + 2 \lg b$. The result follows. ◀

As shown in the following lemma, the overall number of data comparisons performed during the computation of the medians of the middles (i.e., step 2 of the Algorithm 1) is within $O(\sum_{i=1}^{\chi} \log \binom{\rho}{m_i})$, where $m_1, \dots, m_\chi$ are the values of the member of the partition π (see Section 2.2 for the definition of π) and ρ is the number of runs in $\mathcal{M}$.

Consider the instance depicted in Figure 2 for an example illustrating from where the term $\log \binom{\rho}{m_v}$ comes. In this instance, there is a value v that has multiplicity $m_v > 1$ in $\mathcal{M}$ and the rest of the values have multiplicity 1. The elements with value v are present at the end of the last m_v runs and the rest of the runs are formed by only one block. The elements of the i-th run are greater than the elements of the $(i+1)$-th run. During the computation of the medians of the middles, the number of data comparisons that involve elements of value v is within $O(\log \binom{\rho}{m_v})$. The algorithm computes the median μ of the middles and partitions the runs by the value of μ. In the recursive call that involves elements of value v, the number of runs is reduced by half. This is repeated until one occurrence of μ belongs to one of the last m_v runs. The number of data comparisons that involve elements of value v up to this step is within $O(m_v \log \frac{\rho}{m_v}) = O(\log \binom{\rho}{m_v})$, where $\log \frac{\rho}{m_v}$ corresponds to the number of steps where μ does not belong to the last m_v runs. The next recursive call will necessarily choose one element of value v as the median of the middles.

▶ **Lemma 4.** *Let $\mathcal{M}$ be a multiset formed by ρ runs and δ blocks such that these blocks determine a partition π of size χ of the output whose members have values $m_1, \dots, m_\chi$. Consider the steps that compute the medians of the middles and the steps that find the elements $\max_\ell$ and $\min_r$ in the algorithm `Quick Synergy Sort`, the overall number of data comparisons performed during these steps is within $O(\sum_{i=1}^{\chi} \log \binom{\rho}{m_i})$.*

Proof. We prove this lemma by induction over the size χ of π and the number of runs ρ. The number of data comparisons performed by one of these steps is linear in the number of runs in the sub-instance (i.e., ignoring all the empty sets of this sub-instance). Let $\mathcal{T}(\pi, \rho)$ be the overall number of data comparisons performed during the steps 2 and 4 of the algorithm `Quick Synergy Sort`. We prove that $\mathcal{T}(\pi, \rho) \leq \sum_{i=1}^{\chi} m_i \log \frac{\rho}{m_i} - \rho$. Let μ be the first median of the middles computed by the algorithm. Let ℓ and r be the number of runs that are completely to the left and to the right of μ, respectively. Let b be the number of runs that are split in the doubling searches for the value of μ

in all runs. Let π_ℓ and π_r be the partitions determined by the blocks yielded to the left and to the right of μ, respectively. Then, $\mathcal{T}(\pi, \rho) = \mathcal{T}(\pi_\ell, \ell + b) + \mathcal{T}(\pi_r, r + b) + \rho$ because of the two recursive calls and the step that computes μ. By Induction Hypothesis, $\mathcal{T}(\pi_\ell, \ell + b) \le \sum_{i=1}^{\chi_\ell} m_i \log \frac{\ell+b}{m_i} - \ell - b$ and $\mathcal{T}(\pi_r, r + b) \le \sum_{i=1}^{\chi_r} m_i \log \frac{r+b}{m_i} - r - b$. Hence, we need to prove that $\ell + r \le \sum_{i=1}^{\chi_\ell} m_i \log \left(1 + \frac{r}{\ell+b}\right) + \sum_{i=1}^{\chi_r} m_i \log \left(1 + \frac{\ell}{r+b}\right)$, but this is a consequence of $\sum_{i=1}^{\chi_\ell} m_i \ge \ell + b, \sum_{i=1}^{\chi_r} m_i \ge r + b$ (the number of blocks is greater than or equal to the number of runs); $\ell \le r + b, r \le \ell + b$ (at least $\frac{\rho}{2}$ runs are left to the left and to the right of μ); and $\log \left(1 + \frac{y}{x}\right)^x \ge y$ for $y \le x$. ◀

Consider the step that performs doubling searches for the values $\max_\ell$ and $\min_r$ in the run that contains the median μ of the middles, this step results in the finding of the block g that contains μ in at most $4 \log |g|$ data comparisons, where $|g|$ is the size of g. Combining Lemma 3 and Lemma 4 yields an upper bound on the number of data comparisons performed by the algorithm `Quick Synergy Sort`:

▶ **Theorem 5.** *Let $\mathcal{M}$ be a multiset of size n formed by ρ runs and δ blocks of sizes $g_1, \dots, g_\delta$ such that these blocks determine a partition π of size χ of the output whose members have values $m_1, \dots, m_\chi$. The algorithm `Quick Synergy Sort` performs within $n + O(\sum_{i=1}^{\delta} \log g_i + \sum_{i=1}^{\chi} \log \binom{\rho}{m_i})$ data comparisons on $\mathcal{M}$. This number of comparisons is optimal in the worst case over multisets of size n formed by ρ runs and δ blocks of sizes $g_1, \dots, g_\delta$ such that these blocks determine a partition π of size χ of the output whose members have values $m_1, \dots, m_\chi$.*

We extend these results to take advantage of the global order of the multiset in a way that can be combined with the notion of runs (local order).

2.4 Taking Advantage of Global Order

Given a multiset $\mathcal{M}$, a *pivot position* is a position p in $\mathcal{M}$ such that all elements in previous position are smaller than or equal to all elements at p or in the following positions. In 1962, Iverson [13] described an improved version of `BubbleSort` [16] that identifies such pivot positions (as pair of consecutive elements that the algorithm have placed at their final positions and on which it does not make further comparisons). We show that detecting such positions also yields an improved version of `QuickSort` in general, and of our `QuickSort`-inspired solutions in particular. More formally:

▶ **Definition 6** (Pivot positions). Given a multiset $\mathcal{M} = (x_1, \dots, x_n)$ of size n, the "pivot positions" are the positions p such that $x_a \le x_b$ for all a, b such that $a \in [1..p-1]$ and $b \in [p..n]$.

Existing pivot positions in the input order of $\mathcal{M}$ divide the input into subsequences of consecutive elements such that the range of positions of the elements at each subsequence coincide with the range of positions of the same elements in the sorted sequence of $\mathcal{M}$: the more there are of such positions, the more "global" order there is in the input. Detecting such positions takes only a linear number of comparisons by applying the first phase of the algorithm `BubbleSort` [16], which sequentially compares the elements, from left to right in a first phase and then from right to left in a second phase. The positions of the elements that do not interchange their values during both executions are the pivot positions in $\mathcal{M}$.

When there are ϕ such positions, they simply divide the input of size n into $\phi + 1$ sub-instances of sizes $n_0, \dots, n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$). Each sub-instance I_i for $i \in [0..\phi]$ then has its own number of runs r_i and alphabet size σ_i, on which the synergistic solutions described in this work can be applied, from mere SORTING (Section 2) to supporting MULTISELECTION (Section 3) and the more sophisticated DEFERRED DATA STRUCTURES (Section 4).

▶ **Corollary 7.** *Let $\mathcal{M}$ be a multiset of size n with ϕ pivot positions. Let $n_0, \dots, n_\phi$ be integers such that the ϕ pivot positions divide $\mathcal{M}$ into $\phi + 1$ sub-instances of sizes $n_0, \dots, n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$). Let ρ_i and δ_i be such that each sub-instance I_i of size n_i is formed by ρ_i runs and δ_i blocks of sizes $g_{i1}, \dots, g_{i\delta_i}$ such that these blocks determine a partition π_i of size χ_i of the output whose members have values $m_{i1}, \dots, m_{i\chi_i}$ for $i \in [0..\phi]$. There exists an algorithm that performs within $3n + O(\sum_{i=0}^{\phi} \left\{\sum_{j=1}^{\delta_i} \log g_{ij} + \sum_{j=1}^{\chi_i} \log \binom{\rho_i}{m_{ij}}\right\})$ data comparisons for sorting $\mathcal{M}$. This number of comparisons is optimal in the worst case over multisets of size n with ϕ pivot positions which divide the multiset into $\phi + 1$ sub-instances of sizes $n_0, \dots, n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$) and each sub-instance I_i of size n_i is formed by ρ_i runs and δ_i blocks of sizes $g_{i1}, \dots, g_{i\delta_i}$ such that these blocks determine a partition π_i of size χ_i of the output whose members have values $m_{i1}, \dots, m_{i\chi_i}$ for $i \in [0..\phi]$.*

Next, we generalize the algorithm `Quick Synergy Sort` to an offline multiselection algorithm that partially sorts a multiset according to the set of `select` queries given as input. This serves as a pedagogical introduction to the online DEFERRED DATA STRUCTURES for answering `rank` and `select` queries presented in Section 4.

3 MultiSelection Algorithm

Given a linearly ordered multiset $\mathcal{M}$ and a sequence of ranks $r_1, \dots, r_q$, a multiselection algorithm must answer the queries `select`(r_1), ..., `select`(r_q) in $\mathcal{M}$, hence partially sorting $\mathcal{M}$. We describe a MULTISELECTION algorithm based on the sorting algorithm `Quick Synergy Sort` introduced in Section 2.3. This algorithm is an intermediate result leading to the DEFERRED DATA STRUCTURE described in Section 4.

Given a multiset $\mathcal{M}$ and a set of q `select` queries, the algorithm `Quick Synergy MultiSelection` follows the same first steps as the algorithm `Quick Synergy Sort`. But once it has computed the ranks of all elements in the block that contains the pivot μ, it determines which `select` queries correspond to elements smaller than or equal to $\max_\ell$ and which ones correspond to elements greater than or equal to $\min_r$ (see Algorithm 1 for the definitions of $\max_\ell$ and $\min_r$). It then recurses on both sides.

We extend the notion of blocks to the context of partial sorting. Next, we introduce the definitions of *pivot blocks* and *selection blocks* (see Figure 3 for a graphical representation of these definitions).

▶ **Definition 8** (Pivot Blocks). Given a multiset $\mathcal{M}$ formed by ρ runs and δ blocks. The "*pivot blocks*" are the blocks of $\mathcal{M}$ that contain the pivots and the elements of value equals to the pivots during the steps of the algorithm `Quick Synergy MultiSelection`.

In each run, between the pivot blocks and the insertion ranks of the pivots, there are consecutive blocks that the algorithm `Quick Synergy MultiSelection` has not identified as separated blocks, because no doubling searches occurred inside them.

▶ **Definition 9** (Selection Blocks). Given the i-th run, formed of various blocks, and q `select` queries, the algorithm `Quick Synergy MultiSelection` computes ξ pivots in the process of answering the q queries. During the doubling searches, the algorithm `Quick Synergy MultiSelection` finds the insertion ranks of the ξ pivots inside the i-th run. These positions

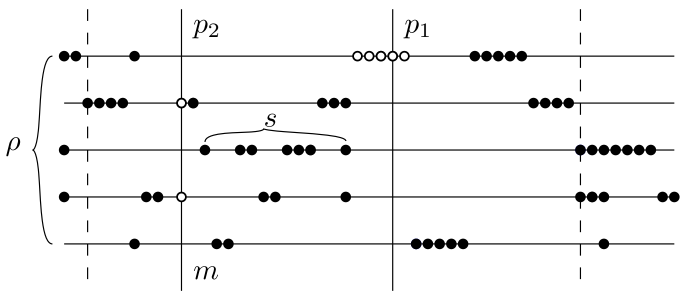


Figure 3 An instance of the MultiSelection problem where the multiset $\mathcal{M}$ is formed by $\rho = 5$ runs. In each run $\mathcal{R}$, the entry $\mathcal{R}[i]$ is represented by a point of x-coordinate $\mathcal{R}[i]$. The dash lines represent the answers of the two `select` queries. The solid vertical lines represent the positions p_1 and p_2 of the first two pivots computed by the `Quick Synergy MultiSelection` algorithm. The pivot blocks corresponding to the pivots p_1 and p_2 are marked by contiguous open disks. The algorithm divides the runs into selection blocks. $s = 7$ is the size of the second selection block, from left to right, into which the third run is divided by the algorithm. $m = 2$ is the number of pivot blocks of size 1 corresponding to the pivot p_2.

determine a partition of size $\xi + 1$ of the i-th run where each element of the partition is formed by consecutive blocks or is empty. We call the elements of this partition "*selection blocks*". The set of all selection blocks contains the set of all pivot blocks.

Using these definitions, we generalize the results proven in Section 2.3 to the more general problem of MultiSelection.

▶ **Theorem 10.** *Given a multiset $\mathcal{M}$ of size n formed by ρ runs and δ blocks; and q offline* `select` *queries over $\mathcal{M}$ corresponding to elements of* `ranks` *$r_1, \dots, r_q$. Let ξ be the number of pivots computed by the algorithm* `Quick Synergy MultiSelection` *in the process of answering the q queries. Let $s_1, \dots, s_\beta$ be the sizes of the β selection blocks determined by these ξ pivots in all runs. Let $m_1, \dots, m_\lambda$ be the numbers of pivot blocks corresponding to the values of the λ pivots with multiplicity greater than 1, respectively. Let $\rho_0, \dots, \rho_\xi$ be the sequence where ρ_i is the number of runs that have elements with values between the pivots i and $i+1$ sorted by* `ranks`*, for $i \in [1..\xi]$. The algorithm* `Quick Synergy MultiSelection` *answers the q* `select` *queries performing within $n + O\left(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i - \sum_{i=0}^{\xi} \rho_i \log \rho_i\right) \subseteq O\left(n \log n - \sum_{i=0}^{q} \Delta_i \log \Delta_i\right)$ data comparisons, where $\Delta_i = r_{i+1} - r_i$, $r_0 = 0$ and $r_{q+1} = n$.*

Proof. The pivots computed by the algorithm `Quick Synergy MultiSelection` for answering the queries are a subset of the pivots computed by the algorithm `Quick Synergy Sort` for sorting the whole multiset. Suppose that the selection blocks determined by every two consecutive pivots form a multiset $\mathcal{M}_j$ such that for every pair of selection blocks in $\mathcal{M}_j$ the elements of one are smaller than the elements of the other one. The algorithm `Quick Synergy Sort` would perform within $n + O\left(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i\right)$ data comparisons in this supposed instance (see the proof of Lemmas 3 and 4 analyzing the algorithm `Quick Synergy Sort` for details). The number of comparisons needed to sort the multisets $\mathcal{M}_j$ is within $\Theta(\sum_{i=0}^{\xi} \rho_i \log \rho_i)$. The result follows. ◀

The process of detecting the ϕ pre-existing pivot positions seen in Section 2.4 can be applied as the first step of the multiselection algorithm. The ϕ pivot positions divide the input of size n into $\phi + 1$ sub-instances of sizes $n_0, \dots, n_\phi$. For each sub-instance I_i for $i \in [0..\phi]$, the multiselection algorithm determines which `select` queries correspond to I_i

and applies then the steps of the algorithm `Quick Synergy MultiSelection` inside I_i in order to answer these queries.

The `Quick Synergy MultiSelection` algorithm takes advantage of the number and sizes of the runs (i.e., the local input order), the number and positions of the pre-existing pivot positions (i.e., the global order), the multiplicities of the elements in the multiset (i.e., the input structure) and the differences between consecutive `select` queries in sorted order (i.e., the query structure).

In the result above, the queries are given all at the same time (i.e., offline). In the context where they arrive one at the time (i.e., online), we define a Deferred Data Structure for answering online `rank` and `select` queries, inspired by the algorithm `Quick Synergy MultiSelection`.

4 `Rank` and `Select` Deferred Data Structures

We describe the Full-Synergistic Deferred Data Structure that answers a set of `rank` and `select` queries, arriving one at the time, over a multiset $\mathcal{M}$, progressively sorting $\mathcal{M}$. This deferred data structure is based in the `Quick Synergy MultiSelection` algorithm described in the previous section. This data structure takes advantage of the order (local and global) and structure in the input, and of the order and structure in the queries.

By "query order", we mean to consider the "distances" between consecutive queries. To take advantage of the query order, we introduce a data structure that finds the nearest pivots to the left and to the right of a position $p \in [1..n]$, while taking advantage of the *distance* between the position of the last computed pivot and p, as measured by the number of computed pivot blocks between the two positions. For that we use a *finger search tree* [8] maintaining *fingers* (i.e., pointers) to elements in the search tree and supporting efficient updates and searches in the vicinity of those. Brodal [8] described an implementation of finger search trees that searches for an element x, starting the search at the element given by the finger f in time within $O(\log d)$, where d is the distance between x and f in the set (i.e, the difference between $\texttt{rank}(x)$ and $\texttt{rank}(f)$ in the set). This operation returns a finger to x if x is contained in the set, otherwise a finger to the largest element smaller than x in the set. This implementation supports the insertion of an element x immediately to the left or to the right of a finger in worst-case constant time.

Given a multiset $\mathcal{M}$ of size n, the Full-Synergistic Deferred Data Structure includes a finger search tree $\mathcal{F}_{\texttt{select}}$, in which it marks the elements in $\mathcal{M}$ that have been computed as pivots when it answers the online queries. For each pivot p in $\mathcal{F}_{\texttt{select}}$, the data structure stores pointers to the insertion ranks of p in each run, to the beginning and to the end of the block g to which p belongs, and to the position of p inside g. This finger search tree is also used to find the two successive pivots between which the query fits.

Once a pivot block g is computed, every element in g is a valid pivot for the rest of the elements in $\mathcal{M}$. In order to capture this idea, we modify the finger search tree $\mathcal{F}_{\texttt{select}}$ so that it contains the pivot blocks (i.e., a sequence of consecutive values) instead of singleton pivots. This modification allows the Full-Synergistic Deferred Data Structure to answer `select` queries, taking advantage of the structure and order in the queries and of the structure and order in the input. But in order to answer `rank` queries taking advantage of the features in the queries and the input, the data structure needs another finger search tree $\mathcal{F}_{\texttt{rank}}$. In $\mathcal{F}_{\texttt{rank}}$ the data structure stores, for each block g identified, the value of one of the elements in g, and pointers in $\mathcal{M}$ to the beginning and to the end of g, and in each run to the position where the elements of g partition the run.

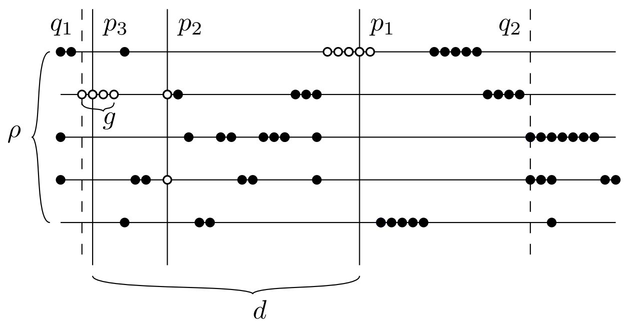


Figure 4 The state of the FULL-SYNERGISTIC DEFERRED DATA STRUCTURE on an instance where the multiset $\mathcal{M}$ is formed by $\rho = 5$ runs. In each run, the entry $\mathcal{M}[i]$ is represented by a point of x-coordinate $\mathcal{M}[i]$. The dash lines represent the positions q_1 and q_2 of the answers of the first two queries. The solid vertical lines represent the positions p_1, p_2 and p_3 of the first three pivots computed by the FULL-SYNERGISTIC DEFERRED DATA STRUCTURE. The pivot blocks corresponding to the pivots p_1, p_2 and p_3 are marked by contiguous open disks. $d = 4$ is the distance (i.e., the number of computed pivot blocks) between the queries q_1 and q_2. If q_1 is a `rank` query, then $g = 4$ is the size of the identified block that contains the answer of the query q_1.

▶ **Theorem 11.** *Consider a multiset $\mathcal{M}$ of size n formed by ρ runs and δ blocks. Let γ and $r_1, \dots, r_q$ be the number of pivot blocks computed by the* FULL-SYNERGISTIC DEFERRED DATA STRUCTURE *in the process of answering q online* `rank` *and* `select` *queries over $\mathcal{M}$, and the* `ranks` *of the elements corresponding to these queries, respectively. Let $s_1, \dots, s_\beta$ be the sizes of the β selection blocks determined by the pivots in the γ blocks in all runs. Let $m_1, \dots, m_\lambda$ be the numbers of pivot blocks corresponding to the values of the λ pivots with multiplicity greater than 1, respectively. Let $\rho_0, \dots, \rho_\gamma$ be the sequence where ρ_i is the number of runs that have elements with values between the elements in the blocks i and $i+1$ sorted by* `ranks`*, for $i \in [1..\gamma]$. Let $d_1, \dots, d_{q-1}$ be the sequence where d_j is the number of computed pivot blocks between the block that answers the $(j-1)$-th query and the one that answers the j-th query before starting the steps to answer the j-th query, for $j \in [2..q]$. Let u and $g_1, \dots, g_u$ be the number of* `rank` *queries and the sizes of the computed and searched pivot blocks in the process of answering the u* `rank` *queries, respectively. The* FULL-SYNERGISTIC DEFERRED DATA STRUCTURE *answers the q online queries by performing within $n + O(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i - \sum_{i=0}^{\gamma} \rho_i \log \rho_i + \sum_{i=1}^{q-1} \log d_i + \sum_{i=1}^{u} \log g_i) \subseteq O\left(n \log n - \sum_{i=0}^{q} \Delta_i \log \Delta_i + q \log n\right)$ data comparisons, where $r_0 = 0$, $r_{q+1} = n$, and $\Delta_i = r_{i+1} - r_i$, for all $i \in [1..n]$.*

Proof. The algorithm answers a new `select`(i) query by searching in $\mathcal{F}_{\texttt{select}}$ for the nearest pivots to the left and right of the query position i. If i is contained in an element of $\mathcal{F}_{\texttt{select}}$, then the block g that contains the element in the position i has already been computed. If i is not contained in an element of $\mathcal{F}_{\texttt{select}}$, then the returned finger f points the nearest block b to the left of i. The block that follows f in $\mathcal{F}_{\texttt{select}}$ is the nearest block to the right of i. It then applies the same steps as the algorithm `Quick Synergy MultiSelection` in order to answer the query. Given f, the algorithm inserts in $\mathcal{F}_{\texttt{select}}$ each pivot block computed in the process of answering the query in constant time, and stores the respective pointers to positions in $\mathcal{M}$. In $\mathcal{F}_{\texttt{rank}}$ the algorithm searches for the value of one of the elements in b. Once the algorithm obtains the finger returned by this search, the algorithm inserts in $\mathcal{F}_{\texttt{rank}}$ the value of one of the elements of each pivot block in constant time, and stores the respective pointers to positions in $\mathcal{M}$ (see Figure 4 for a graphical representation of some of the parameters used in the analysis).

The algorithm answers a new `rank`(x) query by finding the *selection block* s_j in the j-th run such that x is between the smallest and the greatest value of s_j for all $j \in [1..\rho]$. For that the algorithm searches for the value x in $\mathcal{F}_{\texttt{rank}}$. The number of data comparisons performed by this searching process is within $O(\log d)$, where d is the number of blocks in $\mathcal{F}_{\texttt{rank}}$ between the last inserted or searched block and the returned finger f. Given the finger f, there are three possibilities for the `rank` r of x: (i) r is between the `ranks` of the elements at the beginning and at the end of the block pointed by f, (ii) r is between the `ranks` of the elements at the beginning and at the end of the block pointed by the finger following f, or (iii) r is between the `ranks` of the elements in the selection blocks determined by f and the finger following f. In the cases (i) and (ii), a binary search inside the block yields the answer of the query. In the case (iii), the algorithm applies the same steps as the algorithm `Quick Synergy MultiSelection` in order to compute the median μ of the middles, and partitions the selection blocks by μ. The algorithm then decides to which side x belongs. Similar to the algorithm for answering a `select` query, the data structure inserts in $\mathcal{F}_{\texttt{select}}$ every block computed in the process of answering the `rank` query. ◀

The process of detecting the ϕ pivot positions seen in Section 2.4 allows the FULL-SYNERGISTIC DEFERRED DATA STRUCTURE to insert these pivots in $\mathcal{F}_{\texttt{select}}$ and $\mathcal{F}_{\texttt{rank}}$. For each pivot position p in $\mathcal{F}_{\texttt{select}}$ and $\mathcal{F}_{\texttt{rank}}$, the data structure stores pointers to the end of the runs detected on the left of p; to the beginning of the runs detected on the right of p; and to the position of p in the multiset. This concludes the description of our synergistic results. In the next section, we discuss how these results relate to various past results and future work.

5 Discussion

Kaligosi et al.'s multiselection algorithm [14] and Barbay et al's deferred data structure [2] use the very same concept of *runs* as the one described in this work. The difference is, we describe algorithms that *detect* the existing runs in the input in order to take advantage of them, while the algorithms described by those previous works do not take into consideration any pre-existing runs in the input, and rather build and maintain such runs as a strategy to minimize the number of comparisons performed while partially sorting the multiset. We leave the combination of both approaches as a topic for future work, which could probably shave a constant factor off the number of comparisons performed by the SORTING and MULTISELECTION algorithms and by the DEFERRED DATA STRUCTURES supporting `rank` and `select` queries on MULTISETS.

Barbay and Navarro [3] described how any SORTING algorithm taking advantage of specificities in the input, directly implies a compressed encoding for permutations. By using the similarity of the execution tree of the algorithm `MergeSort` with the `Wavelet Tree` data structure, they described a compressed data structure for permutations taking advantage of the local order, i.e., using space proportional to $\mathcal{H}(r_1, \ldots, r_\rho)$ and supporting `direct access` (i.e. $\pi()$) and `inverse access` (i.e. $\pi^{-1}()$) in worst time within $O(1 + \lg \rho)$ and average time within $O(1 + \mathcal{H}(r_1, \ldots, r_\rho))$. We leave as future work the extension of our work into a compressed data structure for multisets taking advantage of both its structure and (local and global) order.

Another perspective is to generalize the synergistic results to related problems in computational geometry: Karp et al. [15] defined the first deferred data structure not only to support `rank` and `select` queries on multisets, but also to support online queries in a deferred way on POINT MEMBERSHIP in a CONVEX HULL in two dimensions and online DOMINANCE queries on sets of multi-dimensional vectors. Preliminary results [4] show that one can refine the results from Karp et al. [15] to take advantage of the blocks between queries (i.e., the structure in the queries) as Barbay et al. [2] did for multisets; but also of the relative position of the points (i.e., the structure in the input) as Afshani et al. [1] did for CONVEX HULLS and MAXIMA; of the order in the points (i.e., the order in the input), as computing the convex hull in two dimensions takes linear time if the points are sorted; and potentially of the order in the queries.

References

1 Peyman Afshani, Jérémy Barbay, and Timothy M. Chan. Instance-optimal geometric algorithms. *J. ACM*, 64(1):3:1–3:38, March 2017. `doi:10.1145/3046673`.

2 Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jonathan Sorenson. Near-optimal online multiselection in internal and external memory. *J. Discrete Algorithms*, 36:3–17, 2016. `doi:10.1016/j.jda.2015.11.001`.

3 Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theor. Comput. Sci.*, 513:109–123, 2013. `doi:10.1016/j.tcs.2013.10.019`.

4 Jérémy Barbay and Carlos Ochoa. Synergistic computation of planar maxima and convex hull, 2017. `arXiv:1702.08545`.

5 Jérémy Barbay, Carlos Ochoa, and Srinivasa Rao Satti. Synergistic sorting and deferred data structures on multisets, August 2016. `arXiv:1608.06666`.

6 Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976. `doi:10.1016/0020-0190(76)90071-5`.

7 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. `doi:10.1016/S0022-0000(73)80033-9`.

8 Gerth S. Brodal. Finger search trees with constant insertion time. In Howard J. Karloff, editor, *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998)*, pages 540–549. ACM/SIAM, 1998. URL: `http://dl.acm.org/citation.cfm?id=314613.314842`.

9 Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In David B. Shmoys, editor, *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 743–752. ACM/SIAM, 2000. URL: `http://dl.acm.org/citation.cfm?id=338219.338634`.

10 David P. Dobkin and J. Ian Munro. Optimal time minimal space selection algorithms. *J. ACM*, 28(3):454–461, 1981. `doi:10.1145/322261.322264`.

11 Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, 1992. `doi:10.1145/146370.146381`.

12 Charles A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, 1961. `doi:10.1145/366622.366647`.

13 Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, 1962. URL: `http://www.softwarepreservation.org/projects/apl/Books/APROGRAMMING%20LANGUAGE`.

14 Kanela Kaligosi, Kurt Mehlhorn, J. Ian Munro, and Peter Sanders. Towards optimal multiple selection. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming (ICALP 2005)*, volume 3580 of *LNCS*, pages 103–114. Springer, 2005. `doi:10.1007/11523468_9`.

15 Richard Karp, Rajeev Motwani, and Prabhakar Raghavan. Deferred data structuring. *SIAM J. Comput.*, 17(5):883–902, 1988. `doi:10.1137/0217055`.

16 Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.

17 Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theor. Comput. Sci.*, 387(3):332–347, 2007. doi:10.1016/j.tcs.2007.07.013.

18 Alistair Moffat and Ola Petersson. An overview of adaptive sorting. *Aust. Comput. J.*, 24(2):70–77, 1992. URL: http://50years.acs.org.au/__data/assets/pdf_file/0017/111464/ACJ-V24-N02-199205.pdf.

19 J. Ian Munro and Philip M. Spira. Sorting and searching in multisets. *SIAM J. Comput.*, 5(1):1–8, 1976. doi:10.1137/0205001.

20 Tadao Takaoka. Partial solution and entropy. In Rastislav Královic and Damian Niwiński, editors, *Proceedings of the 34th International Symposium on Mathematical Foundations of Computer Science (MFCS 2009)*, volume 5734 of *LNCS*, pages 700–711. Springer, 2009. doi:10.1007/978-3-642-03816-7_59.