32nd Annual Symposium on Combinatorial Pattern Matching

CPM 2021, July 5–7, 2021, Wrocław, Poland

Edited by

Paweł Gawrychowski

Tatiana Starikovskaya

Editors

Paweł Gawrychowski
University of Wrocław, Poland
gawry@cs.uni.wroc.pl

Tatiana Starikovskaya
École normale supérieure, France
tat.starikovskaya@gmail.com

ACM Classification 2012
Theory of computation → Pattern matching

ISBN 978-3-95977-186-3

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-186-3.

Publication date
July, 2021

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 https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.CPM.2021.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Invited Talks

Regular Papers

32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021).
Editors: Paweł Gawrychowski and Tatiana Starikovskaya
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The Annual Symposium on Combinatorial Pattern Matching (CPM) has by now over 30 years of tradition and is considered to be the leading conference for the community working on Stringology. The objective of the annual CPM meetings is to provide an international forum for research in combinatorial pattern matching and related applications such as computational biology, data compression and data mining, coding, information retrieval, natural language processing, and pattern recognition.

This volume contains the papers presented at the 32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021) held on July 5–7, 2021 in Wrocław, Poland (in a hybrid mode due to the continuing Covid-19 pandemic). The conference program includes 22 contributed papers and three invited talks by Hideo Bannai (M&D Data Science Center, Tokyo Medical and Dental University, Japan), Michal Koucký (Computer Science Institute of Charles University, Czech Republic), and Nadia Pisanti (University of Pisa, Italy and Erable Team INRIA, France). For the third time, CPM includes the "Highlights of CPM" special session, for presenting the highlights of recent developments in combinatorial pattern matching. In this third edition we invited Travis Gagie (CeBiB – Center for Biotechnology and Bioengineering, Chile and Dalhousie University, Canada) to present a J. of ACM 2020 paper by T. Gagie, G. Navarro, N. Prezza "Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space" and Panagiotis Charalampopoulos (The Interdisciplinary Center Herzliya, Israel) to present a FOCS 2020 paper by P. Charalampopoulos, T. Kociumaka, P. Wellnitz "Faster Approximate Pattern Matching: A Unified Approach". The conference was preceded by a one-day student summer school taught by Jakub Radoszewski (University of Warsaw, Poland) and Martin Farach-Colton (Rutgers University, USA).

The contributed papers were selected out of 49 submissions, corresponding to an acceptance ratio of about 45%. Each submission received at least three reviews. We thank the members of the Program Committee and all the additional external subreviewers who are listed below for their hard, invaluable, and collaborative effort that resulted in an excellent scientific program.

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, Tel Aviv, Warsaw, Qingdao, Pisa, and Copenhagen. From 1992 to the 2015 meeting, all proceedings were published in the LNCS (Lecture Notes in Computer Science) series. Since 2016, the CPM proceedings appear in the LIPIcs (Leibniz International Proceedings in Informatics) series, as volume 54 (CPM 2016), 78 (CPM 2017), 105 (CPM 2018), 128 (CPM 2019), and 161 (CPM 2020). The entire submission and review process was carried out using the EasyChair conference system.

We thank the CPM Steering Committee for their support and advice.

32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021).
Editors: Paweł Gawrychowski and Tatiana Starikovskaya
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Programme Committee

Golnaz Badkobeh
Goldsmiths University of London, UK

Frédérique Bassino
University Paris 13, France

Christina Boucher
University of Florida, USA

Laurent Bulteau
CNRS and Université Paris-Est
Marne-la-Vallée, France

Raphaël Clifford
University of Bristol, UK

Fabio Cunial
MPI-CBG, Germany

Funda Ergun
Indiana University, USA

Paweł Gawrychowski (co-chair)
University of Wrocław, Poland

Inge Li Gørtz
Technical University of Denmark

Stepan Holub
Charles University in Prague, Czech Republic

Tomohiro I
Kyushu Institute of Technology, Japan

Shunsuke Inenaga
Kyushu University, Japan

Tomasz Kociumaka
University of California, Berkeley, USA

Christian Komusiewicz
Philipps-Universität Marburg, Germany

Dmitry Kosolobov
Ural Federal University, Russia

Gad M. Landau
University of Haifa, Israel

Florin Manea
University of Göttingen, Germany

Pierre Peterlongo
INRIA, France

Cinzia Pizzi
University of Padova, Italy

Leena Salmela
University of Helsinki, Finland

Srinivasa Rao Satti
Seoul National University, South Korea

Marinella Sciortino
University of Palermo, Italy

Braha-Riva Shalom
Shenkar College of Engineering and Design, Israel

Tatiana Starikovskaya (co-chair)
Ecole normale supérieure, France

Yasuo Tabei
RIKEN, Japan

Tomasz Waleń
University of Warsaw, Poland

32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021).
Editors: Paweł Gawrychowski and Tatiana Starikovskaya
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

External Subreviewers

Hideo Bannai

Jérémy Barbay

Jacopo Borga

Bastien Cazaux

Davide Cenzato

Rayan Chikhi

Julien Clément

Nadia El-Mabrouk

Gabriele Fici

Arnab Ganguly

Samah Ghazawi

Niels Grüttemeier

Diptarama Hendrian

Jana Holznigenkemper

Varunkumar Jayapaul

Seungbum Jo

Dominik Köppl

Shunsuke Kanda

Dominik Kempa

Gunnar W. Klau

Shmuel Tomi Klein

Tore Koß

Christian Konrad

Maria Kosche

Thierry Lecroq

Inbok Lee

Avivit Levy

Noa Lewenstein

Antoine Limasset

Zsuzsanna Liptak

Bertrand Marchand

Nils Morawietz

Yuto Nakashima

Cyril Nicaud

Takaaki Nishimoto

Taku Onodera

Kunsoo Park

Karol Pokorski

Nicola Prezza

Simon Puglisi

Jakub Radoszewski

Gwenaël Richomme

Giuseppe Romana

Massimiliano Rossi

Paweł Rychlikowski

Arseny Shur

Stefan Siemer

Marcin Smulewicz

Dina Sokol

Frank Sommer

Teresa Anna Steiner

Juliusz Straszyński

Xiaorui Sun

Michelle Sweering

Nathan Wallheimer

Mathias Weller

Samson Zhou

Wiktor Zuba

32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021).
Editors: Paweł Gawrychowski and Tatiana Starikovskaya
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Duncan Adamson
Amihood Amir
Golnaz Badkobeh
Hideo Bannai
Mukul S. Bansal
Djamal Belazzougui
Giulia Bernardini
Noah Bertram
Philip Bille
Itai Boneh
Laurent Bulteau
Bastien Cazaux
Panagiotis Charalampopoulos
Hwan-Gue Cho
Dustin Cobas
Argyrios Deligkas
Chen Ding
Riccardo Dondi
Travis Gagie
Daniel Gildea
Samuele Giraudo
Inge Li Gørtz
Vladimir Gusev
Meng He
Giuseppe F. Italiano
Juha Kärkkäinen
Serikzhan Kazi
Shahbaz Khan
Sung-Hwan Kim
Eitan Kondratovsky
Dominik Köppl
Dmitry Kosolobov
Alberto Marchetti-Spaccamela
Takuya Mieno
Fatemeh Nargesian
Gonzalo Navarro
Abhinav Nellore
Austin Nguyen
Takaaki Nishimoto
Sangsoo Park
Sung Gan Park
Kunsoo Park
Max Pedersen
Marcin Piątkowski
Solon Pissis
Alexandru Popa
Andrei Popa
Igor Potapov
Nicola Prezza
Simon Puglisi
Jakub Radoszewski
Rajeev Raman
Eric Rivals
Wojciech Rytter
Florian Sikora
Blerina Sinaimeri
Joshua Sobel
Teresa Anna Steiner
Leen Stougie
Michelle Sweering
Yasuo Tabei
Reid Thompson

32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021).
Editors: Paweł Gawrychowski and Tatiana Starikovskaya
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Rossano Venturini

Stéphane Vialette

Tomasz Walen

Keegan Yao

Wiktor Zuba

Repetitions in Strings: A "Constant" Problem

Hideo Bannai
M&D Data Science Center, Tokyo Medical and Dental University, Japan

Abstract

Repeating structures in strings is one of the most fundamental characteristics of strings, and has been an important topic in the field of combinatorics on words and combinatorial pattern matching since their beginnings. In this talk, I will focus on squares and maximal repetitions and review the "runs" theorem [1] as well as related results (e.g. [5, 6, 7, 3, 2, 4]) which address the two main questions: how many of them can be contained in a string of given length, and algorithms for computing them.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words

Keywords and phrases Maximal repetitions, Squares, Lyndon words

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.1

Category Invited Talk

Funding *Hideo Bannai*: Supported by JSPS KAKENHI Grant Number JP20H04141.

References

1 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

2 Hideo Bannai, Takuya Mieno, and Yuto Nakashima. Lyndon words, the three squares lemma, and primitive squares. In Christina Boucher and Sharma V. Thankachan, editors, *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 265–273. Springer, 2020. doi:10.1007/978-3-030-59212-7_19.

3 Philip Bille, Jonas Ellert, Johannes Fischer, Inge Li Gørtz, Florian Kurpicz, J. Ian Munro, and Eva Rotenberg. Space efficient construction of Lyndon arrays in linear time. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 14:1–14:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.14.

4 Jonas Ellert and Johannes Fischer. Linear time runs over general ordered alphabets. *CoRR*, abs/2102.08670, 2021. arXiv:2102.08670.

5 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.

6 Yuta Fujishige, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Almost linear time computation of maximal repetitions in run length encoded strings. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th International Symposium on Algorithms and Computation, ISAAC 2017, December 9-12, 2017, Phuket, Thailand*, volume 92 of *LIPIcs*, pages 33:1–33:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ISAAC.2017.33.

7 Ryo Sugahara, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing runs on a trie. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 23:1–23:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CPM.2019.23.

Computing Edit Distance

Michal Koucký
Computer Science Institute of Charles University, Prague, Czech Republic

Abstract

The edit distance (or Levenshtein distance) between two strings x, y is the minimum number of character insertions, deletions, and substitutions needed to convert x into y. It has numerous applications in various fields from text processing to bioinformatics so algorithms for edit distance computation attract lot of attention. In this talk I will survey recent progress on computational aspects of edit distance in several contexts: computing edit distance approximately, sketching and computing it in streaming model, exchanging strings in communication complexity model, and building error correcting codes for edit distance. I will point out many problems that are still open in those areas.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases edit distance, streaming algorithms, approximation algorithms, sketching

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.2

Category Invited Talk

32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021).
Editors: Paweł Gawrychowski and Tatiana Starikovskaya; Article No. 2; pp. 2:1–2:1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

On-Line Pattern Matching on D-Texts

Nadia Pisanti
University of Pisa, Italy

Abstract

The *Elastic Degenerate String Matching* (EDSM) problem is defined as that of finding an occurrence of a pattern P of length m in an ED-text T. A *D-text* (Degenerate text) is a string that actually represents a set of similar and aligned strings (e.g. a pan-genome [5]) by collapsing common fragments into a standard string, and representing variants with sets of alternative substrings. When such substrings are not bound to have the same size, then we talk about *elastic D-strings* (ED-strings). In [6] we gave an $O(nm^2 + N)$ time on-line algorithm for EDSM, where n is the length of T and N is its size, defined as the total number of letters. A fundamental toolkit of our algorithm is the $O(m^2+N)$ time solution of the later called *Active Prefixes* problem (AP). In [2], a $O(m^{1.5}\sqrt{\log m}+N)$ solution for AP was shown, leading to a $O(nm^{1.5}\sqrt{\log m} + N)$ time solution for EDSM. The natural open problem was thus whether the 1.5 exponent could furtherly be decreased. In [3], we prove several properties that answer this and other questions: we give a conditional $O(nm^{1.5} + N)$ lower bound for EDSM, proving that a combinatorial algorithm solving EDSM in $O(nm^{1.5-\epsilon} + N)$ time would break the Boolean Matrix Multiplication (BMM) conjecture; we use this result as a hint to devise a non-combinatorial algorithm that solves EDSM in $O(nm^{1.381} + N)$ time; we do so by successfully combining Fast Fourier Transform and properties of string periodicity.
In my talk I will overview the results above, as well as some interesting side results: the extension to a dictionary rather than a single pattern [7], the introduction of errors [4], and a notion of matching *among* D-strings with its linear time solution [1].

2012 ACM Subject Classification Mathematics of computing

Keywords and phrases pattern matching, elastic-degenerate string, matrix multiplication

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.3

Category Invited Talk

Funding *Nadia Pisanti*: This work is partially supported by the EU funding schemes H2020-MSCA-ITN-2020. Project ALPACA GA 956229, and by the University of Pisa funding scheme PRA (Progetti di Ricerca di Ateneo) Project no. PRA_2020-2021_26.

References

1 Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Comparing degenerate strings. *Fundam. Informaticae*, 175(1-4):41–58, 2020. doi:10.3233/FI-2020-1947.

2 Kotaro Aoyama, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Online Elastic Degenerate String Matching. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, volume 105 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:10, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2018.9.

3 Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Even Faster Elastic-Degenerate String Matching via Fast Matrix Multiplication. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2019.21.

4 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Approximate pattern matching on elastic-degenerate text. *Theor. Comput. Sci.*, 812:109–122, 2020. doi:10.1016/j.tcs.2019.08.012.

5 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings Bioinform.*, 19(1):118–135, 2018. doi:10.1093/bib/bbw089.

6 Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari. On-Line Pattern Matching on Similar Texts. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2017.9.

7 Solon P. Pissis and Ahmad Retha. Dictionary Matching in Elastic-Degenerate Texts with Applications in Searching VCF Files On-line. In Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms (SEA 2018)*, volume 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SEA.2018.16.

Ranking Bracelets in Polynomial Time

Duncan Adamson
Leverhulme Research Centre for Functional Materials Design, Department of Computer Science, University of Liverpool, UK

Vladimir V. Gusev
Leverhulme Research Centre for Functional Materials Design, Department of Computer Science, University of Liverpool, UK

Igor Potapov
Department of Computer Science, University of Liverpool, UK

Argyrios Deligkas
Department of Computer Science, Royal Holloway University of London, UK

Abstract

The main result of the paper is the first polynomial-time algorithm for ranking bracelets. The time-complexity of the algorithm is $O(k^2 \cdot n^4)$, where k is the size of the alphabet and n is the length of the considered bracelets. The key part of the algorithm is to compute the rank of any word with respect to the set of bracelets by finding three other ranks: the rank over all necklaces, the rank over palindromic necklaces, and the rank over enclosing apalindromic necklaces. The last two concepts are introduced in this paper. These ranks are key components to our algorithm in order to decompose the problem into parts. Additionally, this ranking procedure is used to build a polynomial-time unranking algorithm.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words

Keywords and phrases Bracelets, Ranking, Necklaces

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.4

Related Version *Full Version*: `https://arxiv.org/abs/2104.04324`

Funding *Igor Potapov*: Partially funded by EP/R018472/1 and Royal Society Leverhulme Trust Senior Research Fellowship.

Acknowledgements The authors thank the Leverhulme Trust for funding this research via the Leverhulme Research Centre for Functional Materials Design and the reviewers for their helpful comments that improved the quality of the paper.

1 Introduction

Counting, ordering, and generating basic discrete structures such as strings, permutations, set-partitions, etc. are fundamental tasks in computer science. A variety of such algorithms are assembled in the fourth volume of the prominent series "The art of computer programming" by D. Knuth [10]. Nevertheless, this research direction remains very active [8].

If the structures under consideration are linearly ordered, e.g. a set of words under the dictionary (lexicographic) order, then a unique integer can be assigned to every structure. The *rank* (or index) of a structure is the number of structures that are smaller than it. The *ranking* problem asks to compute the rank of a given structure, while the *unranking* problem corresponds to its reverse: compute the structure of a given rank. Ranking has been studied for various objects including partitions [19], permutations [13, 14], combinations [18], etc. Unranking has similarly been studied for objects such as permutations [14] and trees [7, 15].

1.	aaaaaaaa	7.	aaaababb	13.	aaabbabb	19.	aababbbb	25.	ababbabb
2.	aaaaaaab	8.	aaaabbbb	14.	aaabbbbb	20.	aabbaabb	26.	ababbbbb
3.	aaaaaabb	9.	aaabaaab	15.	aabaabab	21.	aabbabbb	27,	abbabbbb
4.	aaaaabab	10.	aaabaabb	16.	aabaabbb	22.	aabbbbbb	28.	abbbabbb
5.	aaaaabbb	11.	aaababab	17.	aabababb	23.	abababab	29.	abbbbbbb
6.	aaaabaab	12.	aaabbabb	18.	aababbab	24.	abababbb	30.	bbbbbbbb

Figure 1 List of all bracelets of length 8 over the alphabet $\{a, b\}$.

Both ranking and unranking are straightforward for the set of all words over a finite alphabet (assuming the standard lexicographic order), but they immediately cease to be so, as soon as additional symmetry is introduced. One of such examples is a class of necklaces [6]. A *necklace*, also known as a *cyclic word*, is an equivalence class of all words under the cyclic shift operation. Necklaces are classical combinatorial objects and they remain an object of study in other contexts such as total search problems [5] or circular splicing systems [4].

The *rank* of a word w for a given set S and its ordering is the number of words in S that are smaller than w. Often the set is a class of words, for instance all words of a given length over some alphabet. The first class of cyclic words to be ranked were *Lyndon words* - fixed length aperiodic cyclic words - by Kociumaka et. al. [11] who provided an $O(n^3)$ time algorithm. An algorithm for ranking necklaces - fixed length cyclic words - was given by Kopparty et. al. [12], without tight bounds on the complexity. A quadratic algorithm for ranking necklaces was provided by Sawada et al. [16].

This paper answers the open problem of ranking *bracelets*, posed by Sawada and Williams [16]. Bracelets are necklaces that are minimal under both *cyclic shifts* and *reflections*. Figure 1 provides an example of the ranks of length 8 bracelets over a binary alphabet. Bracelets have been studied extensively, with results for counting and generation in both the normal and fixed content cases [9, 17].

This paper presents the first algorithm for ranking bracelets of length n over an alphabet of size k in polynomial time, with a time complexity of $O(k^2 \cdot n^4)$. This algorithm is further used to unrank bracelets in $O(n^5 \cdot k^2 \cdot \log(k))$. time. These polynomial time algorithms improve upon the exponential time brute-force algorithm.

2 Preliminaries

2.1 Definitions and Notation

Let Σ be a finite alphabet. We denote by Σ^* the set of all words over Σ and by Σ^n the set of all words of length n. For the remainder of this paper, let $k = |\Sigma|$. The notation $\bar{w}$ is used to clearly denote that the variable w is a word. The length of a word $\bar{u} \in \Sigma^*$ is denoted $|\bar{u}|$. We use $\bar{u}_i$, for any $i \in \{1 \dots |\bar{u}|\}$ to denote the i^{th} symbol of $\bar{u}$. The *reversal* operation on a word $\bar{w} = \bar{w}_1\bar{w}_2 \dots \bar{w}_n$, denoted by $\bar{w}^R$, returns the word $\bar{w}_n \dots \bar{w}_2\bar{w}_1$.

In the present paper we assume that Σ is linearly ordered. Let $[n]$ return the ordered set of integers from 1 to n inclusive. Given two words $\bar{u}, \bar{v} \in \Sigma^*$ where $|\bar{u}| \leq |\bar{v}|$, $\bar{u} = \bar{v}$ if and only if $|\bar{u}| = |\bar{v}|$ and $\bar{u}_i = \bar{v}_i$ for every $i \in [|\bar{u}|]$. A word $\bar{u}$ is *lexicographically smaller* than $\bar{v}$ if there exists an $i \in [|\bar{u}|]$ such that $\bar{u}_1\bar{u}_2 \dots \bar{u}_{i-1} = \bar{v}_1\bar{v}_2 \dots \bar{v}_{i-1}$ and $\bar{u}_i < \bar{v}_i$. For example, given the alphabet $\Sigma = \{a, b\}$ where $a < b$, the word *aaaba* is smaller than *aabaa* as the first two symbols are the same and a is smaller than b. For a given set of words $\mathbf{S}$, the rank of $\bar{v}$ with respect to $\mathbf{S}$ is the number of words in $\mathbf{S}$ that are smaller than $\bar{v}$.

The *rotation* of a word $\bar{w} = \bar{w}_1\bar{w}_2\dots\bar{w}_n$ by $r \in [n-1]$ returns the word $\bar{w}_{r+1}\dots\bar{w}_n\bar{w}_1\dots\bar{w}_r$, and is denoted by $\langle\bar{w}\rangle_r$, i.e. $\langle\bar{w}_1\bar{w}_2\dots\bar{w}_n\rangle_r = \bar{w}_{r+1}\dots\bar{w}_n\bar{w}_1\dots\bar{w}_r$. Under the rotation operation, $\bar{u}$ is equivalent to $\bar{v}$ if $\bar{v} = \langle\bar{w}\rangle_r$ for some r. The t^{th} power of a word $\bar{w} = \bar{w}_1\dots\bar{w}_n$, denoted $\bar{w}^t$, is equal to $\bar{w}$ repeated t times. For example $(aab)^3 = aabaabaab$. A word $\bar{w}$ is *periodic* if there is some word $\bar{u}$ and integer $t \geq 2$ such that $\bar{u}^t = \bar{w}$. Equivalently, word $\bar{w}$ is *periodic* if there exists some rotation $0 < r < |\bar{w}|$ where $\bar{w} = \langle\bar{w}\rangle_r$. A word is *aperiodic* if it is not periodic. The *period* of a word $\bar{w}$ is the length of the smallest word $\bar{u}$ for which there exists some value t for which $\bar{w} = \bar{u}^t$.

A *cyclic word*, also called a *necklace*, is the equivalence class of words under the rotation operation. For notation, a word $\bar{w}$ is written as $\tilde{\mathbf{w}}$ when treated as a necklace. Given a necklace $\tilde{\mathbf{w}}$, the *necklace representative* is the lexicographically smallest element of the set of words in the equivalence class $\tilde{\mathbf{w}}$. The necklace representative of $\tilde{\mathbf{w}}$ is denoted $\langle\tilde{\mathbf{w}}\rangle$, and the r^{th} shift of the necklace representative is denoted $\langle\tilde{\mathbf{w}}\rangle_r$. The reversal operation on a necklace $\tilde{\mathbf{w}}$ returns the necklace $\tilde{\mathbf{w}}^R$ containing the reversal of every word $\bar{u} \in \tilde{\mathbf{w}}$, i.e. $\tilde{\mathbf{w}}^R = \{\bar{u}^R : \bar{u} \in \tilde{\mathbf{w}}\}$. Given a word $\bar{w}$, $\langle\bar{w}\rangle$ will denote the necklace representative of the necklace containing $\bar{w}$, i.e. the representative of $\tilde{\mathbf{u}}$ where $\bar{w} \in \tilde{\mathbf{u}}$.

A *subword* of the cyclic word $\bar{w}$, denoted $\bar{w}_{[i,j]}$ is the word $\bar{u}$ of length $|\bar{w}|+j-i-1 \bmod |\bar{w}||)$ such that $\bar{u}_a = \bar{w}_{i-1+a \bmod |\bar{w}|}$. For notation $\bar{u} \sqsubseteq \bar{w}$ denotes that $\bar{u}$ is a subword of $\bar{w}$. Further, $\bar{u} \sqsubseteq_i \bar{w}$ denotes that $\bar{u}$ is a subword of $\bar{w}$ of length i. If $\bar{w} = \bar{u}\bar{v}$, then $\bar{u}$ is a prefix and $\bar{v}$ is a suffix. A prefix or suffix of a word $\bar{u}$ is *proper* if its length is smaller than $|\bar{u}|$. For notation, the tuple $\mathbf{S}(\bar{v},\ell)$ is defined as the set of all subwords of $\bar{v}$ of length ℓ. Formally let $\mathbf{S}(\bar{v},\ell) = \{\bar{s} \sqsubseteq \bar{v} : |\bar{s}| = \ell\}$. Further, $\mathbf{S}(\bar{v},\ell)$ is assumed to be in lexicographic order, i.e. $\mathbf{S}(\bar{v},\ell)_1 \geq \mathbf{S}(\bar{v},\ell)_2 \geq \dots \mathbf{S}(\bar{v},\ell)_{|\bar{v}|}$.

A *bracelet* is the equivalence class of words under the combination of the rotation and the reversal operations. In this way a bracelet can be thought of as the union of two necklace classes $\tilde{\mathbf{w}}$ and $\tilde{\mathbf{w}}^R$, hence $\hat{\mathbf{w}} = \tilde{\mathbf{w}} \cup \tilde{\mathbf{w}}^R$. Given a bracelet $\hat{\mathbf{w}}$, the *bracelet representative* of $\hat{\mathbf{w}}$, denoted by $[\hat{\mathbf{w}}]$, is the lexicographically smallest word $\bar{u} \in \hat{\mathbf{w}}$.

A necklace $\tilde{\mathbf{w}}$ is *palindromic* if $\tilde{\mathbf{w}} = \tilde{\mathbf{w}}^R$. This means that the reflection of every word in $\tilde{\mathbf{w}}$ is in $\tilde{\mathbf{w}}^R$, i.e. given $\bar{u} \in \tilde{\mathbf{w}}, \bar{u}^R \in \tilde{\mathbf{w}}^R$. Note that for any word $\bar{w} \in \tilde{\mathbf{a}}$, where $\tilde{\mathbf{a}}$ is a palindromic necklace, either $\bar{w} = \bar{w}^R$, or there exists some rotation i for which $\langle\bar{w}\rangle_i = \bar{w}^R$.

Let $\tilde{\mathbf{u}}$ and $\tilde{\mathbf{v}}$ be a pair of necklaces belonging to the same bracelet class. For simplicity assume that $\langle\tilde{\mathbf{u}}\rangle < \langle\tilde{\mathbf{v}}\rangle$. The bracelet $\hat{\mathbf{u}}$ *encloses* a word $\bar{w}$ if $\langle\tilde{\mathbf{u}}\rangle < \bar{w} < \langle\tilde{\mathbf{v}}\rangle$. An example of this is the bracelet $\hat{\mathbf{u}} = aabc$ which encloses the word $\bar{w} = aaca$ as $aabc < aaca < aacb$. The set of all bracelets which enclose $\bar{w}$ are referred to as the set of bracelets *enclosing* $\bar{w}$.

2.2 Bounding Subwords

For both the palindromic and enclosing cases the number of necklaces smaller than $\bar{v} \in \Sigma^n$ is computed by iteratively counting the number of words of length n for which no subword is smaller than $\bar{v}$. The set of such words, denoted by $\mathbf{S}_n$, will be analysed iteratively as well, since it can have an exponential size. In order to relate $\mathbf{S}_i$ to $\mathbf{S}_{i+1}$, we will split $\mathbf{S}_i$ into parts using the positions of length i subwords of $\bar{v}$ with respect to the lexicographic order on $\mathbf{S}_i$. Informally, every $\bar{w} \in \mathbf{S}_i$ can be associated with the unique lower bound from $\mathbf{S}(\bar{v},i)$, which will be used to identify the parts leading us to the following definition.

▶ **Definition 1.** *Let $\bar{w},\bar{v} \in \Sigma^*$ where $|\bar{w}| \leq |\bar{v}|$. The word $\bar{w}$ is* bounded *(resp. strictly bounded) by $\bar{s} \sqsubseteq_{|\bar{w}|} \bar{v}$, if $\bar{s} \leq \bar{w}$ (resp. $\bar{s} < \bar{w}$) and there is no $\bar{u} \sqsubseteq_{|\bar{w}|} \bar{v}$ such that $\bar{s} < \bar{u} \leq \bar{w}$.*

The aforementioned parts $\mathbf{S}_i(\bar{s})$ contain all words $\bar{w} \in \mathbf{S}_i$ such that $\bar{s} \sqsubseteq_{|\bar{w}|} \bar{v}$. The key observation is that words of the form $x\bar{w}$ for all $\bar{w} \in \mathbf{S}_i$ and some fixed symbol $x \in \Sigma$ belong to the same set $\mathbf{S}_{i+1}(\bar{s}')$, where $\bar{s}' \sqsubseteq \bar{v}$. The same holds true for words of the form $\bar{w}x$.

Thus, we can compute the corresponding $\bar{s}'$ for all pairs of $\bar{s}$ and x in order to derive sizes of $\mathbf{S}_{i+1}(\bar{s}')$. Moreover, this relation between $\bar{s}$, x and $\bar{s}'$ is independent of i allowing us to store this information in two $n^2 \times k$ arrays XW and WX. Both arrays will be indexed by the words $\bar{s} \sqsubseteq \bar{v}$ and characters $x \in \Sigma$. Given a word $\bar{w}$ strictly bounded by $\bar{s}$, $XW[\bar{s}, x]$ will contain the word $\bar{s}' \sqsubseteq_{|\bar{s}|+1} \bar{v}$ strictly bounding $x\bar{w}$. Similarly, $WX[\bar{s}, x]$ will contain the word $\bar{s}' \sqsubseteq_{|\bar{s}|+1} \bar{v}$ strictly bounding $\bar{w}x$. By precomputing these arrays, the cost of determining these words can be avoided during the ranking process.

▶ **Proposition 2.** *Let $\bar{v} \in \Sigma^n$. The array $XW[\bar{s} \sqsubseteq \bar{v}, x \in \Sigma]$ such that $XW[\bar{s}, x]$ strictly bounds $x\bar{w}$ for every $\bar{w}$ strictly bounded by $\bar{s}$ can be computed in time $O(k \cdot n^3 \cdot \log(n))$.*

▶ **Proposition 3.** *Let $\bar{v} \in \Sigma^n$. The array $WX[\bar{s} \sqsubseteq \bar{v}, x \in \Sigma]$, such that $WX[\bar{s}, x]$ strictly bounds $\bar{w}x$ for every $\bar{w}$ strictly bounded by $\bar{s}$, can be computed in $O(k \cdot n^3 \cdot \log(n))$ time.*

3 Ranking Bracelets

The main result of the paper is the first algorithm for ranking bracelets. In this paper, we tacitly assume that we are ranking a word $\bar{v}$ of length n. The time-complexity of the ranking algorithm is $O(k^2 \cdot n^4)$, where k is the size of the alphabet and n is the length of the considered bracelets. The key part of the algorithm is to compute the rank of the word $\bar{v}$ with respect to the set of bracelets by finding three other ranks: the rank over all necklaces, the rank over palindromic necklaces, and the rank over enclosing apalindromic necklaces.

A bracelet can correspond to two apalindromic necklaces, or to exactly one palindromic necklace. If a bracelet $\hat{\mathbf{b}}$ corresponds to two necklaces $\tilde{\mathbf{l}}_b$ and $\tilde{\mathbf{r}}_b$, then it is important to take into account the lexicographical positions of these two necklaces $\tilde{\mathbf{l}}_b$ and $\tilde{\mathbf{r}}_b$ with respect to a given word $\bar{v}$. There are three possibilities: $\tilde{\mathbf{l}}_b$ and $\tilde{\mathbf{r}}_b$ could be less than $\bar{v}$; $\tilde{\mathbf{l}}_b$ and $\tilde{\mathbf{r}}_b$ encloses $\bar{v}$, e.g. $\tilde{\mathbf{l}}_b < \bar{v} < \tilde{\mathbf{r}}_b$, or both of necklaces $\tilde{\mathbf{l}}_b$ and $\tilde{\mathbf{r}}_b$ are greater than $\bar{v}$. This is visualised in Figure 2. Therefore the number of bracelets smaller than a given word w can be calculated by adding the number of palindromic necklaces less than $\bar{v}$, enclosing bracelets smaller than $\bar{v}$ and half of all other apalindromic and non-enclosing necklaces smaller than $\bar{v}$. Let us define the following notation is used for the rank of $\bar{v} \in \Sigma^n$ for sets of bracelets and necklaces.

- $RN(\bar{v})$ denotes the rank of $\bar{v}$ with respect to the set of *necklaces* of length n over Σ.
- $RP(\bar{v})$ denotes the rank of $\bar{v}$ with respect to the set of *palindromic necklaces* over Σ.
- $RB(\bar{v})$ denotes the rank of $\bar{v}$ with respect to the set of *bracelets* of length n over Σ.
- $RE(\bar{v})$ denotes the rank of $\bar{v}$ with respect to the set of *bracelets enclosing* $\bar{v}$.

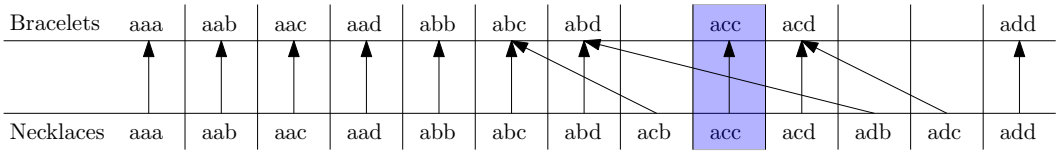


Figure 2 In this example the top line represents the set of bracelets and the bottom line the set of necklaces, with arrows indicated which necklace corresponds to which bracelet. Assuming we wish to rank the word *acc* (highlighted), *abc* and *acb* are apalindromic necklaces smaller than *acc*, while *abd* encloses *acc*. All other necklaces are palindromic.

In Lemma 4 below, we show that $RB(\bar{v})$ can be expressed via $RN(\bar{v})$, $RP(\bar{v})$ and $RE(\bar{v})$. The problem of computing $RN(\bar{v})$ has been solved in quadratic time [16], so the goal of the paper is to design efficient procedures for computing $RP(\bar{v})$ and $RE(\bar{v})$.

▶ **Lemma 4.** *The rank of a word $\bar{v} \in \Sigma^n$ with respect to the set of bracelets of length n over the alphabet Σ is given by $RB(\bar{v}) = \frac{1}{2}(RN(\bar{v}) + RP(\bar{v}) + RE(\bar{v}))$.*

Proof. Simply dividing the number of necklaces by 2 will undercount the number of bracelets, while doing nothing will overcount. Therefore to get the correct number of bracelets, those bracelets corresponding to only 1 necklace must be accounted for. A bracelet $\hat{\mathbf{a}}$ will correspond to 2 necklaces smaller than $\bar{v}$ if and only if $\hat{\mathbf{a}}$ does not enclose $\bar{v}$ and $\hat{\mathbf{a}}$ is apalindromic. Therefore the number of bracelets corresponding to 2 necklaces is $\frac{1}{2}(RN(\bar{v}) - RP(\bar{v}) - RE(\bar{v}))$. The number of bracelets enclosing $\bar{v}$ is equal to $RE(\bar{v})$. The number of bracelets corresponding to palindromic necklaces is equal to $RP(\bar{v})$. Therefore the total number of bracelets is $\frac{1}{2}(RN(\bar{v}) - RP(\bar{v}) - RE(\bar{v})) + RP(\bar{v}) + RE(\bar{v}) = \frac{1}{2}(RN(\bar{v}) + RP(\bar{v}) + RE(\bar{v}))$. ◀

Lemma 4 provides the basis for ranking bracelets. Theorem 5 uses Lemma 4 to get the complexity of the ranking process. The remainder of this paper will prove Theorem 5, starting with the complexity of ranking among palindromic necklaces in Section 4 followed by the complexity of ranking enclosing bracelets in Section 5.

▶ **Theorem 5.** *Given a word $\bar{v} \in \Sigma^n$, the rank of $\bar{v}$ with respect to the set of bracelets of length n over the alphabet Σ, $RB(\bar{v})$, can be computed in $O(k \cdot n^4)$ time.*

The remainder of this paper will prove Theorem 5. For simplicity, the word $\bar{v}$ is assumed to be a necklace representation. It is well established how to find the lexicographically largest necklace smaller than or equal to some given word. Such a word can be found in quadratic time using an algorithm form [16]. Note that the number of necklaces less than or equal to $\bar{v}$ corresponds to the number of necklaces less than or equal to the lexicographically largest necklace smaller than $\bar{v}$. From Lemma 4 it follows that to rank $\bar{v}$ with respect to the set of bracelets, it is sufficient to rank $\bar{v}$ with respect to the set of necklaces, palindromic necklaces, and enclosing bracelets. The rank with respect to the set of palindromic necklaces, $RP(\bar{v})$ can be computed in $O(k \cdot n^3)$ using the techniques given in Theorem 25 in Section 4. The rank with respect to the set of enclosing bracelets, $RE(\bar{v})$ can be computed in $O(k \cdot n^4)$ as shown in Theorem 30 in Section 5. As each of these steps can be done independently of each other, the total complexity is $O(k \cdot n^4)$.

This complexity bound is a significant improvement over the naive method of enumerating all bracelets, requiring exponential time in the worst case. New intuition is provided to rank the palindromic and enclosing cases. The main source of complexity for the problem of ranking comes from having to consider the lexicographic order of the word under reflection. New combinatorial results and algorithms are needed to count the bracelets in these cases.

Before showing in detail the algorithmic results that allow bracelets to be efficiently ranked, it is useful to discus the high level ideas. Lemma 4 shows our approach to ranking bracelets by dividing the problem into the problems of ranking necklaces, palindromic necklaces and enclosing bracelets. For both palindromic necklaces and enclosing bracelets, we derive a *canonical form* using the combinatorial properties of these objects.

Using these canonical forms, the number of necklaces smaller than $\bar{v}$ is counted in an iterative manner. In the palindromic case, this is done by counting the number of necklaces greater than $\bar{v}$, and subtracting this from the total number of palindromic necklaces. In the enclosing case, this is done by directly counting the number of necklaces smaller than $\bar{v}$. For both cases, the counting is done by way of a tree comprised of the set of all prefixes of words of the canonical form. By partitioning the internal vertices of the trees based on the number of children of the vertices, the number of words of the canonical form may be derived in an efficient manner, forgoing the need to explicitly generate the tree. This allows the size of these partitions to be computed through a dynamic programming approach. It follows from these partitions how to count the number of leaf nodes, corresponding to the canonical form.

▶ **Theorem 6.** *The z^{th} bracelet of length n over Σ can be computed in $O(n^5 \cdot k^2 \cdot \log(k))$.*

Theorem 6 is proven by using the ranking technique as a black box alongside a simple binary search in the same manner as [16].

4 Computing the rank $RP(\bar{v})$

To rank palindromic necklaces, it is crucial to analyse their combinatorial properties. This section focuses on providing results on determining unique words representing palindromic necklaces. We study two cases depending on whether the length n of a palindromic necklace is even or odd. The reason for this division can be seen by considering examples of palindromic necklaces. If equivalence under the rotation operation is not taken into account, then a word is palindromic if $\bar{w} = \bar{w}^R$. If the length n of $\bar{w}$ is odd, then if $\bar{w} = \bar{w}^R$, $\bar{w}$ can be written as $\bar{\phi}x\bar{\phi}^R$, where $\bar{\phi} \in \Sigma^{(n-1)/2}$ and $x \in \Sigma$. For example, the word $aaabaaa$ is equal to $\bar{\phi}x\bar{\phi}^R$, where $\bar{\phi} = aaa$ and $x = b$. If the length n of w is even, then if $\bar{w} = \bar{w}^R$, $\bar{w}$ can be written as $\bar{\psi}\bar{\psi}^R$, where $\bar{\psi} \in \Sigma^{n/2}$. For example the word $aabbaa$ is equal to $\bar{\psi}\bar{\psi}^R$, where $\bar{\psi} = aab$.

Once rotations are taken into account, the characterisation of palindromic necklaces becomes more difficult. It is clear that any necklace $\tilde{\mathbf{a}}$ that contains a word of the form $\bar{\phi}x\bar{\phi}^R$ or $\bar{\phi}\bar{\phi}^R$ is palindromic. However this check does not capture every palindromic necklace. Let us take, for example, the necklace $\tilde{\mathbf{a}} = ababab$, which contains two words $ababab$ and $bababa$. While $ababab$ can neither be written as $\bar{\phi}x\bar{\phi}^R$ nor $\bar{\phi}\bar{\phi}^R$, it is still palindromic as $\langle ababab^R \rangle = \langle bababa \rangle = ababab$. Therefore a more extensive test is required. As the structure of palindromic words without rotation is different depending on the length being either odd or even, it is reasonable to split the problem of determining the structure of palindromic necklaces into the cases of odd and even length.

The number of palindromic necklaces are counted by computing the number of these characterisations. This is done by constructing trees containing every prefix of these characterisations. As each vertex corresponds to the prefix of a word, the leaf nodes of these trees correspond to the words in the characterisations. By partitioning the tree in an intelligent manner, the number of leaf nodes and therefore number of these characterisations can be computed. In the odd case this corresponds directly to the number of palindromic necklaces, while in the even case a small transformation of these sets is needed.

4.1 Odd Length Palindromic Necklaces

Starting with the odd-length case, Proposition 7 shows that every palindromic necklace of odd length contains **exactly one word** that can be written as $\bar{\phi}x\bar{\phi}^R$ where $\bar{\phi} \in \Sigma^{(n-1)/2}$ and $x \in \Sigma$. This fact is used to rank the number of bracelets by constructing a tree representing every prefix of a word of the form $\bar{\phi}x\bar{\phi}^R$ that belongs to a bracelet greater than $\bar{v}$.

▶ **Proposition 7.** *A necklace $\tilde{\mathbf{w}}$ of odd length n is palindromic if and only if there exists exactly one word $\bar{u} = \bar{\phi}x\bar{\phi}^R$ such that $\bar{v} \in \tilde{\mathbf{w}}$, where $\bar{\phi} \in \Sigma^{(n-1)/2}$ and $x \in \Sigma$.*

Proof Sketch. If a necklace $\tilde{\mathbf{w}}$ contains a word of the form $\bar{\phi}x\bar{\phi}^R$, then clearly $\tilde{\mathbf{w}}$ is palindromic. The other direction follows a counting argument. In order for $\tilde{\mathbf{w}}$ to be palindromic, the reflection of every word in $\tilde{\mathbf{w}}$ must also be in $\tilde{\mathbf{w}}$. As $\tilde{\mathbf{w}}$ contains an odd number of words, there must exist a word that is its own reflection, i.e. $\bar{u} = \bar{u}^R$. As the length of the word is odd, the only time this can occur is when $\bar{u} = \bar{\phi}x\bar{\phi}^R$. We show uniqueness by observing that no two words in $\tilde{\mathbf{w}}$ can be mapped to the same palindromic word. ◀

▶ **Corollary 8.** *The number of palindromic necklaces of odd length n over Σ equals $k^{(n+1)/2}$.*

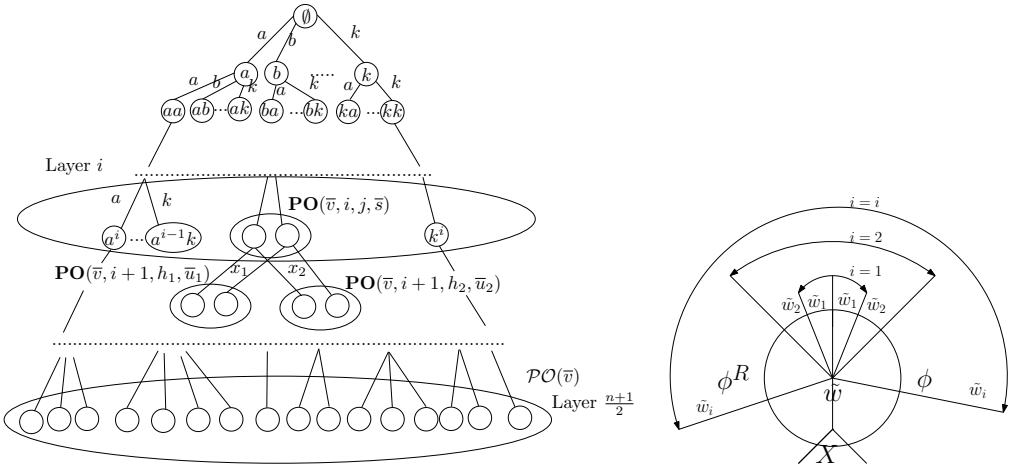


Figure 3 (Left) The relationship between $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ with the tree $\mathcal{TO}(\bar{v})$ and $\mathbf{PO}(\bar{v})$. (right) Example of the order for which characters are assigned. Note that at each step the choices for the symbol $\tilde{\mathbf{w}}_i$ is constrained in the no subword of $\tilde{\mathbf{w}}_{[1,i]}\tilde{\mathbf{w}}^R_{[1,i]}$ is greater than or equal to $\bar{v}$.

The problem now becomes to rank a word $\bar{v}$ with respect to the odd length palindromic necklaces utilising their combinatorial properties. Let $\bar{v} \in \Sigma^n$ be a word of odd length n. We define the set $\mathcal{PO}(\bar{v})$, where $\mathcal{PO}$ stands for palindromic odd length. The set $\mathcal{PO}(\bar{v})$ contains one word representing each palindromic bracelet of odd length n that is greater than $\bar{v}$.

$$\mathcal{PO}(\bar{v}) := \left\{ \bar{w} \in \Sigma^n : \bar{w} = \bar{\phi} x \bar{\phi}^R, \text{ where } \langle \bar{w} \rangle > \bar{v},\ \bar{\phi} \in \Sigma^{(n-1)/2},\ x \in \Sigma \right\}.$$

As each word will correspond to a unique palindromic necklace of length n greater than $\bar{v}$, and every palindromic necklace greater than $\bar{v}$ will correspond to a word in $\mathcal{PO}(\bar{v})$, the number of palindromic necklaces greater than $\bar{v}$ is equal to $|\mathcal{PO}(\bar{v})|$. Using this set the number of necklaces less than $\bar{v}$ can be counted by subtracting the size of $\mathcal{PO}(\bar{v})$ from the total number of odd length palindromic necklaces, equal to $k^{(n+1)/2}$ (Corollary 8).

High level idea for the Odd Case. Here we provide a high level idea for the approach we follow for computing $\mathcal{PO}(\bar{v})$. Let $\bar{v}$ have a length n. Since $\mathcal{PO}(\bar{v})$ only contains words of the form $\bar{\phi} x \bar{\phi}^R$, where $\bar{\phi} \in \Sigma^{(n-1)/2}$ and $x \in \Sigma$, we have that $\bar{w}_i = \bar{w}_{n-i}$ for every $\bar{w} \in \mathcal{PO}(\bar{v})$. As the lexicographically smallest rotation of every $\bar{w} \in \mathcal{PO}(\bar{v})$ must be greater than $\bar{v}$, it follows that any word rotation of $\bar{w}$ must be greater than $\bar{v}$ and therefore every subword of $\bar{w}$ must also be greater than or equal to the prefix of $\bar{v}$ of the same length. This property is used to compute the size of $\mathcal{PO}(\bar{v})$ by iteratively considering the set of prefixes of each word in $\mathcal{PO}(\bar{v})$ in increasing length representing them with the tree $\mathcal{TO}(\bar{v})$. As generating $\mathcal{TO}(\bar{v})$ directly would require an exponential number of operations, a more sophisticated approach is needed for the calculation of $|\mathcal{PO}(\bar{v})|$ based on partial information.

As the tree $\mathcal{TO}(\bar{v})$ is a tree of prefixes, vertices in $\mathcal{TO}(\bar{v})$ are referred to by the prefix they represent. So $\bar{u} \in \mathcal{TO}(\bar{v})$ refers to the unique vertex in $\mathcal{TO}(\bar{v})$ representing $\bar{u}$. The root vertex of $\mathcal{TO}(\bar{v})$ corresponds to the empty word. Every other vertex $\bar{u} \in \mathcal{TO}(\bar{v})$ corresponds to a word of length i, where i is the distance between $\bar{u}$ and the root vertex. Given two vertices $\bar{p}, \bar{c} \in \mathcal{TO}(\bar{v})$, $\bar{p}$ is the parent vertex of a child vertex $\bar{c}$ if and only if $\bar{c} = \bar{p}x$ for some symbol $x \in \Sigma$. The i^{th} layer of $\mathcal{TO}(\bar{v})$ refers to all representing words of length i in $\mathcal{TO}(\bar{v})$.

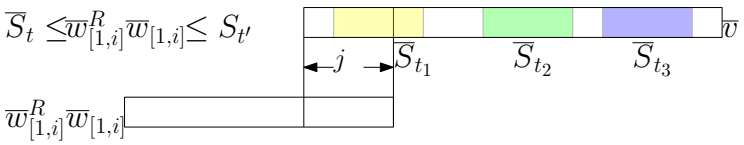


Figure 4 Visual representation of the properties of $\bar{w}^R_{[1,i]}\bar{w}_{[1,i]} \in \mathbf{PO}(\bar{v}, i, j, \bar{s})$.

The size of $\mathcal{PO}(\bar{v})$ is equivalent to the number of unique prefixes of length $\frac{n+1}{2}$ of words of the palindromic form $\bar{\phi}x\bar{\phi}^R$ in $\mathcal{PO}(\bar{v})$. This set of prefixes corresponds to the vertices in the layer $\frac{n+1}{2}$ of $\mathcal{TO}(\bar{v})$. Therefore the maximum depth of $\mathcal{TO}(\bar{v})$ is $\frac{n+1}{2}$.

To speed up computation, each layer of $\mathcal{TO}(\bar{v})$ is partitioned into sets that allow the size of $\mathcal{PO}(\bar{v})$ to be efficiently computed. This partition is chosen such that the size of the sets in layer $i+1$ can be easily derived from the size of the sets in layer i. As these sets are tied to the tree structure, the obvious property to use is the number of children each vertex has. As each vertex $\bar{u} \in \mathcal{TO}(\bar{v})$ represents a prefix of some word $\bar{w} \in \mathcal{PO}(\bar{v})$, the number of children of $\bar{u}$ is the number of symbols $x \in \Sigma$ such that $\bar{u}x$ is a prefix of some word in $\mathcal{PO}(\bar{v})$. Recall that every word in $\bar{w} \in \mathcal{PO}(\bar{v})$ has the form $\bar{\phi}x\bar{\phi}^R$, and that there is no subword of $\bar{w}$ that is less than $\bar{v}$. Therefore if $\bar{u} \in \mathcal{TO}(\bar{v})$, there must be no subword of $\bar{u}^R\bar{u}$ that is less than $\bar{v}$. Hence the number of children of $\bar{u}$ is the number of symbols $x \in \Sigma$ such that no subword of $x\bar{u}^R\bar{u}x$ is less than the prefix of $\bar{v}$ of the same length. As $\bar{u}^R\bar{u}$ has no subword less than $\bar{v}$, $x\bar{u}^R\bar{u}x$ will only have a subword that is less than $\bar{v}$ if either (1) $x\bar{u}^R\bar{u}x < \bar{v}$ or (2) there exists some suffix of length j such that $(\bar{u}^R\bar{u})_{[2i-j,2i]} = \bar{v}_{[1,j]}$ and $x < \bar{v}_{j+1}$. For the first condition, let $\bar{s} \sqsubseteq_{2i} \bar{v}$. By the definition of strictly bounding subwords (Definition 1), $x\bar{u}^R\bar{u}x < \bar{v}$ if and only if $x\bar{s}x < \bar{v}$. Note that this ignores any word $\bar{u}$ where $\bar{u}^R\bar{u} \sqsubseteq \bar{v}$. The restriction to strictly bounded words is to avoid the added complexity caused by Proposition 2, where the word that bounds $x\bar{s}x$ might not be the word that bounds $x\bar{u}^R\bar{u}x$. For the second property, let j be the length of the longest suffix of $\bar{u}^R\bar{u}$ that is a prefix of $\bar{v}$. From Lemma 1 due to Sawada and Williams [16], there is some suffix of $\bar{u}^R\bar{u}x$ that is smaller than $\bar{v}$ if and only if $x < \bar{v}_{j+1}$. The i^{th} layer of $\mathcal{TO}(\bar{v})$ is partitioned into n^2 sets $\mathbf{PO}(\bar{v}, i, j, \bar{s})$, for every $i \in [\frac{n+1}{2}], j \in [2i]$ and $\bar{s} \sqsubseteq_{2i} \bar{v}$.

▶ **Definition 9.** *Let $i \in [\frac{n+1}{2}], j \in [2i]$ and $\bar{s} \sqsubseteq_{2i} \bar{v}$. The set $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ contains every prefix $\bar{u} \in \mathcal{TO}(\bar{v})$ of length i where (1) the longest suffix of $\bar{u}^R_{[1,i]}\bar{u}_{[1,i]}$ which is a prefix of $\bar{v}$ has a length of j and (2) The word $\bar{u}^R_{[1,i]}\bar{u}_{[1,i]}$ is strictly bounded by $\bar{s}$.*

An overview of the properties used by $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ is given in Figures 3 and 4. It follows from the earlier observations that each vertex in $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ has the same number of children. Lemma 10 strengthens this observation, showing that given $\bar{a}, \bar{b} \in \mathbf{PO}(\bar{v}, i, j, \bar{s})$, $\bar{a}x \in \mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$ if and only if $\bar{b}x \in \mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$.

The remainder of this section establishes how to count the size of $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ and the number of children vertices for each vertex in $\mathbf{PO}(\bar{v}, i, j, \bar{s})$. The first step is to formally prove that all vertices in $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ have the same number of children vertices. This is shown in Lemma 10 by proving that given two vertices $\bar{a}, \bar{b} \in \mathbf{PO}(\bar{v}, i, j, \bar{s})$, if the vertex $\bar{a}' = \bar{a}x$ for $x \in \Sigma$ belongs to the set $\mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$, so to does $\bar{b}' = \bar{b}x$.

▶ **Lemma 10.** *Let $\bar{a}, \bar{b} \in \mathbf{PO}(\bar{v}, i, j, \bar{s})$ and let $x \in \Sigma$. If the vertex $\bar{a}' = \bar{a}x$ belongs to $\mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$, the vertex $\bar{b}' = \bar{b}x$ also belongs to $\mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$. Furthermore the value of j' and $\bar{s}'$ can be computed in constant time from the values of $j, \bar{s}$ and x.*

Proof Sketch. The arrays XW and WX are used to determine the value of $\bar{s}'$. The longest suffix of both $x\bar{a}x$ and $x\bar{b}x$ that is a prefix of $\bar{v}$ is determined by the value of j and x, either $j+1$, if $x = \bar{v}_{j+1}$, or 0 otherwise. Hence if $\bar{a}x \in \mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$, $\bar{b}x \in \mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$. Further the values of j' and $\bar{s}'$ can be determined in constant time. ◀

Computing the size of $\mathbf{PO}(\bar{v}, i, j', \bar{s}')$. Lemma 10, provides enough information to compute the size of $\mathbf{PO}(\bar{v}, i, j', \bar{s}')$ once the size of $\mathbf{PO}(\bar{v}, i-1, j, \bar{s})$ has been computed for each value of $j \in [2(i-1)]$ and $\bar{s} \in \mathbf{S}(\bar{v}, 2(i-1))$. At a high level, the idea is to create an array, $SizePO$, storing the size of the $\mathbf{PO}(\bar{v}, i, j', \bar{s}')$ for every value of $i \in [\frac{n-1}{2}], j \in [2i]$ and $\bar{s} \sqsubseteq_{2i} \bar{v}$. For simplicity, let the value of $SizePO[i, j, \bar{s}]$ be the size of $|\mathbf{PO}(\bar{v}, i, j, \bar{s})|$.

Lemma 11 formally provides the method of computing $SizePO[i, j, \bar{s}]$ for every $j \in [2i]$ and $\bar{s} \sqsubseteq_{2i} \bar{v}$ once $SizePO[i-1, j', \bar{s}']$ has been computed for every $j' \in [2i-2]$ and $\bar{s} \sqsubseteq_{2i-2} \bar{v}$. Observe that each vertex $a \in \mathbf{PO}(\bar{v}, i, j', \bar{s}')$ represents a prefix $\bar{a}'x$ where $\bar{a}'$ is either in $\mathbf{PO}(\bar{v}, i-1, j, \bar{s})$, for some value of j and $\bar{s}$, or $\bar{a}' \sqsubseteq \bar{v}$. Using this, the high level idea is to derive the values of j' and $\bar{s}'$ for each $j \in [2(i-1)], \bar{s} \in \mathbf{S}(\bar{v}, 2(i-1))$ and $x \in \Sigma$. Once the values j' and $\bar{s}'$ have been derived, the value of $SizePO[i, j', \bar{s}']$ is increased by the size of $\mathbf{PO}(\bar{v}, i-1, j, \bar{s})$. Repeating this for every value of $j, \bar{s}$ and x will leave the value of $SizePO[i, j', \bar{s}']$ as the number of vertices in $\mathbf{PO}(\bar{v}, i, j', \bar{s}')$ representing words of the form $\bar{a}x$ where $\bar{a} \not\sqsubseteq \bar{v}$. As each set $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ may have children in at most k sets $\mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$, the number of vertices in $\mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$ with a parent vertex in $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ can be computed in $O(k \cdot n^2)$ by looking at every argument of $j \in [2i]$ and $\bar{s} \sqsubseteq_{2i} \bar{v}$.

To account for the vertices in $\mathbf{PO}(\bar{v}, i, j', \bar{s}')$ of the form $\bar{b}x$ where $\bar{b}^R\bar{b} \sqsubseteq \bar{v}$, a similar process is applied to each pair $\bar{s} \in \mathbf{S}(\bar{v}, 2(i-1))$ and $x \in \Sigma$. For each pair, the values $\bar{s}'$ and j' are derived in the same manner as Lemma 10 utilising the tables XW and WX. Once derived, the value of $SizePO[i, j', \bar{s}']$ is increased by one, to account for the vertex $\bar{s}x$. As the values of j' and $\bar{s}'$ can be computed in $O(n)$ time from the value of x and $\bar{s}$, the number of vertices in $\mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$ where the parent vertex is a subword of $\bar{v}$ can be computed in $O(k \cdot n^2)$ time.

▶ **Lemma 11.** *Given the size of $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ for $i \in \left[\frac{n-3}{2}\right]$ and every $j \in [2i], \bar{s} \sqsubseteq_{2i} \bar{v}$, the size of $\mathbf{PO}(\bar{v}, i+1, j', \bar{s}')$ for every $j' \in [2i+2], \bar{s}' \sqsubseteq_{2i+2} \bar{v}$ can be computed in $O(k \cdot n^2)$ time.*

Proof Sketch. $SizePO[i+1, j', \bar{s}']$ is computed by looking at every argument of $j \in [2i], \bar{s} \sqsubseteq_{2i} \bar{v}$ and $x \in \Sigma$. For each set of arguments, $\bar{j}'$ and $\bar{s}'$ are derived by Lemma 10, and the size of $SizePO[i+1, j', \bar{s}']$ is increased by $SizePO[i, j, \bar{s}]$. Similarly, for each $x \in \Sigma$ and $\bar{s} \sqsubseteq_{2i} \bar{v}$, $\bar{j}'$ and $\bar{s}'$ are derived and the size of $SizePO[i+1, j', \bar{s}']$ is increased by 1. ◀

Once the size of $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ has been computed for every $i \in [\frac{n-1}{2}], j \in [2i], \bar{s} \in \mathbf{S}(\bar{v}, 2i)$, the final step is to compute $|\mathcal{PO}(\bar{v})|$. The high level idea is to determine the number of vertices in $\mathcal{PO}(\bar{v})$ are children of a vertex in $\mathbf{PO}(\bar{v}, \frac{n-1}{2}, j, \bar{s})$.The set $\mathbf{X}(\bar{v}, j, \bar{s}) \subseteq \Sigma$ is introduced to help with this goal. Let $\mathbf{X}(\bar{v}, j, \bar{s})$ contain every symbol $x \in \Sigma$ such that $\bar{a}x\bar{a}^R \in \mathcal{PO}(v)$ where $\bar{a} \in \mathbf{PO}(\bar{v}, \frac{n-1}{2}, j, \bar{s})$. By the definition of $\mathbf{X}(\bar{v}, j, \bar{s})$, $|\mathbf{X}(\bar{v}, j, \bar{s})| \cdot |\mathbf{PO}(\bar{v}, \frac{n-1}{2}, j, \bar{s})|$ equals the number of words $\bar{w} \in \mathcal{PO}(\bar{v})$ where $(\bar{w}_1 \dots \bar{w}_{(n-1)/2}) \in \mathbf{PO}(\bar{v}, i, j, \bar{s})$. Lemma 12 shows how to compute the size of $\mathbf{X}(\bar{v}, j, \bar{s})$ in $O(k \cdot n)$ time.

▶ **Lemma 12.** *Let $\mathbf{X}(\bar{v}, j, \bar{s})$ contain every symbol in Σ such that $\bar{a}x\bar{a}^R \in \mathcal{PO}(v)$ where $\bar{a} \in \mathbf{PO}(\bar{v}, \frac{n-1}{2}, j, \bar{s})$. The size of $\mathbf{X}(\bar{v}, j, \bar{s})$ can be computed in $O(k \cdot n)$ time.*

Proof Sketch. The size of $\mathbf{X}(\bar{v}, j, \bar{s})$ is computed by checking if $x \in \mathbf{X}(\bar{v}, j, \bar{s})$ for each $x \in \Sigma$, where j and $\bar{s}$ are used to bound x from below. As x must be such that $\bar{v}_{[1,j]}x\bar{s} > \bar{v}$, $x \geq \bar{v}_{j+1}$. Further, if $\bar{s} < \bar{v}_{[j+2,n]}$ then $x > \bar{v}_{j+1}$. ◀

Converting *SizePO* to $|\mathcal{PO}(\bar{v})|$. The final step in computing $\mathcal{PO}(\bar{v})$ is to convert the cardinality of $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ to the size of $\mathcal{PO}(\bar{v})$. Lemma 13 provides a formula for counting the size of $\mathcal{PO}(\bar{v})$. Combining this formula with the techniques given in Lemma 11 an algorithm for computing the size of $\mathcal{PO}(\bar{v})$ directly follows.

It follows from Lemma 10 that the number of words in $\mathcal{PO}(\bar{v})$ with a prefix in $\mathbf{PO}\left(\bar{v}, \frac{n-1}{2}, j, \bar{s}\right)$ is equal to the cardinality of $\mathbf{PO}\left(\bar{v}, \frac{n-1}{2}, j, \bar{s}\right)$ multiplied by the size of $\mathbf{X}(\bar{v}, j, \bar{s})$. Similarly the number of words in $\mathcal{PO}(\bar{v})$ with a prefix $\bar{u}$ of length $\frac{n-1}{2}$ where $\bar{u}^R\bar{u} \sqsubseteq \bar{v}$ can be determined using $\mathbf{X}(\bar{v}, j, \bar{u}^R\bar{u})$. The main difference in this case is that if $\bar{u}^R\bar{u} = \bar{v}_{[j+2,n+j]}$, where j is the length of the longest suffix of $\bar{u}^R\bar{u}$ that is a prefix of $\bar{v}$, then the number of words in $\mathcal{PO}(\bar{v})$ where $\bar{u}$ is a prefix is 1 fewer than for the number of words strictly bounded by $\bar{u}^R\bar{u}$, i.e. $|\mathbf{X}(\bar{v}, J(\bar{s}, \bar{v}), \bar{s})| - 1$. Lemma 13 provides the procedure to compute $|\mathcal{PO}(\bar{v})|$.

▶ **Lemma 13.** *Let $J(\bar{s}, \bar{v})$ return the length of the longest suffix of $\bar{s}$ that is a prefix of $\bar{v}$. The size of $\mathcal{PO}(\bar{v})$ is equal to*

$$\sum_{\bar{s} \in \mathbf{S}(\bar{v}, n-1)} \left(\sum_{j=1}^{n-1} |\mathbf{X}(\bar{v}, j, \bar{s})| \cdot |\mathbf{PO}\left(\bar{v}, \tfrac{n-1}{2}, j, \bar{s}\right)| \right) + \begin{cases} 0 & \bar{s} \neq \phi\phi^R \\ |\mathbf{X}(\bar{v}, J(\bar{s}, \bar{v}), \bar{s})| & \bar{s} \neq \bar{v}_{[j+2,n+j]} \\ |\mathbf{X}(\bar{v}, J(\bar{s}, \bar{v}), \bar{s})| - 1 & \bar{s} = \bar{v}_{[j+2,n+j]} \end{cases}$$

Further this can be computed in $O(k \cdot n^3 \cdot \log(n))$ time.

Proof Sketch. By the definition of $\mathbf{X}(\bar{v}, j, \bar{s})$, the number of words in $\mathcal{PO}(\bar{v})$ with a parent vertex in $\mathbf{PO}(\bar{v}, \frac{n-1}{2}, j, \bar{s})$ equals $|\mathbf{X}(\bar{v}, j, \bar{s})|$. Similarly, given $\bar{s} \sqsubseteq_{n-1} \bar{v}$, the number words in $\mathcal{PO}(\bar{v})$ of the form $\bar{s}_{[(n+1)/2,n-1]}x\bar{s}_{[1,(n-1)/2]}$ are equal to either $|\mathbf{X}(\bar{v}, j, \bar{s})|$, if $\bar{s} \neq \bar{v}_{[j+2,n+j]}$, or $|\mathbf{X}(\bar{v}, j, \bar{s})| - 1$ if $\bar{s} = \bar{v}_{[j+2,n+j]}$. ◀

4.2 Even Length Palindromic Necklaces

Section 4.1 shows how to rank $\bar{v}$ within the set of odd length palindromic necklaces. This leaves the problem of counting even length palindromic necklaces. As in the odd case, the first step is to determine how to characterise these words. Proposition 14 shows that every palindromic necklace will have at least one word of either the form $\bar{\phi}\bar{\phi}^R$, where $\bar{\phi} \in \Sigma^{n/2}$, or $x\bar{\phi}y\bar{\phi}^R$, where $x, y \in \Sigma$ and $\bar{\phi} \in \Sigma^{(n/2)-1}$. Proposition 14 is strengthened by Propositions 15 and 16, showing that each palindromic necklace of even length will have no more than two words of either form. Lemmas 21, 22, 23 and 24 use these results a similar manner to Section 4.1 to count the number of palindromic necklaces of even length.

▶ **Proposition 14.** *A necklace $\tilde{\mathbf{w}}$ of even length n is palindromic if and only if there exists some word $\bar{u} \in \tilde{\mathbf{w}}$ where either (1) $\bar{u} = x\bar{\phi}y\bar{\phi}^R$ where $x, y \in \Sigma$ and $\bar{\phi} \in \Sigma^{(n/2)-1}$, or (2) $\bar{u} = \bar{\phi}\bar{\phi}^R$ where $\bar{\phi} \in \Sigma^{n/2}$.*

Proof Sketch. Recall that if the necklace $\tilde{\mathbf{w}}$ is palindromic, then for any word $\bar{u} \in \tilde{\mathbf{w}}$, $\bar{u}^R \in \tilde{\mathbf{w}}$. As such, $\bar{u} = \langle \bar{u}^R \rangle_r$ for some rotation r. It follows from this that $\bar{u}_1 = \bar{u}_{n-r}, \bar{u}_2 = u_(n - r - 1) \ldots \bar{u}_j = \bar{u}_{n-r-j+1}$. The word $\bar{u}$ is split into the subwords $\bar{a} = \bar{u}_{[1,n-r]}$ and $\bar{b}_{[n-r+1,n]}$. As $\bar{a} = \bar{a}^R$ and $\bar{b} = \bar{b}^R$, the smaller subword can be lengthened by appending the first and last symbol of the longer word while maintaining this property, i.e. $\bar{b}' = \bar{a}_1\bar{b}\bar{a}_{|\bar{a}|} = \bar{b}'^R$. Repeating this until either both words have length $\frac{n}{2}$, or one has length $\frac{n}{2} - 1$, allows $\bar{u}$ to be rewritten, using these words to derive the values of $\bar{\phi} \in \Sigma^{n/2}$, or $x, y \in \Sigma$ and $\bar{\phi} \in \Sigma^{(n/2)-1}$. ◀

▶ **Proposition 15.** *The word $\bar{u} \in \Sigma^*$ equals both $x\bar{\phi}y\bar{\phi}^R = \bar{\psi}\bar{\psi}^R$ if and only if $\bar{u} = x^n$.*

▶ **Proposition 16.** *For an even length palindromic necklace* $\tilde{\mathbf{a}}$ *there are at most two words* $\bar{w}, \bar{u} \in \tilde{\mathbf{a}}$ *where either (1)* $\bar{w}$ *and* $\bar{u}$ *are of the form* $x\bar{\phi}y\bar{\phi}^R$ *where* $x, y \in \Sigma$ *and* $\bar{\phi} \in \Sigma^{(n/2)-1}$ *or (2)* $\bar{w}$ *and* $\bar{u}$ *are of the form* $\bar{\phi}\bar{\phi}^R$ *where* $\bar{\phi} \in \Sigma^{n/2}$.

Proof Sketch. For both $\bar{w} = x\bar{\phi}y\bar{\phi}^R$, and $\bar{w} = \bar{\phi}\bar{\phi}^R$, a similar approach is used. For $\bar{w} = x\bar{\phi}y\bar{\phi}^R$ assume that $\langle\bar{w}\rangle_r = \bar{u}$ for the smallest rotation r such that $\langle\bar{w}\rangle_r \neq \bar{w}$ and $\langle\bar{w}\rangle_r = x\bar{\phi}y\bar{\phi}^R$. By showing that the period of $\bar{w}$ must be $2r$, it follows that if there is some other rotation s such that $\bar{w} \neq \langle\bar{w}\rangle_s \neq \langle\bar{w}\rangle_r$ and $\langle\bar{w}\rangle_s$ is of the form $x\bar{\phi}y\bar{\phi}^R$, s must be less than r contradicting the initial assumption. For $\bar{w} = \bar{\phi}\bar{\phi}^R$, a similar process is done, showing that not only must there be no more than two words of the form $\bar{\phi}\bar{\phi}^R$, but that if $\bar{w} = \bar{\phi}\bar{\phi}^R$ then $\bar{u} = \bar{\psi}\bar{\psi}^R$ if and only if $\bar{\phi} = \bar{\psi}^R$. ◀

Propositions 14, 15 and 16 show that every palindromic necklace of even length has 1 or 2 words of either the form $x\bar{\phi}y\bar{\phi}^R$ or $\bar{\phi}\bar{\phi}^R$. To count the number of words of each form, the problem is split into two sub problems, counting words of the form $x\bar{\phi}y\bar{\phi}^R$ and counting the number of words of the form $\bar{\phi}\bar{\phi}^R$. This is done using the same basic ideas as in Section 4.1. Two new sets $\mathcal{PE}(\bar{v})$ and $\mathcal{PS}(\bar{v})$ are introduced, serving the same function as $\mathcal{PO}(\bar{v})$ for words of the from $x\bar{\phi}y\bar{\phi}^R$ and $\bar{\phi}\bar{\phi}^R$ respectively.

$$\mathcal{PE}(\bar{v}) := \left\{\bar{w} \in \Sigma^n : \bar{w} = x\bar{\phi}y\bar{\phi}^R, \text{ where } \langle\bar{w}\rangle > \bar{v}, \bar{\phi} \in \Sigma^{(n/2)-1}, x, y, \in \Sigma\right\}$$

$$\mathcal{PS}(\bar{v}) := \left\{\bar{w} \in \Sigma^n : \bar{w} = \bar{\phi}\bar{\phi}^R, \text{ where } \langle\bar{w}\rangle > \bar{v}, \bar{\phi} \in \Sigma^{(n/2)-1}\right\}$$

Unlike the set $\mathcal{PO}(\bar{v})$ in Section 4.1 the sets $\mathcal{PE}(\bar{v})$ and $\mathcal{PS}(\bar{v})$ do not correspond directly to bracelets greater than $\bar{v}$. For notation let $\mathcal{GE}(\bar{v})$ and $\mathcal{GS}(\bar{v})$ denote the number of bracelets greater than $\bar{v}$ of the form $x\bar{\phi}y\bar{\phi}^R$ and $\bar{\phi}\bar{\phi}^R$ respectively. The number of even length necklaces greater than $\bar{v}$ equals $\mathcal{GE}(\bar{v}) + \mathcal{GS}(\bar{v}) - (k - \bar{v}_1)$, where $k - \bar{v}_1$ denotes the number of symbols in Σ greater than $\bar{v}_1$. Before showing how to compute the size of these sets, it is useful to first understand how they are used to compute the rank amongst even length palindromic necklaces. Lemmas 19 and 18 shows how to covert the cardinalities of these sets into the number of even length palindromic necklaces smaller than $\bar{v}$. The main idea is to use the observations given by Propositions 14 and 16 to determine how many even length palindromic necklaces have either one or two words of the form $x\bar{\phi}y\bar{\phi}^R$ or $\bar{\phi}\bar{\phi}^R$.

▶ **Proposition 17.** *Let* $l = \frac{n+2}{4}$ *if* $\frac{n}{2}$ *is odd or* $l = \frac{n}{4}$ *if* $\frac{n}{2}$ *is even. The number of even length palindromic necklaces is given by* $\frac{1}{2}\left(k^{n/2}(k+2) + k^l\right) - k$.

Proof Sketch. This equation is derived by first determining the number of necklaces that has only one word of the form $x\bar{\phi}y\bar{\phi}^R$, from which the first $k^{n/2}$ term comes from. The number of necklaces with two representatives can be computed by subtracting the number of necklaces with one representative from the number of words of the form $x\bar{\phi}y\bar{\phi}^R$, giving $\frac{1}{2}(k^{n/2+1} - k^{n/2})$ By adding these two values together, the total number of necklaces of this form can be counted as $\frac{1}{2}(k^{n/2+1} - k^{n/2}) + k^{n/2} = \frac{1}{2}(k^{n/2+1} + k^{n/2})$. The number of necklaces with two words of the form $\bar{\phi}\bar{\phi}^R$ is counted by determining the number of necklaces with only one word of the form $\bar{\phi}\bar{\phi}^R$, giving the k^l term. Subtracting this from the number of words of the form $\bar{\phi}\bar{\phi}^R$, giving a total of $\frac{1}{2}(k^{n/2} - k^l) + k^l = \frac{1}{2}(k^{n/2} + k^l)$ necklaces. Finally, to avoid over counting words of the form x^n, the k necklaces of this form are subtracted, giving a total of $\frac{1}{2}\left(k^{n/2+1} + k^{n/2} + k^{n/2} + k^l\right) - k$. ◀

▶ **Lemma 18.** *The number of necklaces greater than* $\bar{v}$ *containing at least one word of the form* $x\bar{\phi}y\bar{\phi}^R$ *is given by* $GE(\bar{v}) = \frac{1}{2}\left(|\mathcal{PE}(\bar{v})| + \begin{cases} |\mathcal{PO}(\bar{v}_{[1,n/2]})| & \frac{n}{2} \text{ is odd.} \\ GE(\bar{v}_{[1,n/2]}) & \frac{n}{2} \text{ is even.} \end{cases}\right)$.

Proof Sketch. This claim is shown by dividing necklaces greater than $\bar{v}$ in to two sets, those with one word of the form $x\bar{\phi}y\bar{\phi}^R$, and those with two. By subtracting the set of necklaces with only a single such word from the total set, the number of necklaces with two such representatives are counted. The equation comes from adding the size of these sets. ◀

▶ **Lemma 19.** *The number of necklaces greater than $\bar{v}$ containing at least one word of the form $\bar{\phi}\bar{\phi}^R$ is given by* $GS(\bar{v}) = \frac{1}{2}\left(|\mathcal{PE}(\bar{v})| + \begin{cases} |\mathcal{PO}(\bar{v})| & \frac{n}{2} \text{ is odd.} \\ GS(\bar{v}_{[1,n/2]}) & \frac{n}{2} \text{ is even.} \end{cases}\right)$.

Proof Sketch. This claim is shown by dividing necklaces greater than $\bar{v}$ in to two sets, those with one word of the form $\bar{\phi}\bar{\phi}^R$, and those with two. By subtracting the set of necklaces with only a single such word from the total set, the number of necklaces with two such representatives can be counted. The final equation comes from adding the size of these sets together. ◀

High Level Idea for the Even Case. Lemmas 18 and 19 show how to use the sets $\mathcal{PS}(\bar{v})$ and $\mathcal{PE}(\bar{v})$ to get the number of necklaces of the form $x\bar{\phi}y\bar{\phi}^R$ and $\bar{\phi}\bar{\phi}^R$ respectively. This leaves the problem of computing the size of both sets. This is achieved in a manner similar to the one outlined in Section 4.1. At a high level the idea is to use two trees analogous to $\mathcal{TO}(\bar{v})$ as defined in Section 4.1. The tree $\mathcal{TE}(\bar{v})$ is introduced to compute the cardinality of $\mathcal{PE}(\bar{v})$ and the tree $\mathcal{TS}(\bar{v})$ is introduced to compute the cardinality of $\mathcal{PS}(\bar{v})$. As in Section 4.1, the trees $\mathcal{TE}(\bar{v})$ and $\mathcal{TS}(\bar{v})$ contain every prefix of a word in $\mathcal{PS}(\bar{v})$ or $\mathcal{PE}(\bar{v})$ respectively. The leaf vertices of these trees correspond to the words in these sets.

To compute the size of $\mathcal{PE}(\bar{v})$ using $\mathcal{TE}(\bar{v})$, the same approach as in Section 4.1 is used. A word $\bar{u}$ of length less than $\frac{n}{2}$ is a prefix of some word in $\mathcal{PE}(\bar{v})$ if and only if no subword of $(\bar{u}_{[1,|\bar{u}|-1]})^R\bar{u}$ is less than the prefix of $\bar{v}$ of the same length. This is slightly different from the odd case, where $\bar{u} \in \mathcal{PE}(\bar{v})$ if and only if there is no subword of $\bar{u}^R\bar{u}$ smaller than the prefix of $\bar{v}$ of the same length. To account for this difference the sets $\mathbf{PE}(\bar{v}, i, j, \bar{s})$ are introduced as analogies to the sets $\mathbf{PO}(\bar{v}, i, j, \bar{s})$.

▶ **Definition 20.** *Let $i \in [\frac{n+1}{2}], j \in [2i]$ and $\bar{s} \sqsubseteq_{2i} \bar{v}$. The set $\mathbf{PE}(\bar{v}, i, j, \bar{s})$ contains every word $\bar{u} \in \mathcal{TE}(\bar{v})$ of length i where (1) the longest suffix of $(\bar{u}_{[1,i-1]})^R\bar{u}_{[1,i]}$ which is a prefix of $\bar{v}$ has a length of j and (2) the word $(\bar{u}_{[1,i-1]})^R\bar{u}_{[1,i]}$ is strictly bounded by $\bar{s} \sqsubseteq_{2i-1} \bar{v}$.*

As in Section 4.1, the size of $\mathbf{PE}(\bar{v}, i, j, \bar{s})$ is computed via dynamic programming. The array $SizePE$ is introduced, storing the size of $\mathbf{PE}(\bar{v}, i, j, \bar{s})$ for every value of $i \in \left[\frac{n}{2}\right], j \in [2i-1]$ and $\bar{s} \sqsubseteq_{2i-1} \bar{v}$. Let $SizePE$ be and $n \times n \times n$ array such that $SizePE[i, j, \bar{s}] = |\mathbf{PE}(\bar{v}, i, j, \bar{s})|$. Lemma 21 shows that the techniques used in Lemma 11 can be used to compute $SizePE$ in $O(k \cdot n^3 \log(n))$ time. This is done by proving that the properties established by Lemma 10 regarding the relationship between the sets $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ also hold for the sets $\mathbf{PE}(\bar{v}, i, j, \bar{s})$. As words in $\mathcal{PS}(\bar{v})$ are of the form $\bar{\phi}\bar{\phi}^R$, a word $\bar{u}$ is in $\mathcal{TS}(\bar{v})$ if and only if no subword of $\bar{u}^R\bar{u}$ is less than the prefix of $\bar{v}$ of the same length. Note that this corresponds to the same requirement as the odd case. As such the internal vertices in the tree $\mathcal{TS}(\bar{v})$ may be partitioned in the same way as those of $\mathcal{TO}(\bar{v})$. Lemma 23 shows how to convert the array $SizePO$ as defined is Section 4.1 to the size of $\mathcal{PS}(\bar{v})$.

▶ **Lemma 21.** *Given $\bar{u}, \bar{w} \in \mathbf{PE}(\bar{v}, i, j, \bar{s})$ and $x \in \Sigma$. If $\bar{u}x \in \mathbf{PE}(\bar{v}, i+1, j', \bar{s}')$ then $\bar{v}x \in \mathbf{PE}(\bar{v}, i+1, j', \bar{s}')$. Further the values of j' and $\bar{s}'$ can be computed in constant time from the values of $j, \bar{s}$ and x. Therefore the array $SizePE[i, j, \bar{s}]$ can be computed for every value $i \in \left[\frac{n}{2}\right], j \in [2i-1]$ and $\bar{s} \sqsubseteq_{2i-1} \bar{v}$ in $O(k \cdot n^3 \cdot \log(n))$ time.*

Proof Sketch. By proving that all properties of $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ from Lemma 10 hold for $\mathbf{PE}(\bar{v}, i, j, \bar{s})$, the approach in Lemma 11 is used to compute $SizePE[i, j, \bar{s}]$ for every $i \in \left[\frac{n}{2}\right], j \in [2i-1], \bar{s} \sqsubseteq_{2i-1} \bar{v}$ in $O(k \cdot n^3 \cdot \log(n))$ time. ◀

▶ **Lemma 22.** *Let $\bar{v} \in \Sigma^n$. The size of $\mathcal{PE}(\bar{v})$ can be computed in $O(k \cdot n^3 \cdot \log(n))$ time.*

Proof Sketch. The size of $\mathcal{PE}(\bar{v})$ is computed in a similar manner to $\mathcal{PO}(\bar{v})$ using $SizePE$ in the same manner as $SizePO$. As before there are two cases to consider for each $\bar{w} \in \mathcal{PE}(\bar{v})$. The first case is where $\bar{w}_{[1,(n/2)]} \in \mathbf{PE}(\bar{v}, \frac{n}{2}, j, \bar{s})$ for some set of arguments $j \in [n-1], \bar{s} \sqsubseteq_{n-1} \bar{v}$. The second is where $\bar{w}_{[1,(n/2)-1]})^R \bar{w}_{[1,(n/2)]} \sqsubseteq_{n-1} \bar{v}$. In both cases the same approach as used in Lemma 13 can be used, determining the number of symbols $x \in \Sigma$ where $\langle (\bar{w}_{[1,(n/2)-1]})^R \bar{w}_{[1,(n/2)]} x \rangle \in \mathcal{PS}(\bar{v})$, using the values of j and $\bar{s}$ rather than directly computing this value for each word. ◀

The size of $\mathcal{PS}(\bar{v})$ is calculated in a similar manner. As the words in $\mathcal{PS}(\bar{v})$ are of the form $\bar{\phi}\bar{\phi}^R$, the prefixes of length i correspond to subwords of length $2i$ with the form $\bar{u}^R\bar{u}$. Note that these are the same as the prefixes used in Section 4.1 for odd length palindromic necklaces. As such, the sets $\mathbf{PO}(\bar{v}, i, j, \bar{s})$ are used to partition internal vertices of the tree $\mathcal{TS}(\bar{v})$. Lemma 23 shows how to use these sets to compute the size of $\mathcal{PS}(\bar{v})$.

▶ **Lemma 23.** *Let $\bar{v} \in \Sigma^n$. The size of $\mathcal{PS}(\bar{v})$ can be computed in $O(k \cdot n^3 \cdot \log(n))$ time.*

Proof Sketch. The size of $\mathcal{PS}(\bar{v})$ is computed using two cases. As before, for every word $\bar{w} \in \mathcal{PS}(\bar{v})$ either $(\bar{w}_{[1,(n/2)-1]})^R \bar{w}_{[1,(n/2)-1]} \sqsubseteq \bar{v}$. or there exists some set $\mathbf{PO}(\bar{v}, \frac{n}{2}-1, j, \bar{s})$ such that $\bar{w}_{[1,(n/2)-1]} \in \mathbf{PS}(\bar{v}, \frac{n}{2}-1, j, \bar{s})$. Following the same approach as in Lemmas 13 and 22, set of arguments $j \in [n-2], \bar{s} \sqsubseteq_{n-2} \bar{v}$ are used to compute the number of symbols $x \in \Sigma$ such that for $\bar{u} \in \mathbf{PS}(\bar{v}, \frac{n}{2}-1, j, \bar{s})$, $\langle \bar{u}x\bar{u}^R \rangle > \bar{v}$. Similarly, for every subword $\bar{u}\bar{u}^R \sqsubseteq_{n-2} \bar{v}$, the number of symbols $x \in \Sigma$ where $\langle \bar{u}x\bar{u}^R \rangle > \bar{v}$. ◀

Combining Lemmas 22 and 23 with Lemmas 18 and 19 provides the tools to compute the rank of $\bar{v}$ among even length palindromic necklaces. Lemma 24 shows how to combine these values to get the rank of $\bar{v}$ among even length palindromic necklaces.

▶ **Lemma 24.** *The rank of $\bar{v} \in \Sigma^n$ among even length palindromic necklaces can be computed in $O(k \cdot n^3 \cdot \log(n)^2)$ time.*

Proof. From Proposition 17, the number of even length palindromic necklaces is equal to $\frac{1}{2}\left(k^{n/2+1} + 2k^{n/2} + k^l\right) - k$, where $l = \frac{n+2}{4}$ if $\frac{n}{2}$ is odd, or $l = \frac{n}{4}$ if $\frac{n}{2}$ is even. Lemma 18 provides an equation to count the number of necklaces greater than $\bar{v}$ containing at least one word of the form $x\bar{\phi}y\bar{\phi}^R$. The equation given by Lemma 18 requires the size of $\mathcal{PE}(\bar{v})$ to be computed, needing at most $O(k \cdot n^3 \cdot \log(n))$ operations, and either $|\mathcal{PE}(\bar{v}_{[1,n/2]})|$ or $GE(\bar{v}_{[1,n/2]})$. As both $|\mathcal{PE}(\bar{v})|$ and $|\mathcal{PO}(\bar{v})|$ require $O(k \cdot n^3 \cdot \log(n))$ operations, the total complexity comes from the number of such sets that must be considered. As the prefixes of $\bar{v}$ that need to be computed is no more than $\log_2(n)$, the total complexity of computing $GE(\bar{v})$ is $O(k \cdot n^3 \cdot \log^2(n))$. Similarly as the complexity of computing $\mathcal{PS}(\bar{v})$ is $O(k \cdot n^3 \cdot \log(n))$, the complexity of computing $GS(\bar{v})$ is $O(k \cdot n^3 \cdot \log^2(n))$. ◀

▶ **Theorem 25.** *Give a word $\bar{v} \in \Sigma^n$, the rank of $\bar{v}$ with respect to the set of palindromic necklaces, $RP(\bar{v})$, can be computed in $O(k \cdot n^3 \cdot \log^2(n))$ time.*

Proof. The number of odd length palindromic necklaces is given by Proposition 8 as $k^{(n-1)/2}$. Lemma 13 shows that the size of set $\mathcal{PO}(\bar{v})$, corresponding to the number of odd length palindromic bracelets, can be computed in $O(k \cdot n^3 \cdot \log(n))$ time. By subtracting the size of

$\mathcal{PO}(\bar{v})$ from $k^{(n-1)/2}$, the rank of $\bar{v}$ can be computed in $O(k \cdot n^3 \cdot \log(n))$ time. Lemma 24 shows that of $RP(\bar{v})$ can be computed in $O(k \cdot n^3 \cdot \log^2(n))$ time if the length of $\bar{v}$ is even. Hence the total complexity is $O(k \cdot n^3 \cdot \log^2(n))$. ◀

5 Enclosing Bracelets

Following Lemma 4 and Theorem 25, the remaining problem is counting the number of enclosing words. This section will provide a technique to count the number of necklaces enclosing some word $\bar{v}$. As in the palindromic case, the structure of these words will first be analysed so that a more efficient algorithm can be derived.

▶ **Proposition 26.** *The bracelet representation of every bracelet $\hat{\mathbf{w}}$ enclosing the word $\bar{v} \in \Sigma^n$ can be written as $\bar{v}_{[1,i]}x\bar{\phi}$ where; $x \in \Sigma$ is a symbol that is strictly smaller than $\bar{v}_{[i+1]}$, and $\bar{\phi} \in \Sigma^*$ is a word such that every rotation of $(\bar{v}_{[1,i]}x\bar{\phi})^R$ is greater than $\bar{v}$.*

Proof Sketch. The claim is shown by proving that any word not of this form has a rotation of the reflection smaller than $\bar{v}$, contradicting the assumption that the bracelet encloses $\bar{v}$. ◀

▶ **Proposition 27.** *Given a bracelet $\hat{\mathbf{w}}$ enclosing the word $\bar{v} \in \Sigma^n$ of the form $\bar{v}_{[1,j]}x\bar{\phi}$ as given in Proposition 26. The value of x must be greater than or equal to $\bar{v}_{[(j+1) \bmod l]}$ where l is the length of the longest Lyndon word that is a prefix of $\bar{v}_{[1,j]}$.*

High Level Idea for the Enclosing Case. Similar to Sections 4.1 and 4.2, the main idea is to use the structure given in Proposition 26 as a basis for counting the number of enclosing bracelets. For each value of i and x, the number of possible values of $\bar{\phi}$ are counted. This is done in a recursive manner, working backwards from the last symbol. For each combination of i and x, the key properties to observe are that (1) every suffix of $\bar{\phi}$ must be greater than or equal to $\bar{v}_{[1,i]}x$ and (2) every rotation of $\bar{\phi}^R x\bar{v}^R_{[1,i]}$ is greater than $\bar{v}$.

These observations are used to create a tree, $\mathcal{TEN}(\bar{v}, i, x)$, where each vertex represents a suffix of some possible value of $\bar{\phi}$. Equivalently, the vertices of $\mathcal{TEN}(\bar{v}, i, x)$ can be thought of as representing the prefixes of $\bar{\phi}^R$. The leaf vertices of $\mathcal{TEN}(\bar{v}, i, x)$ represent the possible values of $\bar{\phi}$. As in Section 4, each layer of $\mathcal{TEN}(\bar{v}, i, x)$ is grouped into sets based on the lexicographical value of the reflection of the suffixes, and the prefixes of the suffixes. Let $t \in [|\bar{w}| - i]$, $j \in [t + i + 1]$ and $\bar{s} \sqsubseteq_{t+i+1} \bar{v}$. For the t^{th} layer of $\mathcal{TEN}(\bar{v}, i, x)$, the set $\mathcal{E}(\bar{v}, i, x, j, \bar{s})$ is introduced containing a subset of the vertices at layer t. The idea is to use the values of j and $\bar{s}$ to divide the prefixes at layer t by lexicographic value and suffix respectively. Let $\bar{u} \in \mathcal{E}(\bar{v}, i, x, j, \bar{s})$ be a suffix of some word $\bar{w}$ such that $\bar{v}_{[1,i]}x\bar{w}$ is a bracelet enclosing $\bar{v}$. To ensure that the necklace represented by the reflection is strictly greater than $\bar{v}$, j is used to track the longest prefix of $\bar{u}^R$ that is a prefix of $\bar{v}$. To ensure that there is no rotation of $x\bar{v}^R_{[1,i]}\bar{w}^R$, the subword $\bar{s} \sqsubseteq_t \bar{v}$ is used to bound the value of $\bar{u}^R$. Formally, $\mathcal{E}(\bar{v}, i, x, j, \bar{s})$ contains every suffix $\bar{u} \in \mathcal{TEN}(\bar{v}, i, x)$ of length i where (1) the longest prefix of $\bar{u}^R$ that is also a prefix of $\bar{v}$ and (2) the subword $\bar{s} \sqsubseteq_t \bar{v}$ bounds $\bar{u}^R$.

As in Section 4 the number of leaf vertices are calculated by determining the size of the sets $\mathcal{E}(\bar{v}, i, x, j, \bar{s})$ at layer $|\bar{v}| - i - 2$, and the number of children of each set. To determine the size of the sets, two key observations must be made. The first is that given the word $\bar{u} \in \mathcal{E}(\bar{v}, i, x, j, \bar{s})$ and the symbol $y \in \Sigma$, if $y\bar{u} \in \mathcal{TEN}(\bar{v}, i, x)$ then there exists some pair $j' \in [n], \bar{s}' \sqsubseteq_{|\bar{u}|+1} \bar{v}$ such that $y\bar{u} \in \mathcal{E}(\bar{v}, i, x, j', \bar{s}')$. Secondly, if $y\bar{u} \in \mathcal{E}(\bar{v}, i, x, j', \bar{s}')$, then $y\bar{w} \in \mathcal{E}(\bar{v}, i, x, j', \bar{s}')$ for every $\bar{w} \in \mathcal{E}(\bar{v}, i, x, j, \bar{s})$. These observations are proven in Lemma 28, as well as showing how to determine the values of j' and $\bar{s}'$.

▶ **Lemma 28.** *Given $\bar{u} \in \mathcal{E}(\bar{v}, i, x, j, \bar{s})$ and symbol $y \in \Sigma$, the pair $j' \in [n], \bar{s}' \sqsubseteq_{|\bar{u}|+1} \bar{v}$ such that $y\bar{u} \in \mathcal{E}(\bar{v}, i, x, j', \bar{s}')$ can be computed in constant time. Further, if $y\bar{u} \in \mathcal{E}(\bar{v}, i, x, j', \bar{s}')$, then $y\bar{w} \in \mathcal{E}(\bar{v}, i, x, j', \bar{s}')$ for every $\bar{w} \in \mathcal{E}(\bar{v}, i, x, j, \bar{s})$.*

Proof Sketch. The proof follows the same arguments as Lemma 10: the value of j' is either $j+1$ if $y = \bar{v}_{j+1}$, or 0 otherwise. Then, the value of $\bar{s}'$ is derived using the array XW. ◀

From Lemma 28, the size of $\mathcal{E}(\bar{v}, i, x, j, \bar{s})$ are computed using the sizes of $\mathcal{E}(\bar{v}, i, x, j', \bar{s}')$ for $j' \in [0, n]$ and $\bar{s}' \in \mathbf{S}(\bar{v}, |\bar{s}|+1)$. To compute the value of $\mathcal{E}(\bar{v}, i, x, j, \bar{s})$, an array SE of size $k \times n \times n \times n^2$ is introduced such that the value of $SE[x, i, j, \bar{s}] = |\mathcal{E}(\bar{v}, i, x, j, \bar{s})|$.

▶ **Lemma 29.** *Let $\bar{v} \in \Sigma^n$. Let SE be a $n \times n^2$ array such that $SE[x, i, j, \bar{s}] = |\mathcal{E}(\bar{v}, i, x, j, \bar{s})|$ for $j \in [0, n]$ and $\bar{s} \sqsubseteq \bar{v}$. Every value of $SE[x, i, j, \bar{s}]$ is computed in $O(k^2 \cdot n^4)$ time.*

Proof Sketch. SE is computed using Lemma 28. The value of $SE[x, i, j, \bar{s}]$ is computed starting with $|\bar{s}| = n-1$ for every value of x, i, and j. Then, by iteratively decreasing the length of $\bar{s}$, the value of $SE[x, i, j, \bar{s}]$ for each of the $n^3 \cdot k$ arguments of $x, i, j, \bar{s}$ can be computed in $O(n \cdot k)$ time; hence we need $O(k^2 \cdot n^4)$ time in total. ◀

Once SE has been computed, the number of enclosing words can be computed using SE and each valid combination of i and x. This is done in a direct manner. Note that the number of possible values of $\bar{\phi}$ such that $\bar{v}_{[1,i]}x\bar{\phi}$ represents a bracelet enclosing $\bar{v}$ is equal to $SE[x, i, j, \bar{s}]$ where j is the longest suffix of $\bar{v}_{[2,i]}x$ that is a prefix of $\bar{v}$ and $\bar{s}$ is the subword that bounds $x\bar{v}^R_{[1,i]}$. As both values can be computed naively in $O(n^2)$ operations, the complexity of this problem comes predominately from computing SE.

▶ **Theorem 30.** *The number of bracelets enclosing $\bar{v} \in \Sigma^n$ can be computed in $O(n^4 \cdot k^2)$.*

Proof. From Lemma 29 the array SE may be computed in $O(n^4 \cdot k^2)$ operations. Using SE, let $i \in [1, n]$ and $x \in \Sigma$. Further let l be the length of the longest Lyndon word that is a prefix of $\bar{v}_{[1,i]}$. If the value of x is less than $\bar{v}_{i+1 \bmod l}$ or greater than or equal to $\bar{v}_{i+1}$ then there is no bracelet represented by $\bar{v}_{[1,i]}x\bar{\phi}$. Similarly if $x\bar{v}^R_{[1,i]} < \bar{v}_{[1,i+1]}$, then any bracelet of the form $\bar{v}_{1,i}x\bar{\phi}$ does not enclose $\bar{v}$. Otherwise, the number of enclosing bracelets represented by $\bar{v}_{[1,i]}x\bar{\phi}$ is equal to $SE[x, i, j, \bar{s}']$ where j is the longest suffix of $\bar{v}_{[2,i]}x$ that is a prefix of $\bar{v}$ and $\bar{s}$ is the subword that bounds $x\bar{v}^R_{[1,i]}$. By summing the value of $SE[x, i, j, \bar{s}']$ for each value of $i \in [1, n]$ and $x \in \Sigma$ such that $\bar{v}_{[1,i]}x$ is the prefix of the representation of some bracelet enclosing $\bar{v}$ gives the number of enclosing bracelets. Therefore

$$RE(\bar{v}) = \sum_{i \in [1,n-1]} \sum_{x \in \Sigma} \begin{cases} 0 & x\bar{v}^R_{[1,i]} < \bar{v} \\ 0 & x \leq \bar{v}_{i+1 \bmod l} \text{ or } x > \bar{v}_{i+1} \\ SE[x, i, j, \bar{s}'] & \textit{Otherwise.} \end{cases}$$ ◀

Proof of Theorem 5. The tools are now available to prove Theorem 5 and show that it is possible to rank a word $\bar{v} \in \Sigma^n$ with respect to the set of bracelets of length n over the alphabet Σ in $O(k^2 \cdot n^4)$ steps. To rank bracelets, it is sufficient to use the results of ranking $\bar{v}$ with respect to necklaces, palindromic necklaces and bracelets enclosing $\bar{v}$, combining them as shown in Lemma 4. Sawada et. al. provided an algorithm to rank v with respect to necklaces in $O(n^2)$ time. It follows from Theorem 25 that the rank with respect to palindromic necklaces can be computed in $O(k \cdot n^3)$ time. Theorem 30 shows that the rank with respect to bracelets enclosing v can be computed in $O(k^2 \cdot n^4)$ time. As combining these results can be done in $O(1)$ steps, therefore the overall complexity is $O(k^2 \cdot n^4)$. ◀

Conclusions and Future Work. In this work we have presented an algorithm for the ranking of bracelets in $O(k \cdot n^4)$ time. Additionally, we have presented a complimentary $O(n^4 \cdot k^2 \cdot \log(k))$ time algorithm for unranking. This expands upon the previous work on ranking necklaces and Lyndon words in $O(n^2)$ time, and unranking in $O(n^3)$ time. This leaves the question of if there is a faster algorithm for ranking bracelets, which may be found by deriving a faster algorithm to count the number of enclosing bracelets.

In addition to the importance of the results from the perspective of combinatorics on words, a practical application of combinatorial necklaces and bracelets can be found in the field of chemistry, since they provide discrete representation of periodic motives in crystals. The problems on finding diverse and representative samples of languages of necklaces and bracelets has served as a heuristic in the exploration of the space of crystal structures [2, 3], since the problem is considered to be NP-hard [1]. The essential component for building representative samples require efficient procedures for ranking bracelets.

References

1 Duncan Adamson, Argyrios Deligkas, Vladimir V. Gusev, and Igor Potapov. On the Hardness of Energy Minimisation for Crystal Structure Prediction. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12011 LNCS, pages 587–596. Springer, January 2020. doi:10.1007/978-3-030-38919-2_48.

2 Duncan Adamson, Argyrios Deligkas, Vladimir V. Gusev, and Igor Potapov. The K-Centre Problem for Necklaces. *CoRR*, May 2020. arXiv:2005.10095.

3 C. Collins, M. S. Dyer, M. J. Pitcher, G. F. S. Whitehead, M. Zanella, P. Mandal, J. B. Claridge, G. R. Darling, and M. J. Rosseinsky. Accelerated discovery of two crystal structure types in a complex inorganic phase field. *Nature*, 546(7657):280–284, 2017.

4 Clelia De Felice, Rocco Zaccagnino, and Rosalba Zizza. Unavoidable sets and circular splicing languages. *Theoretical Computer Science*, 658:148–158, 2017. Formal Languages and Automata: Models, Methods and Application In honour of the 70th birthday of Antonio Restivo. doi:10.1016/j.tcs.2016.09.008.

5 Aris Filos-Ratsikas and Paul W. Goldberg. The complexity of splitting necklaces and bisecting ham sandwiches. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 638–649. ACM, 2019. doi:10.1145/3313276.3316334.

6 R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete mathematics : a foundation for computer science*. Addison-Wesley, 1994.

7 U. I. Gupta, D. T. Lee, and C. K. Wong. Ranking and unranking of B-trees. *Journal of Algorithms*, 4(1):51–60, March 1983. doi:10.1016/0196-6774(83)90034-2.

8 Elizabeth Hartung, Hung Phuc Hoang, Torsten Mütze, and Aaron Williams. Combinatorial generation via permutation languages. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1214–1225. SIAM, 2020. doi:10.1137/1.9781611975994.74.

9 S. Karim, J. Sawada, Z. Alamgir, and S. M. Husnine. Generating bracelets with fixed content. *Theoretical Computer Science*, 475:103–112, March 2013. doi:10.1016/j.tcs.2012.11.024.

10 Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 1st edition, 2011.

11 T. Kociumaka, J. Radoszewski, and W. Rytter. Computing k-th Lyndon word and decoding lexicographically minimal de Bruijn sequence. In *Symposium on Combinatorial Pattern Matching*, pages 202–211. Springer, 2014.

12 S. Kopparty, M. Kumar, and M. Saks. Efficient indexing of necklaces and irreducible polynomials over finite fields. *Theory of Computing*, 12(1):1–27, 2016.

13 Martin Mareš and Milan Straka. Linear-time ranking of permutations. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms – ESA 2007*, pages 187–193, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

14 Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001. doi:10.1016/S0020-0190(01)00141-7.

15 J. M. Pallo. Enumerating, Ranking and Unranking Binary Trees. *The Computer Journal*, 29(2):171–175, February 1986. doi:10.1093/comjnl/29.2.171.

16 J. Sawada and A. Williams. Practical algorithms to rank necklaces, Lyndon words, and de Bruijn sequences. *Journal of Discrete Algorithms*, 43:95–110, 2017.

17 Joe Sawada. Generating bracelets in constant amortized time. *SIAM Journal on Computing*, 31(1):259–268, January 2001. doi:10.1137/S0097539700377037.

18 Toshihiro Shimizu, Takuro Fukunaga, and Hiroshi Nagamochi. Unranking of small combinations from large sets. *Journal of Discrete Algorithms*, 29:8–20, 2014. doi:10.1016/j.jda.2014.07.004.

19 S. G. Williamson. Ranking algorithms for lists of partitions. *SIAM Journal on Computing*, 5(4):602–617, 1976. doi:10.1137/0205039.

The k-Mappability Problem Revisited*

Amihood Amir
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
Georgia Tech, Atlanta, GA, USA

Itai Boneh
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel

Eitan Kondratovsky
Department of Computer Science, Bar Ilan University, Ramat Gan, Israel
Cheriton School of Computer Science, Waterloo University, Waterloo, Canada

Abstract

The k-mappability problem has two integers parameters m and k. For every subword of size m in a text S, we wish to report the number of indices in S in which the word occurs with at most k mismatches.

The problem was lately tackled by Alzamel et al. [1]. For a text with constant alphabet Σ and $k \in O(1)$, they present an algorithm with linear space and $O(n \log^{k+1} n)$ time. For the case in which $k = 1$ and a constant size alphabet, a faster algorithm with linear space and $O(n \log(n) \log\log(n))$ time was presented in [2].

In this work, we enhance the techniques of [2] to obtain an algorithm with linear space and $O(n \log(n))$ time for $k = 1$. Our algorithm removes the constraint of the alphabet being of constant size. We also present linear algorithms for the case of $k = 1$, $|\Sigma| \in O(1)$ and $m = \Omega(\sqrt{n})$.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Sorting and searching

Keywords and phrases Pattern Matching, Hamming Distance, Suffix Tree, Suffix Array

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.5

Funding *Amihood Amir*: Partly supported by ISF grant 1475/18, BSF grant 2018141 and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme grant agreement No 683064.
Itai Boneh: Partly supported by ISF grant 1475/75.
Eitan Kondratovsky: Partly supported by ISF grant 1475/18.

Acknowledgements We warmly thank Tomasz Kociumaka for useful discussions.

1 Introduction

Many real world applications need to identify events that repeat very often. Examples of such applications are road traffic peaks [12], load peaks on web servers [9], monitoring events in computer networks [3], life event histories [10] and many others. Finding such events often leads to useful insights by shedding light on the structure of the data, and giving a basis to predicting future events and behavior. Moreover, in some applications frequent events can point out a problem. In a computer network, for example, repeating error messages can indicate a misconfiguration, or even a security intrusion such as a port scan [7].

In Stringology, the problem of counting the occurrences of every subword of length m that appears in text S is a well-known exercise in the power of suffix trees [13] or suffix arrays [6,8]. However, in reality one seldom finds *exact* repetitions of a substring. The situation becomes more complex when we seek the most frequent subword that *approximately* occurs in the string.

* This work is part of the second author's Ph. D. dissertation.

Let $S[1 \dots n]$ be a text and k and m two integers. The *k-mappability problem* is defined as follows:

▶ **Definition 1.** *For every index $i \in [1 \dots n-m+1]$, report the number of indices j such that $HD(S[i \dots i+m-1], S[j \dots j+m-1]) \leq k$. With $HD(X,Y)$ denoting the Hamming distance between X and Y.*

The k-mappability problem was lately tackled by Alzamel et al. [1]. For a text with constant alphabet and $k \in O(1)$, they present an algorithm with linear space and $O(n \log^{k+1} n)$ time. Additionally, they present a quadratic algorithm for reporting the k-mappability for a fixed value of k and every $m \in [k \dots n]$ or a fixed value of m and every $k \in [0 \dots m]$. Finally, they show that the k-mappability problem can not be solved in truly subquadratic time unless the Strong Exponentional Time Hypothesis is false. For the case in which $k = 1$ and a constant size alphabet, a faster algorithm with linear space and $O(n \log(n) \log\log(n))$ time was presented in [2]. [2] also presented an algorithm with average case linear time for $k = 1$, and provided some experimental results.

Our results:

1. By enhancing the techniques of [2], we construct an algorithm for k-mappability with linear space and $O(n \log n)$ time for $k = 1$ and infinite integer alphabet. This is an improvement over the $O(n \log^{k+1} n)$ time achieved by [1] for $k \in O(1)$. It also improves the faster $O(n \log(n) \log\log(n))$ time for $k = 1$ achieved by [2]. In the settings in which infinite integer alphabet is allowed, our algorithm is optimal.
2. We present a linear time algorithm for k-mappability in the case in which $k = 1$, the alphabet size is constant and $m \in \Omega(\sqrt{n})$.

The paper is organized as follows. In Section 2 we define the basic notions used. Section 3 presents a linear space $O(n \log n)$ time algorithm for 1-mappability. In Section 4 we present a linear algorithm for 1-mappability with constant sized alphabet and $m \in \Omega(\sqrt{n})$.

2 Preliminaries

Let Σ be an alphabet. A *string* S over Σ is a finite sequence of characters from Σ. By $S[i]$, for $1 \leq i \leq |S|$, we denote the i^{th} character of S. The *empty string* is denoted by ϵ. By $S[i \dots j]$ we denote the string $S[i] \dots S[j]$ called a *substring*, or *factor*, of S (if $i > j$, then the substring is the empty string). A substring is called a *prefix* if $i = 1$ and a *suffix* if $j = |S|$. The prefix of length j is denoted by $S[\dots j]$, while by $S[i \dots]$ we denote the suffix which starts from index i in S. We say that a string S of length n has a *period* p, for some $1 \leq p \leq \frac{n}{2}$ if $S[i] = S[i+p]$ for every $i \in [1 \dots n-p]$. *The period* of S, denoted as $per(S)$, is the smallest p that is a period of S. We say that a substring of S, denoted as $A = S[a \dots b]$ is a *run* with period p if its period is p, but $S[a-1] \neq S[a-1+p]$ and $S[b+1] \neq S[b+1-p]$. This means that no substring containing A has a period p. The *Hamming distance* of two n-length strings, S_1, and S_2, denoted as $HD(S_1, S_2)$, is the number of indices in which they differ. We say that an m-length word w has a k-ham occurrence in location i of string S if $HD(w, S[i \dots i+m-1]) \leq k$.

The longest common prefix (suffix) of two indexes $i, j \in [1 \dots n]$ is the maximal integer ℓ such that $S[i \dots i+\ell-1] = S[j \dots j+\ell-1]$ ($S[i-\ell+1 \dots i] = S[j-\ell+1 \dots j]$). We denote the $LCP(i,j) = \ell$ ($LCS(i,j) = \ell$). LCP and LCS are collectively referred to as longest common extensions (LCE).

The *suffix tree* [13] is a useful string data structure.

▶ **Definition 2.** *Let $S_1, \ldots, S_k$ be strings over alphabet Σ and let $\$ \notin \Sigma$.*

An uncompacted trie of strings $S_1, \ldots, S_k$ *is an edge-labeled tree with k leaves. Every path from the root to a leaf corresponds to a string S_i with a $\$$ symbol appended to its end. The edges on this path are labeled by the symbols of S_i. Strings with a common prefix start at the root and follow the same path of the prefix, the paths split where the strings differ.*

A compacted trie *is the uncompacted trie with every chain of edges connected by degree-2 nodes contracted to a single edge whose label is the concatenation of the symbols on the edges of the chain.*

Let $S = S[1], \ldots, S[n]$ be a string over alphabet Σ. Let $\{S_1, \ldots, S_n\}$ be the set of suffixes of S, where $S_i = S[i \ldots]$, $i = 1, \ldots, n$. A suffix tree *of S is the compacted trie of the suffixes $S_1, \ldots, S_n$.*

For every node u, we call the concatenation of the labels on the path from the root to u the *locus* of u denoted as $\mathcal{L}(u)$. For an edge e in the compact trie, we use the same notation $\mathcal{L}(e)$ to denote the label (or the locus) of e. Finally, for a downwards path P in the compact trie, the locus $\mathcal{L}(P)$ is the concatenation of the loci of the edges in P. In a compact trie, an edge e can have label s.t. $|\mathcal{L}(e)| > 1$. We refer to the symbol $\mathcal{L}(e)[1]$ as the symbol of e.

▶ **Theorem 3** (Weiner [13])**.** *For finite alphabet Σ, the suffix tree of a length-n string can be constructed in time $O(n)$. For general alphabets it can be constructed in time $O(n \log \sigma)$, where $\sigma = \min(|\Sigma|, n)$.*

The suffix tree can be preprocessed in $O(n)$ [4] to be used as a data structure for LCE queries with $O(1)$ query time.

We assume that every node u in the suffix tree contains some auxiliary information about $\mathcal{L}(u)$, that is the number of occurrences of $\mathcal{L}(u)$ in the text S and a pointer to the list of indices in which $\mathcal{L}(u)$ occurs. This information can be evaluated for all the nodes of a given suffix tree ST in $O(|ST|)$ time and require an additional $O(|ST|)$ space.

Over finite alphabet Σ, the adjacency list of a node $u \in ST$ is represented as an array $A_u[1 \ldots |\Sigma|]$ with the edge with symbol $\sigma \in \Sigma$ in $A_u[\sigma]$ (or an emptiness indicator if there is no edge with that symbol).

Over infinite alphabet, the adjacency list of $u \in ST$ is represented as a balanced search tree storing the edges emerging from u in a sorted order of their symbols. In our algorithm, we assume that the representation of the adjacency list allows linear time DFS iteration on the subtree rooted in a node $u \in ST$. This is indeed the case for most balanced trees.

▶ **Definition 4.** *The* suffix array *of a string S, denoted as $SA(S)$, is an integer array of size $n + 1$ storing the starting positions of all (lexicographically) sorted non-empty suffixes of S, i.e. for all $1 < r \leq n + 1$ we have $S[SA(S)[r - 1]..n] < S[SA(S)[r]..n]$. Note that the empty suffix is explicitly added to the array.*

The suffix array of S corresponds to a pre-order traversal of all the leaves of the suffix tree of S. Various algorithms exist for efficient time and space construction of the suffix array [5, 6, 11]. In particular, the suffix array over a fixed finite alphabet can be constructed in linear time.

3 $O(n\log(n))$ time and $O(n)$ space algorithm for $k = 1$

3.1 An overview of the $O(n\log(n)\log\log(n))$ algorithm for $k = 1$

We start with an overview of the ideas for the $O(n\log(n)\log\log(n))$ algorithm of [2]. They present an algorithm for counting the number of occurrences with **exactly** one mismatch, for every word of size m. Since there is a textbook algorithm for counting the number of exact occurrences of every word, this is sufficient for solving the 1-mappability problem.

They start by evaluating the suffix tree T of S and trimming the tree at word length m. That is, every node v with $|\mathcal{L}(v)| > m$ is removed. Implicit nodes with $|\mathcal{L}| = m$ are made explicit leaves in the trimming process. They proceed to evaluate the heavy paths decomposition of T.

▶ **Definition 5** (Heavy Path Decomposition). *Let T be a rooted tree. For every non-leaf vertex u, the edge (u, v) is heavy if $|I_u| < 2|I_v|$ with I_x denoting the set of leaves in the subtree rooted in the vertex x. An edge that is not heavy is called a light edge. The heavy path of a vertex v is the maximal path of heavy edges going through v (it may contain 0 edges). For every heavy path P, a vertex $u \in P$, and a light edge (u, v) emerging from u, we call $T(v)$ a sidetree of P (emerging from u).*

It is easy to observe that every root-to-leaf path in T consists of at most $\log(n)$ heavy paths and $\log(n)$ light edges. The following observation is the key for the complexity achieved by [2]:

▶ **Observation 6.** *For every $w = S[i \dots i + m - 1]$, every 1-ham occurrence of w $w' = w[1 \dots x - 1]\sigma w[x + 1 \dots m]$ with a mismatch in index x corresponds to a node u in T with $\mathcal{L}(u) = w[1 \dots x - 1]$. u must have two edges e_1, e_2 s.t. there is a downwards path starting with e_1 (resp. e_2) and ending in a leaf with path label $w[x \dots m]$ (resp. $\sigma w[x + 1 \dots n]$).*

Consider the following procedure: For every node $u \in T$ with path label w, let the heavy edge emerging from u be e_h with label d. Inspect every light edge $e = (u, v)$ with label c emerging from u. For every leaf $z \in T(v)$ with label $L(z) = w \cdot c \cdot w_z$ and for every $c' \neq c \in \Sigma$, find the leaf z' with label $\mathcal{L}(z') = w \cdot c' \cdot w_z$, if it exists. If it does, add the number of occurrences of $\mathcal{L}(z')$ to a counter associated with z. For the leaf z_d with $\mathcal{L}(z_d) = w \cdot d \cdot w_z$, also increment a counter associated with z_d by the amount of occurrences of $\mathcal{L}(z)$.

It is straightforward from Observation 6 that for every index i, every 1-ham occurrence is counted by the above procedure. As for complexity - every leaf z is iterated once per light edge in the path from the root to z. A single iteration on a leaf z consists of a constant number of counter increments and a single query for finding z' with $L(z') = w \cdot c' \cdot w_z$ per symbol $c' \in \Sigma$. Since $|\Sigma| = O(1)$, the bottleneck of the iteration is finding z'. The following is proven in [2]:

▶ **Theorem 7.** *A text $S[1 \dots n]$ can be preprocessed in time $O(n\log\log n)$ and linear space to allow the following query in $O(\log\log n)$ time:*

Given a node u in the suffix tree of S with $L(u) = w_1 \cdot c \cdot w_2$ ($w_1, w_2 \in \Sigma^*$ and $c \in \Sigma$) and a symbol $c \neq c' \in \Sigma$, find the node u' with $L(u') = w_1 \cdot c' \cdot w_2$ if it exists.

We call the queries described in Theorem 7 *concatenation* queries.

With Theorem 7 the final complexity is clear – every leaf is iterated $O(\log n)$ times and the iteration costs $O(\log\log n)$ after an $O(n\log\log n)$ preprocessing time. The overall time complexity is $O(n\log(n)\log\log(n))$

3.2 Linear space $O(n \log n)$ algorithm for $k = 1$

Intuition: Our algorithm is based on the ideas of [2]. For every light edge (u, v) we iterate every leaf $z \in T(v)$ and wish to find the vertices corresponding to a 1-ham occurrence of $\mathcal{L}(z)$ with a mismatch in index $|\mathcal{L}(u)| + 1$. Instead of using concatenation queries, we construct a lexicographically sorted array of the words W we need to find. Given the sorted array of words, finding the vertices corresponding to these words in the suffix tree can be done in $O(|W|)$. If we manage to construct this sorted array in $O(|W|)$, the amortized time for inspecting a leaf is constant (rather than $O(\log \log n)$).

Terminology. Let $P = (u_1, u_2, \ldots u_x)$ be a heavy path in the heavy path decomposition of the suffix tree ST of S. Let $\mathcal{L}(u_i) = w_i$ and let $e_i = (u_i, u_{i+1})$ be the i'th heavy edge in P with symbol d_i. Let (u_i, v) be a light edge emerging from u_i with symbol c and let $z \in T(v)$ be a leaf. It holds that $\mathcal{L}(z) = w_i \cdot c \cdot s_z$ for some suffix $s_z \in \Sigma^*$.

▶ **Definition 8.** *The node $z' \in ST$ is a P-light occurrence of z if $L(z') = w_i \cdot c' \cdot s_z$ for some $c' \in \Sigma \setminus \{c, d_i\}$. We call the word $hw(z) = w_i \cdot d_i \cdot s_z$ the P-heavy word of z. The node $z' \in ST$ is a P-heavy occurrence of z if $L(z') = hw(z)$.*

Note that the above definitions are with respect to a heavy path P. z may be a leaf in the sidetrees of multiple heavy paths. In every such path, the P-light occurrences, P-heavy occurrence and the P-heavy word of z are different. Also note that the P-heavy word and the P-heavy occurrence are undefined for leaves in the sidetrees emerging from u_x, as the last heavy edge in P is e_{x-1}.

In our algorithm, we count the P-heavy occurrences and the P-light occurrences of every node z in a sidetree of P independently. For every heavy occurrence z', we also count the occurrences of $w(z)$ as 1-ham occurrences of $w(z')$. We do this for every heavy path P. Surely, this process counts all the 1-ham occurrences.

We start by showing how to efficiently count the P-light occurrences.

▶ **Observation 9.** *For every vertex z with $\mathcal{L}(z) = w$ in a sidetree $T(v)$ emerging from u_i, all the P-light occurrences of z are also leaves in (different) sidetrees emerging from u_i. Furthermore, a leaf z' with $\mathcal{L}(z') = w'$ in a sidetree $T(v') \neq T(v)$ emerging from u_i is a light occurrence of z iff $w[|w_i + 2| \ldots m] = w'[|w_i + 2| \ldots m]$.*

Observation 9 is directly derived from the definition of a light occurrence. For every $u_i \in P$, we wish to construct a sorted array consisting of the suffixes starting in index $|w_i| + 2$ of the labels of the leaves of the sidetrees emerging from u_i.

We present the following routine:

Algorithm 1 Suffix Sorting.

As a preprocess procedure, construct the suffix array SA of S.

Initialize an array A of size n consisting of empty lists

Alignment step: Iterate the leaves in the sidetrees of u_i. For every leaf z, extract j_z - a starting index of $L(z)$. We add z to $A[j_z + |w_i| + 2]$.

Insertion step: Initialize an empty list L. Iterate SA from left to right. When iterating $SA[j]$, add all the nodes in $A[SA[j]]$ to the end of L.

▷ Claim 10. After running Suffix Sorting, L is sorted by the lexicographic order of the suffixes starting in index $|w_i| + 2$ of z. The running time is $O(n + |SE|)$ with SE being the set of sorted elements.

Proof. leaf z with $\mathcal{L}(z) = w_z$ occurring in index j_z is inserted to L before the leaf y with $L(y) = w_y$ occurring in index j_y only if the suffix of S starting in $j_z + |w_i| + 2$ is lexicographically smaller or equal to the suffix of S starting in $j_y + |w_i| + 2$. Therefore, it can not be the case that $w_z[|w_i| + 2 \dots m] >_L w_y[|w_i| + 2 \dots m]$.

As for complexity- the alignment step takes $O(|SE|)$ time as it executes a constant amount of list insertions and basic arithmetic operations for every leaf. The insertion step takes $O(|SE| + n)$ time as it iterates over the entire suffix array. The sum of the sizes of the lists in A is identical to the amount of iterated leaves in the alignment step. We assume that the suffix array was evaluated prior to the run of Suffix Sorting. Therefore, we exclude the complexity of computing the suffix array from our running time. ◁

This is not exactly what we want. If we execute Suffix Sorting for every node, the n factor will dominate the complexity and the overall time will be quadratic. To avoid that, we present the following algorithm for sorting a batch of sidetrees.

Algorithm 2 Batched Suffix Sorting.

As a preprocess procedure, construct the suffix array SA of S.

Input: A batch of vertices $v_1, v_2 \dots v_b$

Initialize an array A of size n consisting of empty lists.

Batched Alignment step:

For every $i \in [1 \dots b]$:

1. Initialize an empty list L_i
2. Iterate the leaves in the sidetrees of v_i with $L(v_i) = w_i$. For every leaf z, extract j_z - a starting index of $\mathcal{L}(z)$. Add the pair (z, L_i) to $A[j_z + |w_i| + 2]$.

Batched Insertion step: Iterate SA from left to right. When iterating $SA[j]$, for every $(z, L) \in A[SA[j]]$, add z to the end of L.

The same arguments as in the proof of claim 10 can be made to prove the following:

▷ Claim 11. After running Batched Suffix Sort, every list L_i has the leaves in the sidetrees of v_i sorted by the lexicographic order of the suffixes starting in index $|w_i| + 2$ of z. The running time is $O(n + |SE|)$ with SE being the set of sorted elements in the batch.

To sort the sidetrees in amortized linear time, we set a counter $se = 0$ for the amount of leaves in the sidetrees that need to be sorted and an empty list $Sort$. We iterate the vertices in ST. For every vertex u, we count the number of leaves in the sidetrees of u, add this number to se and add u to $Sort$. Once $se > n$, we execute Batched Suffix Sort on $Sort$.

Since the number of leaves in the sidetrees of a vertex u never exceeds n, it is guaranteed that $se \leq 2n$ when we execute Batched Suffix Sorting. Therefore, the overall complexity is $O(n + 2n) = O(n)$. Since we only execute the batched insertion with $se \geq n$, the amortized time for placing every leaf in the sorted list is constant.

Once we have the sorted list $L = z_1, z_2 \dots z_t$ of the leaves in the sidetrees emerging from $u_i \in P$, a simple iteration can be implemented to count the number of P-light occurrences for every node in L. We start by preprocessing S for constant time lcp queries. We iterate

L. For every consecutive pair of leaves z_a and z_{a+1} with $\mathcal{L}(z_a) = s_1$ occurring in j_1 and $\mathcal{L}(z_{a+1}) = s_2$ occurring in j_2, we query $l = lcp(j_1 + |w_i| + 2, j_2 + |w_i| + 2)$. If we have $l \geq m - |w_i| - 1$, then z_a and z_{a+1} are P-light occurrences of each other (Observation 9). Once we identify a pair z_a, z_{a+1} of P-light occurrences, we proceed in L until we reach a leaf z_{b+1} that is not a P-light occurrence. Of course, all the pairs z_x, z_y with $x \neq y$ and $x, y \in [a \dots b]$ are P-light occurrences of each other. We evaluate the sum Oc of occurrences of $\mathcal{L}(z_x)$ for $x \in [a \dots b]$ and increment the counter of ham-1 occurrences of z_x by $Oc - Oc(z_x)$ with $Oc(z_x)$ being the number of occurrences of $\mathcal{L}(z_x)$.

It can be easily verified that the iteration is linear. For every leaf we execute a single lcp query and a constant number of basic arithmetic operations. We conclude the handling of P-light occurrences with the following theorem:

▶ **Theorem 12.** *The 1-ham occurrences of $\mathcal{L}(z)$ that are corresponding to P-light occurrences of some heavy path P can be computed for every leaf $z \in ST$, in $O(n \log n)$ time and linear space.*

Proof. For every heavy path P and vertex $u_i \in P$, we compute the sorted list of the suffixes starting in $|w_i| + 2$ of the words of the leaves of the sidetrees emerging from u_i. We use the sorted list to find the P-light occurrences of the leaves in the sidetrees of u_i. Sorting the leaves is done using Batched Suffix Sorting with batches of size between n and $2n$ and takes a constant amortized time per sorted leaf. There may be one 'remainder' batch with size $se < n$ that takes an additional $O(n)$ time to sort. Given the sorted lists, finding the P-light occurrences is linear in the number of leaves in the sidetrees of u_i. Every leaf z participates in at most $\log(n)$ different sidetrees, so the overall time is $O(n \log n)$. We also build the Suffix array as a preprocess step, which takes an additional $O(n \log n)$ time.

As for space - the only non-trivially linear part of our solution is the array A used in Batched Suffix Sort. Since we never let se the number of sorted elements exceed $2n$, the lists in A never contain more than $2n$ elements collectively. So the size of A is always linear. After executing the Batched Suffix Sorting, we iterate the sorted lists to count the P-light occurrences and then reuse the space occupied by these lists as they are no longer required. ◀

We are left with the task of counting the P-heavy occurrences of every leaf z. Consider a heavy path $P = u_1, u_2 \dots u_x$. Our key sub-task for finding all the P-heavy occurrences of all the leaves in the sidetrees of P is constructing a sorted list of the P-heavy words of the leaves.

Note that unlike P-light occurrences, P-heavy occurrences of a leaf z of a sidetree emerging from u_i can not be in a sidetree emerging from u_i. However, they must be leaves of a sidetree emerging from u_j for some $j > i$.

The process of building the sorted list of P-heavy words relies on the same principles we used for the P-light occurrences. However, there is a further difficulty to tackle. With P-light occurrences that lie on the same u_i - we have a guarantee that the words match until the index $|w_i|$. Therefore, it is sufficient to sort by the suffixes starting right after the mismatch in index $|w_i| + 2$. With the P-heavy words, we may have to compare P- heavy words from sidetrees of different nodes u_i and u_j with $i < j$. In this case, there is no guarantee that the words match in the indices in $[|w_i + 1| \dots |w_j|]$.

To handle this difficulty, we partition the leaves into classes prior to sorting them. Our partition will have the property that the P-heavy words of leaves in the same class have a certain common prefix that exceeds the index in which the error occurs (c is replaced by d_i). This property will allow us to sort the P-heavy words in every class using Batched Suffix Sorting.

The first step for sorting the P-heavy words is to partition the leaves in the sidetrees of $P = u_1, u_2 \dots u_x$ by the *lcp* of their P-heavy words with w_x. This is done with the following procedure:

Algorithm 3 LCP Partition.

Input: A heavy path $P = u_1, u_2 \dots u_x$

Initialize an array $LCP[1 \dots m]$ of size m of empty lists. Let j_x be an index in which w_x occurs.

Alignment Step: For every $i \in [1 \dots x]$:
For every leaf z in a sidetree emerging from u_i:

1. Extract an index j_z in which $\mathcal{L}(z)$ occurs in S.
2. Find $l_z = lcp(hw(z), w_x)$ by computing $l_z = min(|w_i| + 1 + lcp(j_z + |w_i| + 2, j_x + |w_i| + 2), |w_x|)$.
3. Compare between the symbols in index $l_z + 1$ in $hw(z)$ and in w_x in order determine the lexicographical order $o_z \in \{<, >, =\}$ between $hw(z)$ and w_x (For example, $o_z =<$ if $hw(z) <_L w_x$). If $l_z = |w_x|$, o_z is set to $' ='$.
4. Add the tuple (z, o_z) to $LCP[l_z]$.

Insertion Step:

For every $l \in [1 \dots |w_x| - 1]$ (in increasing order):

1. If the list $L = LCP[l]$ is empty - do nothing.
2. Otherwise, create 2 lists $L_l^>$ and $L_l^<$.
3. For every tuple (z, o_z), add z to $L_l^{o_z}$.

If $L = LCP[|w_x|]$ is not empty, construct a new list $L_{|w_x|}$ and add z to $L_{|w_x|}$ for every pair $(z, =) \in L$.

Note that $l_z \geq |w_i|+1$ since $hw(z)[1 \dots |w_i|] = w_i = w_x[1 \dots |w_i|]$ and $hw(z)[|w_i+1|] = d_i$. With that observation, it is clear that the formula for finding l_z in Step 2 works.

We make the following observation:

▶ **Observation 13.** *For every list $L_l^>$ (or $L_l^<$ or $L_{|w_x|}$), every vertex $z \in L_l^>$ has $lcp(hw(z), w_x) = l$ and $hw(z)[l + 1 \dots m] = \mathcal{L}(z)[l + 1 \dots m]$. The running time of LCP partition is $O(m + |SE|)$ with SE being the set of leaves in sidetrees of P.*

Proof. The *lcp* property is derived directly from the construction of $LCP[1 \dots m]$. As for complexity, every leaf is processed with a single *lcp* query and a constant amount of basic operations. The iteration and construction of LCP takes an additional $O(m)$ ◀

It follows from Observation 13 that the lexicographical order between the P-heavy words of the vertices in $L_l^>$ (or $L_l^<$ or L_{w_x}) are determined by the suffixes starting in index $l + 1$ of $L(z)$. Therefore, sorting $L_l^>$ by the lexicographical order of the heavy words can be done using the algorithm Batched Suffix Sorting.

▶ **Theorem 14.** *The lexicographically sorted list of P-heavy words of a heavy path P can be evaluated in $O(|SL|)$ amortized time with SL the set of leaves in the sidetrees of P*

Proof. We want to use LCP Partition on a batch of heavy paths. Transforming LCP Partition to a batched algorithm can be done with the same technique that was used to generate Batched Suffix Sorting from Suffix Sorting.

As in the Batched Suffix Sorting algorithm, we execute the alignment step of LCP Partition for possibly multiple heavy paths P until the collective amount of leaves considered is between n and $2n$. Once this amount is met, we construct the lists $L_l^>$, $L_l^<$ and $L_{|w_x|}$ for all the paths in the batch by applying the insertion step. We then sort the lists by the lexicographic order of $hw(z)$ with Batched Suffix Sorting. The overall time is $O(n+m) = O(n)$.

We are left with the task of merging the sorted lists $L_l^>$,$L_l^<$ and $L_{|w_x|}$ into a single sorted list L containing all the P-heavy words. This is done by applying the following observation:

▶ **Observation 15.** *Let $l_1, l_2, \ldots l_c$ be the set of indices for which either $L_{l_i}^<$ or $L_{l_i}^>$ is constructed for the path P by LCP Partition. The sorted list L of the P-heavy words is of the form $L = L_{l_1}^<, L_{l_2}^<, \ldots L_{l_c}^<, L_{|w_x|}, L_{l_c}^>, L_{l_{c-1}}^> \ldots L_{l_1}^>$. (If L_{l_i} was not constructed, it is considered as an empty list)*

Proof. We start by showing that the lists $L_{l_i}^<$ must appear in increasing order of l_i in L. Let $a, b \in \{l_1, l_2 \ldots l_c\}$ be two indices for which $a < b$. Let $w_a \in L_a^<$ and $w_b \in w_b^<$. Since the LCP of w_b and w_x is $b \geq a+1$, we have $w_b[a+1] = w_x[a+1]$. Since the LCP of w_a and w_x is a and $w_a <_L w_x$, we have $lcp(w_a, w_b) = a$ and $w_a[a+1] <_L w_x[a+1] = w_b[a+1]$ and therefore $w_a <_L w_b$. Similar arguments can be made to prove that $w_b <_L w_a$ for every $w_a \in L_{l_i}^>$ and $w_b \in L_b^>$. It is straight forward from the construction of the lists $L_{l_i}^<$ and $L_{l_i}^>$ that for every l_i and every $W \in L_{l_i}^>$, $w \in L_{l_i}^<$ and $w' \in L_{|w_x|}$ we have $w <_L w' <_L W$. ◀

With Observation 15, the construction of the sorted P-heavy words list is completed. Observe that LCP Partition naturally generates $L_l^>$ and $L_l^<$ in increasing order of l. Therefore, the concatenation of the lists in the order dictated by Observation 15 does not require any further sorting and can be executed in linear time, and the proof of Theorem 14 is completed. ◀

Given the sorted list $L_P[1 \ldots h]$ of the P-heavy words, we are interested in finding the node z' with $L(z') = hw(z)$ for every word $hw(z) \in L_P$. We can do this in linear time as follows: First, observe that every P-heavy word has the prefix w_1. So z' ,if it exists, must be a descendant of u_1 and therefore is a leaf in a sidetree of P. Let $L[1 \ldots l]$ be the sorted list of occurrences of w_1 stored in u_1. These are actually all the leaves in the sidetrees of P. The following procedure matches every $hw(z) \in L_P$ with its corresponding z':

Algorithm 4 Count P-Heavy.

Input: The lexicographically sorted lists $L_P[1 \ldots h]$ of P-heavy words and $L[1 \ldots l]$ the list of lexicographically sorted vertexes in the sidetrees of P

Initialize two indices $i = j = 1$.

While $i \leq h$ and $j \leq l$:

1. Let $hw(z) = L_P[i]$ and $\mathcal{L}(z') = L[j]$
2. **If $hw(z) = \mathcal{L}(z')$:**
   a. Increase the counter associated with z by $Oc(z')$.
   b. Increase the counter associated with z' by $Oc(z)$.
   c. Increase i by 1.
3. **If $hw(z) <_L \mathcal{L}(z')$:** Increase i by 1.
4. **If $hw(z) >_L \mathcal{L}(z')$:** Increase j by 1.

It can be easily verified that Count P-Heavy counts the P-heavy occurrence z' of every leaf z in a sidetree of P. Notice that double counting will not occur. That is due to the following:

▶ **Fact 16.** *Let z and z' be two leaves in sidetrees of P emerging from u_i and u_j respectively such that $hw(z) = \mathcal{L}(z')$. It must be the case that $i < j$.*

Fact 16 guarantees that if we count the occurrences of z as 1-ham occurrences of z' and vice versa when $hw(z)$ and $\mathcal{L}(z')$ are be visited in Count P-Heavy, we will not count them as 1-ham occurrences of each other again, because it can't be the case that $hw(z') = \mathcal{L}(z)$.

The lexicographic comparisons between $hw(z)$ and $\mathcal{L}(z')$ can be executed in constant time using *lcp* queries. To efficiently execute an *lcp* query with a P-heavy word, we store the P-heavy word $hw(z)$ as a pair (z, i) with i the index in which $\mathcal{L}(z)$ is modified. With that representation, two *lcp* queries can be used to find $a = lcp(hw(z), \mathcal{L}(z'))$ in a 'kangooroo' jump manner. If $a < m$, the following symbol can be compared to determine the lexicographic order between $hw(z)$ and $\mathcal{L}(z')$. With the constant time lexicographic comparing, it is easy to see that the complexity of Count P-Heavy is $O(|L_P| + |L|) = O(|SE|)$ with SE being the set of leaves in the sidetrees of P.

Note that when the equality $hw(z) = L(z')$ is met, it is crucial to increase i rather than j. That is due to the fact that L_P may contain duplicates while L does not. Alternatively, L_P can be preprocessed to group duplicates together. We conclude the counting of P-heavy occurrences with the following:

▶ **Theorem 17.** *The P-heavy occurrences of every leaf z can be counted over all the heavy paths P such that z is a leaf in a sidetree of P in $O(n \log n)$ time and linear space.*

Proof. For every heavy path $P = u_1, u_2 \dots u_x$, we use Theorem 14 to obtain the list L_P of sorted P-heavy words and obtain L from u_1. We then apply Count P-Heavy on L_P and P to match every $hw(z) \in L_P$ with its P-heavy occurrence z' if exists, and update the corresponding counters accordingly.

The amortized time for applying Theorem 14 for a path P is $O(|SE(P)|)$ with $SE(P)$ being the set of leaves in the sidetrees of P. Every leaf in ST is a leaf in the sidetree of at most $\log(n)$ heavy paths, so the overall complexity is $O(n \log n)$. We also construct the suffix array as a preprocess procedure, which takes an additional $O(n \log n)$ time.

As for space, the only non-trivially linear part is the array $LCP[1 \dots m]$ used in LCP Partition. As before, we apply LCP Partition on batches of size at most $2n$, so the collective size of the lists in $LCP[1 \dots m]$ never exceeds $O(n)$. After obtaining L_P for all the paths in the batch, we apply Count P-Heavy for every path in the batch and then reuse the space occupied by the sorted lists L_P as they are no longer required. ◀

When put together, Theorem 12 and Theorem 17 yield the main result of this section:

▶ **Theorem 18.** *The 1-mappability problem can be solved using $O(n \log n)$-time and linear space on a text with infinite integer alphabet.*

Note that for infinite integer alphabet, better time can not be achieved unless certain values of m are excluded. For example:

▶ **Observation 19.** *For a text S over infinite integer alphabet and $m = 2$, there is an index $i \in [1 \dots n]$ with at least 1-ham occurrence iff the symbols of $S \cdot \sigma'$ are not distinct for some $\sigma' \notin \Sigma$.*

The above straight forward observation shows a trivial relation between the k-mappability problem and reporting whether or not all the elements of a set are distinct - which can not be done in $o(n \log n)$. It can be easily generalized for every fixed value of m.

4 $O(\frac{n^2}{m^2} + n)$ and Linear Space Algorithm for 1-Mappability with Constant sized Alphabet

As a warm up, we present a technique for counting the 1-ham occurrence of a word with size m in $O(\frac{n}{m})$ time. Applying this technique to every m-sized word yields an $O(\frac{n^2}{m})$ algorithm for 1-mappability. We then proceed to show how to process all the words of size m not one by one, but in batches of size $O(m)$. We extend the technique used in the warm up to handle a batch in $O(\frac{n}{m} + m)$ time. Since there are $O(\frac{n}{m})$ batches, this yields an $O((\frac{n}{m})^2 + n)$ time algorithm.

4.1 Warm up – $O(\frac{n^2}{m} + n)$

Let $w = S[i \dots i + m - 1]$ be a subword of S with length m.

▶ **Definition 20.** *Let $w_1 = S[j \dots j+m-1]$ be a 1-ham occurrence of w. w_1 is an l-occurrence of w if $w_1[1 \dots \lceil \frac{m}{2} \rceil] = w[1 \dots \lceil \frac{m}{2} \rceil]$. w_1 is an r occurrence of w if $w_1[\lceil \frac{m}{2} \rceil + 1 \dots m] = w[\lceil \frac{m}{2} \rceil + 1 \dots m]$. We respectively denote as $Lo(w)$ and $Ro(w)$ the sets of l-occurrences and r-occurrences of w in S.*

It is easy to see that $|Lo(w)| + |Ro(w)| - \#w$ is the number of 1-ham occurrences of w, with $\#w$ denoting the number of proper occurrences of w in S. In this section, we show how to evaluate the number of l-occurrences of a given word w in $O(\frac{n}{k})$ time. A symmetrical approach can be applied to count the number of r-occurrences of w. $\#w$ can be evaluated for all the subwords of S in $O(n)$ time using the suffix tree.

▶ **Theorem 21.** *All the occurrences of a string w of size m in a text of size n can be represented by a set of $O(\frac{n}{m})$ arithmetic progressions of the form $A = (s, e, d)$ such that $A = (s, e, d)$ represent a sequence of occurrences with starting indexes $\{i_x = s + d \cdot x | x \geq 0, i_x \leq e\}$. If w is periodic, every arithmetic progression $A = (s, e, d)$ has $d = per(w)$. $|A| = e - s + 1$ represents the number of occurrences represented by A. Every arithmetic progression that has $A > 1$ corresponds to a periodic set of instances contained within a run with period d. This representation is called the periodic occurrences representation of w and it can be obtained in $O(\frac{n}{m})$ time from the suffix tree following an $O(n)$ time preprocessing.*

A proof for the above can be found in Section A.2.

Given a words $w = S[i \dots i + m - 1]$, we use Theorem 21 to obtain all the occurrences of $w_L = w[1 \dots \lceil \frac{m}{2} \rceil]$ in periodic occurrences representation. For every occurrence of w_L in this representation, we wish to check if it is a prefix of an l-occurrence.

We process every arithmetic progression $A = (s, e, d)$ of occurrences of w_L. If A only represents a single occurrence of w_L in index s, we query $l_1 = LCP(s, i)$. If $l_1 \geq m$, we have a proper instance of w. Otherwise, we have a mismatch. Proceed to query $l_2 = LCP(s + l_1 + 1, i + l_1 + 1)$. If it is the case that $l_2 + l_1 + 1 \geq m$, we count s as an l-occurrence of w.

If $A = (s, e, d)$ represents multiple occurrences of w_L, then w_L must have a period d. We exploit the periodic structure of the occurrences represented by A to compute l_2 for all the occurrences in A using constant time. The following lemma proven in Section A.2 is the key for doing so.

▶ **Lemma 22.** *Let $A = (s, e, d)$ be an arithmetic progression representing a set of indexes $s_j = s + j \cdot d$ for $j \in [0 \dots |A| - 1]$ within a run with period d.*

Let $i \in [1 \dots n]$ be an index and let $l_p = lcp(i, s)$. Let Ex_i be the maximal extension of a run with period d containing i to the right of i (regardless of periodicity, $Ex_i \geq d$), and let Ex_s be the maximal extension of the period d to the right of s.

1. *If $l_p < d$: $LCP(i, s_j) = l_p$ for every $j \in [0 \dots |A| - 2]$.*
2. *Otherwise, $LCP(i, s_j) = min(Ex_i, Ex_s - j \cdot d)$ for every $j \in [0 \dots |A| - 1]$ such that $Ex_i \neq Ex_s - j \cdot d$.*

We exploit Lemma 22 to efficiently implement the following subroutine (details proof for the following can be found in Section A.2.

▶ **Lemma 23.** *Given an arithmetic progression $A = (s, e, d)$ representing the indexes $\{s_j = i + j \cdot d | j \in [0 \dots |A| - 1]\}$ that are contained within the same run with period d, and an index $s \in [1 \dots n]$. The values $lcp_j = LCP(s, s_j)$ can be evaluated and represented in $O(1)$ following $O(n)$ preprocess time on S.*

The representation consists of pairs (I, L) such that $I = [a \dots b]$ is a consecutive interval of j values and L is an integer such that one of the following holds:
1. *$lcp_j = L$ for every $j \in I$*
2. *$lcp_j = L - j \cdot d$ for every $j \in I$.*

Every pair is stored alongside with a bit indicating which one of the above holds for this pair.

In the process of evaluating the representation of lcp_j for $A = (s, e, d)$ and i, at most one of the indexes in A is called the aligned index. In the case in which $l_p < d$, the aligned index is $|A| - 1$. In the case in which $l_p \geq d$, j^* such that $Ex_s - j^* \cdot d$ is the aligned index, provided that it is an integer. We mark the pair representing the LCP value of the aligned index.

We employ Lemma 23 to obtain a representation of l_1^j for every $j \in [0 \dots |A| - 1]$. After obtaining this representation, we are left with the task of applying a second LCP query after the mismatch index for every s_j (That will be the equivalent of finding l_2). Namely, for every s_j we need to compute $l_2^j = LCP(i + l_1^j + 1, s_j + l_1^j + 1)$. More precisely , we need to count the number of j values for which $l^j = l_1^j + l_2^j + 1 \geq m$.

For every pair $(I = [a \dots b], L)$, we wish to evaluate l_2^j for $j \in I$ by employing Lemma 23 again. In order to do that, we first need to prove that the settings of Lemma 23 are satisfied in the second evaluation. We prove the following lemma in Section A.2.

▶ **Lemma 24.** *For every pair (I, L) in the output of Lemma 23 on $A = (s, e, d)$ and i that is not corresponding to an aligned occurrence, one of the below holds for $j \in I$.*
1. *$i + lcp_j + 1$ is a fixed value and $s_j + lcp_j + 1$ is an arithmetic progression of indexes within a run with period d with difference d.*
2. *$i + lcp_j + 1$ is an arithmetic progression of indexes within a run with period d with difference d and $s_j + lcp_j + 1$ is a fixed value*

It follows from Lemma 24 that Lemma 23 can be applied to each of the pairs representing the non aligned indexes to evaluate a representation of lcp_2^j for every non aligned index j in $O(1)$ time. The aligned index, if exists, has its l_2^j evaluated individually.

The above process outputs a set of (at most) 4 non-singular intervals for which the values of l_2^j and l_1^j are represented either as an arithmetic progression or as a fixed value. We can easily deduce the amount of occurrences s_j with $l_1^j + l_2^j + 1 \geq m$ from this representation. We sum the amount of occurrences of w_L with $l_1^j + l_2^j + 1 \geq m$ over all the arithmetic progressions of occurrences to obtain $|Lo(w)|$. A symmetric procedure can be constructed to evaluate

$|Ro(w)|$ using occurrences of $w_R = w[\lceil \frac{m}{2} + 1 \rceil \dots m]$. We can use the suffix tree to obtain $\#w$ (the number of occurrences of w within S) for every m-length word in S in $O(n)$ time. The number of occurrences of w with at most one mismatch is $|Ro(w)| + |Lo(w)| - \#w$. substracting $\#w$ is required to omit double counting. We do this process for every word of size m.

Complexity. For a word w of size m, we process the arithmetic progressions of occurrences of w_L. For every arithmetic progression A, we evaluate a representation of lcp_1^j and lcp_2^j in constant time using Lemma 23. We deduce the number of l-occurrences corresponding to the occurrence of w_L represented by A from the representation of lcp_1^j and lcp_2^j in constant time. We execute a symmetric procedure to deduce the number of r-occurrences of w as well. There are $O(\frac{n}{m})$ arithmetic progressions in periodic occurrences representation of w_L, so counting the 1-ham occurrences of a single word takes $O(\frac{n}{m})$. We do this for every word of size m, so it adds up to $O(\frac{n^2}{m})$. There is an additional $O(n)$ preprocessing time prior to the iteration on the words to enable LCP queries, suffix tree construction and access to the periodic occurrences representation. The overall complexity is $O(\frac{n^2}{m} + n) = O(\frac{n^2}{m})$.

4.2 Reducing the complexity to $O(\frac{n^2}{m^2} + n)$

For reducing the complexity by a factor of m, we present a technique for obtaining $|Lo(w)|$ for a batch containing $O(m)$ words in $O(\frac{n}{m} + m)$ time. Consider the consecutive set of words with length m starting in the indices $[i \dots i + \frac{m}{4}]$. For every word $w^t = S[i + \frac{m}{4} - t \dots i + \frac{m}{4} - t + m - 1]$ with $t \in [0 \dots \frac{m}{4}]$ in this set, the left half of w^t denoted as $w_L^t = S[i + \frac{m}{4} - t \dots i + \frac{m}{4} - t + \lceil \frac{m}{2} \rceil]$ contains the word $w_L^i = S[i + \frac{m}{4} \dots i + \frac{m}{2} - 1]$. We use the occurrences of w_L^i to evaluate $|Lo(w^t)|$ for every $t \in [0 \dots \frac{m}{4}]$ as we did in the previous section. A symmetric process can be constructed for computing $|Ro(w)|$.

We start by finding the arithmetic progression representation of the occurrences of w_L^i. For simpler notation, we denote the starting and ending indices of w_L^i as $w_L^i = S[s_i \dots e_i]$. For every cluster A with occurrences $\{s_j = s + d \cdot j | j \in [0 \dots |A| - 1]\}$, let $r_1^j = lcp(s_j, s_i)$ and $r_2^j = lcp(s_j + r_1^j + 1, s_i + r_1^j + 1)$. As in the previous section, we use Lemma 23 to obtain a compact representation of $r^j = r_1^j + r_2^j + 1$ for every index s_j represented by A.

Using the following Lemma ,that can be proved similarly to Lemma 23, we obtain a compact representation of $l^j = LCS(s_i - 1, s_j - 1)$ for every $j \in [0 \dots |A| - 1]$.

▶ **Lemma 25.** *Given an arithmetic progression $A = (s, e, d)$ representing the indexes $\{s_j = i + j \cdot d | j \in [0 \dots |A| - 1]\}$ that are contained within the same run with period d, and an index $s \in [1 \dots n]$. The values $lcs_j = LCS(s, s_j)$ can be evaluated and represented in $O(1)$ following $O(n)$ preprocess time on S.*

The representation consists of pairs (I, L) such that $I = [a \dots b]$ is a consecutive interval of j values and L is an integer such that one of the following holds:

1. *$lcs_j = L$ for every $j \in I$*
2. *$lcs_j = L + j \cdot d$ for $j \in I$.*

After obtaining the values of l_j and r_j, our next task is deducing for every s_j represented by A, what are the values of t for which s_j is corresponding to an l-occurrence of w^t. The following observation is the key for doing so.

▶ **Observation 26.** *$s_j - t$ is an l-occurrence of w^t iff $r^j \geq m - t$ and $l^j \geq t$.*

Observation 26 allows us to associate every occurrence s_j with a continuous interval $I = [a \dots b]$ such that s_j is extendable to an l-occurrence of w^t for and only for $t \in I$.

We initialize a data structure D for maintaining $\frac{m}{4}$ counters $C_0, C_1, C_2 \dots C_{\frac{m}{4}}$. C_t counts l-occurrences of w^t. Initially, $C_t = 0$ for every $t \in [0 \dots \frac{k}{4}]$. We already know from observation 26 that for every s_j, the indexes C_t with $t \in [m - r_j \dots l_j]$ need to be increased by 1. We call this type of updates, in which a consecutive interval of counters is increased by a constant value, an interval increment update. There are folklore techniques for applying this kind of updates to an array of counters efficiently.

Unfortunately, an efficient data structure for applying interval increment updates will not be sufficient for our cause, as we wish to process the effect of a **set** of occurrences on D. We therefore need to explore the structure of the set of updates $[m - r_j \dots l_j]$ derived from the occurrences s_j represented by a cluster A.

We present the following types of updates to be applied to an array of counters D.

▶ **Definition 27.** *An interval increment is represented by a triplet (a, b, x). Applying (a, b, x) to D results in every counter C_t with $t \in [a \dots b]$ being increased by x.*

An increasing stairs update *is represented by a triplet (a, b, p). The update requires applying the following modifications on D:*

For every $d \in 1 \dots \lfloor \frac{b-a+1}{x} \rfloor$ Counters C_i with $i \in [a + p \cdot (d-1) \dots a + p \cdot d - 1]$ are increased by d. The counters with $t \in [a + p \cdot \lfloor \frac{b-a+1}{x} \rfloor \dots b]$ are increased by $\lfloor \frac{b-a+1}{x} \rfloor + 1$

A decreasing *stairs update is also represented by a triplet (a, b, p). The update requires applying the following modifications on D:*

For every $d \in 1 \dots \lfloor \frac{b-a+1}{x} \rfloor$ Counters C_i with $i \in [b - p \cdot d + 1 \dots b - p \cdot (d-1)]$ are increased by d. The counters with $t \in [a \dots b - p \cdot \lfloor \frac{b-a+1}{x} \rfloor]$ are increased by $\lfloor \frac{b-a+1}{x} \rfloor + 1$

We call the interval $[a \dots b]$ the span *of the stairs. We call the interval that is increased by d the dth step of the update. p is called the* width *of the stairs update.*

A negative *stairs update (either increasing or decreasing) is a stairs update in which the counter in the dth step is decreased by d rather than being increased by d.*

▶ **Example 28.** Let $x = 10$ and an array $D = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$. An increasing stairs update $(3, 8, 2)$ on D will result in the counters being set to $(0, 0, 1, 1, 2, 2, 3, 3, 4, 0, 0)$. Applying a decreasing stairs update $(1, 5, 2)$ on the updated counters in D will result in the counters being set to $(3, 2, 3, 2, 3, 2, 3, 3, 4, 0, 0)$.

It turns out that a constant number of interval updates and stairs updates can be used to express the updates derived from the occurrences s_j represented by a cluster A. In Section A.2, we prove the following:

▶ **Lemma 29.** *Given a cluster A of occurrences of w_L^i, the set of updates that need to be applied to D in order to represent the l-occurrences corresponding to occurrences s_j with $j \in [0 \dots |A| - 1]$ can be represented by a constant number of stairs updates and interval increment updates. Given A and the representation of l_j and r_j, this set of stairs and interval increment updates can be retrieved in $O(1)$ time.*

Over all the clusters representing occurrences of w_L^i, every stairs update (a, b, p) in the representation has the same stairs width p which is the period of w_L^i.

Our algorithm runs as follows: Initialize a data structure D for maintaining a set of $\frac{m}{4}$ counters. Find all the occurrences of w_L^i in arithmetic progression representation. For every one of the $O(\frac{n}{m})$ arithmetic progressions, find the arithmetic progressions representing r^j and l^j. Apply Lemma 29 to obtain an $O(1)$ size set of interval increment update and stairs update that represents the required modifications to be applied to D. The final ingredient for our algorithm is a data structure that enables the efficient application of these updates. In the full version of this paper, we prove the following.

▶ **Theorem 30.** *An array of t counters can be maintained to support stairs updates in $O(1)$ time per update. Retrieving the values of all the counters in the array takes $O(t+u)$ time with u being the amount of applied updates. The data structure works in the restricted settings in which every update (a, b, p) has the same p value.*

Note that the restriction on the queries hold in our case, since the step width is always p the period of w_L^i in all the stairs updates constructed in Lemma 29.

Every update corresponding to a set of occurrences of a certain type is applied to D in $O(1)$ by employing the data structure of Theorem 30 all the updates take $O(\frac{n}{m})$ by applying Theorem 30.

Note that we need a data structure for handling interval increment updates with the same complexities as the data structure of Theorem 30. The construction of such a data structure is quite simple and may be considered folklore. We therefore omit the implementation details of this data structure.

After applying the updates, we query our data structure for the values of all the counters. This process takes $O(m + \frac{n}{m})$ time. This is done for batches of $\frac{m}{4}$ consecutive indices. The indices of S are partitioned to $4\frac{n}{m}$ such batches. We also preprocess the text for constant time *lcp* and *lcs* queries and construct the suffix tree. The total running time is $O(n + \frac{n}{m}(m + \frac{n}{m})) = O(n + \frac{n^2}{m^2})$. Recall that we described a procedure for evaluating $|Lo(w)|$. A symmetric procedure can be constructed to evaluate $|Ro(w)|$.

Note that for $m \in \Omega(\sqrt{n})$, $O(\frac{n^2}{m^2} + n)$ is dominated by $O(n)$. The main result of this section immediately follows.

▶ **Theorem 31.** *For constant size alphabet and $m \in \Omega(\sqrt{n})$, the 1-mappability problem can be solved in time $O(n)$.*

References

1. Mai Alzamel, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, and Juliusz Straszynski. Efficient computation of sequence mappability. In Travis Gagie, Alistair Moffat, Gonzalo Navarro, and Ernesto Cuadros-Vargas, editors, *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, volume 11147 of *Lecture Notes in Computer Science*, pages 12–26. Springer, 2018. doi:10.1007/978-3-030-00479-8_2.
2. Mai Alzamel, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, Jakub Radoszewski, and Wing-Kin Sung. Faster algorithms for 1-mappability of a sequence. *Theor. Comput. Sci.*, 812:2–12, 2020. doi:10.1016/j.tcs.2019.04.026.
3. S. Bagchi, E. Hung, A. Iyengar, N. G. Vogl, and N. Wadia. Capacity planning tools for web and grid environments. In *Proc. 1st International Conference on Performance Evaluation Methodolgies and Tools (VALUETOOLS)*, 2006. ISBN = 1-59593-504-5, article number 25, http://doi.acm.org/10.1145/1190095.1190127.
4. Farach-Colton M. Bender M.A. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004. Latin American Theoretical Informatics. doi:10.1016/j.tcs.2003.05.002.
5. L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006.
6. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 03)*, pages 943–955, 2003. LNCS 2719.

7 S. Ma and J.L. Hellerstein. Mining partially periodic event patterns with unknown periods. In *Proc. 17th International Conference on Data Engineering (ICDE)*, pages 205–214. IEEE Computer Society, 2001.

8 U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 319–327, 1990.

9 V. V. Panteleenko. *Instantaneous Offloading of Web Server Loads*. PhD thesis, University of Notre Dame, 2002.

10 G. Ritschard, R. Bürgin, and M. Studer. Exploratory mining of life event histories. In J.J.McArdle and G. Ritschard, editors, *Contemporary Issues in Exploratory Data Mining in Behavioral Sciences*, pages 221–253. Routeledge, New York, 2013.

11 K. Sadakane. A fast algorithm for making suffix arrays and for burrows-wheeler transformation. In *Proc. Data Compression Conference (DCC)*, pages 129–138, 1998.

12 Federal Highway Administration U.S. Department of Transportation. Conjestion: a national issue. http://www.ops.fhwa.dot.gov/aboutus/opstory.htm, August 2011.

13 P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

A Appendix

A.1 Complementary Figures

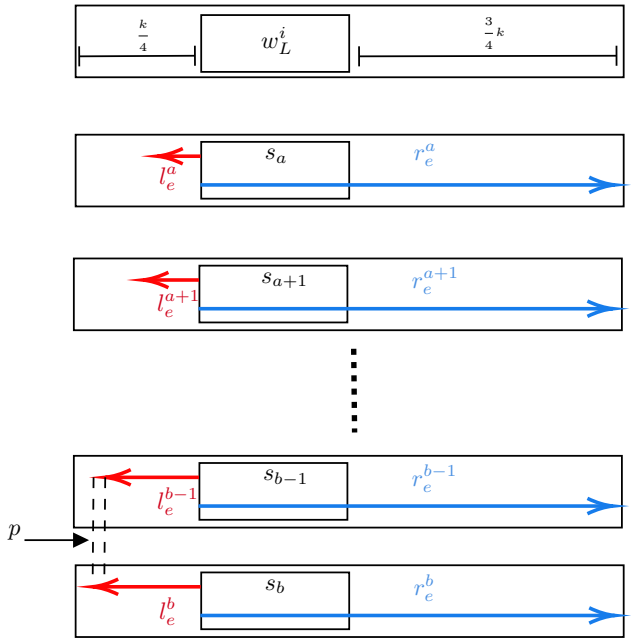


Figure 1 An illustration of the stairs update derived from a set of type 2 occurrences. Note that for every type 2 occurrence, the red arrow representing l^j represents the interval of values of t for which C_t should be incremented due to an occurrence of w_t in $s_j - t$. The p indices that are only contained by the lowest step will be increased by 1. The next p indices are contained within two stairs and will be increased by 2. And so on.

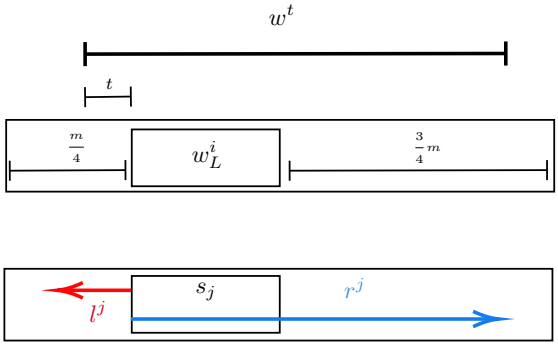


Figure 2 A demonstration of w_L^i aligned with an occurrence s_j. Every m sized word that fits within the interval spanned by l^j and r^j (Red arrow and blue arrow, respectively) is an l-occurrence of a word w_t that occurs in $s_j - t$.

A.2 Complementary Proofs For Section 4

Proof for Theorem 21. The existence of the representation specified by Theorem 21 follows directly from the following facts:

▶ **Fact 32.** *An aperiodic string of length m can have up to $O(\frac{n}{m})$ occurrences in a string S of length n*

▶ **Fact 33.** *Let w be a periodic word with period p and length k. The distance between the starting points of two occurrences of w in a string S is either p or greater than $\frac{m}{2}$.*

As for efficiently obtaining the periodic occurrences representation from the suffix tree, we present the following algorithm for preprocessing the suffix tree.

Algorithm 5 Periodic Occurrences Representation Preprocess.

Input: A suffix tree ST

For every node $v \in ST$ with $|L(v)| = m$:
1. Initialize an empty list L_v that is linked to v.
2. Initialize a period $p_v = -1$
3. Initialize two auxiliary integers $pre_v = 0$ and $runstart_v = 0$.

Initialize an array $A[1 \dots n - m + 1]$ with $A[i] = v$ such that v is the node in ST with $L(v) = S[i \dots i + m - 1]$.

For every $i \in [1 \dots n]$:
1. Let $v = A[i]$
2. If $pre_v = 0$, set $pre_v = i$ and $runstart_v = i$.
3. Otherwise:
   a. If $i - pre_v > \frac{m}{2}$, add the pair $(runstart_v, pre_v)$ to L_v and set $pre_v = i$ and $runstart_v = i$.
   b. Otherwise, set $i - pre_v = p_v$ and $pre_v = i$.

For every L_v:
1. Add the pair $(runstart_v, pre_v)$ to L_v.
2. Replace every pair $(s, e) \in L_v$ with the tuple (s, e, p_v).

▷ Claim 34. L_v contains the periodic occurrences representation of $L(v)$.

Proof. For a periodic $L(v)$, the correctness of Claim 34 directly follows from Fact 32 and the value of p_v is irrelevant since every arithmetic progression will be a singleton. If $L(v)$ is periodic, every sequence of occurrences such that every occurrence starts $p = per(L(v))$ indexes to the right of the previous one will be represented as a single arithmetic progression. According to Fact 33, the distance between the starting indexes of two such sequences of occurrences is at least $\frac{m}{2}$, and therefore $|L_v| \in O(\frac{n}{m})$. ◁

A can be initialized in time $O(n)$ using the suffix tree. The rest of the algorithm is obviously linear. With that, the proof of Theorem 21 is complete. ◀

Proof for Lemma 22. The correctness of the first case follows from the fact that every s_j within the run has the same d symbols to its right, possibly excluding the rightmost s_j.

As for the second case, note that the extension of the period from occurrence s_j is $Ex_s - j \cdot d$. It holds that $S[i + x] = S[s_j + x]$ for every $d \leq x < min(Ex_i, Ex_s - j \cdot d)$. This is due to the fact that for every such x , $S[i + x - d] = S[i + x]$. And the first d symbols to the right of $s_j$and i are equal. If $Ex_i < Ex_s - j \cdot d$, The equality is broken in $S[i + Ex_i] \neq S[s_j + Ex_i]$ since $S[s_j + Ex_i] = S[s_j + Ex_i - d] = S[i + Ex_i - d] \neq S[i + Ex_i]$. Symmetrical arguments can be made for the case in which $Ex_i > Ex_s - j \cdot d$. ◀

Proof for Lemma 23. We preprocess S for constant time LCP queries. Given A and s, we evaluate $l_p = lcp(s, i)$ using an lcp query. We find the extension of the period d to the right from i and to the right from s in constant time by querying $Ex_i = LCP(i, i + d)$ and $Ex_s = LCP(s, s + d)$ respectively.

If $l_p < d$, Observation 22 suggests that $lcp_j = l_p$ for $[1 \dots |A| - 2]$. $lcp_{|A|-1}$ can be evaluated independently using an additional LCP query. Our representation consists of the pairs $([1 \dots |A| - 2], l_p)$ and $(|A - 1 \dots |A| - 1], l_{|A|-1})$. Both are pairs of type (1).

In the case in which $l_p \geq d$, let j^* be the number satisfying $Ex_i = Ex_s - j^* \cdot d$. The following fact is directly derived from Lemma 22:

▶ **Fact 35.**
1. $lcp_j = Ex_i$ *for* $j \in [0 \dots min(\lceil j^* \rceil - 1, |A| - 1)]$
2. $lcp_j = Ex_s - j \cdot d$ *for* $j \in [max(\lfloor j^* \rfloor + 1, 0) \dots |A| - 1]$

The above fact provides a representation for lcp_j for every $j \neq j^*$. Specifically, the pair $([0 \dots min(\lceil j^* \rceil - 1, |A| - 1)], Ex_i)$ of type (1) and the pair $([max(\lfloor j^* \rfloor + 1, 0) \dots |A| - 1], Ex_s)$ of type (2). In the case in which j^* is an integer, another pair of type (1) with a singleton interval is required to represent lcp_{j^*}. lcp_{j^*} can be independently evaluated using an LCP query.

The evaluation of l_p, Ex_s, Ex_i, $lcp_{|A|-1}$ and lcp_{j^*} is done using a constant LCP query each and therefore consumes constant time. j^* can be calculated from Ex_s, Ex_i and d using a constant number of basic arithmetic operations. The overall time for obtaining the representation of lcp_j is constant. ◀

Proof of Lemma 24. In the case in which $l_p < d$, we have one pair $(I = [0 \dots |A| - 2], l_p)$ that is corresponding to the non aligned occurrences. $i + l_p + 1$ is a fixed value and $s_j + l_p + 1$ is an arithmetic progression with difference d. Let R be the run with period d containing the indexes of A. Since $l_p < d$, and I does not contain the rightmost index in the run , for

every $j \in I$ s_j has at least d indexes to its right contained within R. Therefore, the index $s_j + l_p + 1 \leq s_j + d$ is within R for every $j \in I$ and condition (1) in the statement of the lemma holds.

In the case in which $l_p \geq d$, we distinguish between the two pairs corresponding to the non-aligned indexes in the representation of lcp_j.

The indexes represented by the pair $(I = [0 \dots min(\lceil j^* \rceil - 1, |A| - 1)], Ex_i)$ have $Ex_i < Ex_s - j \cdot d$. Since $lcp_j = Ex_i$ is a fixed value for $j \in I$, the sequence $i + lcp_j + 1$ is fixed and $s_j + lcp_j + 1$ is an arithmetic progression with difference d. we also have $s_j + lcp_j + 1 = s_j + Ex_i + 1 \leq Ex_s - j \cdot d = s + Ex_s$. Recall that $s + Ex_s$ is the right border of R, so $s \leq s_j + lcp_j + 1 \leq s_E x_s$ suggests that $s_j + lcp_j + 1$ is within R. We therefore proved that condition (1) holds in this case.

The indexes represented by the pair $(I = [max(\lfloor j^* \rfloor + 1, 0) \dots |A| - 1], Ex_s)$ have $Ex_i > Ex_s - j \cdot d$. Since $lcp_j = Ex_s - j \cdot d$ is an arithmetic progression with difference $-d$ for $j \in I$, $i + lcp_j + 1$ is an arithmetic progression with difference $-d$ and $s_j + lcp_j + 1$ is a fixed value. Symmetric arguments to the ones in the previous case can be made to show that the indexes $i + lcp_j + 1$ are within the run with period d containing i and condition (2) holds. ◀

Proof for Lemma 29. We partition the occurrences s_j into four distinct types:

1. s_j with $r^j \geq m$ and $l^j_e \geq \frac{m}{4}$. According to Observation 26, $s_j - t$ is an l-occurrence of w^t for every $t \in [0 \dots \frac{m}{4}]$.
2. s_j with $r^j \geq m$ and $l^j < \frac{m}{4}$. According to Observation 26, $s_j - t$ is an l-occurrence of w^t for $t \in [0 \dots l^j]$.
3. s_j with $r^j < m$ and $l^j \geq \frac{m}{4}$. According to Observation 26, $s_j - t$ is an l-occurrence of w^t for $t \in [m - r^j \dots \frac{m}{4}]$ in this case.
4. s_j with $r^j < m$ and $l^j \geq \frac{m}{4}$. According to Observation 26, $s_j - t$ is an l-occurrence of w^t for $t \in [m - r^j \dots l^j]$ in this case.

Fig. 2 demonstrates the fourth type listed above and can be used to understand the rest of the types. Recall that r^j (resp. l^j) is partition into a constant number of intervals of values of j. For every such interval $I = [s \dots e]$, an arithmetic progression represents the values of r^j (resp. l^j) with $j \in I$. This representation can be easily processed in $O(1)$ time to obtain a partition P of the values of j into a constant number of intervals, such that every interval $I = [a \dots b] \in P$ contains occurrences of exactly one of the types listed above.

We treat every type independently.

Type 1: An interval $I = [a \dots b]$ of type 1 occurrences contributes $b - a + 1$ l-occurrences of w^t for every $t \in [0 \dots \frac{m}{4}]$. This is naturally represented by the interval increment update $(0, \frac{m}{4}, b - a + 1)$

Type 2: Consider an interval $I = [a \dots b]$ of type 2 occurrences. $s_j - t$ with $j \in I$ is an l-occurrence for every w^t with $t \in [0 \dots l^j]$. Recall l^j is either an increasing arithmetic progression or a fixes value in $[a \dots b]$. If it is a fixed value l', every occurrence s_j with $j \in I$ contributes an l-occurrence of w_t for the same interval of t values $[0 \dots l']$. The overall contribution of all the occurrences in I can be therefore represented with the interval increment update $(0, l', b - a + 1)$.

The more complicated case is the case in which l^j is an increasing arithmetic progression. Recall that the difference p of this arithmetic progression is the period of w^i_L. The occurrence s_b with the maximal LCP value l^b contributes an l-occurrence of w^t for $t \in [0 \dots l^b]$. The occurrence s_{b-1} contributes an l-occurrence for w^t for $t \in [0 \dots l^b - p]$ and so on. The effect of the entire progression on the counters C_t can be described as follows: The counters C_t

for $t \in [l^b - p + 1 \dots l_b]$ are increased by 1, the counters with $t \in [l^b - 2p + 1 \dots l^b - p]$ are increased by 2 and so on. In general: the counters C_t with $t \in [l^b - x \cdot p + 1 \dots l^b - (x-1) \cdot p]$ are increased by x for $x \in [1 \dots b - a]$ and the counters C_t with $t \in [0 \dots l^a]$ are increased by $b - a + 1$. The modification of indexes in $[l^a + 1 \dots l^b]$ can be equivalently described as an application of a decreasing stairs update $(l^a + 1, l^b, p)$. The modification of the indexes $[0 \dots l^a]$ can be described as an interval increment update $(0, l^a, b - a + 1)$. See Fig. 1 for an illustration of the stairs update derived from type 2 occurrences.

Type 3: Having a symmetric structure to an interval of type 2 occurrences, the effect of an interval of type 3 occurrences on D can be represented by a stairs update and an interval increment update as well.

Type 4: Consider a consecutive interval $I = [a \dots b]$ of type 4 occurrences. Recall that, similarly to l^j, the arithmetic progression r^j must be either decreasing or a fixed value. If both l^j and r^j are fixed in I, the counters C_t with $t \in [m - r^b \dots l^a]$ need to be increased by $a - b + 1$ which can be represented with an interval increment update. If either l^j or r^j are fixed, and the other is an increasing or decreasing arithmetic progression, the required modification for D can be represented with a stairs update and an interval increment update similarly to the representation of type 2 updates.

If both r^j and l^j are arithmetic progressions, the updates to C_t have a "sliding window" structure. Namely, Counters with $t \in [m - r^a \dots l^a]$ are increased due to occurrence s_a. Counters with $t \in [m - r^a + p \dots l_a + p]$ are increased due to occurrence s_{a+1} and so on (Notice that these intervals may overlap). We proceed to show how to represent this kind of modification to the clusters using a constant number of stairs updates and interval increment updates.

For clearer presentation, assume that the required modification to be applied to the counters is given as a pair (x, y) such that for every $j \in [0 \dots |I| - 1]$ the interval $[x + j \cdot p \dots y + j \cdot p]$ is increased by 1. Every such interval $[x + j \cdot p \dots y + j \cdot p]$ is called a window, with $x + j \cdot p$ being the start of the window and $y + j \cdot p$ being the end of the window. We represent the modification to the updates using two increasing stairs updates and one interval increment update.

The first increasing stairs update is $\mathit{Starts} = (x, x + (|I| - 1) \cdot p - 1, p)$. Note the the d-th step of *Starts* starts in the same index as the start of the d-th window. The second update is a **negative** increasing stairs updates $\mathit{Ends} = (y + 1, y + (|I| - 1) \cdot p, p)$. Note that the d-th step of *Ends* starts one index to the right of the end of the d-th window. Finally, we have the interval increment update $\mathit{Remainder} = (x + (|I| - 1) \cdot p, y + (|I| - 1) \cdot p, |I|)$ which can be considered an extended last step for *Starts*. It is easy to see that all the updates only apply to the indexes affected by the sliding window. Furthermore, a counter C_t is increased by *Starts* (or by *Remainders*) by the number of starting indexes of windows that are not to the right of t. C_t is decreased by *Ends* by the number of windows with ending indexes strictly to the left of t. Overall, the counter C_t is increased by the number of windows containing it. With this, we proved that *Starts*, *Ends* and *Remainders* are equivalent to the sliding window update given as (x, y).

Every stairs or interval update we constructed in the above discussion can be easily obtained in $O(1)$ time from I and from the representation of l^j and r^j. The proof of the Lemma 29 is completed. ◀

Internal Shortest Absent Word Queries

Golnaz Badkobeh
Department of Computing, Goldsmiths University of London, UK

Panagiotis Charalampopoulos
Efi Arazi School of Computer Science, The Interdisciplinary Center Herzliya, Israel

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, we are to preprocess T so that given a range $[i, j]$, we can return a representation of a shortest string over Σ that is absent in the fragment $T[i] \cdots T[j]$ of T. For any positive integer $k \in [1, \log\log_\sigma n]$, we present an $\mathcal{O}((n/k) \cdot \log\log_\sigma n)$-size data structure, which can be constructed in $\mathcal{O}(n \log_\sigma n)$ time, and answers queries in time $\mathcal{O}(\log\log_\sigma k)$.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, internal queries, shortest absent word, bit parallelism

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.6

Related Version *Extended Version (with Dmitry Kosolobov)*: https://arxiv.org/abs/2106.01763

Funding *Panagiotis Charalampopoulos*: Supported by the Israel Science Foundation grant 592/17.

1 Introduction

Range queries are a classic data structure topic [63, 13, 12]. In 1d, a range query $q = f(A, i, j)$ on an array of n elements over some set U, denoted by $A[1\,..\,n]$, takes two indices $1 \leq i \leq j \leq n$, a function f defined over arrays of elements of U, and outputs $f(A[i\,..\,j]) = f(A[i], \ldots, A[j])$. Range query data structures in 1d can thus be viewed as data structures answering queries on a string in the internal setting, where U is the considered alphabet.

Asking internal queries on a string has received much attention in recent years. In the internal setting, we are asked to preprocess a string T of length n over an alphabet Σ of size σ, so that queries about substrings of T can be answered efficiently. Note that an arbitrary substring of T can be encoded in $\mathcal{O}(1)$ words of space by the indices i, j of an occurrence of it as a fragment $T[i] \cdots T[j] = T[i\,..\,j]$ of T. Data structures for answering internal queries are interesting in their own sake, but also have numerous applications in the design of algorithms and (more sophisticated) data structures in stringology. Because of these numerous applications, we usually place particular emphasis on the construction time – other than on space/query-time tradeoffs, which is the main focus in the classic data structure literature.

The most widely-used internal query is that of asking for the *longest common prefix* of two suffixes $T[i\,..\,n]$ and $T[j\,..\,n]$ of T. The classic data structure for this problem [48] consists of the suffix tree of T [25] and a lowest common ancestor data structure [37] over the suffix tree. It occupies $\mathcal{O}(n)$ space, it can be constructed in $\mathcal{O}(n)$ time, and it answers queries in $\mathcal{O}(1)$ time. In the word RAM model of computation with word size $\Theta(\log n)$ bits the construction time is not necessarily optimal. A sequence of works [61, 52, 14] has culminated in the recent optimal data structure of Kempa and Kociumaka [40]: it occupies $\mathcal{O}(n/\log_\sigma n)$ space, it can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time, and it answers queries in $\mathcal{O}(1)$ time.

Another fundamental problem in this setting is the *internal pattern matching* (IPM) problem. It consists in preprocessing T so that we can efficiently compute the occurrences of a substring U of T in another substring V of T. For the decision version of the IPM problem, Keller et al. [39] presented a data structure of nearly-linear size supporting sublogarithmic-time queries. Kociumaka et al. [45] presented a data structure of linear size supporting constant-time queries when the ratio between the lengths of V and U is bounded by a constant. The $\mathcal{O}(n)$-time construction algorithm of the latter data structure was derandomised in [42]. In fact, Kociumaka et al. [45], using their efficient IPM queries as a subroutine, managed to show efficient solutions for other internal problems, such as for computing the periods of a substring (*period queries*, introduced in [44]), and for checking whether two substrings are rotations of one another (*cyclic equivalence queries*). Other problems that have been studied in the internal setting include string alignment [62, 18], approximate pattern matching [21], dictionary matching [20, 19], longest common substring [4], counting palindromes [59], range longest common prefix [3, 1, 49, 34], the computation of the lexicographically minimal or maximal suffix, and minimal rotation [7, 41], as well as of the lexicographically kth suffix [8]. We refer the interested reader to the Ph.D dissertation of Kociumaka [42], for a nice exposition.

In this work, we extend this line of research by investigating the following basic internal query, which, to the best of our knowledge, has not been studied previously. Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$, preprocess T so that given a range $[i, j]$, we can return a shortest string over Σ that does not occur in $T[i\,..\,j]$. The latter shortest string is also known as a shortest absent word in the literature. We work on the standard unit-cost word RAM model with machine word-size $w = \Theta(\log n)$ bits. We measure the space used by our algorithms and data structures in machine words, unless stated otherwise. We assume that we have random access to T and so our algorithms return a constant-space representation of a shortest string (a witness) consisting of a substring of T and a letter. A naïve solution for this problem precomputes a table of size $\mathcal{O}(n^2)$ that stores the answer for every possible query $[i, j]$. Our main result is the following.

▶ **Theorem 1.** *Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, for any positive integer $k \in [1, \log\log_\sigma n]$, we can construct a data structure of size $\mathcal{O}((n/k) \cdot \log\log_\sigma n)$, in $\mathcal{O}(n \log_\sigma n)$ time, so that if query $[a, b]$ is given, we can compute a shortest string over Σ that does not occur in $T[a\,..\,b]$ in $\mathcal{O}(\log\log_\sigma k)$ time.*

By setting $k = 1$ we get an $\mathcal{O}(n \log\log_\sigma n)$-size data structure with $\mathcal{O}(1)$ query time.

In the related *range shortest unique substring* problem, defined by Abedin et al. [2], the task is to construct a data structure over T to be able to answer the following type of online queries efficiently. Given a range $[i, j]$, return a shortest string with exactly one occurrence (starting position) in $[i, j]$. Abedin et al. presented a data structure of size $\mathcal{O}(n \log n)$ supporting $\mathcal{O}(\log_w n)$-time queries, where $w = \Theta(\log n)$ is the word size. Additionally, Abedin et al. [2] presented a data structure of size $\mathcal{O}(n)$ supporting $\mathcal{O}(\sqrt{n} \log^\epsilon n)$-time queries, where ϵ is an arbitrarily small positive constant.

Our Techniques

For clarity of exposition, in this overview, we skip the time-efficient construction algorithms of our data structures and only describe how to compute the *length* of a shortest absent word (without a witness) in $T[a\,..\,b]$; note that this length is at most $\log_\sigma n$. Let us also note that the length of a shortest absent word of T can be computed in $\mathcal{O}(n)$ time using the suffix tree of T [25]. It suffices to traverse the suffix tree of T recording the shortest string-depth ℓ, where an implicit or explicit node has less than σ outgoing edges.

First approach: We precompute, for each position i and for each length $j \in [1, \log_\sigma n]$, the ending position of the shortest prefix of $T[i\,..\,n]$ that contains an occurrence of each of the σ^j distinct words of length j. Then, a query for the length of a shortest absent word of $T[a\,..\,b]$ reduces to a predecessor query among the ending positions we have precomputed for position a. By maintaining these $\mathcal{O}(\log_\sigma n)$ ending positions in a fusion tree [32], we obtain a data structure of size $\mathcal{O}(n \log_\sigma n)$ supporting queries in $\mathcal{O}(\log_w \log n) = \mathcal{O}(1)$ time.

Second approach: We precompute, for each length $j \in [1, \log_\sigma n]$, all minimal fragments of T that contain an occurrence of each of the distinct σ^j words of length j. As these fragments are inclusion-free, we can encode them using two n-bit arrays storing their starting and ending positions in T, respectively. We thus require $\mathcal{O}(n)$ words of space in total over all js. Observe that $T[a\,..\,b]$ does not have an absent word of length j if and only if it contains a minimal fragment for length j; we can check this condition in $\mathcal{O}(1)$ time after augmenting the computed bit arrays with succinct rank and select data structures [38]. Finally, due to monotonicity (if $T[a\,..\,b]$ contains all strings of length $j+1$ then T contains all strings of length j), we can binary search for the answer in $\mathcal{O}(\log \log_\sigma n)$ time.

Third approach: We rely on the following combinatorial observation: if the length of a shortest absent word of a string X over Σ is λ, we need to prepend $\Omega(\sigma^{d-1} \cdot \lambda)$ letters to X in order to obtain a string with a shortest absent word of length $\lambda + d$. (For intuition, think of $|X|$ as a constant; then, we essentially need to prepend the de Bruijn sequence of order d over Σ to X in order to achieve the desired result.) This observation allows us to sparsify the information we stored in our first approach: for each length $j \in [1, \log_\sigma n]$, we use the value (previously) stored for some position i for an interval of positions. We then maintain a dynamic fusion tree over the stored values, which are now $o(n \log_\sigma n)$ in total, and make it persistent so that we can later query any version of it. As we will show, in the end we get the correct answer up to a small additive error, which we then eliminate by utilising the data structure developed in our second approach.

Let us remark that our partially persistent fusion trees allow us to obtain an alternative time-optimal data structure for the weighted ancestors problem [26] when the input tree of size n is of depth polylogarithmic in n. Such a data structure can be also easily derived from [47, 36].

Other Related Work

Let us recall that a string S that does not occur in T is called *absent* from T, and if all its proper substrings appear in T it is called a *minimal absent word* of T. It should be clear that every shortest absent word is also a minimal absent word. Minimal absent words (MAWs) are used in many applications [60, 56, 29, 35, 15, 54, 24] and their theory is well developed [51, 28, 30], also from an algorithmic and data structure point of view [50, 22, 9, 17, 16, 6, 33, 10, 23]. For example, it is well known that, given two strings X and Y, one has $X = Y$ if and only if X and Y have the same set of MAWs [51].

Paper Organization

Section 2 provides some preliminaries. The first approach is detailed in Section 3 and the second one in Section 4. Section 5 provides the combinatorial foundations for the third approach, which is detailed in Section 6.

2 Preliminaries

An *alphabet* Σ is a finite nonempty set whose elements are called *letters*. A *string* (or *word*) $S = S[1\,..\,n]$ is a sequence of *length* $|S| = n$ over Σ. The *empty* string ε is the string of length 0. The *concatenation* of two strings S and T is the string composed of the letters of S followed by the letters of T. It is denoted by $S \cdot T$ or simply by ST. The set of all strings (including ε) over Σ is denoted by Σ^*. The set of all strings of length $k > 0$ over Σ is denoted by Σ^k. For $1 \leq i \leq j \leq n$, $S[i]$ denotes the ith letter of S, and the fragment $S[i\,..\,j]$ denotes an *occurrence* of the underlying *substring* $P = S[i] \cdots S[j]$. We say that P *occurs* at (starting) *position* i in S. P is called *absent* from S if it does not occur in S. A substring $S[i\,..\,j]$ is a *suffix* of S if $j = n$ and it is a *prefix* of S if $i = 1$.

The following proposition is straightforward (as explained in Section 1).

▶ **Proposition 2.** *Let T be a string of length n. A shortest absent word of T can be computed in $\mathcal{O}(n)$ time.*

Given an array A of n items taken from a totally ordered set, the *range minimum query* $\mathsf{RMQ}_A(\ell, r) = \arg\min A[k]$ (with $1 \leq \ell \leq k \leq r \leq n$) returns the position of the minimal element in $A[\ell\,..\,r]$. The following result is known.

▶ **Theorem 3** ([12])**.** *Let A be an array of n integers. A data structure of size $\mathcal{O}(n)$ can be constructed in $\mathcal{O}(n)$ time supporting RMQs on A in $\mathcal{O}(1)$ time.*

We make use of *rank and select* data structures constructed over bit vectors. For a bit vector H we define $\mathsf{rank}_q(i, H) = |\{k \in [1, i] : H[k] = q\}|$ and $\mathsf{select}_q(i, H) = \min\{k \in [1, n] : \mathsf{rank}_q(k, H) = i\}$, for $q \in \{0, 1\}$. The following result is known.

▶ **Theorem 4** ([38, 53])**.** *Let H be a bit vector of n bits. A data structure of $o(n)$ additional bits can be constructed in $\mathcal{O}(n)$ time supporting rank and select queries on H in $\mathcal{O}(1)$ time.*

The *static predecessor* problem consists in preprocessing a set Y of integers, over an ordered universe U, so that, for any integer $x \in U$ one can efficiently return the predecessor $\mathsf{pred}(x) := \max\{y \in Y : y \leq x\}$ of x in Y. The successor problem is defined analogously: upon a queried integer $x \in U$, the successor $\min\{y \in Y : y \geq x\}$ of x in Y is to be returned. Willard and Fredman designed the *fusion tree* data structure for this problem [32]. In the dynamic variant of the problem, updates to Y are interleaved with predecessor and successor queries. Pătraşcu and Thorup [57] presented a dynamic version of fusion trees, which, in particular yields an efficient construction of this data structure.

▶ **Theorem 5** ([32, 57])**.** *Let Y be a set of at most n w-bit integers. A data structure of size $\mathcal{O}(n)$ can be constructed in $\mathcal{O}(n \log_w n)$ time supporting insertions, deletions, and predecessor queries on Y in $\mathcal{O}(\log_w n)$ time.*

If $|U| = \mathcal{O}(n)$, then, after an $\mathcal{O}(n)$-time preprocessing, we can answer predecessor queries in $\mathcal{O}(1)$ time. For each $y \in Y$, we set the yth bit of an initially all-zeros $|U|$-size bit vector. We then preprocess this bit vector as in Theorem 4. Then, a predecessor query for any integer x can be answered in $\mathcal{O}(1)$ time due to the following readily verifiable formula: $\mathsf{pred}(x) = \mathsf{select}_1(\mathsf{rank}_1(x))$.

The main problem considered in this paper is formally defined as follows.

Internal Shortest Absent Word (ISAW)
Input: A string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size $\sigma > 1$.
Output: Given integers a and b, with $1 \leq a \leq b \leq n$, output a shortest string in Σ^* with no occurrence in $T[a\,..\,b]$.

If $a = b$ then the answer is trivial. So, in what follows we assume that $a < b$. Let us also remark that the output (shortest absent word) can be represented in $\mathcal{O}(1)$ space using: either a range $[i, j] \subseteq [1, n]$ and a letter α of Σ, such that the shortest string in Σ^* with no occurrence in $T[a\,..\,b]$ is $T[i\,..\,j]\alpha$; or simply a range $[i, j] \subseteq [1, n]$ such that the shortest string in Σ^* with no occurrence in $T[a\,..\,b]$ is $T[i\,..\,j]$.

▶ **Example 6.** Given the string $T = \texttt{abaabaaabbabbbaaab}$ and the range $[a, b] = [8, 14]$ (shown in red), the only shortest absent word of $T[8\,..\,14]$ is $T[i\,..\,j] = T[7\,..\,8] = \texttt{aa}$.

3 $\mathcal{O}(n \log_\sigma n)$ Space and $\mathcal{O}(1)$ Query Time

Let T be a string of length n. We define $S_T(j)$ as the function counting the cardinality of the set of length-j substrings of T. This is known as the *substring complexity* function [27, 58]. Note that $S_T(j) \leq n$, for all j. We have the following simple fact.

▶ **Fact 7.** *The length ℓ of a shortest absent word of a string T of length n over an alphabet of size σ is equal to the smallest j for which $S_T(j) < \sigma^j$ and hence $\ell \in [1, \lfloor \log_\sigma n \rfloor]$.*

We denote the set of shortest absent words of T by SAW_T. Recall that, by Proposition 2, a shortest absent word of T can be computed in $\mathcal{O}(n)$ time. We denote the length of the shortest absent words of T by ℓ. By Fact 7, $\ell \leq \lfloor \log_\sigma n \rfloor$. Since ℓ is an upper bound on the length of the answer for any ISAW query on T, in what follows, we consider only lengths in $[1, \ell - 1]$. Let one such length be denoted by j. By constructing and traversing the suffix tree of T, we can assign to each $T[i\,..\,i + j - 1]$ its lexicographic rank in Σ^j. The time required for each length j is $\mathcal{O}(n)$, since the suffix tree of T can be constructed within this time [25]. Thus, the total time for all lengths $j \in [1, \ell - 1]$ is $\mathcal{O}(n \log_\sigma n)$ by Fact 7.

We design the following warm-up solution to the ISAW problem. For all $j \in [1, \ell - 1]$ we store an array RNK_j of n integers such that $\mathsf{RNK}_j[i]$ is equal to the lexicographic rank of $T[i\,..\,i + j - 1]$ in Σ^j. Then, given a range $[a, b]$, in order to check if there is an absent word of length j in $T[a\,..\,b]$ we only need to compute the number of distinct elements in $\mathsf{RNK}_j[a\,..\,b - j + 1]$. It is folklore that using a persistent segment tree, we can preprocess an array A of n integers in $\mathcal{O}(n \log n)$ time so that upon a range query $[a, b]$ we can return the number of distinct elements in $A[a\,..\,b]$ in $\mathcal{O}(\log n)$ time. Thus, we could use this tool as a black box for every array RNK_j resulting, however, in $\Omega(\log n)$-time queries. We improve upon this solution as follows.

We employ a range minimum query (RMQ) data structure [12] over a slight modification of RNK_j. For each j, we have an auxiliary procedure checking whether all strings from Σ^j occur in $T[a\,..\,b]$ or not (i.e., it suffices to check whether any lexicographic rank is absent from the corresponding range). Similar to the previous solution, we rank the elements of Σ^j by their lexicographic order. We append RNK_j with all integers in $[1, \sigma^j]$. Let this array be APP_j. By Fact 7, we have that $|\mathsf{APP}_j| \leq 2n$. Then, we construct an array PRE_j of size $|\mathsf{APP}_j|$: $\mathsf{PRE}_j[i]$ stores the position of the rightmost occurrence of $\mathsf{APP}_j[i]$ in $\mathsf{APP}_j[1\,..\,i - 1]$ (or 0 if such an occurrence does not exist). This can be done in $\mathcal{O}(n)$ time per j by sorting the list of pairs $(T[i\,..\,i + j - 1], i)$, for all i, using the suffix tree of T to assign ranks for $T[i\,..\,i + j - 1]$ and then radix sort to sort the list of pairs.

We now rely on the following fact.

▶ **Fact 8.** *$S_{T[a\,..\,b]}(j) = \sigma^j$ if and only if $\min\{\mathsf{PRE}_j[i] : i \in [b - j + 2, |\mathsf{PRE}_j|]\} \geq a$.*

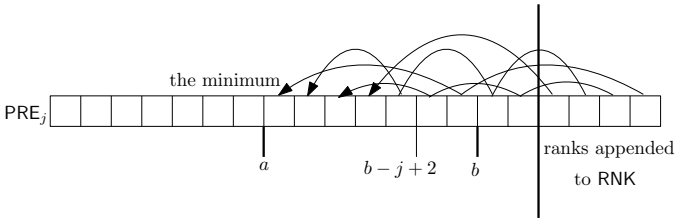


Figure 1 Illustration of the setting in Fact 8.

Proof. If the smallest element in $\mathsf{PRE}_j[b-j+2\,..\,|\mathsf{PRE}_j|]$, say $\mathsf{PRE}_j[k]$, is such that $\mathsf{PRE}_j[k] \geq a$, then all ranks of elements in Σ^j occur in $\mathsf{APP}_j[a\,..\,b-j+1]$. This is because all elements (ranks) in Σ^j occur at least once after $b-j+2$ (due to appending all integers in $[1,\sigma^j]$ to RNK_j), thus all must have a representative occurrence after $b-j+2$. Inspect Figure 1 for an illustration. (The opposite direction is analogous.) ◀

The following two examples illustrate the construction of arrays RNK_j, APP_j, and PRE_j as well as Fact 8.

▶ **Example 9** (Construction). Let $T = \mathtt{abaabaaabbabbbaaab}$ and $\Sigma = \{\mathtt{a},\mathtt{b}\}$. The set SAW_T of shortest absent words of T over Σ, each of length $\ell = 4$, is $\{\mathtt{aaaa},\mathtt{abab},\mathtt{baba},\mathtt{bbbb}\}$. Arrays RNK_j, APP_j, and PRE_j, for all $j \in [1,\ell-1]$, are as follows: For instance, $\mathsf{RNK}_2[15] =$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T	a	b	a	a	b	a	a	a	b	b	a	b	b	b	a	a	a	b						
RNK_1	1	2	1	1	2	1	1	1	2	2	1	2	2	2	1	1	1	2						
APP_1	1	2	1	1	2	1	1	1	2	2	1	2	2	2	1	1	1	2	1	2				
PRE_1	0	0	1	3	2	4	6	7	5	9	8	10	12	13	11	15	16	14	17	18				
RNK_2	2	3	1	2	3	1	1	2	4	3	2	4	4	3	1	1	2							
APP_2	2	3	1	2	3	1	1	2	4	3	2	4	4	3	1	1	2	1	2	3	4			
PRE_2	0	0	0	1	2	3	6	4	0	5	8	9	12	10	7	15	11	16	17	14	13			
RNK_3	3	5	2	3	5	1	2	4	7	6	4	8	7	5	1	2								
APP_3	3	5	2	3	5	1	2	4	7	6	4	8	7	5	1	2	1	2	3	4	5	6	7	8
PRE_3	0	0	0	1	2	0	3	0	0	0	8	0	9	5	6	7	15	16	4	11	14	10	13	12

$\mathsf{APP}_2[15] = 1$ denotes that the lexicographic rank of $\mathtt{aa}$ in Σ^2 is 1; and $\mathsf{PRE}_2[15] = 7$ denotes that the previous rightmost occurrence of $\mathtt{aa}$ is at position 7.

▶ **Example 10** (Fact 8). Let $[a,b] = [7,11]$ and $j = 2$ (see Example 9). The smallest element in $\{\mathsf{PRE}_2[11],\ldots,\mathsf{PRE}_2[21]\}$ is $\mathsf{PRE}_2[15] = 7 \geq a = 7$, which corresponds to rank $\mathsf{APP}_2[15] = 1$. Indeed all other ranks $2,3,4$ have at least one occurrence within $\mathsf{APP}_2[7\,..\,11] = 1,2,4,3,2$.

To apply Fact 8, we construct an RMQ data structure over PRE_j. By Theorem 3 it takes $\mathcal{O}(n)$ time and space and answers RMQs in $\mathcal{O}(1)$ time. This results in $\mathcal{O}(n\ell) = \mathcal{O}(n\log_\sigma n)$ preprocessing time and space for all j.

For querying, let us observe that $\sigma^j - S_{T[a\,..\,b]}(j)$, for any T,a,b and increasing j, is non-decreasing. We can thus apply binary search on j to find the smallest length j such that $S_{T[a\,..\,b]}(j) < \sigma^j$. This results in $\mathcal{O}(\log\ell) = \mathcal{O}(\log\log_\sigma n)$ query time. We obtain the following proposition (retrieving a witness shortest absent word is detailed later).

▶ **Proposition 11.** *Given a string T of length n over an alphabet $\Sigma \subset \{1,2,\ldots,n^{\mathcal{O}(1)}\}$ of size σ, we can construct a data structure of size $\mathcal{O}(n\log_\sigma n)$ in $\mathcal{O}(n\log_\sigma n)$ time, so that if query $[a,b]$ is given, we can compute a shortest string over Σ that does not occur in $T[a\,..\,b]$ in $\mathcal{O}(\log\log_\sigma n)$ time.*

We further improve the query time via employing fusion trees as follows. We create a 2d array $\mathsf{FTR}[1\,..\,\ell-1][1\,..\,n]$ of integers, where

$$\mathsf{FTR}[j][i] = \min\{\mathsf{PRE}_j[i-j+2], \ldots, \mathsf{PRE}_j[|\mathsf{PRE}_j|]\},$$

for all $j \in [1, \ell-1]$ and $i \in [1, n]$. Intuitively, $\mathsf{FTR}[j][i]$ is the rightmost index of T such that $T[\mathsf{FTR}[j][i]\,..\,i]$ contains all strings of length j over Σ.

Array FTR can be constructed in $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$ time by scanning each array PRE_j from right to left maintaining the minimum. Within the same complexities we also maintain satellite information specifying the index $k \in [i-j+2, |\mathsf{PRE}_j|]$ where the range minimum $\mathsf{FTR}[j][i]$ came from in the sub-array $\mathsf{PRE}_j[i-j+2\,..\,|\mathsf{PRE}_j|]$. We then construct n fusion trees, one for every collection of $\ell-1$ integers in $\mathsf{FTR}[1\,..\,\ell-1][i]$. This takes total preprocessing time and space $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$ by Theorem 5. Given the range query $[a, b]$, we need to find the smallest $j \in [1, \ell-1]$ such that $\mathsf{FTR}[j][b] < a$. By Theorem 5, we find where the predecessor of a lies in $\mathsf{FTR}[1\,..\,\ell-1][b]$ in $\mathcal{O}(\log_w \ell)$ time, where w is the word size; this time cost is $\mathcal{O}(1)$ since $w = \Theta(\log n)$.

We finally retrieve a *witness* shortest absent word as follows. If there is no $j < \ell$ such that $\mathsf{FTR}[j][b] < a$, then we output any shortest absent word of length ℓ of T arbitrarily. If such a $j < \ell$ exists, by the definition of $\mathsf{FTR}[j]$, we output $T[\mathsf{FTR}[j][b]\,..\,\mathsf{FTR}[j][b]+j-1]$ if $\mathsf{FTR}[j][b] > 0$ or $T[k\,..\,k+j-1]$ if $\mathsf{FTR}[j][b] = 0$, where k is the index of PRE_j, where the minimum came from. Inspect the following illustrative example.

▶ **Example 12** (Querying). We construct array FTR for T from Example 9. For a given $[a, b]$ we look up column b, and find the topmost entry whose value is less than a. If all entries have values greater than or equal to a, we output any element from SAW_T arbitrarily.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	a	b	a	a	b	a	a	a	b	b	a	b	b	b	a	a	a	b
FTR[1]	0	1	2	2	4	5	5	5	8	8	10	11	11	11	14	14	14	17
FTR[2]	0	0	0	0	0	0	0	0	0	5	7	7	7	7	7	11	11	13
FTR[3]	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4	4

If $[a, b] = [3, 14]$ then no entry in column $b = 14$ is less than $a = 3$, which means the length of the shortest absent word is 4; we output one from $\{\mathtt{aaaa}, \mathtt{abab}, \mathtt{baba}, \mathtt{bbbb}\}$ arbitrarily. If $[a, b] = [5, 14]$ then $\mathsf{FTR}[3][14] = 4 < 5$ so the length of a shortest absent word of $T[5\,..\,14]$ is 3; a shortest absent word is $T[\mathsf{FTR}[3][14]\,..\,\mathsf{FTR}[3][14]+3-1] = T[4\,..\,6] = \mathtt{aba}$.

If $[a, b] = [7, 9]$, $\mathsf{FTR}[2][9] = 0 < 7$ so the length of a shortest absent word is 2; a shortest absent word is $T[k\,..\,k+j-1] = T[9\,..\,10] = \mathtt{bb}$ because $\mathsf{FTR}[2][9] = \min\{\mathsf{PRE}_2[9], \ldots, \mathsf{PRE}_2[|\mathsf{PRE}_2|]\} = \mathsf{PRE}_2[9] = 0$ tells us that the minimum in this range came from index $k = 9$.

We obtain the following result.

▶ **Theorem 13.** *Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, we can construct a data structure of size $\mathcal{O}(n \log_\sigma n)$ in $\mathcal{O}(n \log_\sigma n)$ time, so that if query $[a, b]$ is given, we can compute a shortest string over Σ that does not occur in $T[a\,..\,b]$ in $\mathcal{O}(1)$ time.*

4 $\mathcal{O}(n)$ Space and $\mathcal{O}(\log \log_\sigma n)$ Query Time

▶ **Definition 14** (Order-j Fragment). *Given a string T over an alphabet of size σ and an integer j, V is called an* order-j fragment *of T if and only if V is a fragment of T and $S_V(j) = \sigma^j$. V is further called a* minimal order-j fragment *of T if $S_U(j) < \sigma^j$ and $S_Z(j) < \sigma^j$ for $U = V[1\,..\,|V|-1]$ and $Z = V[2\,..\,|V|]$.*

In particular, minimal order-j fragments are pairwise not included in each other. The following fact follows directly.

▶ **Fact 15.** *Given a string T of length n over an alphabet of size σ and an integer j we have $\mathcal{O}(n)$ minimal order-j fragments. Moreover, an arbitrary fragment F of T has $S_F[j] = \sigma^j$ if and only if it contains at least one of these minimal fragments.*

For each $j \in [1, \log_\sigma n]$, we consider all minimal order-j fragments T, separately. We encode the minimal order-j fragments of T using two bit vectors SP_j and EP_j, standing for starting positions and ending positions. Inspect the following example.

▶ **Example 16.** We consider T from Example 9 and $j = 2$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
T	a	b	a	a	b	a	a	a	b	b	a	b	b	b	a	a	a	b			
APP_2	2	3	1	2	3	1	1	2	4	3	2	4	4	3	1	1	2	1	2	3	4
PRE_2	0	0	0	1	2	3	6	4	0	5	8	9	12	10	7	15	11	16	17	14	13
SP_2	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	0	0			
EP_2	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1			

For instance, $\mathsf{SP}_2[13] = 1$ and $\mathsf{EP}_2[18] = 1$ denote the minimal order-2 fragment $V = T[13\,..\,18] = \mathtt{bbaaab}$.

We construct a rank and select data structure on SP_j and EP_j, for all $j \in [1, \ell - 1]$ supporting $\mathcal{O}(1)$-time queries. The overall space is $\mathcal{O}(n)$ by Theorem 4 and Fact 7.

Let us now explain how this data structure enables fast computation of absent words of length j. Given a range $[a, b]$, by Fact 15, we only need to find whether $T[a\,..\,b]$ contains a minimal order-j fragment. We can do this in $\mathcal{O}(1)$ time using one rank and two select queries: $t = \mathsf{rank}_1(a-1, \mathsf{SP}_j) + 1$; and $\mathsf{select}_1(t, \mathsf{SP}_j)$ and $\mathsf{select}_1(t, \mathsf{EP}_j)$.

▶ **Example 17.** We consider T, SP_2 and EP_2 from Example 16. Let $[a, b] = [5, 14]$. We have $t = \mathsf{rank}_1(a-1, \mathsf{SP}_2)+1 = \mathsf{rank}_1(4, \mathsf{SP}_2)+1 = 1$, $\mathsf{select}_1(t, \mathsf{SP}_2) = \mathsf{select}_1(1, \mathsf{SP}_2) = 5 < b = 14$ and $\mathsf{select}_1(t, \mathsf{EP}_2) = \mathsf{select}_1(1, \mathsf{EP}_2) = 10 < b = 14$, which means $T[5, 14]$ contains a minimal order-2 fragment.

Let us now describe a time-efficient construction of SP_j and EP_j. We use arrays PRE_j and APP_j of T, which are constructible in $\mathcal{O}(n)$ time (see Section 3). Recall that $\mathsf{PRE}_j[i]$ stores the starting position of the rightmost occurrence of rank $\mathsf{APP}_j[i]$ in $\mathsf{APP}_j[1\,..\,i-1]$ (or 0 if such an occurrence does not exist). We apply Fact 8 as follows. We start with all bits of SP_j and EP_j unset. Then, for each $b \in [1, n]$ for which $\mathsf{PRE}_j[b-j+1] < \min\{\mathsf{PRE}_j[i] : i \in [b-j+2, |\mathsf{PRE}_j|]\} = a$, we set the bth bit of EP_j and the ath bit of SP_j. This can be done online in a right-to-left scan of PRE_j in $\mathcal{O}(n)$ time.

▶ **Example 18.** We consider T, SP_2 and EP_2 from Example 16. We start by setting $b = n = 18$ and scan PRE_2 from right to left: we have $a = 13$ because $\min\{\mathsf{PRE}_2[21] = 13 : i \in [18, 21]\} \geq a = 13$. This gives fragment $T[13\,..\,18]$. Then we set $b = n - 1 = 17$ and have $a = 11$ because

$\min\{\mathsf{PRE}_2[21] = 13 : i \in [17, 20]\} \geq a = 11$. This gives fragment $T[11\,..\,17]$. Then we set $b = n - 2 = 16$ and have $a = 11$ because $\min\{\mathsf{PRE}_2[21] = 13 : i \in [16, 19]\} \geq a = 11$. This gives fragment $T[11\,..\,16]$. At this point note that $T[11\,..\,17]$ contains $T[11\,..\,16]$, and so $T[11\,..\,17]$ is removed as it is non-minimal.

▶ **Lemma 19.** *SP_j and EP_j can be constructed in $\mathcal{O}(n)$ time.*

For all j, the construction time is $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$ by Theorem 4, Lemma 19, and Fact 7. We obtain the following lemma.

▶ **Lemma 20.** *Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, we can construct a data structure of size $\mathcal{O}(n)$ in $\mathcal{O}(n \log_\sigma n)$ time, so that if query $(j, [a, b])$ is given, we can check in $\mathcal{O}(1)$ time whether there is any string in Σ^j that does not occur in $T[a\,..\,b]$, and if so return such a string.*

We can now apply Lemma 20 using binary search on j to find the smallest length j such that $S_{T[a\,..\,b]}(j) < \sigma^j$. This results in $\mathcal{O}(\log \ell) = \mathcal{O}(\log \log_\sigma n)$ query time by Fact 7. It should now be clear that when we find the j corresponding to the length of a shortest absent word, we can output the length-j suffix of the leftmost minimal order-j fragment starting after a. Note that outputting this suffix is correct by the definition of minimal order-j fragments.

▶ **Example 21.** We consider T, SP_2 and EP_2 from Example 16. Let $[a, b] = [2, 7]$. The length of a shortest absent word of $T[2\,..\,7]$ is 2. We output `bb`, which is the length-2 suffix of the leftmost minimal order-2 fragment $T[5\,..\,10] =$ `baaabb` starting after $a = 2$.

We obtain the following result.

▶ **Theorem 22.** *Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, we can construct a data structure of size $\mathcal{O}(n)$ in $\mathcal{O}(n \log_\sigma n)$ time, so that if query $[a, b]$ is given, we can compute a shortest string over Σ that does not occur in $T[a\,..\,b]$ in $\mathcal{O}(\log \log_\sigma n)$ time.*

5 Combinatorial Insights

A positive integer p is a *period* of a string S if $S[i] = S[i + p]$ for all $i \in [1, |S| - p]$. We refer to the smallest period as *the period* of the string. Let us state the periodicity lemma, one of the most elegant combinatorial results on strings.

▶ **Lemma 23** (Periodicity Lemma (weak version) [31])**.** *If a string S has periods p and q such that $p + q \leq |S|$, then $\mathsf{gcd}(p, q)$ is also a period of S.*

▶ **Lemma 24.** *If all strings in $\{WU : U \in \Sigma^k\}$ for $W \neq \varepsilon$ occur in some string S, then $|S| \geq |W| \cdot \sigma^k/4$.*

Proof. Let p be the period of W, and let $a \in \Sigma$ be such so that the period of Wa is also p. All strings WbZ for a letter $b \neq a$ and $Z \in \Sigma^{k-1}$ must occur in S. Let $A = \{WU : U \in \Sigma^k\} \setminus \{WaZ : Z \in \Sigma^{k-1}\}$, and note that it is of size $\sigma^k - \sigma^{k-1} \geq \sigma^k/2$. The following claim immediately implies the statement of the lemma.

▷ Claim. Let i and j be starting positions of occurrences of different strings $WU, WV \in A$ in S, respectively. Then, we have $|j - i| \geq |W|/2$.

Proof. Let us assume, without loss of generality, that $j > i$. Further, let us assume towards a contradiction that $j - i < |W|/2$. Then, $j - i$ is a period of W and $p + j - i \leq |W|$ since $p \leq j - i$. Therefore, due to the periodicity lemma (Lemma 23), $j - i$ must be divisible by the period p of W. Hence, U starts with the letter a and $WU \notin A$, a contradiction. ◁

This concludes the proof of this lemma. ◀

▶ **Lemma 25.** *If a shortest absent word of a string Y is of length λ, then the length of a shortest absent word of XY is in $[\lambda, \lambda + \max\{10, 4 + \log_\sigma(|X|/\lambda)\}]$.*

Proof. Let W and W' be shortest absent words of Y and XY, respectively. Further, let $d = |W'| - |W|$. In order to have $d > 0$, all strings WU for $U \in \Sigma^{d-1}$ must occur in XY, and hence in $X \cdot Y[1\,..\,|WU| - 1]$, since none of them occurs in Y. Lemma 24 implies that $|X| + \lambda + d > \lambda \cdot \sigma^{d-1}/4$. Then, since $\lambda + d \leq 2\lambda d$ for any positive integers λ, d, we have $|X| > \lambda \cdot (\sigma^{d-1}/4 - 2d)$. Assuming that $d \geq 10$, and since $\sigma \geq 2$, we conclude that $|X| > \lambda \cdot \sigma^{d-1}/8$. Consequently, $\log_\sigma(8|X|/\lambda) + 1 > d$. Since $\log_\sigma 8 \leq 3$ we get the claimed bound. ◀

▶ **Lemma 26.** *If a shortest absent word of XY is of length m, a shortest absent word of Y is of length λ, and $|X| \leq m \cdot \tau$, for a positive integer $\tau \geq 16$, then $m - \lambda \leq 10 + 2\log_\sigma \tau$.*

Proof. From Lemma 25 we have $\lambda \in [m - \max\{10, 4 + \log_\sigma(|X|/\lambda)\}, m]$. If $\max\{10, 4 + \log_\sigma(|X|/\lambda)\} = 10$, then $m - \lambda \leq 10$ and we are done.

In the complementary case, since $|X| \leq m \cdot \tau$, we get the following:

$$\lambda \geq m - \log_\sigma(m \cdot \tau/\lambda) - 4 \iff \lambda \geq m + \log_\sigma \lambda - \log_\sigma m - \log_\sigma \tau - 4.$$

In particular, $\lambda \geq m - \log_\sigma m - \log_\sigma \tau - 4$.

From the above, if $m \leq \tau$, then $m - \lambda \leq 4 + 2\log_\sigma \tau$.

In what follows we assume that $m > \tau \geq 16$. Rearranging the original equation, and since $\log_\sigma(\cdot)$ is an increasing function and $\lambda \geq m - \log_\sigma m - \log_\sigma \tau - 4$, we have

$$\begin{aligned} m - \lambda \leq 4 + \log_\sigma(m \cdot \tau/\lambda) &\leq 4 + \log_\sigma\left(\frac{m}{m - \log_\sigma m - \log_\sigma \tau - 4}\right) + \log_\sigma \tau \\ &\leq 4 + \log_\sigma\left(\frac{m}{m - 2\log_\sigma m - 4}\right) + \log_\sigma \tau. \end{aligned}$$

Then, we have $m - 2\log_\sigma m - 4 \geq m/5$ since, for any $\sigma \geq 2$, $4x/5 - 2\log_\sigma x - 4$ is an increasing function on $[16, \infty)$ and positive for $x = 16$. Hence, $m - \lambda \leq 4 + \log_\sigma 5 + \log_\sigma \tau \leq 7 + \log_\sigma \tau$.

By combining the bounds on $m - \lambda$ we get the claimed bound. ◀

6 $\mathcal{O}(n \log\log_\sigma n)$ Space and $\mathcal{O}(1)$ Query Time

Recall that we denote by ℓ the length of a shortest absent word of T. We start by constructing the 2d array $\mathsf{FTR}[1\,..\,\ell - 1][1\,..\,n]$ from Section 3 in time $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$. Further recall that $\mathsf{FTR}[j][i]$ is the rightmost index of T such that $T[\mathsf{FTR}[j][i]\,..\,i]$ contains all strings of length j over Σ. Then, to answer a query $[a, b]$, it suffices to find the smallest j such that $\mathsf{FTR}[j][b] < a$. We do this by finding where the predecessor of a lies in $\mathsf{FTR}[1\,..\,\ell - 1][b]$. To this end, we construct n fusion trees: one per $\mathsf{FTR}[1\,..\,\ell - 1][i]$, resulting in a data structure of size $\Theta(n\ell) = \mathcal{O}(n \log_\sigma n)$ with $\mathcal{O}(1)$ query time.

The main idea in this section is to rather maintain a collection of lazy dynamic fusion trees going from position n to 1, and apply the combinatorial lemmas from Section 5 to answer the query. Using the lazy dynamic fusion trees, for a parameter τ, we will compute an interval of size $\Theta(\log_\sigma \tau)$ that contains the length of a shortest absent word of $T[a\,..\,b]$. Then we will perform binary search employing Lemma 20 to compute the desired length and output a shortest absent word.

6.1 Lazy FTR Arrays

With *lazy* we mean that instead of array FTR, we consider array

$$\mathsf{LFTR}[j][i] = \mathsf{FTR}[j][i + ((n-i) \pmod{\tau \cdot j})],$$

for an integer parameter $\tau \geq 16$. We will later set τ to be some function of n – the reader may think of it as a constant. Intuitively, for an integer k, we use the value $\mathsf{FTR}[j][n-\tau\cdot j\cdot k]$ for $\tau \cdot j$ positions, namely for $\mathsf{LFTR}[j][i]$, $i \in (n-\tau\cdot j\cdot(k-1), n-\tau\cdot j\cdot k]$. Overall, the number of values that we consider is

$$\sum_{j\in[1,\ell-1]} \frac{n}{\tau\cdot j} = \mathcal{O}\left(\frac{n\log\log_\sigma n}{\tau}\right).$$

Inspect Figure 2 in this regard.

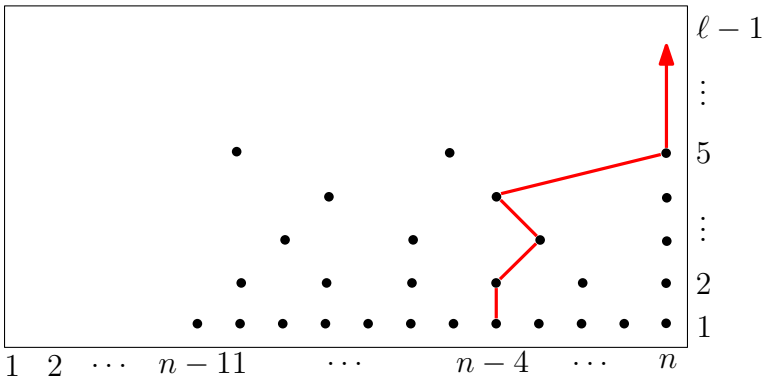


Figure 2 For simplicity, we have set $\tau = 1$. The black dots represent the entries stored in $\mathsf{LFTR}[1\,..\,5][n-11\,..\,n]$ and the red line represents $\mathsf{LFTR}[1\,..\,5][n-4]$.

We implement LFTR array using a collection of 1d arrays that occupy $\mathcal{O}((n/\tau)\cdot\log\log_\sigma n)$ space in total and allow $\mathcal{O}(1)$-time access to $\mathsf{LFTR}[j][i]$ for any j, i. Specifically, we store in array R_j, for all $j \in [1, \ell-1]$ and in decreasing order of i, the entries $\mathsf{FTR}[j][i]$ with $n-i \equiv 0 \pmod{\tau\cdot j}$. Then, we have $\mathsf{LFTR}[j][i] = R_j[1 + \lfloor (n-i)/(\tau\cdot j)\rfloor]$.

▶ **Fact 27.** *$LFTR[j][i] \geq FTR[j][i]$, for all i, j.*

Proof. It follows by the definition of LFTR and the fact that $\mathsf{FTR}[j][i]$ is monotonically non-decreasing for increasing i and fixed j. ◀

Note that the fact that $\mathsf{FTR}[1\,..\,\ell-1][i]$ is decreasing for all i allowed us to use predecessor queries in our previous solution. We prove an analogous, slightly weaker, statement for LFTR.

▶ **Lemma 28.** *Let $j_1, j_2 \in [1, \ell-1]$ such that $j_2 - j_1 > 10 + 2\log_\sigma \tau$, and suppose that $\tau \geq 16$. Then, for all i, $LFTR[j_1][i] > LFTR[j_2][i]$.*

Proof. Let $X_1 = T[\mathsf{LFTR}[j_1][i] + 1 \,..\, i]$. Then, X_1Y_1 has a shortest absent word of length j_1, for some Y_1 with $|Y_1| \leq \tau \cdot j_1$.

Let $X_2 = T[\mathsf{LFTR}[j_2][i] + 1 \,..\, i]$. Then, X_2Y_2 has a shortest absent word of length j_2, for some Y_2 with $|Y_2| \leq \tau \cdot j_2$.

Note that $||Y_2| - |Y_1|| \leq \tau \cdot j_2$.

Suppose, towards a contradiction, that $\mathsf{LFTR}[j_1][i] \leq \mathsf{LFTR}[j_2][i]$, and hence $|X_1| \geq |X_2|$. Then, we must have $|Y_2| > |Y_1|$ as otherwise X_2Y_2 would be a substring of X_1Y_1 and its shortest absent word cannot be longer than the one of X_1Y_1. Let $Y_2 = Y_1V$, with $|V| \leq \tau \cdot j_2$.

Then, we have that a shortest absent word of X_1Y_2 is of length at least j_2, since X_2Y_2 is a suffix of X_1Y_2. By Lemma 26 applied to $X_1Y_2 = X_1Y_1V$ and X_1Y_1, we have $j_2 - j_1 \leq 10 + 2\log_\sigma \tau$, a contradiction. ◀

In particular, Lemma 28 tells us that in column i of LFTR, we cannot have too many values that are equal. More formally, for each j, we have $\mathcal{O}(\log_\sigma \tau)$ indices $j' \neq j$ such that $\mathsf{LFTR}[j'][i] = \mathsf{LFTR}[j][i]$. Our goal is to have, for every position i, a snapshot of one of our dynamic fusion trees to contain as keys the entries of $\mathsf{LFTR}[1 \,..\, \ell - 1][i]$. The satellite information (value) of key $\mathsf{LFTR}[j][i]$ is a bit vector of size $\ell - 1$ bits. For each key, we maintain the corresponding lengths j in the bit vector. Whenever a key is returned, we can also return the largest of the lengths j stored in the bit vector: it corresponds to the highest set bit. In the next subsection we show the following result.

▶ **Lemma 29.** *We can preprocess LFTR in $\mathcal{O}((n/\tau) \cdot \log\log_\sigma n)$ time and space to answer predecessor and successor queries over $\mathsf{LFTR}[1 \,..\, \ell - 1][i]$, for any $i \in [1, n]$, in $\mathcal{O}(1)$ time.*

We denote by $\mathsf{top}(a, b)$ the largest j such that $\mathsf{LFTR}[j][b]$ is equal to the successor of a in $\mathsf{LFTR}[1 \,..\, \ell - 1][b]$.

▶ **Lemma 30.** *Given LFTR, after $\mathcal{O}((n/\tau) \cdot \log\log_\sigma n)$ time and space preprocessing, we can answer $\mathsf{top}(a, b)$ queries in $\mathcal{O}(1)$ time.*

Proof. At preprocessing, construct the data structure underlying Lemma 29. Upon a query $\mathsf{top}(a, b)$, answer a successor query for a in $\mathsf{LFTR}[1 \,..\, \ell - 1][b]$ using this data structure. For the corresponding bit vector, we find the highest set bit in $\mathcal{O}(1)$ time, thus retrieving $\mathsf{top}(a, b)$. ◀

Our data structure mainly relies on what we show next using Lemma 26: the sought answer is "close" to $\mathsf{top}(a, b)$. Let us denote the length of a shortest absent word of $T[c \,..\, b]$ by $\ell_{[c,b]}$ and the length of a shortest absent word of $T[a \,..\, b]$ by $\ell_{[a,b]}$.

By the definition of $\mathsf{top}(a, b)$, we have that for some $c = \mathsf{LFTR}[\mathsf{top}(a, b)][b] \geq a$ and a prefix X of $T[b + 1 \,..\, n]$ with $|X| \leq \mathsf{top}(a, b) \cdot \tau$, the length of a shortest absent word of $T[c \,..\, b]X$ is $\mathsf{top}(a, b) + 1$. By Lemma 26, $\mathsf{top}(a, b) - \ell_{[c,b]} \leq 10 + 2\log_\sigma \tau$. Thus $\ell_{[a,b]} \geq \ell_{[c,b]} \geq \mathsf{top}(a, b) - 10 - 2\log_\sigma \tau$.

In addition, we have $\mathsf{LFTR}[\mathsf{top}(a, b)][b] \geq a > \mathsf{LFTR}[j][b]$, for all $j > \mathsf{top}(a, b) + 10 + 2\log_\sigma \tau$, by Lemma 28 and the definition of $\mathsf{top}(a, b)$. Hence, $\ell_{[a,b]} \leq \mathsf{top}(a, b) + 11 + 2\log_\sigma \tau$ by Fact 27.

Thus, the sought answer $\ell_{[a,b]}$ is in $[\mathsf{top}(a, b) - 10 - 2\log_\sigma \tau, \mathsf{top}(a, b) + 11 + 2\log_\sigma \tau]$. We employ Lemma 20 to perform binary search over this interval in $\mathcal{O}(\log\log_\sigma \tau)$ time – after an $\mathcal{O}(n)$-time preprocessing. Recall that Lemma 20 also gives us a witness shortest absent word.

We thus arrive at the main result of this paper, by setting $\tau = 15 + k$, for any positive integer $k \in [1, \log\log_\sigma n]$.

▶ **Theorem 1.** *Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, for any positive integer $k \in [1, \log\log_\sigma n]$, we can construct a data structure of size $\mathcal{O}((n/k) \cdot \log\log_\sigma n)$, in $\mathcal{O}(n \log\log_\sigma n)$ time, so that if query $[a, b]$ is given, we can compute a shortest string over Σ that does not occur in $T[a\,..\,b]$ in $\mathcal{O}(\log\log_\sigma k)$ time.*

In particular, we get the following tradeoffs:

- an $\mathcal{O}(n \log\log_\sigma n)$-size data structure with $\mathcal{O}(1)$ query time (for $k = 1$);
- an $\mathcal{O}(n)$-size data structure with $\mathcal{O}(\log\log_\sigma \log\log_\sigma n)$ query time (for $k = \lfloor \log\log_\sigma n \rfloor$).

6.2 Partially Persistent Fusion Trees (using Fusion Trees)

We now describe the construction of the fusion trees over the LFTR array, which underlies Lemma 29. We provide the description for answering predecessor queries but it can be trivially adapted for successor queries. We will make our fusion trees "static partially persistent": for each position $i \in [1, n]$, we will be able to answer predecessor queries in the version of the fusion tree corresponding to $\mathsf{LFTR}[1\,..\,\ell - 1][i]$.

Recall that we work in the word RAM model. For implementing partially persistent fusion trees, we view each memory cell as a collection of pairs of values and timestamps; a timestamp indicates when the respective value was written in the cell. (This is a standard persistence trick, see e.g. [55].) For each cell, we want to construct a predecessor data structure over the timestamps to simulate these operations. The key idea is that, in each such cell, we would like to keep the number of updates small so as to employ fusion trees for implementing it as a predecessor data structure. Let us stress that the latter fusion trees should not be confused with the partially persistent fusion trees we construct over the LFTR array for our problem.

We now process T from right to left to construct the collection of partially persistent fusion trees. For each position of T, we perform the updates as per the LFTR array. Specifically, for position n, we initialise the partially persistent fusion tree with keys $\mathsf{LFTR}[1\,..\,\ell - 1][n]$. Then, for position i from $n - 1$ to 1, for all $j \in [1, \ell - 1]$, such that $(n - i) \pmod{\tau \cdot j} = 0$, we remove key $\mathsf{LFTR}[j][i + \tau \cdot j]$ and insert key $\mathsf{LFTR}[j][i]$. However, after processing every $\tau \cdot \log n / \log\log n$ positions of T, and hence $\sum_{j \in [1, \ell - 1]} (\tau \log n / \log\log n)/(\tau \cdot j) = \Theta(\log n)$ updates have been performed, we create a completely new instance of a partially persistent fusion tree. We initialise this new instance with the LFTR values of the currently unprocessed position of T. Let us note that the $\mathcal{O}(\log_\sigma n)$ time cost for reinitialisation amortises, because we can charge it to the $\Theta(\log n)$ $\mathcal{O}(1)$-time updates we have previously performed. For each position i of T, we store the timestamp $t(i)$ at which its processing ended and a pointer to the partially persistent fusion tree of the collection corresponding to it.

Upon a query $[a, b]$, we wish to find the smallest j such that $\mathsf{LFTR}[j][b] < a$. We thus need to find where the predecessor of a lies in $\mathsf{LFTR}[1\,..\,\ell - 1][b]$. We retrieve the partially persistent fusion tree and ask the query, using timestamp $t(b)$. Note that each memory access performed by the query requires $\mathcal{O}(1)$ time, as it translates to a predecessor query in a fusion tree with $\mathcal{O}(\log n)$ keys; and there are $\mathcal{O}(1)$ such accesses because this is a (partially persistent) fusion tree with $\mathcal{O}(\log_\sigma n)$ keys. The query thus takes $\mathcal{O}(1)$ time.

6.3 Weighted Ancestor Data Structure for Shallow Trees

A tree is a *weighted tree* if it is a rooted tree with an integer weight on each node v, denoted by $w(v)$, such that the weight of the root is zero and $w(u) < w(v)$ if u is the parent of v. We say that a node v is a *weighted ancestor at depth δ* of a node u if v is the highest ancestor of u with weight of at least δ. The problem of constructing a data structure to answer weighted ancestor queries was introduced by Farach and Muthukrishnan in [26].

After $\mathcal{O}(n)$-time preprocessing, weighted ancestor queries for nodes of a weighted tree $\mathcal{T}$ of size n with integer weights from a universe $[1\,..\,U]$ can be answered in $\mathcal{O}(\log\log U)$ time [26, 5]. Later, it was shown that a dynamic variant of the weighted ancestors problem admits a solution with the same time bounds as those for dynamic predecessor structures [47, 36]. Further, Kopelowitz et al. [46] introduced another $\mathcal{O}(n)$-size data structure that achieves faster query time in many special cases. For the offline version, Kociumaka et al. [43] showed how to answer a batch of q weighted ancestor queries in the optimal $\mathcal{O}(n+q)$ time.

The weighted ancestors problem has numerous applications if the input tree is a suffix tree of some string; see [36] for a nice exposition of these applications. In this context, the weighted ancestors problem translates to preprocessing the suffix tree of a string $T[1\,..\,n]$, so that, given i and j, we can retrieve the implicit or explicit node corresponding to substring $T[i\,..\,j]$. Observe that, since the weighted ancestor is a generalisation of the predecessor problem, it cannot admit better bounds. Nevertheless, the problem on suffix trees is a special case of the general problem. This led to the challenge of solving the problem on suffix trees in $\mathcal{O}(n)$ preprocessing time and $\mathcal{O}(1)$ query time [26]. Gawrychowski et al. [36] partly settled this question by presenting an $\mathcal{O}(n)$-size data structure with $\mathcal{O}(1)$ query time; the construction time, however, is superlinear in n. Very recently, Belazzougui et al. [11] have settled this question by presenting an $\mathcal{O}(n)$-size data structure for weighted ancestors in suffix trees with $\mathcal{O}(1)$ query time and an $\mathcal{O}(n)$-time construction algorithm.

Given a tree $\mathcal{T}$ of size n and depth d, we can do the following trick to reduce the weighted ancestors problem to $\mathcal{O}(n/d)$ instances on trees of both size and depth $\mathcal{O}(d)$: Cut along every dth root-to-leaf path in $\mathcal{T}$, and duplicate its nodes and edges. Then, we can apply the result of Kopelowitz and Lewenstein [47, 36] to each of the $\mathcal{O}(n/d)$ smaller trees. In the specific case of $d=\log^{\mathcal{O}(1)} n$, this gives an $\mathcal{O}(n)$-size data structure that answers weighted ancestor queries in $\mathcal{O}(1)$ time. By applying the machinery we have developed in the previous subsection, we achieve the same result for this special case in an alternative way. For each of the $\mathcal{O}(n/d)$ trees of size and depth $\mathcal{O}(d)$, we perform a depth-first traversal, maintaining a partially persistent fusion tree that when visiting node v stores the weights of all (weak) ancestors of v. We obtain the following corollary for shallow trees.

▶ **Corollary 31.** *Let $\mathcal{T}$ be a weighted tree of size n and depth $d=\log^{\mathcal{O}(1)} n$, with weights polynomial in n. We can preprocess $\mathcal{T}$ in $\mathcal{O}(n)$ time so that weighted ancestor queries on $\mathcal{T}$ can be answered in $\mathcal{O}(1)$ time.*

References

1 Paniz Abedin, Arnab Ganguly, Wing-Kai Hon, Yakov Nekrich, Kunihiko Sadakane, Rahul Shah, and Sharma V. Thankachan. A linear-space data structure for Range-LCP queries in poly-logarithmic time. In *Computing and Combinatorics - 24th International Conference, COCOON 2018*, pages 615–625, 2018. doi:10.1007/978-3-319-94776-1_51.

2 Paniz Abedin, Arnab Ganguly, Solon P. Pissis, and Sharma V. Thankachan. Efficient data structures for range shortest unique substring queries. *Algorithms*, 13(11):276, 2020. doi:10.3390/a13110276.

3 Amihood Amir, Alberto Apostolico, Gad M. Landau, Avivit Levy, Moshe Lewenstein, and Ely Porat. Range LCP. *J. Comput. Syst. Sci.*, 80(7):1245–1253, 2014. doi:10.1016/j.jcss.2014.02.010.

4 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/s00453-020-00744-0.

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

6 Lorraine A. K. Ayad, Golnaz Badkobeh, Gabriele Fici, Alice Héliou, and Solon P. Pissis. Constructing antidictionaries in output-sensitive space. In *Data Compression Conference, DCC 2019*, pages 538–547. IEEE, 2019. doi:10.1109/DCC.2019.00062.

7 Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, Ignat I. Kolesnichenko, and Tatiana Starikovskaya. Computing minimal and maximal suffixes of a substring. *Theor. Comput. Sci.*, 638:112–121, 2016. doi:10.1016/j.tcs.2015.08.023.

8 Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 572–591. SIAM, 2015. doi:10.1137/1.9781611973730.39.

9 Carl Barton, Alice Héliou, Laurent Mouchard, and Solon P. Pissis. Linear-time computation of minimal absent words using suffix array. *BMC Bioinform.*, 15:388, 2014. doi:10.1186/s12859-014-0388-9.

10 Carl Barton, Alice Héliou, Laurent Mouchard, and Solon P. Pissis. Parallelising the computation of minimal absent words. In *Parallel Processing and Applied Mathematics - 11th International Conference, PPAM 2015. Revised Selected Papers, Part II*, volume 9574 of *Lecture Notes in Computer Science*, pages 243–253. Springer, 2015. doi:10.1007/978-3-319-32152-3_23.

11 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021*, volume 191 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CPM.2021.8.

12 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

13 Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993. doi:10.1137/0222017.

14 Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, pages 607–626. SIAM, 2020. doi:10.1137/1.9781611975994.37.

15 Supaporn Chairungsee and Maxime Crochemore. Using minimal absent words to build phylogeny. *Theor. Comput. Sci.*, 450:109–116, 2012. doi:10.1016/j.tcs.2012.04.031.

16 Panagiotis Charalampopoulos, Maxime Crochemore, Gabriele Fici, Robert Mercaş, and Solon P. Pissis. Alignment-free sequence comparison using absent words. *Inf. Comput.*, 262:57–68, 2018. doi:10.1016/j.ic.2018.06.002.

17 Panagiotis Charalampopoulos, Maxime Crochemore, and Solon P. Pissis. On extended special factors of a word. In *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018*, volume 11147 of *Lecture Notes in Computer Science*, pages 131–138. Springer, 2018. doi:10.1007/978-3-030-00479-8_11.

18 Panagiotis Charalampopoulos, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. An almost optimal edit distance oracle. *CoRR*, abs/2103.03294, 2021. arXiv:2103.03294.

19 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Counting distinct patterns in internal dictionary matching. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, volume 161 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.8.

20 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal dictionary matching. In *30th International Symposium on Algorithms and Computation, ISAAC 2019*, volume 149 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ISAAC.2019.22.

21 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 978–989. IEEE, 2020. doi:10.1109/FOCS46700.2020.00095.

22 Maxime Crochemore, Alice Héliou, Gregory Kucherov, Laurent Mouchard, Solon P. Pissis, and Yann Ramusat. Absent words in a sliding window with applications. *Inf. Comput.*, 270, 2020. doi:10.1016/j.ic.2019.104461.

23 Maxime Crochemore, Filippo Mignosi, and Antonio Restivo. Automata and forbidden words. *Inf. Process. Lett.*, 67(3):111–117, 1998. doi:10.1016/S0020-0190(98)00104-5.

24 Maxime Crochemore, Filippo Mignosi, Antonio Restivo, and Sergio Salemi. Data compression using antidictionaries. *Proceedings of the IEEE*, 88(11):1756–1768, 2000. doi:10.1109/5.892711.

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

26 Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Combinatorial Pattern Matching, 7th Annual Symposium, CPM 1996*, volume 1075 of *Lecture Notes in Computer Science*, pages 130–140. Springer, 1996. doi:10.1007/3-540-61258-0_11.

27 Sébastien Ferenczi. Complexity of sequences and dynamical systems. *Discret. Math.*, 206(1-3):145–154, 1999. doi:10.1016/S0012-365X(98)00400-2.

28 Gabriele Fici and Pawel Gawrychowski. Minimal absent words in rooted and unrooted trees. In *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019*, volume 11811 of *Lecture Notes in Computer Science*, pages 152–161. Springer, 2019. doi:10.1007/978-3-030-32686-9_11.

29 Gabriele Fici, Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Word assembly through minimal forbidden words. *Theor. Comput. Sci.*, 359(1-3):214–230, 2006. doi:10.1016/j.tcs.2006.03.006.

30 Gabriele Fici, Antonio Restivo, and Laura Rizzo. Minimal forbidden factors of circular words. *Theor. Comput. Sci.*, 792:144–153, 2019. doi:10.1016/j.tcs.2018.05.037.

31 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. URL: http://www.jstor.org/stable/2034009.

32 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.

33 Yuta Fujishige, Yuki Tsujimaru, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing DAWGs and minimal absent words in linear time for integer alphabets. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016*, volume 58 of *LIPIcs*, pages 38:1–38:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.MFCS.2016.38.

34 Arnab Ganguly, Manish Patil, Rahul Shah, and Sharma V. Thankachan. A linear space data structure for range LCP queries. *Fundam. Inform.*, 163(3):245–251, 2018. doi:10.3233/FI-2018-1741.

35 Sara P. Garcia, Armando J. Pinho, João M. O. S. Rodrigues, Carlos A. C. Bastos, and Paulo J. S. G. Ferreira. Minimal absent words in prokaryotic and eukaryotic genomes. *PLoS ONE*, 6, 2011. doi:10.1371/journal.pone.0016065.

36 Pawel Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. Weighted ancestors in suffix trees. In *Algorithms - ESA 2014 - 22th Annual European Symposium*, volume 8737 of *Lecture Notes in Computer Science*, pages 455–466. Springer, 2014. doi:10.1007/978-3-662-44777-2_38.

37 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.

38 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, FOCS 1989*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

39 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:42–54, 2014. doi:10.1016/j.tcs.2013.10.010.

40 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

41 Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, pages 28:1–28:12, 2016. doi:10.4230/LIPIcs.CPM.2016.28.

42 Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, 2018. URL: https://mimuw.edu.pl/~kociumaka/files/phd.pdf.

43 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A linear-time algorithm for seeds computation. *ACM Trans. Algorithms*, 16(2):27:1–27:23, 2020. doi:10.1145/3386369.

44 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Efficient data structures for the factor periodicity problem. In *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012*, volume 7608 of *Lecture Notes in Computer Science*, pages 284–294, 2012. doi:10.1007/978-3-642-34109-0_30.

45 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

46 Tsvi Kopelowitz, Gregory Kucherov, Yakov Nekrich, and Tatiana Starikovskaya. Cross-document pattern matching. *J. Discrete Algorithms*, 24:40–47, 2014. doi:10.1016/j.jda.2013.05.002.

47 Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007*, pages 565–574. SIAM, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283444.

48 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

49 Kotaro Matsuda, Kunihiko Sadakane, Tatiana Starikovskaya, and Masakazu Tateshita. Compressed orthogonal search on suffix arrays with applications to range LCP. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, pages 23:1–23:13, 2020. doi:10.4230/LIPIcs.CPM.2020.23.

50 Takuya Mieno, Yuki Kuhara, Tooru Akagi, Yuta Fujishige, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Minimal unique substrings and minimal absent words in a sliding window. In *46th SOFSEM*, volume 12011 of *Lecture Notes in Computer Science*, pages 148–160. Springer, 2020. doi:10.1007/978-3-030-38919-2_13.

51 Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Words and forbidden factors. *Theor. Comput. Sci.*, 273(1-2):99–117, 2002. doi:10.1016/S0304-3975(00)00436-9.

52 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Text indexing and searching in sublinear time. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, volume 161 of *LIPIcs*, pages 24:1–24:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.24.

53 Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016. URL: http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/compact-data-structures-practical-approach?format=HB.

54 Takahiro Ota and Hiroyoshi Morita. On the adaptive antidictionary code using minimal forbidden words with constant lengths. In *Proceedings of the International Symposium on Information Theory and its Applications, ISITA 2010*, pages 72–77. IEEE, 2010. doi:10.1109/ISITA.2010.5649621.

55 Mihai Patrascu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011. doi:10.1137/09075336X.

56 Diogo Pratas and Jorge M Silva. Persistent minimal sequences of SARS-CoV-2. *Bioinformatics*, July 2020. btaa686. doi:10.1093/bioinformatics/btaa686.

57 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 166–175. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.26.

58 Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013. doi:10.1007/s00453-012-9618-6.

59 Mikhail Rubinchik and Arseny M. Shur. Counting palindromes in substrings. In *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017*, volume 10508 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 2017. doi:10.1007/978-3-319-67428-5_25.

60 Raquel M. Silva, Diogo Pratas, Luísa Castro, Armando J. Pinho, and Paulo J. S. G. Ferreira. Three minimal sequences found in Ebola virus genomes and absent from human DNA. *Bioinform.*, 31(15):2421–2425, 2015. doi:10.1093/bioinformatics/btv189.

61 Yuka Tanimura, Takaaki Nishimoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Small-space LCE data structure with constant-time queries. In *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017*, volume 83 of *LIPIcs*, pages 10:1–10:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.MFCS.2017.10.

62 Alexander Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Math. Comput. Sci.*, 1(4):571–603, 2008. doi:10.1007/s11786-007-0033-3.

63 Andrew C. Yao. Space-time tradeoff for answering range queries (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC 1982, pages 128–136. ACM, 1982. doi:10.1145/800070.802185.

Constructing the Bijective and the Extended Burrows–Wheeler Transform in Linear Time

Hideo Bannai
M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan

Juha Kärkkäinen
Helsinki Institute of Information Technology (HIIT), Finland

Dominik Köppl
M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan

Marcin Piątkowski
Nicolaus Copernicus University, Toruń, Poland

Abstract

The Burrows–Wheeler transform (BWT) is a permutation whose applications are prevalent in data compression and text indexing. The *bijective BWT* (BBWT) is a bijective variant of it. Although it is known that the BWT can be constructed in linear time for integer alphabets by using a linear time suffix array construction algorithm, it was up to now only conjectured that the BBWT can also be constructed in linear time. We confirm this conjecture in the word RAM model by proposing a construction algorithm that is based on SAIS, improving the best known result of $\mathcal{O}(n \lg n / \lg \lg n)$ time to linear. Since we can reduce the problem of constructing the extended BWT to constructing the BBWT in linear time, we obtain a linear-time algorithm computing the extended BWT at the same time.

2012 ACM Subject Classification Theory of computation; Mathematics of computing → Combinatorics on words

Keywords and phrases Burrows–Wheeler Transform, Lyndon words, Circular Suffix Array, Suffix Array Construction Algorithm

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.7

Related Version *Full Version*: `https://arxiv.org/abs/1911.06985`

Funding *Hideo Bannai*: JSPS KAKENHI Grant Number `JP20H04141`
Dominik Köppl: JSPS KAKENHI Grant Numbers `JP18F18120` and `JP21K17701`

1 Introduction

The Burrows–Wheeler transform (BWT) [4] is a transformation permuting the characters of a given string $T\$$, where $\$$ is a character that is strictly smaller than all characters occurring it T. The i-th entry of the BWT of $T\$$ is the character preceding the i-th lexicographically smallest suffix of $T\$$, or $\$$ if this suffix is $T\$$ itself. Strictly speaking, the BWT is not a bijection since its output contains $\$$ at an arbitrary position while it requests the input T to have $\$$ as a delimiter at its end in order to restore T. A variant, called the bijective BWT [19, 12], is a *bijective* transformation, which does not require the artificial delimiter $\$$. It is based on the Lyndon factorization [5] of T. In this variant, the output consists of the last characters of the lexicographically sorted cyclic rotations of all factors composing the Lyndon factorization of T.

In the following, we call the BWT *traditional* to ease the distinguishability of both transformations. It is well known that the traditional BWT has many applications in data compression [1] and text indexing [8, 9, 10]. Recently, such a text index was adapted to work with the bijective BWT [2].

Related Work. In what follows, we review the traditional BWT construction via suffix arrays, and some algorithms computing the BBWT or the extended BWT. For the complexity analysis, we take a text T of length n whose characters are drawn from a polynomial bounded integer alphabet $\{1, \ldots, n^{\mathcal{O}(1)}\}$. Let us start with the traditional BWT, which we can construct thanks to linear time suffix array construction algorithms [23, 17] in linear time. That is because the traditional BWT, denoted by $\mathsf{BWT}[1..n]$, is determined by $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$ for $\mathsf{SA}[i] > 1$ and $\mathsf{BWT}[i] = T[n]$ for $\mathsf{SA}[i] = 1$. Considering the bijective BWT, Gil and Scott [12] postulated that it can be built in linear time, but did not give a construction algorithm. It is clear that the time is upper bounded by the total length of all conjugates [22, after Example 9], which is $\mathcal{O}(n^2)$. In the same paper, Mantaci et al. [22] also introduced the *extended* BWT, a generalization of the BBWT in that it is a BWT based on a set $\mathcal{S}$ of *primitive* strings, i.e., strings that are not periodic. Hon et al. [15] provided an algorithm building the extended BWT in $\mathcal{O}(n \lg n)$ time. Their idea is to construct the *circular suffix array* $\mathsf{SA}_\circ$ such that the i-th position of the extended BWT is given by $T[\mathsf{SA}_\circ[i] - 1]$, where T is the concatenation of all strings in $\mathcal{S}$. Bonomo et al. [3] presented the most recent algorithm building the bijective BWT online in $\mathcal{O}(n \lg n / \lg \lg n)$ time. In [3, Sect. 6], they also gave a linear time reduction from computing the extended BWT to computing the BBWT. Knowing that an irreducible word has exactly one conjugate being a Lyndon word, the reduction is done by exchanging each element of the set of irreducible strings $\mathcal{S}$ by the conjugate being a Lyndon word, and concatenating these Lyndon words after sorting them in descending order. Consequently, a linear-time BBWT construction algorithm can be used to compute the extended BWT in linear time.

On the practical side, we are aware of the work of Branden Brown[1], Yuta Mori in his OpenBWT library[2], and of Neal Burns[3]. While the first is a naive but easily understandable implementation calling a general sorting algorithm on all conjugates to directly compute the BBWT, the second seems to be an adaptation of the suffix array – induced sorting (SAIS) algorithm [23] to induce the BBWT. The last one takes an already computed suffix array SA as input, and modifies SA such that reading the characters $T[\mathsf{SA}[i] - 1]$ gives the BBWT. For that, this algorithm shifts entries in SA to the right until they fit. Hence, the running time is based on the lengths of these shifts, which can be $\mathcal{O}(n^2)$, but seem to be negligible in practice for common texts.

Our Result. In this article, we present a linear time algorithm computing the BBWT in the word RAM model. The main idea is to adapt SAIS to compute the circular suffix array of the Lyndon factors. We obtain linear running time by exploiting some facts based on the nature of the Lyndon factorization.

2 Preliminaries

Our computational model is the word RAM model with word size $\Omega(\lg n)$. Accessing a word costs $\mathcal{O}(1)$ time. In this article, we study strings on an *integer* alphabet $\Sigma = \{1, \ldots, \sigma\}$ with size $\sigma = n^{\mathcal{O}(1)}$.

[1] https://github.com/zephyrtronium/bwst
[2] https://web.archive.org/web/20170306035431/https://encode.ru/attachment.php?attachmentid=959&d=1249146089
[3] https://github.com/NealB/Bijective-BWT

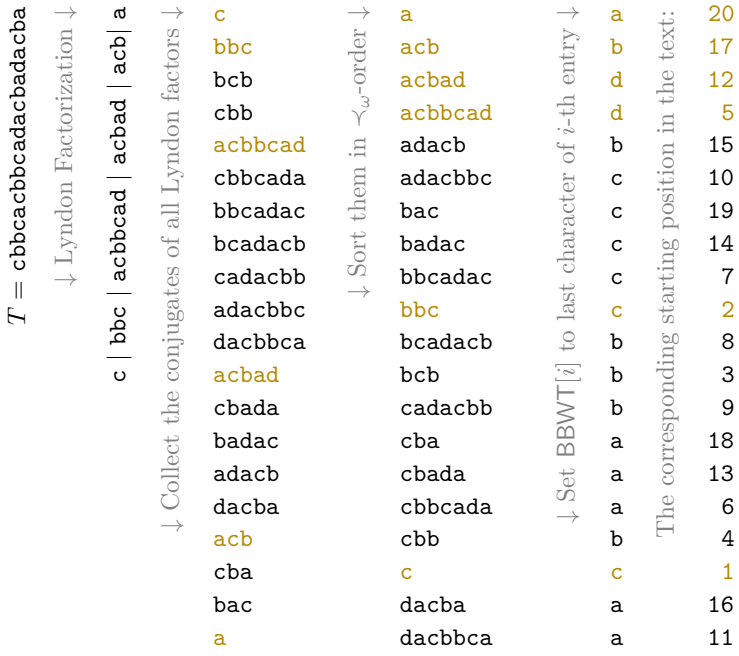


Figure 1 Constructing BBWT of $T =$ `cbbcacbbcadacbadacba`. The Lyndon factors are highlighted (■). Reading the characters of the penultimate column top-down yields BBWT. The last column shows in its i-th row the starting position of the i-th smallest conjugate of a Lyndon factor in the text. It is the circular suffix array studied later in Sect. 4.1. Note that `cbb` $\prec_{\text{lex}}$ `cbbcada`, but `cbbcada` $\prec_\omega$ `cbb`.

Strings. We call an element $T \in \Sigma^*$ a *string*. Its length is denoted by $|T|$. Given an integer $j \in [1..|T|]$, we access the j-th character of T with $T[j]$. Given a string $T \in \Sigma^*$, we denote with T^k that we concatenate k times the string T. When T is represented by the concatenation of $X, Y, Z \in \Sigma^*$, i.e., $T = XYZ$, then X, Y, and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. A prefix X, substring Y, or suffix Z is called *proper* if $X \neq T$, $Y \neq T$, or $Z \neq T$, respectively. A proper prefix X of T is called a *border* of T if it is also a suffix of T. T is called *border-free* if it has no border. For two integers i and j with $1 \leq i \leq j \leq |T|$, let $T[i..j]$ denote the substring of T that begins at position i and ends at position j in T. If $i > j$, then $T[i..j]$ is the empty string. In particular, the suffix starting at position j of T is denoted with $T[j..n]$. A string T is called *primitive* if it cannot be written as $T = S^k$ for a string $S \in \Sigma^+$ and $k \geq 2$.

Orders on Strings. We denote the *lexicographic order* with $\prec_{\text{lex}}$. Given two strings S and T, then $S \prec_{\text{lex}} T$ if S is a proper prefix of T or there exists an integer ℓ with $1 \leq \ell \leq \min(|S|, |T|)$ such that $S[1..\ell-1] = T[1..\ell-1]$ and $S[\ell] < T[\ell]$. We write $S \prec_\omega T$ if the infinite concatenation $S^\omega := SSS\cdots$ is lexicographically smaller than $T^\omega := TTT\cdots$. For instance, `ab` $\prec_{\text{lex}}$ `aba` but `aba` $\prec_\omega$ `ab`. The relation $\prec_\omega$ induces an order on the set of *primitive* strings[4], which we call $\prec_\omega$*-order*.

[4] The order cannot be generalized to strings in general since `a` $\neq$ `aa` but neither `a` $\prec_\omega$ `aa` nor `aa` $\prec_\omega$ `a` holds.

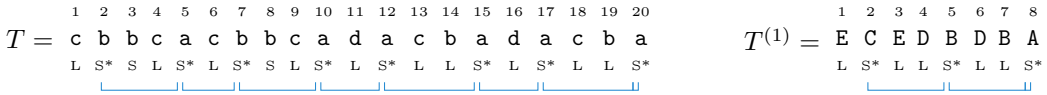


Figure 2 Splitting T and $T^{(1)}$ into LMS substrings. The rectangular brackets below the types represent the LMS substrings. $T^{(1)}$ is T after the replacement of its LMS substrings with their corresponding ranks defined in Sect. 4.3 and on the left of Fig. 3.

Lyndon Words. Given a primitive string $T = T[1..n]$, its i-th *conjugate* $\mathsf{conj}_i(T)$ is defined as $T[i+1..n]T[1..i]$ for an integer $i \in [0..n-1]$. Since T is primitive, all its conjugates are distinct. We say that T and every one of its conjugates belongs to the *conjugate class* $\mathsf{conj}(T) := \{\mathsf{conj}_0(T), \ldots, \mathsf{conj}_{n-1}(T)\}$. If a conjugate class contains *exactly* one conjugate that is lexicographically smaller than all other conjugates, then this conjugate is called a *Lyndon word* [21]. Equivalently, a string T is said to be a Lyndon word if and only if $T \prec S$ for every proper suffix S of T. A consequence is that a Lyndon word is *border-free*.

The *Lyndon factorization* [5] of $T \in \Sigma^+$ is the unique factorization of T into a sequence of Lyndon words $F_1 \cdots F_z$, where (a) each $F_x \in \Sigma^+$ is a Lyndon word, and (b) $F_x \succeq_{\text{lex}} F_{x+1}$ for each $x \in [1..z)$.

▶ **Lemma 1** ([7, Algo. 2.1]). *The Lyndon factorization of a string can be computed in linear time.*

Each Lyndon word F_x for $x \in [1..z]$ is called a *Lyndon factor*. For what follows, we fix a string $T[1..n]$ over an alphabet Σ of size σ. We use the string $T :=$ `cbbcacbbcadacbadacba` as our running example. Its Lyndon factorization is `c`, `bbc`, `acbbcad`, `acbad`, `acb`, `a`.

Bijective Burrows–Wheeler Transform. We denote the bijective BWT of T by BBWT, where $\mathsf{BBWT}[i]$ is the last character of the i-th string in the list storing the conjugates of all Lyndon factors $F_1, \ldots, F_z$ of T sorted with respect to $\prec_\omega$. Figure 1 shows the BBWT of our running example.

3 Reviewing SAIS

Our idea is to adapt SAIS to compute $\mathsf{SA}_\circ$ instead of the suffix array. To explain this adaptation, we briefly review SAIS. First, SAIS assigns each suffix a type, which is either `L` or `S`:

- $T[i..|T|]$ is an `L` suffix if $T[i..|T|] \succ_{\text{lex}} T[i+1..|T|]$, or
- $T[i..|T|]$ is an `S` suffix otherwise, i.e., $T[i..|T|] \prec_{\text{lex}} T[i+1..|T|]$,

where we stipulate that $T[|T|]$ is always type `S`. Since it is not possible that $T[i..|T|] = T[i+1..|T|]$, SAIS assigns each suffix a type. An `S` suffix $T[i..|T|]$ is additionally an `S`* suffix (also called LMS suffix in [23]) if $T[i-1..|T|]$ is an `L` suffix. The substring between two succeeding `S`* suffixes is called an *LMS substring*. In other words, a substring $T[i..j]$ with $i < j$ is an LMS substring if and only if $T[i..|T|]$ and $T[j..|T|]$ are `S`* suffixes and there is no $k \in (i..j)$ such that $T[k..|T|]$ is an `S`* suffix. A border case is $T[|T|..|T|]$, which has to be the smallest suffix of T (and can be achieved by appending the artificial character `$` to T lexicographically smaller than all other characters appearing it T) such that $T||T|..|T|]$ in an `S`* suffix. We additionally treat $T[|T|..|T|]$ as an LMS substring. The types for the suffixes of our running example are given in Fig. 2. Regarding the defined types, we make no distinction between suffixes and their starting positions (e.g., the statements that (a) $T[i]$ is type `L` and (b) $T[i..|T|]$ is an `L` suffix are equivalent).

LMS Substring	Contents	Non-Terminal
$T[2..5]$	bbca	E
$T[5..7]$	acb	C
$T[7..10]$	bbca	E
$T[10..12]$	ada	D
$T[12..15]$	acba	B
$T[15..17]$	ada	D
$T[17..20]$	acba	B
$T[20..20]$	a	A

S* Suffix	Contents
$T[20]$	a
$T[17..20]$	acba
$T[12..20]$	acbadacba
$T[5..20]$	acbbcadacbadacba
$T[15..20]$	adacba
$T[10..20]$	adacbadacba
$T[2..20]$	bbcacbbcadacbadacba
$T[7..20]$	bbcadacbadacba

Figure 3 Ranking of the LMS substrings and the S* suffixes of our running example given in Sect. 4.3 and Fig. 2. *Left*: LMS substrings assigned with non-terminals reflecting their corresponding rank in $\prec_{\text{LMS}}$-order. *Right*: S* suffixes of T sorted in $\prec_{\text{lex}}$-order. Note that $T[5..7] = \texttt{acb} \prec_{\text{lex}} \texttt{acba} = T[12..15] = T[17..20]$, but $\texttt{acba} \prec_{\text{LMS}} \texttt{acb}$.

Next, Nong et al. [23, Def. 3.3] define a relation $\prec_{\text{LMS}}$ on substrings of T based on the lexicographic order and the types: Given two substrings S and U. Let i be the smallest integer such that (1) $S[i] < U[i]$ or (2) $S[i]$ is type L and $U[i]$ is type S or S*. If such an i exists, then we write $S \prec_{\text{LMS}} U$. For two LMS substrings S and U with $S \neq U$, either $S \prec_{\text{LMS}} U$ or $U \prec_{\text{LMS}} S$, even if S is a prefix of U (cf. the discussion below of Def. 3.3 in [23]). So $\prec_{\text{LMS}}$ is an order on the LMS substrings. The $\prec_{\text{LMS}}$-order is shown on the left side of Fig. 3 for the LMS substrings listed of the left side of Fig. 2. The crucial observation is that the $\prec_{\text{LMS}}$-order of the LMS substrings coincides with the lexicographic order of the suffixes starting with the LMS substrings [23, Lemma 3.8].

Nong et al. [23, A3.4] compute the $\prec_{\text{LMS}}$-order of all LMS substrings with the induced sorting (which we describe below for the step of computing the rank of all suffixes). Figure 4 visualizes this computation on our running example. Hence, we can assign each LMS substring a rank based on the $\prec_{\text{LMS}}$-order. Next, we build a string $T^{(1)}$ of LMS substring ranks with $T^{(1)}[i]$ being the rank of the i-th LMS substring of T in text order.[5] See the right side of Fig. 2 for our running example. We recursively call SAIS on this text of ranks until the ranks of all LMS substrings are distinct. Given that we have computed $T^{(k)}$ and all characters of $T^{(k)}$ (i.e., the ranks of the respective LMS substrings) are distinct, then these ranks determine the order of the S* suffixes of $T^{(k)}$. The order of the S* suffixes of our running example are given in Fig. 3 on the right side. Having the order of the S* suffixes, we allocate space for the suffix array, and divide the suffix array into buckets, grouping each suffix with the same starting character and same type (either L or S) into one bucket. Among all suffixes with the same starting character, the L suffixes precede the S suffixes [18, Corollary 3]. Putting S* suffixes in their respective buckets according to their order (smallest elements are the leftmost elements in the buckets), we can induce the L suffixes, as these precede either L or S* suffixes. For that, we scan SA from left to right, and take action only for suffix array entries that are not empty: When accessing the entry $\mathsf{SA}[k] = i$ with $i > 1$, write $i-1$ to the leftmost available slot of the L bucket with the character $T[i-1]$ if $T[i-1..|T|]$ is an L suffix. Finally, we can induce the $\prec_{\text{lex}}$-order of the S suffixes by scanning the suffix array from right

[5] We can obtain $T^{(1)}$ by scanning T from left to right and replacing each LMS substring by its respective rank, but keep its last character in T if this character is the first character of the subsequent LMS substring. We further omit the first characters of T that are not part of an LMS substring (which must be of type L).

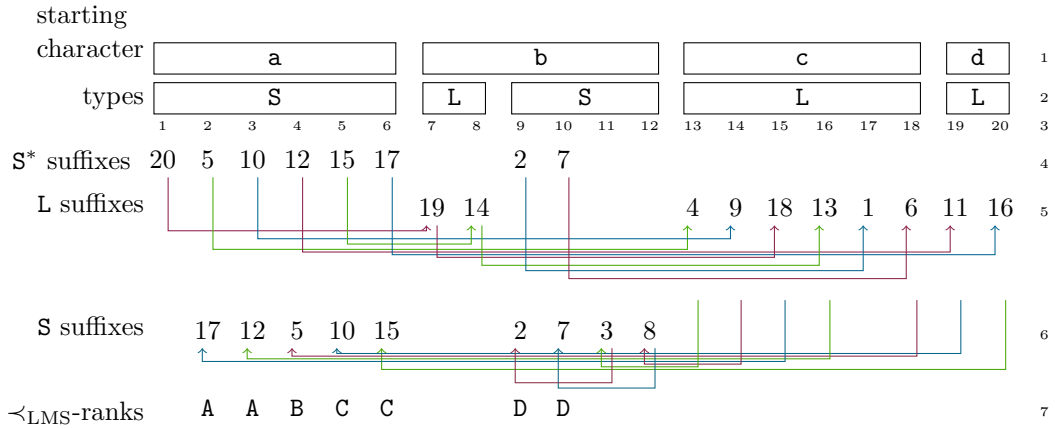


Figure 4 Inducing LMS substrings. Rows 1 and 2 show the partitioning of SA into buckets, first divided by the starting characters of the respective LMS substrings, and second by the types L and S. In Row 4, the S* suffixes are inserted into their respective S buckets. Here it is sufficient to only put the smallest S* suffix in the correct order among all other S* suffixes in the same bucket. This suffix is $T[20..20]$ in our example, stored at the suffix array entry 1. The S* (resp. L) suffixes induce the L (resp. S) suffixes in Row 5 (resp. Row 6). The last row assigns each S* suffix a meta-character representing its $\prec_{\text{LMS}}$-rank. We can compute two subsequent suffixes by character-wise comparison, spending $\mathcal{O}(|T|)$ time in total since the LMS substrings have a total length of $\mathcal{O}(|T|)$.

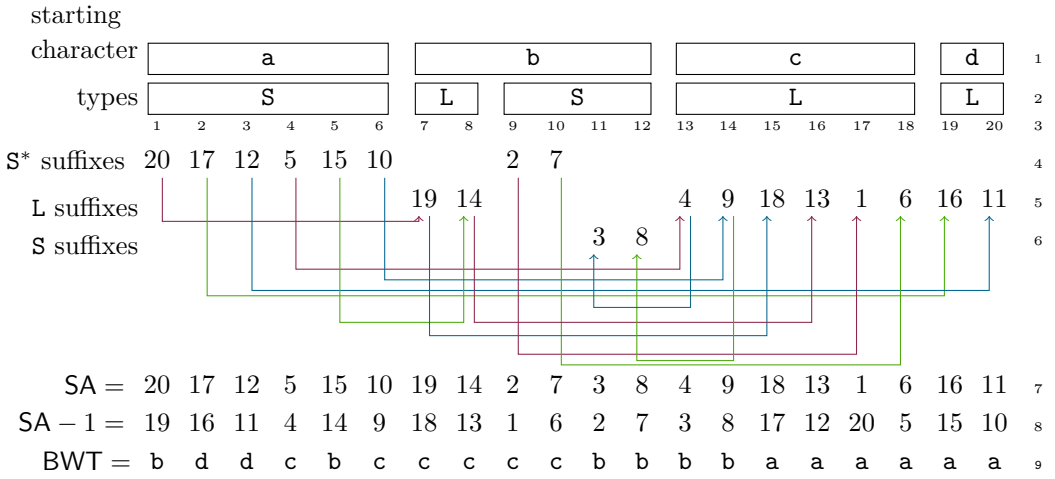


Figure 5 Inducing L and S suffixes from the $\prec_{\text{lex}}$-order of the S* suffixes given in Fig. 2. Rows 1 and 2 show the partitioning of SA into buckets, first divided by the starting characters of the respective suffixes, and second by the types L and S. Row 4 is SA after inserting the S* suffixes according to their $\prec_{\text{lex}}$-order rank obtained from the right of Fig. 3. The S* (resp. L) suffixes induce the L (resp. S) suffixes in Row 5 (resp. Row 6). Putting all together yields SA in Row 7. In the penultimate row SA − 1, each text position stored in SA is decremented by one, or set to n if this position was 1. The last row shows $T[(\mathsf{SA}-1)[i]] = \mathsf{BWT}[i]$ in its i-th column, which is the BWT of T. This BWT is not reversible since the input is not terminated with a unique character like $. To obtain the BWT of $T$$, we first write $T[\mathsf{SA}[1]] = T[20] =$ a to the output, and then BWT, but exchanging $\mathsf{BWT}[\mathsf{SA}^{-1}[1]] = \mathsf{BWT}[17] =$ a with $, i.e., abddcbcccccbbbbaa$aaa.

U	V	$\prec_{\text{lex}}$	$\prec_{\omega}$	$\prec_{\text{LMS}}$
aba	aca	$<$	$<$	$<$
adc	adcb	$<$	$<$	$>$
acb	acba	$<$	$>$	$>$

Figure 6 Comparison of the three orders studied in this paper applied to LMS substrings. Assume that U and V are substrings of the text surrounded by a character d (i.e., $T = \ldots \mathtt{d}U\mathtt{d}\ldots\mathtt{d}V\mathtt{d}\ldots$) such that the first and the last character of both U and V start with an $\mathtt{S}^*$ suffix. We mark with the signs $<$ and $>$ whether U is smaller or respectively larger than V according to the corresponding order. The orders can differ only when one string is the prefix of another string, as this is the case in the last two rows. Finally, occurrences of U and V can be $\prec_{\text{LMS}}$-incomparable in different contexts such as $\ldots\mathtt{d}U\mathtt{a}\ldots\mathtt{d}V\mathtt{d}\ldots$, for instance.

to left: When accessing the entry $\mathsf{SA}[k] = i$, write $i-1$ to the rightmost available slot of the S type bucket with the character $T[i-1]$ if $T[i-1..|T|]$ is an S suffix. As an invariant, we always fill an L bucket and an S bucket from left to right and from right to left, respectively. So we can think of each L bucket and each S bucket as a list with an insertion operation at the end or at the beginning, respectively. We conduct these steps for our running example in Fig. 5.

In total, the induction takes $\mathcal{O}(|T|)$ time. The recursion step takes also $\mathcal{O}(|T|)$ time since there are at most $|T|/2$ LMS substrings (there are no two text positions $T[i]$ and $T[i+1]$ with type $\mathtt{S}^*$ for $i \in [1..n-1]$). This gives $\mathcal{T}(n) = \mathcal{T}(n/2) + \mathcal{O}(n) = \mathcal{O}(n)$ total time, where $\mathcal{T}(n)$ denotes the time complexity for computing a suffix array of length n.

However, with SAIS we cannot obtain $\mathsf{SA}_\circ$ ad-hoc since we need to exchange $\prec_{\text{lex}}$ with $\prec_{\omega}$. Although these orders are the same for Lyndon words [3, Thm. 8], they differ for LMS substrings as can be seen in Fig. 6. Hence, we need to come up with an idea to modify SAIS in such way to compute $\mathsf{SA}_\circ$.

4 Our Adaptation

We want SAIS to sort Lyndon conjugates in $\prec_{\omega}$-order instead of suffixes in $\prec_{\text{lex}}$-order. For that, we first get rid of duplicate Lyndon factors to facilitate the analysis, and then subsequently introduce a slightly different notion to the types of suffixes and LMS substrings, which translates the suffix sorting problem into computing the BBWT.

4.1 Reduced String and Composed Lyndon Factorization

In a pre-computation step, we want to facilitate our analysis by removing all identical Lyndon factors from T yielding a reduced string R. We want to remove them to make conjugates unique; thus we can linearly order them. Consequently, the first step is to show that we can obtain the BBWT of T from the circular suffix array of R (which we will subsequently define):

The (composed) *Lyndon factorization* [5] of $T \in \Sigma^+$ is the factorization of T into $T_1^{\tau_1} \cdots T_t^{\tau_t} = T$, where $T_1, \ldots, T_t$ is a sequence of lexicographically decreasing Lyndon words and $\tau_x \geq 1$ for $x \in [1..t]$. Let $R := T_1 \cdots T_t$ denote the text, in which all duplicate Lyndon factors are removed. Obviously, the Lyndon factorization of R is $T_1, \ldots, T_t$. Let $\mathsf{b}(T_x)$ and $\mathsf{e}(T_x)$ denote the starting and ending position of the x-th Lyndon factor in R, i.e., $R[\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ is the x-th Lyndon factor T_x of R.

Our aim is to compute the $\prec_\omega$-order of all conjugates of all Lyndon factors of R, which are given by the set $\mathcal{S} := \bigcup_{x=1}^{t} \mathsf{conj}(T_x)$. Like Hon et al. [14], we present this order in the so-called *circular suffix array* $\mathsf{SA}_\circ$ of $\{T_1, \ldots, T_t\}$, i.e., an array of length $|R|$ with $\mathsf{SA}_\circ[k] = i$ if $R[i..\mathsf{e}(T_x)]R[\mathsf{b}(T_x)..i-1]$ is the k-th smallest string in $\mathcal{S}$ with respect to $\prec_\omega$, where $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$. The length of $\mathsf{SA}_\circ$ is $|R|$ since we can associate each text position $\mathsf{SA}_\circ[k]$ in R with a conjugate starting with $R[\mathsf{SA}_\circ[k]]$.

Having the circular suffix array $\mathsf{SA}_\circ$ of $\{T_1, \ldots, T_t\}$, we can compute the BBWT of T by reading $\mathsf{SA}_\circ[k]$ for $k \in [1..|R|]$ from left to right: Given $\mathsf{SA}_\circ[k] = i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$, we append $T[i^-]$ exactly τ_x times to BBWT, where i^- is $i-1$ or $\mathsf{e}(T_x)$ if $i = \mathsf{b}(T_x)$. (This is analogous to the definition of BWT where we set $\mathsf{BWT}[i] = T[n]$ for $\mathsf{SA}[i] = 1$, but here we wrap around each Lyndon factor.)

4.2 Translating Types to Inf-Suffixes

In what follows, we continue working with R defined in Sect. 4.1 instead of T. Let $R[i..]$ denote the infinite string $R[i..\mathsf{e}(T_x)]T_xT_x\cdots = \mathsf{conj}_k(T_x)\mathsf{conj}_k(T_x)\cdots$ with x such that $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ and $k = i - \mathsf{b}(T_x)$. We say that $R[i..]$ is an *inf-suffix*. As a shorthand, we also write $T_x[i..] = \mathsf{conj}_{i-1}(T_x)\mathsf{conj}_{i-1}(T_x)\cdots$ for the inf-suffix starting at $R[\mathsf{b}(T_x)+i-1]$. In particular, $T_x[|T_x|+1..] = T_x[1..] = T_xT_x\cdots$.

Like in SAIS, we distinguish between $\mathtt{L}$ and $\mathtt{S}$ inf-suffixes:

- $R[i..]$ is an $\mathtt{L}$ inf-suffix if $R[i..] \succ_{\text{lex}} R[i^+..]$, and
- $R[i..]$ is an $\mathtt{S}$ inf-suffix if $R[i..] \prec_{\text{lex}} R[i^+..]$,

where i^+ is either $i+1$ or $\mathsf{b}(T_x)$ if $i = \mathsf{e}(T_x)$, and x is given such that $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$. Finally, we introduce the $\mathtt{S}^*$ inf-suffixes as a counterpart to the $\mathtt{S}^*$ suffixes: If $R[i..]$ is an $\mathtt{S}$ inf-suffix, it is further an $\mathtt{S}^*$ inf-suffix if $R[i^-..]$ is an $\mathtt{L}$ inf-suffix with i^- being either $i-1$ or $\mathsf{e}(T_x)$ if $i = \mathsf{b}(T_x)$, and $x \in [1..t]$ chosen such that $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$.

When speaking about types, we do not distinguish between an inf-suffix and its starting position in R. This definition assigns all positions of R a type except those belonging to a Lyndon factor of length one. We solve this by stipulating that all Lyndon factors of length one start with an $\mathtt{S}^*$ inf-suffix. However, in what follows, we temporarily omit all Lyndon factors of length one because we will later see that they can be placed at the beginning of their corresponding buckets in the circular suffix array. They nevertheless appear in the examples for completeness. To show that suffixes and inf-suffixes starting at the same position have the same type (except for some border-cases), the following lemma will be particularly useful:

▶ **Lemma 2** ([3, Lemma 7]). *For $i, j \in [1..|T_x|]$ and $x \in [1..t]$, the following statements are equivalent:*

1. $\mathsf{conj}_{i-1}(T_x) = T_x[i..|T_x|]T_x[1..i-1] \prec_{\text{lex}} T_x[j..|T_x|]T_x[1..j-1] = \mathsf{conj}_{j-1}(T_x)$;
2. $\mathsf{conj}_{i-1}(T_x) \prec_\omega \mathsf{conj}_{j-1}(T_x)$, *i.e.,* $T_x[i..] \prec_{\text{lex}} T_x[j..]$;
3. $T_x[i..|T_x|] \prec_{\text{lex}} T_x[j..|T_x|]$.

▶ **Lemma 3.** *Omitting all Lyndon factors of length one from R, the types of all positions match the original SAIS types, except maybe $R[1]$ and $R[\mathsf{b}(T_t)+1..|R|]$, where $R[1..]$ and $R[|R|..|R|]$ are always an $\mathcal{S}^*$ inf-suffix and an $\mathcal{S}^*$ suffix, respectively.*

Proof. We show that inf-suffixes as well as suffixes starting with Lyndon factors have the same type $\mathtt{S}^*$:

inf-suffxes. Assume that $R[\mathsf{b}(T_x)..]$ is an $\mathtt{L}$ inf-suffix for an $x \in [1..t]$. According to the definition $R[\mathsf{b}(T_x)+1..] \prec_{\text{lex}} R[\mathsf{b}(T_x)..]$, i.e., $T_x[2..] \prec_{\text{lex}} T_x[1..]$, and with Lemma 2, $T_x[2..|T_x|] \prec_{\text{lex}} T_x$, contradicting that T_x is a Lyndon word. Finally, $R[\mathsf{b}(T_x)..]$ is an $\mathtt{S}^*$ inf-suffix because $T_x \prec_{\text{lex}} T_x[|T_x|]$ and hence $T_x[1..] \prec_{\text{lex}} T_x[|T_x|..]$, again with Lemma 2.

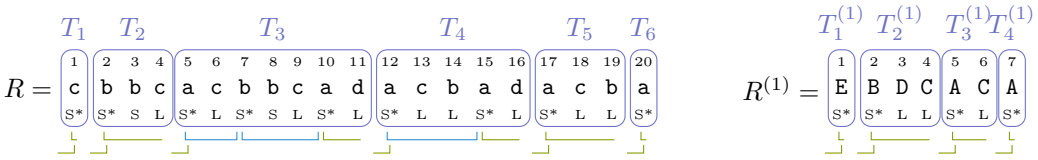


Figure 7 Splitting R and $R^{(1)}$ into LMS inf-substrings. The rectangular brackets below the types represent the LMS inf-substrings. Broken brackets denote that the corresponding LMS inf-substring ends with the first character of the Lyndon factor in which it is contained. They are colored in green (■); all other LMS inf-substrings are represented by brackets colored in blue (■). $R^{(1)}$ is R after the replacement of its LMS inf-substrings with their corresponding ranks defined in Sect. 4.3 and on the left of Fig. 8.

suffixes. Due to the Lyndon factorization, $R[\mathsf{b}(T_x)..|R|] \succ_{\text{lex}} R[\mathsf{b}(T_{x+1})..|R|]$ for $x \in [1..t-1]$. Hence, the suffix $R[\mathsf{e}(T_x)..|R|]$ starting at $R[\mathsf{e}(T_x)]$ has to be lexicographically larger than the suffix $R[\mathsf{e}(T_x)+1..|R|] = R[\mathsf{b}(T_{x+1})..|R|]$, otherwise we could extend the Lyndon factor T_x.

Consequently, $R[\mathsf{b}(T_x)..|R|]$ and $R[\mathsf{b}(T_x)..]$ are an $\mathtt{S}^*$ suffix and an $\mathtt{S}^*$ inf-suffix, respectively, and $R[\mathsf{e}(T_x)..|R|]$ and $R[\mathsf{e}(T_x)..]$ are an $\mathtt{L}$ suffix and an $\mathtt{L}$ inf-suffix.

The claim for all other positions $(\bigcup_{x=1}^{t-1}[\mathsf{b}(T_x)+1..\mathsf{e}(T_x)-1])$ follows by observing that $T_x[1..]$ is the $\prec_{\text{lex}}$-smallest inf-suffix among all inf-suffixes starting in T_x and $R[\mathsf{b}(T_{x+1})..|R|]$ is $\prec_{\text{lex}}$-smaller than all suffixes starting in $R[\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ for $x \in [1..t-1]$. ◀

A corollary is that $R[i..|R|] \prec_{\text{lex}} R[i..]$ for $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ and $x \in [1..t-1]$ since $T_{x+1} \prec_{\text{lex}} T_x$.[6] Next, we define the equivalent to the LMS substrings for the inf-suffixes, which we call *LMS inf-suffixes*: For $1 \le i < j \le |T_x|+1$, the substring $(T_xT_x)[i..j]$ is called an *LMS inf-substring* if and only if $T_x[i..]$ and $T_x[j..]$ are $\mathtt{S}^*$ inf-suffixes and there is no $k \in (i..j)$ such that $T_x[k..]$ is an $\mathtt{S}^*$ inf-suffix. This definition differs from the original LMS substrings (omitting the last one $R[|R|..|R|]$ being a border case) only for the last LMS inf-substring of each Lyndon factor. Here, we append $T_x[1]$ instead of $T_{x+1}[1]$ to the suffix starting with the last type $\mathtt{S}^*$ position of T_x.

4.3 Example

The LMS inf-substrings of our running example $T :=$ `cbbcacbbcadacbadacba` with $R = T$ are given in Fig. 7. Their $\prec_{\text{LMS}}$-ranking is given on the left side of Fig. 8, where we associate each LMS inf-substring, except those consisting of a single character, with a non-terminal reflecting its rank. By replacing the LMS inf-substrings by their $\prec_{\text{LMS}}$-ranks in the text while discarding the single character Lyndon factors, we obtain the string $T^{(1)} :=$ `EBDCACA`, whose LMS inf-substrings are given on the right side of Fig. 7. Among these LMS inf-substrings, we only continue with `BDC` and `AC`. Since all LMS-inf substrings are distinct, their $\prec_{\text{LMS}}$-ranks determine the $\prec_\omega$-order of the $\mathtt{S}^*$ inf-suffixes as shown on the right side of Fig. 8. It is left to induce the $\mathtt{L}$ and $\mathtt{S}$ suffixes, which is done exactly as in the SAIS algorithm. We conduct these steps in Fig. 9, which finally lead us to $\mathsf{SA}_\circ$.

[6] Consequently, for transforming SA into $\mathsf{SA}_\circ$, one only needs to shift values in SA to the right, as this is done by one of the implementations mentioned in the related work.

LMS Inf-Substring	Contents	Non-Terminal
$R[1]R[1]$	cc	-
$R[2..4]R[2]$	bbcb	E
$R[5..7]$	acb	B
$R[7..10]$	bbca	D
$R[10..11]R[10]$	ada	C
$R[12..15]$	acba	A
$R[15..16]R[12]$	ada	C
$R[17..19]R[17]$	acba	A
$R[20]R[20]$	aa	-

S* Inf-Suffix	Contents
$R[20..]$	a...
$R[17..]$	acb...
$R[12..]$	acbad...
$R[5..]$	acbbcad...
$R[15..]$	adacb...
$R[10..]$	adacbbc...
$R[7..]$	bbcadac...
$R[2..]$	bbc...
$R[1..]$	c...

Figure 8 Ranking of the LMS inf-substrings and the S* suffixes of our running example $T = R$ given in Sect. 4.3 and Fig. 7. *Left*: LMS inf-substrings assigned with non-terminals reflecting their corresponding rank in $\prec_{\text{LMS}}$-order. They have the same color as the respective rectangular brackets on the left of Fig. 7. The first and the last LMS substring do not receive a non-terminal since their lengths are one (remember that we omit Lyndon factors of length 1 in the recursive call). *Right*: S* inf-suffixes of T sorted in $\prec_{\text{lex}}$-order, which corresponds to the $\prec_{\omega}$ of the conjugate starting with this inf-suffix. Compared with Fig. 3, the suffixes $R[2..20]$ and $R[7..20]$ in the $\prec_{\text{lex}}$-order are order differently than their respective inf-suffixes $R[2..]$ and $R[7..]$ in the $\prec_{\text{lex}}$-order.

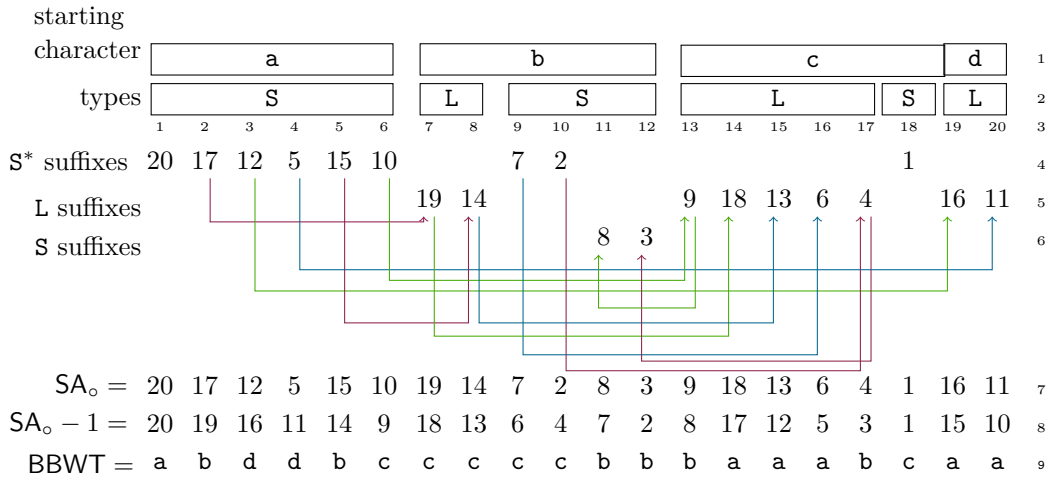


Figure 9 Inducing L and S inf-suffixes from the $\prec_{\text{lex}}$-order of the S* inf-suffixes given in Fig. 7. Rows 1 and 2 show the partitioning of $\mathsf{SA}_\circ$ into buckets, first divided by the starting characters of the respective inf-suffixes, and second by the types L and S. Row 4 is $\mathsf{SA}_\circ$ after inserting the S* inf-suffixes according to their $\prec_{\text{lex}}$-order rank obtained from the right of Fig. 8. The S* (resp. L) inf-suffixes induce the L (resp. S) inf-suffixes in Row 5 (resp. Row 6). Putting all together yields $\mathsf{SA}_\circ$ in Row 7. In the penultimate row $\mathsf{SA}_\circ - 1$, each text position stored in $\mathsf{SA}_\circ$ is decremented by one, wrapping around a Lyndon factor if necessary (for instance, $(\mathsf{SA}_\circ - 1)[2] = 19 = \mathsf{e}(T_5)$ since $\mathsf{SA}_\circ[2] = 17 = \mathsf{b}(T_5)$). The last row shows $R[(\mathsf{SA}_\circ - 1)[i]]$ in its i-th column, which is the BBWT of R as given in Fig. 1.

4.4 Correctness and Time Complexity

Let us recall that our task is to compute the $\prec_\omega$-order of the conjugates $\mathsf{conj}_{i_x-1}(T_x)$ for $i_x \in [1..|T_x|]$ of all Lyndon factors $T_1, \ldots, T_t$ of R. We will frequently use that $\mathsf{conj}_{i_x-1}(T_x) \prec_\omega \mathsf{conj}_{i_y-1}(T_y)$ is equivalent to $T_x[i_x..] \prec_{\text{lex}} T_y[i_y..]$ for $i_x \in [1..|T_x|]$ and $i_y \in [1..|T_y|]$. We start with showing that the $\prec_{\text{LMS}}$-ranks of the LMS inf-substrings determine the $\prec_{\text{lex}}$-order of the $\mathtt{S}^*$ inf-suffixes[7], whenever the LMS inf-suffixes are all distinct.

▶ **Lemma 4.** *Let S_x and S_y be two LMS inf-substrings that are prefixes of $T_x[i_x..]$ and $T_y[i_y..]$, respectively, for $i_x \in [1..|T_x|]$ and $i_y \in [1..|T_y|]$. If $S_x \prec_{\text{LMS}} S_y$ then $T_x[i_x..] \prec_{\text{lex}} T_y[i_y..]$.*

Proof. Given $S_x \prec_{\text{LMS}} S_y$, there is a position i such that (a) $S_x[i] < S_y[i]$ or (b) $S_x[i]$ is type $\mathtt{L}$ and $S_y[i]$ is type $\mathtt{S}$; let i be the smallest such position. In the latter case (b), there is a position $j > i$ such that $T_x[i_x + j - 1] = S_x[j] < S_x[i] = S_y[i] < S_y[j] = T_y[i_y + j - 1]$ and $T_x[i_x..i_x + j - 2] = T_y[i_y..i_y + j - 2]$, where we abused the notation that $T_x[k] = (T_xT_x \cdots)[k]$ for a $k \in [1..2|T_x|]$. In both cases (a) and (b), $T_x[i_x..] \prec_{\text{lex}} T_x[i_y..]$. ◀

Exactly as in the SAIS recursion step, we map each LMS inf-substring to its respective meta-character via its $\prec_{\text{LMS}}$-rank, obtaining a string $R^{(1)}$ whose characters are $\prec_{\text{LMS}}$-ranks. The lexicographic order $\prec_{\text{lex}}$ induces a natural order on the strings whose characters are drawn from the $\prec_{\text{LMS}}$-ranks. With that, we can determine the Lyndon factorization on $R^{(1)}$, which is given by the following connection:

▶ **Lemma 5.** *There is a one-to-one correspondence between Lyndon factors of R and $R^{(1)}$, meaning that each Lyndon factor of $R^{(1)}$ generates a Lyndon factor in R by expanding each of its $\prec_{\text{LMS}}$-ranks to the characters of the respective LMS inf-substring (while omitting the last character if it is the beginning of another LMS inf-substring), and vice-versa by contracting the characters of R to non-terminals.*

Proof. We first observe that each LMS inf-substring is contained in $T_x[1..|T_x|]T_x[1]$ for an $x \in [1..t]$. Now, let L be a Lyndon factor of $R^{(1)}$ with $L = r_1 \cdots r_\ell$ such that each r_i is a $\prec_{\text{LMS}}$-rank. Suppose that there is a $d \in [1..\ell - 1]$ such that $r_1 \cdots r_d$ expands to a suffix $T_x[s..|T_x|]$ of T_x (again omitting the last character of each expanded LMS inf-substring) and $r_{d+1} \cdots r_\ell$ expands to a prefix P of T_{x+1}. Since L is a Lyndon word, $r_1 \cdots r_d \prec_{\text{lex}} r_1 \cdots r_\ell \prec_{\text{lex}} r_{d+1} \cdots r_\ell$. Hence, $T_x[s..|T_x|] \prec_{\text{LMS}} T_x[s..|T_x|]T_x[1] \prec_{\text{LMS}} P$, and with Lemma 4, $T_x[1..] \prec_{\text{lex}} T_x[s..] \prec_{\text{lex}} T_{x+1}[1..]$, contradicting the Lyndon factorization of R with Lemma 2.

Finally, suppose that a Lyndon factor L_1 of $R^{(1)}$ expands to a proper prefix of a Lyndon factor T_x. Let L_2 be its subsequent Lyndon factor, which has to end inside T_x according to the above observation. Then $L_2 \prec_{\text{lex}} L_1$, which means that T_x contains an inf-suffix smaller than T_x due to Lemma 2, contradicting that T_x is a Lyndon factor. ◀

Thanks to Lemma 5, we do not have to compute the Lyndon factorization of $R^{(1)}$ needed in the recursive step, but can infer it from the Lyndon factorization of R. Additionally, we have the property that the order of the LMS inf-substrings in the recursive step only depends on the Lyndon factors they are (originally) contained in. It remains to show how the $\prec_{\text{LMS}}$-ranks of the LMS inf-substrings can be computed:

▶ **Lemma 6.** *We can compute the $\prec_{\text{LMS}}$-ranks of all LMS inf-substrings in linear time.*

[7] This is a counterpart to the property that the $\prec_{\text{LMS}}$-ranks of the LMS substrings determine the $\prec_{\text{lex}}$-order of the $\mathtt{S}^*$ suffixes [23, Theorem 3.12].

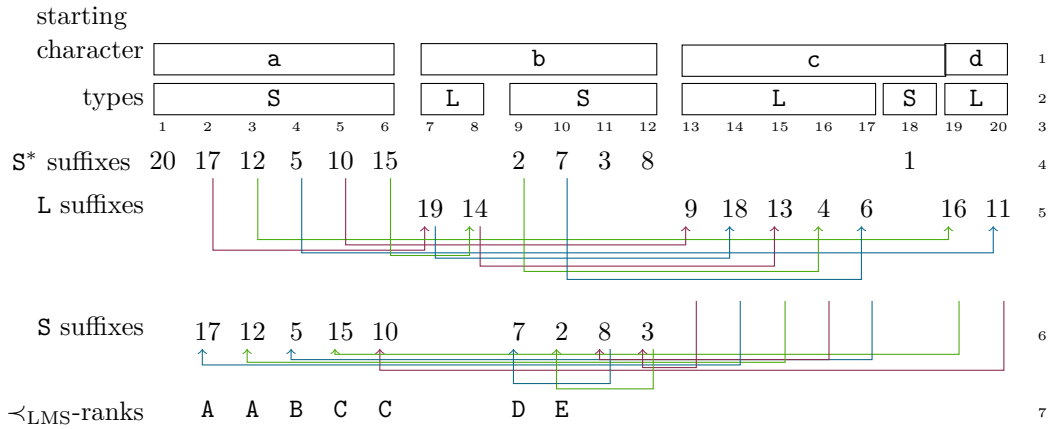


Figure 10 Inducing LMS inf-substrings. Thanks to the Lyndon factorization, we know the $\prec_\omega$-order of the inf-suffixes starting with the Lyndon factors, which is $T[20..] \prec_\omega T[17..] \prec_\omega T[12..] \prec_\omega T[5..] \prec_\omega T[2..] \prec_\omega T[1..]$. We insert the starting positions of these inf-suffixes in this order into their respective buckets, and fill the $\mathtt{S}^*$ buckets with the rest of $\mathtt{S}^*$ inf-suffixes by an arbitrary order (here we used the text order). Like Fig. 4, the $\mathtt{S}^*$ (resp. $\mathtt{L}$) suffixes induce the $\mathtt{L}$ (resp. $\mathtt{S}$) suffixes in Row 5 (resp. Row 6), but we skip those belonging to Lyndon factors of length one, since each of them is always stored at the leftmost position of its respective bucket. In the last row, we assign each LMS inf-substring a non-terminal based on its $\prec_{\text{LMS}}$-rank, but omitting those that correspond to factors of length one.

Proof. We follow the proof of [23, Theorem 3.12]. The idea is to know the $\prec_{\text{lex}}$-order among some smallest $\mathtt{S}^*$ inf-suffixes with which we can induce the $\prec_{\text{LMS}}$-ranks of all LMS inf-substrings. Here, we use the one-to-one correlation between each LMS inf-substring $R[i..j]$ and the respective $\mathtt{S}^*$ inf-suffix $R[i..]$ by using the starting position i for identification. To compute the order of the (traditional) LMS substrings, it sufficed to know the lexicographically smallest $\mathtt{S}^*$ suffix (cf. Fig. 4), which can be determined by appending an artificial character such as $\mathtt{\$}$ to R with the property that it is smaller than all other characters appearing in R. Here, we need to know the order of at least one $\mathtt{S}^*$ inf-suffix per Lyndon factor. That is because an inf-suffix can only induce the order of another inf-suffix of the *same* Lyndon word. However, this is not a problem since we know that the inf-suffix starting with a Lyndon factor T_x is smaller in $\prec_\omega$-order than all other inf-suffixes of T_x, for each $x \in [1..t]$. In particular, we know that $T_x \succ_{\text{lex}} T_{x+1}$ is equivalent to $T_x \succ_\omega T_{x+1}$ due to [3, Thm. 8], and hence we know the $\prec_{\text{lex}}$-ranks among all inf-suffixes starting with the Lyndon factors.[8] In what follows, we use the inf-suffixes starting with the Lyndon factors to induce the $\prec_{\text{LMS}}$-ranks of all LMS inf-substrings.

However, the inducing only works if we include all text positions: While an ordered suffix $R[i..|R|]$ induces the order of $R[i-1..|R|]$ in the traditional SAIS, here we want an inf-suffix $R[i..]$ to induce the order of $R[i-1..]$. For that, we define a superset of the LMS inf-substrings, whose elements are called LMS-prefixes [23, Sect. 3.4]: Let $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ for an $x \in [1..t]$ be a text position, and let $j > i$ be the next $\mathtt{S}^*$ position in R. Then the *LMS-prefix* P_i starting at position i is $P_i := R[i..j]$ if $j \leq \mathsf{e}(T_x)$ or $P_i := R[i..j-1]\mathsf{b}(T_x)$

[8] Since T_t is the smallest Lyndon word, we have the invariants that $\mathsf{SA}_\circ[1] = \mathsf{b}(T_t)$ and $\mathsf{BBWT}[1] = R[\mathsf{e}(T_t)] = R[|R|]$.

if $j = \mathsf{b}(T_{x+1})$. In particular, if i is the starting position of an LMS inf-substring S, then $P_i = S$. The LMS-prefixes inherit the types (`L` or `S`) from their starting positions. We show that we can compute the $\prec_{\mathrm{LMS}}$-ranks of all P_i's by induce sorting:

Initialize the Suffix Array. We create $\mathsf{SA}_\circ$ of size $|R|$ to store the $\prec_{\mathrm{LMS}}$-ranks of all LMS-prefixes, where the entries are initially empty. Like in SAIS, we divide $\mathsf{SA}_\circ$ into buckets, and put the LMS-prefixes corresponding to the LMS inf-substrings into the `S` buckets of the respective starting characters in lexicographically sorted order. See also Fig. 10 for an example.

Inducing `L` LMS-prefixes. We scan the suffix array from left to right, and take action whenever we access a non-empty value i stored in $\mathsf{SA}_\circ$: Given $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ and $i^- = i-1$ or $i^- = \mathsf{e}(T_x)$ for $i = \mathsf{b}(T_x)$, we insert i^- into the `L` bucket of the character $T_x[i^-]$ if $R[i^-..]$ is an `L` inf-suffix. By doing so, we compute the $\prec_{\mathrm{LMS}}$ -order of all `L` LMS-prefixes in ascending lexicographic order per `L` bucket. The correctness follows by induction over the number k of inserted `L` LMS-prefixes. Since we know that all LMS-prefixes $P_{\mathsf{b}(T_x)}$ for $x \in [1..t]$ starting with the Lyndon factors are stored correctly in $\prec_{\mathrm{LMS}}$-order, and each of them is preceded by an `L` LMS-prefix, we perform the insertion of the first `L` LMS-prefix correctly, which is induced by the lexicographically smallest `S`* LMS-prefix $P_{T_t[1]}$. For the induction step, assume that there is a $k > 1$ such that when we append the $(k+1)$-th `L` LMS-prefix P_i into its corresponding bucket, we have stored an `L` LMS-prefix P_j with larger $\prec_{\mathrm{LMS}}$-rank in the same bucket. In this case, we have that $R[i] = R[j]$, $P_{j+1} \succ_{\mathrm{LMS}} P_{i+1}$ and P_{j+1} is stored to the left of P_{i+1}. This implies that when we scanned $\mathsf{SA}_\circ$ from left to right, before appending P_i to its bucket, we already did a mistake.

The inducing step for the `S` LMS-prefixes works exactly in the same way by symmetry. Finally, we scan the computed $\mathsf{SA}_\circ$, and for each pair of subsequent positions i and j with $i < j$ corresponding to the starting positions of two LMS inf-suffixes, we perform a character-wise comparison whether the LMS inf-substring starting at i is $\prec_{\mathrm{LMS}}$-smaller than the one starting at j. By doing so, we can compute the $\prec_{\mathrm{LMS}}$-ranks of all LMS inf-substrings in linear time because the number of character comparisons is bounded by the number of characters covered by all LMS inf-substrings, which is $\mathcal{O}(|R|)$. ◀

With Lemma 6, we can determine the $\prec_\omega$ -order of the `S`* inf-suffixes R. It is left to perform the induction step to induce first the order of the `L` inf-suffixes, and subsequently the `S` inf-suffixes, which we do in the same manner as SAIS, but access $(T_xT_x\cdots)[i^-]$ instead of $R[i-1]$ when accessing a suffix array entry with value i, where x chosen such that $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ and $i^- = i-1$ or $i^- = \mathsf{e}(T_x)$ if $i = \mathsf{b}(T_x)$. The correctness follows by construction: Instead of partitioning the suffixes into LMS substrings (maybe omitting a prefix of R with `L` suffixes), we refine the Lyndon factors into a partitioning of LMS inf-substrings.

Lyndon Factors of Length One. It is left to reintroduce the Lyndon factors of lengths one to obtain the complete $\mathsf{SA}_\circ$ of R. Remember that we omitted these factors at the recursive call. After the recursive call, we reinsert each of them at the smallest position in the `S` bucket of its respective starting character. By doing so, we correctly sort them due to the following observation: Suppose that there is a Lyndon factor consisting of a single character `b` (the following holds if $\mathtt{b} \in \Sigma$ or if `b` is a rank of an LMS substring considered in the recursive call). All LMS inf-substrings larger than one starting with `b` are larger than `bb` in the $\prec_\omega$-order because such an LMS inf-substring starting with $R[i]$ having type `S`* is lexicographically

smaller than $R[i+1..]$. Consequently, $\mathtt{bb}\cdots \prec_{\text{lex}} R[i..] = \mathtt{b}R[i+1..]$ since $\mathtt{b}\cdots \prec_{\text{lex}} R[i+1..]$. Thus, the Lyndon factor consisting of the single character $\mathtt{b}$ does not have to be tracked further in the recursive call since we know that its rank precedes the ranks of all other LMS inf-substrings starting with $\mathtt{b}$.

Time Complexity. By omitting Lyndon factors in the recursive calls, reducing R to a string R' where no two subsequent inf-suffixes $R[i..]$ and $R[i+1..]$ are $\mathtt{S}^*$, we can bound the maximum number of all $\mathtt{S}^*$ inf-suffixes by $n/2$ for the recursive call. After the recursion, we can simply insert all omitted LMS inf-substrings into the order returned by the recursive call by a linear scan. Hence, we obtain that $\mathcal{T}(n) = \mathcal{T}(n/2) + \mathcal{O}(n) = \mathcal{O}(n)$, where $\mathcal{T}(n)$ is the time complexity for computing a circular suffix array of length n. Note that the omission of the single character Lyndon factors is crucial for obtaining this time complexity. Without, there may be more than $n/2$ many $\mathtt{S}^*$ inf-suffixes, and because we keep the same Lyndon factorization in all recursive levels, we could have $\Theta(n)$ LMS inf-suffixes at each recursion level. The final step of computing the BBWT of T from the circular suffix array $\mathsf{SA}_\circ$ of R can be done in linear time with a linear scan of $\mathsf{SA}_\circ$ as described in Sect. 4.1.

4.5 Space Complexity

Given that $z = \sum_{x=1}^{t} \tau_x$ is the number of all non-composed Lyndon factors $F_1 \cdots F_z$, the algorithm of Lemma 1 computing the Lyndon factorization online only needs to maintain three integer variables of $\mathcal{O}(\lg n)$ bits to find $F_1 \cdots F_z$. We can represent the non-composed Lyndon factorization by a bit vector B of length n marking the ending position of each factor F_x ($x \in [1..z]$) with a one. We additionally create a bit vector B_2 of length z, and mark the first occurrence of each non-composed Lyndon factor F_x in B_2 for $x \in [1..z]$ such that B_2 stores t ones. Then the x-th '1' in B_2 corresponds to the x-th composed Lyndon factor T_x, and the number of '0's between the x-th and $(x+1)$-th '1' in B_2 is $\tau_x - 1$. It is now possible to replace T by R and store the Lyndon factorization of R in B (and resizing B to length $|R|$) since we can restore T later with B_2. (Alternatively, we can simulate R having T and B_2.) This saves at least $(z-t)\lg\sigma \geq z-t$ bits, such that our working space is at most $n+t+n\lg\sigma$ bits including the text space, before starting the actual algorithm computing $\mathsf{SA}_\circ$. Building a rank-support data structure on B helps us to identify the Lyndon factor covering a text position of R in constant time [16]. A rank-support data structure provides support for a *rank query*, i.e., retrieving the number of ones up to a queried position in B. Since a recursive call of SAIS works on a text instance of at most $|R|/2$ characters, we can rebuild B from scratch by rerunning the algorithm of Lemma 1 on $R^{(1)}$ or after finalizing the recursive call. In total, we can maintain the Lyndon factorization in $n + o(n)$ bits with $\mathcal{O}(n)$ total time throughout all recursive calls. When a recursive call ends, we need to insert the omitted Lyndon factors of length one into the list of sorted $\mathtt{S}^*$ inf-suffixes. But this can be done with a linear scan of the sorted $\mathtt{S}^*$ inf-suffixes and their initial characters, since we know that the omitted Lyndon factors have to be inserted at the first position among all inf-suffixes sharing the same initial character. Additionally, we can achieve this within the space used for storing the circular suffix array $\mathsf{SA}_\circ$, since all $\mathtt{S}^*$ inf-suffixes use up at most half of the positions of the inf-suffix array. Overall, we have an algorithm running with $n + t + o(n)$ bits on top of our modified SAIS, which uses $\mathcal{O}(\sigma \lg n)$ bits of working space additionally to $\mathsf{SA}_\circ$. If σ is not constant, one may consider an option to get rid of this additional space requirement. Luckily, we can do so with the in-place suffix array construction algorithm of Goto [13] (or similarly with [20]), which is a variation of SAIS, storing an implicit representation of these $\mathcal{O}(\sigma \lg n)$ bits within the space of $\mathsf{SA}_\circ$. Since B_2 is only needed for the final step computing the BBWT

of T, we can compute $\mathsf{SA}_\circ$ with $n + o(n)$ additional bits of working space, and BBWT with $|\mathsf{SA}_\circ| + n + t + o(n)$ additional bits of working space, where $|\mathsf{SA}_\circ| = n \lg n$ denotes the size of $\mathsf{SA}_\circ$ in bits.

5 Conclusion

We proposed an algorithm computing the bijective Burrows–Wheeler transform (BBWT) in linear time. Consequently, we can also compute the extended Burrows–Wheeler transform (eBWT) within the same time bounds by a linear-time reduction of the problem to compute the eBWT to computing the BBWT.

Our trick was to first reduce our input text T to a text R by removing all duplicate Lyndon factors. Second, we slightly modified the suffix array – induce sorting (SAIS) algorithm to compute the $\prec_\omega$-order of the conjugates of all Lyndon factors of R instead of the $\prec_{\text{lex}}$-order of all suffixes of R. For that, we introduced the notion of inf-suffixes and inf-substrings, and adapted the typing system of `L`, `S`, and `S`* types from SAIS. By some properties of the Lyndon factors, we could show that there are only some border cases, where a text position receives a different type in our modification. Thanks to that, we could directly translate the induce sorting techniques of SAIS, and obtain the correctness of our result.

Open Problems

The BBWT is bijective in the sense that it transforms a string of Σ^n into another string of Σ^n while preserving distinctness. Consequently, given a string of length n, there is an integer $k \geq 1$ with $\mathsf{BBWT}^k(T) = \mathsf{BBWT}^{k-1}(\mathsf{BBWT}(T)) = T$. With our presented algorithm we can compute the smallest such number k in $\mathcal{O}(nk)$ time. However, we wonder whether we can compute this number faster, possible by scanning only the text in $\mathcal{O}(n)$ time independent of k.

We also wonder whether we can define the BBWT for the generalized Lyndon factorization [6]. Contrary to the Lyndon factorization, the generalized Lyndon factorization uses a different order, called the *generalized lexicographic order* $\prec_{\text{gen}}$. In this order, two strings $S, T \in \Sigma^*$ are compared character-wise like in the lexicographic order. However, the generalized lexicographic order $\prec_{\text{gen}}$ can use different orders $<_1, <_2, \ldots$ for each text position, i.e., $S \prec_{\text{gen}} T$ if and only if S is a proper prefix of T or there is an integer ℓ with $1 \leq \ell \leq \min(|S|, |T|)$ such that $S[1..\ell-1] = T[1..\ell-1]$ and $S[\ell] <_\ell T[\ell]$.

Recently, Gibney and Thankachan [11] showed that finding an order of the alphabet such that the number of Lyndon factors of a given string is minimized or maximized is NP-complete. This is an important but negative result for finding an advantage of the BBWT over the BWT, since the hope is to find a way to increase the number of Lyndon factors and therefore the chances of having multiple equal factors that are contracted to a single composed factor in the BBWT index of [2]. However, it is left open, whether we can find an efficient algorithm that approximates the alphabet order maximizing the number of Lyndon factor.

Another direction would be to find a string family for which we $\mathsf{SA}_\circ$ and SA differ, for instance, with a relatively high Hamming distance.

References

1 Donald Adjeroh, Timothy Bell, and Amar Mukherjee. *The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching.* Springer, 2008.

2 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Indexing the bijective BWT. In *Proc. CPM*, volume 128 of *LIPIcs*, pages 17:1–17:14, 2019.

3 Silvia Bonomo, Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting conjugates and suffixes of words in a multiset. *Int. J. Found. Comput. Sci.*, 25(8):1161, 2014.

4 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

5 Kuo Tsai Chen, Ralph H. Fox, and Roger C. Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Annals of Mathematics*, pages 81–95, 1958.

6 Francesco Dolce, Antonio Restivo, and Christophe Reutenauer. On generalized Lyndon words. *Theor. Comput. Sci.*, 777:232–242, 2019.

7 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

8 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.

9 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

10 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018.

11 Daniel Gibney and Sharma V. Thankachan. Finding an optimal alphabet ordering for Lyndon factorization is hard. In *Proc. STACS*, volume 187 of *LIPIcs*, pages 35:1–35:15, 2021.

12 Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *ArXiv 1201.3077*, 2012. `arXiv:1201.3077`.

13 Keisuke Goto. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In *Proc. PSC*, pages 111–125, 2019.

14 Wing-Kai Hon, Tsung-Han Ku, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Efficient algorithm for circular Burrows–Wheeler transform. In *Proc. CPM*, volume 7354 of *LNCS*, pages 257–268, 2012.

15 Wing-Kai Hon, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Succinct indexes for circular patterns. In *Proc. ISAAC*, volume 7074 of *LNCS*, pages 673–682, 2011.

16 Guy Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989.

17 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

18 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.

19 Manfred Kufleitner. On bijective variants of the Burrows–Wheeler transform. In *Proc. PSC*, pages 65–79, 2009.

20 Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. In *Proc. SPIRE*, volume 11147 of *LNCS*, pages 268–284, 2018.

21 R. C. Lyndon. On Burnside's problem. *Transactions of the American Mathematical Society*, 77(2):202–215, 1954.

22 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.

23 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

Weighted Ancestors in Suffix Trees Revisited

Djamal Belazzougui
Centre de Recherche sur l'Information Scientifique et Technique, Algiers, Algeria

Dmitry Kosolobov
Institute of Natural Sciences and Mathematics, Ural Federal University, Ekaterinburg, Russia

Simon J. Puglisi
Department of Computer Science, University of Helsinki, Helsinki, Finland

Rajeev Raman
Department of Informatics, University of Leicester, Leicester, United Kingdom

Abstract

The weighted ancestor problem is a well-known generalization of the predecessor problem to trees. It is known to require $\Omega(\log\log n)$ time for queries provided $\mathcal{O}(n \operatorname{polylog} n)$ space is available and weights are from $[0..n]$, where n is the number of tree nodes. However, when applied to suffix trees, the problem, surprisingly, admits an $\mathcal{O}(n)$-space solution with constant query time, as was shown by Gawrychowski, Lewenstein, and Nicholson (Proc. ESA 2014). This variant of the problem can be reformulated as follows: given the suffix tree of a string s, we need a data structure that can locate in the tree any substring $s[p..q]$ of s in $\mathcal{O}(1)$ time (as if one descended from the root reading $s[p..q]$ along the way). Unfortunately, the data structure of Gawrychowski et al. has no efficient construction algorithm, limiting its wider usage as an algorithmic tool. In this paper we resolve this issue, describing a data structure for weighted ancestors in suffix trees with constant query time and a linear construction algorithm. Our solution is based on a novel approach using so-called irreducible LCP values.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases suffix tree, weighted ancestors, irreducible LCP, deterministic substring hashing

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.8

Funding *Dmitry Kosolobov*: Supported by the grant 20-71-00050 of the Russian Foundation of Basic Research (for the final version of the data structure and for its construction algorithm).

1 Introduction

The suffix tree is one of the central data structures in stringology. Its primary application is to determine whether an arbitrary string t occurs as a substring of another string, s, which can be done in time proportional to the length of t by traversing the suffix tree of s downward from the root and reading off the symbols of t along the way. Many algorithms using suffix trees perform this procedure for substrings $t = s[p..q]$ of s itself. In this important special case, the traversal can be executed much faster than $\mathcal{O}(q - p + 1)$ time provided the tree has been preprocessed to build some additional data structures [1, 11, 13]; particularly surprising is that the traversal can be performed in constant time using only linear space, as shown by Gawrychowski et al. [13]. In this paper, we describe the first linear construction algorithm for such a data structure. The lack of an efficient construction algorithm for the result of [13] has been, apparently, the main obstacle hindering its wider adoption. Our solution is completely different from that of [13] and is based on a combinatorial result of Kärkkäinen et al. [17] (see also [16]) that estimates the sum of irreducible LCP values (precise definitions follow).

As one might expect, the described data structure has a multitude of applications: [1, 7, 8, 11, 19, 20, 21], to name a few. We, however, do not dive further into a discussion of these applications and refer the reader to the overview in [13, Sect. 1] and references therein for more details.

In order to "locate" a substring $t = s[p..q]$ in the suffix tree of s, it suffices to answer the following query: given a node of the tree that corresponds to the suffix of s starting at position p (usually, it is a leaf), we should find the farthest (closest to the root) ancestor v of this node such that the string written on the root–v path has length at least $q - p + 1$. This problem is a particular case of the general *weighted ancestor problem* [1, 11, 22]: given a tree whose nodes are associated with integer weights such that the weights decrease if one ascends from any node to the root (the weight of a node in the suffix tree is the length of the string written on the root–node path), the tree should be preprocessed in order to answer *weighted ancestor queries* that, for a given node v and a number w, return the farthest ancestor of v whose weight is at least w. The problem can be viewed as a generalization to trees of the predecessor search problem, in which we preprocess a set of integers to support predecessor queries: for any given number w, return the largest integer from the set that precedes w.

Clearly, any linear-space solution for weighted ancestors can be used as a solution for predecessor search. As was shown in [22] and [13], a certain converse reduction is also possible: the weighted ancestor queries for a tree with n nodes and integer weights from a range $[0..U]$ can be answered in $\mathcal{O}(\mathrm{pred}(n, U))$ time using linear space, where $\mathrm{pred}(n, U)$ is the time required to answer predecessor search queries for any set of $k \leq n$ integers from the range $[0..U]$ using $\mathcal{O}(k)$ space. Therefore, when $U = n$ (as in the case of suffix tree), both problems can be solved in linear space with $\mathcal{O}(\log\log n)$-time queries using the standard van Emde Boas or y-fast trie data structures [30, 31].

Due to the lower bound of Pătraşcu and Thorup [25], the time $\mathcal{O}(\log\log n)$ is optimal for $U = n$ when the available space is linear, and moreover, any solution of the weighted ancestor problem that uses $\mathcal{O}(n \operatorname{polylog} n)$ space must spend $\Omega(\log\log n)$ time on queries (see [13, Appendix A]). In view of this lower bound, it is all the more unexpected that the special case of suffix trees admits an $\mathcal{O}(n)$-space solution with constant query time. In order to circumvent the lower bound, Gawrychowski et al. [13] solve predecessor search problems on some paths of the suffix tree using $\mathcal{O}(n)$ *bits* of space (or slightly less), which admits a constant time solution by a so-called rank data structure; because of the internal repetitive structure of the suffix tree, the solution for one path can be reused in many different paths in such a way that, in total, the utilized space is linear. Our approach essentially relies on the same intuition but we perform path predecessor queries on different trees closely related to the suffix tree and the advantages of repetitive structures come implicitly; in particular, we do not explicitly treat periodic regions of the string separately so that, in this regard, our solution is more "uniform", in a sense, than that of [13].

This paper is organized as follows. In Section 2 the problem is reduced to certain path-counting queries using (unweighted) level ancestor queries. Section 3 describes a simple solution for the queries locating substrings $s[p..q]$ whose position p corresponds to an irreducible LCP value. In Section 4 we reduce general queries to the queries at irreducible positions and a certain geometric orthogonal predecessor problem. Section 5 describes a solution for this special geometric problem. Conclusions and reflections are then offered in Section 6.

2 Basic Data Structures

Let us fix a string s of length n, denoting its letters by $s[0], s[1], \ldots, s[n-1]$. We write $s[p..q]$ for the *substring* $s[p]s[p+1]\cdots s[q]$, assuming it is empty if $p > q$; $s[p..q]$ is called a *suffix* (resp., *prefix*) of s if $q = n-1$ (resp., $p = 0$). For any string t, let $|t|$ denote its length. We say that t *occurs* at position p in s if $s[p..p+|t|-1] = t$. Denote $[p..q] = \{k \in \mathbb{Z} \colon p \le k \le q\}$.

A *suffix tree* of s is a compacted trie containing all suffixes of s. The labels on the edges of the tree are stored as pointers to corresponding substrings of s. For each tree node v, denote by $\mathsf{str}(v)$ the string written on the root–v path. The number $|\mathsf{str}(v)|$ is called the *string depth* of v. The *locus* of a substring $s[p..q]$ is the (unique) node v of minimal string depth such that $s[p..q]$ is a prefix of $\mathsf{str}(v)$.

The string s and its suffix tree T are the input to our algorithm. To simplify the exposition, we assume that $s[n-1]$ is equal to a special letter \$ that is smaller than all other letters in s, so that there is a one-to-one correspondence between the suffixes of s and the leaves of T. Our computational model is the word-RAM and space is measured in $\Theta(\log n)$-bit machine words. Our goal is to construct in $\mathcal{O}(n)$ time a data structure that can find in the tree the locus of any substring $s[p..q]$ of s in $\mathcal{O}(1)$ time.

A *level ancestor query* in a tree asks, for a given node v and an integer $d \ge 1$, the dth node on the v–root path (provided the path has at least d nodes). It is known that any tree can be preprocessed in linear time to answer such queries in constant time [5, 6]. With such a structure, the locus of a substring $s[p..q]$ can be found by first locating the leaf v of T corresponding to $s[p..n-1]$ (i.e., $\mathsf{str}(v) = s[p..n-1]$, which can be located using a precomputed array of length n) and, then, counting the number d of nodes u on the v–root path such that $|\mathsf{str}(u)| > q - p$; then, evidently, the locus of $s[p..q]$ is given by the level ancestor query on v and d.

We have to complicate this scheme slightly since the machinery that we develop in the sequel allows us to count only those nodes u on the v–root path that have a branch to the "left" of the path (or, symmetrically, to the "right"). More formally, given a node v, a node u on the v–root path is called *left-branching* (resp., *right-branching*) if there is a suffix $s[p..n-1]$ that is lexicographically smaller (resp., greater) than the string $\mathsf{str}(v)$ and its longest common prefix with $\mathsf{str}(v)$ is $\mathsf{str}(u)$. For instance, the path from the leaf corresponding to the string *sippi*\$ in Figure 1 has four nodes but only one of them (namely, the root) is left-branching.

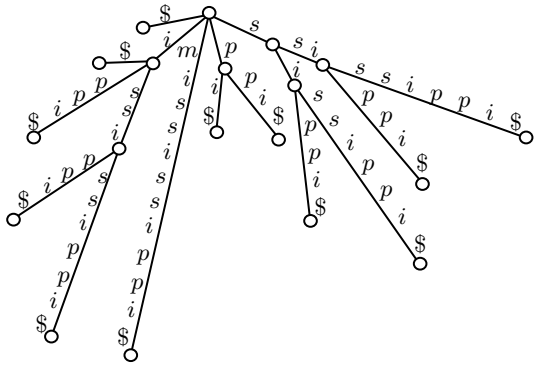
▶ **Lemma 1.** *For any suffix tree, one can build in linear time a data structure that, for any node v and integer $d \ge 1$, can return in $\mathcal{O}(1)$ time the dth left-branching (or right-branching) node on the v–root path.*

Proof. Traversing the suffix tree, we construct another tree on the same set of nodes in which the parent of each non-root node is either its nearest left-branching ancestor in the suffix tree (if any) or the root. The queries for left-branching nodes can be answered by the level ancestor structure [5, 6] built on this new tree. The right-branching case is symmetric. ◀

Thus, to find the locus for $s[p..q]$, we have to count on a leaf–root path the number of left- and right-branching nodes whose string depths are greater than the threshold $q - p$; we then use these two numbers in the data structure of Lemma 1 in order to find two candidate nodes and the node with smaller string depth is the locus. In the remaining text, we focus only on this counting problem and only on left-branching nodes as the right-branching case is symmetric. First, however, we define a number of useful standard structures.

The *suffix array* of s is an array $\mathsf{sa}[0..n-1]$ containing integers from 0 to $n-1$ such that $s[\mathsf{sa}[0]..n-1] < s[\mathsf{sa}[1]..n-1] < \cdots < s[\mathsf{sa}[n-1]..n-1]$ lexicographically [23]. The *inverse suffix array*, denoted $\mathsf{isa}[0..n-1]$, is defined as $\mathsf{sa}[\mathsf{isa}[p]] = p$, for all $p \in [0..n-1]$. For any

positions p and q, denote by $\mathsf{lcp}(p,q)$ the length of the longest common prefix of $s[p..n{-}1]$ and $s[q..n{-}1]$. The *longest common prefix (LCP) array* is an array $\mathsf{lcp}[0..n{-}1]$ such that $\mathsf{lcp}[0] = 0$ and $\mathsf{lcp}[i] = \mathsf{lcp}(\mathsf{sa}[i{-}1], \mathsf{sa}[i])$, for $i \in [1..n{-}1]$. The *permuted longest common prefix (PLCP) array* is an array $\mathsf{plcp}[0..n{-}1]$ such that $\mathsf{plcp}[p] = \mathsf{lcp}[\mathsf{isa}[p]]$, for $p \in [0..n{-}1]$. The *Burrows–Wheeler transform* [9] is a string $\mathsf{bwt}[0..n{-}1]$ such that $\mathsf{bwt}[i] = s[\mathsf{sa}[i]{-}1]$ if $\mathsf{sa}[i] \neq 0$, and $\mathsf{bwt}[i] = s[n{-}1]$ otherwise. All the arrays sa, isa, lcp, plcp and the string bwt can be built from T in $\mathcal{O}(n)$ time. Some of these structures are depicted in Figure 1.



i	lcp	sa	bwt	sorted suffixes
0	**0**	11	i	$
1	**0**	10	p	i
2	**1**	7	s	$ippi$
3	1	4	s	$issippi$
4	**4**	1	m	$ississippi$
5	**0**	0	$	$mississippi$
6	**0**	9	p	pi
7	**1**	8	i	ppi
8	**0**	6	s	$sippi$
9	2	3	s	$sissippi$
10	**1**	5	i	$ssippi$
11	3	2	i	$ssissippi$

Figure 1 The suffix tree T, lcp, sa, and bwt of the string $s = mississippi\$$. All irreducible LCP values are in bold and their sum is 7; all positions of s except $2, 3, 4$ are irreducible.

3 Queries at Irreducible Positions

An LCP value $\mathsf{lcp}[i]$ is called *irreducible* if either $i = 0$ or $\mathsf{bwt}[i{-}1] \neq \mathsf{bwt}[i]$. In other words, the irreducible LCP values are those values of $\mathsf{lcp}[0..n{-}1]$ that correspond to the first positions in the runs of bwt (e.g., all values $\mathsf{lcp}[i]$ except for $i = 3, 9, 11$ in Figure 1). The central combinatorial fact that is at the core of our construction is the following surprising lemma proved by Kärkkäinen et al. [17].

▶ **Lemma 2** (see [17]). *The sum of all irreducible LCP values is at most* $2n \log n$.

We say that a position p of the string s is *irreducible* if $\mathsf{lcp}[\mathsf{isa}[p]]$ (=$\mathsf{plcp}[p]$) is an irreducible LCP value.

Consider a query that asks to find the locus of a substring $s[p..q]$ such that p is an irreducible position. Denote $\ell = q - p + 1$. We first locate in $\mathcal{O}(1)$ time the leaf v corresponding to $s[p..n{-}1]$ using a precomputed array. As was discussed above, the problem is reduced to the counting of left-branching nodes u on the v–root path such that $|\mathsf{str}(u)| \geq \ell$. We will construct for p a bit array $b_p[0..\ell_p{-}1]$ of length $\ell_p = \mathsf{plcp}[p]$ such that, for any d, $b_p[d] = 1$ iff there is a left-branching node with string depth d on the v–root path. Since the suffix $s[p..n{-}1]$ and its lexicographical predecessor among all suffixes (if any, i.e., if $\mathsf{isa}[p] \neq 0$) have the longest common prefix of length $\mathsf{plcp}[p]$, the lowest left-branching node on the v–root path has string depth $\mathsf{plcp}[p]$. Therefore, the number of left-branching nodes u on the v–root path such that $|\mathsf{str}(u)| \geq \ell$ is equal to the number of ones in the subarray $b_p[\ell..\ell_p{-}1]$ plus 1. The bit counting query is answered using the following *rank data structure*.

▶ **Lemma 3** (see [10, 15]). *For any bit array $b[0..m{-}1]$ packed into $\mathcal{O}(m/w)$ w-bit machine words, one can construct in $\mathcal{O}(m/w)$ time a rank data structure that can count the number of ones in any range $b[p..q]$ in $\mathcal{O}(1)$ time, provided a table computable in $o(2^w)$ time and independent of the array has been precomputed.*

By Lemma 2, the total length of all arrays b_p for all irreducible positions p of s is $\mathcal{O}(n \log n)$. Therefore, one can construct the rank data structures of Lemma 3 for them in $\mathcal{O}(n)$ overall time provided $w = \lfloor \log n \rfloor$. It remains to build the arrays b_p themselves.

To this end, we traverse the suffix tree T in depth first (i.e., lexicographical) order, maintaining along the way a bit array b of length n such that, when we are in a node v, the subarray $b[0..|\mathsf{str}(v)|-1]$ marks the string depths of all left-branching ancestors of v, i.e., for $0 \le d < |\mathsf{str}(v)|$, we have $b[d] = 1$ iff there is a left-branching node with string depth d on the v–root path. The array is stored in a packed form in $\lfloor \log n \rfloor$-bit machine words. For each node v, we consider its outgoing edges. Each pair of adjacent edges (in the lexicographical order of their labels) corresponds to a unique value $\mathsf{lcp}[i]$ such that $s[\mathsf{sa}[i]..n-1]$ is the suffix corresponding to the leftmost (lexicographically smallest) leaf of the subtree connected to the larger one of the two edge labels. We check whether the value $\mathsf{lcp}[i]$ is irreducible comparing $\mathsf{bwt}[i-1]$ and $\mathsf{bwt}[i]$ and, if so, store the subarray $b[0..|\mathsf{str}(v)|-1]$ into $b_{\mathsf{sa}[i]}$ in $\mathcal{O}(1 + |\mathsf{str}(v)|/\log n)$ time. As we touch in this way every value $\mathsf{lcp}[i]$ only once, the overall time is $\mathcal{O}(n)$ by the same argument using Lemma 2.

4 Reduction to Irreducible Positions

Consider a query for the locus of a substring $s[p..q]$ such that the position p is not irreducible. Let v be the leaf corresponding to $s[p..n-1]$. As was discussed in Section 2, to answer the query, we have to count the number of left-branching nodes with string depths at least $q-p+1$ on the v–root path. As a first approach, we do the following precalculations for this.

We build in $\mathcal{O}(n)$ time on the suffix tree T a data structure that allows us to find the lowest common ancestor for any pair of nodes in $\mathcal{O}(1)$ time [4, 14]. In one tree traversal, we precompute in each node u the number of left-branching nodes on the u–root path. We create two arrays $R_{\le}[0..n-1]$ and $R_{\ge}[0..n-1]$ such that, for any i, $R_{\le}[i] = \max\{j \le i \colon j = 0 \text{ or } \mathsf{bwt}[j-1] \ne \mathsf{bwt}[j]\}$ and $R_{\ge}[i] = \min\{j \ge i \colon j = 0 \text{ or } \mathsf{bwt}[j-1] \ne \mathsf{bwt}[j]\}$ ($R_{\ge}[i]$ is undefined if there is no such j), i.e., $R_{\le}[i]$ and $R_{\ge}[i]$ are indexes of irreducible LCP values, respectively, preceding and succeeding $\mathsf{lcp}[i]$. Thus, for any suffix $s[p..n-1]$, the suffixes $s[\mathsf{sa}[R_{\le}[\mathsf{isa}[p]]]..n-1]$ and $s[\mathsf{sa}[R_{\ge}[\mathsf{isa}[p]]]..n-1]$ are, respectively, the closest lexicographical predecessor and successor of $s[p..n-1]$ with irreducible starting position. The *closest irreducible lexicographical neighbour of* $s[p..n-1]$ is that suffix $s[r..n-1]$ among these two that has longer common prefix with $s[p..n-1]$, i.e., $r = \mathsf{sa}[R_{\le}[\mathsf{isa}[p]]]$ if either $\mathsf{lcp}(\mathsf{sa}[R_{\le}[\mathsf{isa}[p]]], p) \ge \mathsf{lcp}(\mathsf{sa}[R_{\ge}[\mathsf{isa}[p]]], p)$ or $R_{\ge}[\mathsf{isa}[p]]$ is undefined, and $r = \mathsf{sa}[R_{\ge}[\mathsf{isa}[p]]]$ otherwise. Since $\mathsf{lcp}(t, p)$ can be calculated in $\mathcal{O}(1)$ time for any positions t and p by one lowest common ancestor query on their corresponding leaves, the closest irreducible lexicographical neighbour can be computed in $\mathcal{O}(1)$ time.

Now, consider a query for the locus of $s[p..q]$ with non-irreducible p. We first find in $\mathcal{O}(1)$ time the closest irreducible lexicographical neighbour $s[r..n-1]$ for $s[p..n-1]$. Let v' and v be the leaves corresponding to $s[r..n-1]$ and $s[p..n-1]$, respectively. We compute in $\mathcal{O}(1)$ time the lowest common ancestor u of v' and v. Thus, $|\mathsf{str}(u)| = \mathsf{lcp}(r, p)$. Using the number of left-branching nodes on the v–root and u–root paths that were precomputed in v and u, we calculate in $\mathcal{O}(1)$ time the number k of left-branching nodes on the v–root path that lie between the nodes v and u (inclusively).

Denote $\ell = q - p + 1$. If $\ell \le |\mathsf{str}(u)|$, then we can count in the v–root path the number of left-branching nodes w such that $|\mathsf{str}(w)| \ge \ell$ as follows. The number of nodes w such that $|\mathsf{str}(w)| \ge |\mathsf{str}(u)|$ is equal to k. The number of nodes w such that $\ell \le |\mathsf{str}(w)| < |\mathsf{str}(u)|$ can be found by counting the number of ones in the subarray $b_r[\ell..|\mathsf{str}(u)|-1]$ of the bit

array b_r associated with the irreducible position r (assuming that all values $b_r[t]$ with $t \geq \ell_r$ are zeros in case the length ℓ_r of b_r is less than $|\mathsf{str}(u)|$), which can be performed in $\mathcal{O}(1)$ time by Lemma 3. Thus, the problem is solved for the case $\ell \leq |\mathsf{str}(u)|$.

To address the case $\ell > |\mathsf{str}(u)|$, we have to develop more sophisticated techniques that allow us to reduce the counting of left-branching nodes on the v–root path to counting on a different leaf–root path with a different threshold ℓ' that meets the condition $\ell' \leq |\mathsf{str}(u')|$, for an analogously appropriately defined node u'. The remainder of the text describes the reduction.

Similar to irreducible positions, let us define, for each non-irreducible position p, a number $\ell_p = \mathsf{plcp}[p]$ and an array $b_p[0..\ell_p{-}1]$ such that, for any d, $b_p[d] = 1$ iff there is a left-branching node of string depth d on the path from the leaf corresponding to $s[p..n{-}1]$ to the root. We do not actually store the arrays b_p and use them only in the analysis.

▶ **Lemma 4.** *For any non-irreducible position p, we have $\ell_{p-1} = \ell_p + 1$ and, if $b_{p-1}[d{+}1] = 1$ for some $d \geq 0$, then $b_p[d] = 1$.*

Proof. Let $s[t..n{-}1]$ be the suffix lexicographically preceding $s[p..n{-}1]$, i.e., $t = \mathsf{sa}[\mathsf{isa}[p]{-}1]$ and $\ell_p = \mathsf{lcp}(t, p)$. Since p is not irreducible, we have $\mathsf{bwt}[\mathsf{isa}[t]] = \mathsf{bwt}[\mathsf{isa}[p]]$, i.e., $s[t{-}1] = s[p{-}1]$. Hence, $s[t{-}1..n{-}1]$ is the suffix lexicographically preceding $s[p{-}1..n{-}1]$: $\mathsf{isa}[t{-}1] = \mathsf{isa}[p{-}1] - 1$. Therefore, $\ell_{p-1} = \mathsf{lcp}(t-1, p-1)$, which is equal to $\mathsf{lcp}(t, p) + 1 = \ell_p + 1$.

If $b_{p-1}[d{+}1] = 1$ for $d \geq 0$, then there is a suffix $s[r..n{-}1]$ that is smaller than $s[p{-}1..n{-}1]$ and $\mathsf{lcp}(r, p-1) = d + 1$. Then, $\mathsf{lcp}(r+1, p) = d$, which implies that $b_p[d] = 1$. ◀

Consider two consecutive irreducible positions p' and p'' in s (i.e., all positions r between p' and p'' are not irreducible). Lemma 4 states that the arrays $b_{p'}, b_{p'+1}, \ldots, b_{p''-1}$ form a trapezoidal structure as depicted in Figure 2 in which each 1 value is inherited by all arrays above it while it fits in their range.

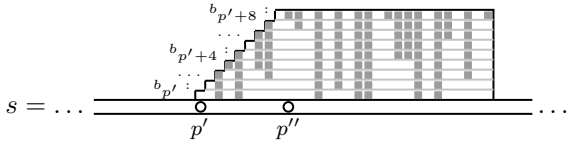


Figure 2 Here, p' and p'' are consecutive irreducible positions. Each line of the trapezoid depicts an array $b_p[0..\ell_p{-}1]$, for $p \in [p'..p''{-}1]$, in which gray and white positions signify, respectively, ones and zeros.

The query for the locus of $s[p..q]$ was essentially reduced to the counting of ones in the subarray $b_p[\ell..\ell_p{-}1]$, where $\ell = q - p + 1$. The problem now is that the array b_p is not stored explicitly since the position p is not irreducible. Denote the length of $b_p[\ell..\ell_p{-}1]$ by $m = \ell_p{-}\ell$, so that $b_p[\ell..\ell_p{-}1] = b_p[\ell_p{-}m..\ell_p{-}1]$, which is a more convenient notation for what follows. Let p' be the closest irreducible position preceding p, i.e., $p' = \max\{r \leq p\colon r \text{ is irreducible}\}$. If we are lucky and neither of the arrays $b_{p'+1}, b_{p'+2}, \ldots, b_p$ introduces new 1 values in the subarray $b_{p'}[\ell_{p'}{-}m..\ell_{p'}{-}1]$ (in other words if $b_{p'}[\ell_{p'}{-}m..\ell_{p'}{-}1] = b_p[\ell_p{-}m..\ell_p{-}1]$), then we can simply count the number of ones in $b_{p'}[\ell_{p'}{-}m..\ell_{p'}{-}1]$ in $\mathcal{O}(1)$ time using Lemma 3 and, thus, solve the problem. Unfortunately, new 1 values indeed could be introduced. But, as it turns out, such new values are, in a sense, "covered" by other arrays b_r at some irreducible positions r as the following lemma suggests.

▶ **Lemma 5.** *Let p be a non-irreducible position and $s[r..n{-}1]$ be the closest irreducible lexicographical neighbour of $s[p..n{-}1]$. Denote $c_p = \mathsf{lcp}(r,p)$. If, for some d, we have $b_p[d] = 1$ but $b_{p-1}[d{+}1] = 0$, then $d \le c_p$.*

Proof. The condition $b_p[d] = 1$ implies that there is a suffix $s[t..n{-}1]$ that is smaller than $s[p..n{-}1]$ and $\mathsf{lcp}(t,p) = d$. If $\mathsf{bwt}[\mathsf{isa}[t]] = \mathsf{bwt}[\mathsf{isa}[p]]$, then $s[t{-}1] = s[p{-}1]$ and, hence, the suffix $s[t{-}1..n{-}1]$ is smaller than $s[p{-}1..n{-}1]$ and $\mathsf{lcp}(t-1,p-1) = d+1$. This gives $b_{p-1}[d{+}1] = 1$, which contradicts the equality $b_{p-1}[d{+}1] = 0$. Hence, $\mathsf{bwt}[\mathsf{isa}[t]] \neq \mathsf{bwt}[\mathsf{isa}[p]]$. Thus, at least one of the values $\mathsf{lcp}[\mathsf{isa}[t]{+}1], \mathsf{lcp}[\mathsf{isa}[t]{+}2], \ldots, \mathsf{lcp}[\mathsf{isa}[p]{-}1]$ must be irreducible (note that $\mathsf{isa}[t]+1 < \mathsf{isa}[p]$ since otherwise p would be irreducible). Let r' be an irreducible position such that $\mathsf{isa}[t] < \mathsf{isa}[r'] < \mathsf{isa}[p]$. We obtain $\mathsf{lcp}(r,p) \ge d$ since $\mathsf{lcp}(r',p) \ge \mathsf{lcp}(t,p) = d$ and, for any irreducible r'' (in particular, for r'), we have $\mathsf{lcp}(r,p) \ge \mathsf{lcp}(r'',p)$. ◀

Observe that c_p in Lemma 5 is equal to $|\mathsf{str}(u)|$, where u is the lowest common ancestor of the leaves v' and v corresponding to $s[r..n{-}1]$ and $s[p..n{-}1]$. Thus, the numbers c_p can be precomputed for all non-irreducible positions p in $\mathcal{O}(n)$ time using lowest common ancestor queries and the arrays $R_{\le}$ and $R_{\ge}$ described in the beginning of the present section. For irreducible positions p', we put $c_{p'} = \ell_{p'}$ by definition. To illustrate Lemma 5 and its consequences, we depict in Figure 3 the same trapezoidal structure as in Figure 2 coloring in each b_p a prefix of length $c_p + 1$ (and also depicting the range $b_p[\ell_p{-}m..\ell_p{-}1]$).

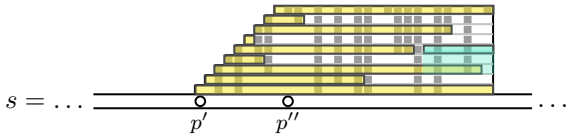


Figure 3 The yellow stripe in each array b_p is a prefix of length $\min\{c_p + 1, \ell_p\}$. Note that, according to Lemma 5, each gray position whose corresponding position below is white is covered in yellow. The blue stripe is $b_p[\ell_p{-}m..\ell_p{-}1]$, where $p = p' + 4$. It coincides with two corresponding regions below it that are in light blue. Here we have $t = p - 2$.

The problematic condition $\ell > |\mathsf{str}(u)|$ that we consider can be reformulated as $\ell_p{-}m > c_p$. It follows from Lemmas 4 and 5 that in this case $b_p[\ell_p{-}m..\ell_p{-}1] = b_{p-1}[\ell_{p-1}{-}m..\ell_{p-1}{-}1]$. Let t be the first position preceding p such that $\ell_t - m \le c_t$. Note that $t \ge p'$ since $c_{p'} = \ell_{p'}$ by definition and, thus, $\ell_{p'} - m \le c_{p'}$. Then, applying Lemma 5 consecutively to all positions $p, p-1, \ldots, t+1$, we conclude that $b_t[\ell_t{-}m..\ell_t{-}1] = b_p[\ell_p{-}m..\ell_p{-}1]$.

Informally, in terms of Figure 3, the searching of t corresponds to the moving of the range $b_p[\ell_p{-}m..\ell_p{-}1]$ down until we encounter an "obstacle", a colored part of b_t of length $c_t + 1$. The specific algorithm finding t is discussed in Section 5; let us assume for the time being that t is already known. Then, Lemma 5 implies that all the ranges $b_r[\ell_r{-}m..\ell_r{-}1]$ with $r \in [t..p]$ coincide and, thus, the whole problem was reduced to the counting of ones in the subarray $b_t[\ell_t{-}m..\ell_t{-}1]$. But since $\ell_t - m \le c_t$, the problem can be solved by the method described at the beginning of the section: we find an irreducible position r such that $\mathsf{lcp}(r,t) = c_t$, then locate the leaves v' and v'' corresponding to $s[r..n{-}1]$ and $s[t..n{-}1]$, respectively, find the lowest common ancestor u' of v' and v'', and separately count the number of left-branching nodes w on the v''–root path such that $|\mathsf{str}(w)| \ge c_t$ and such that $\ell_t - m \le |\mathsf{str}(w)| < c_t$.

Let us briefly recap the reductions described above: the searching for the locus of $s[p..q]$ was essentially reduced to the counting of ones in the subarray $b_p[\ell_p{-}m..\ell_p{-}1]$, where $m = \ell_p{-}(q{-}p{+}1)$, for which we first compute the position $t = \max\{t \le p \colon \ell_t - m \le c_t\}$ (this

is discussed in Section 5) and, then, count the number of ones in the subarray $b_t[\ell_t-m..\ell_t-1]$ (the subarray coincides with $b_p[\ell_p-m..\ell_p-1]$ by Lemmas 4 and 5) using lowest common ancestor queries, the arrays $R_{\leq}$ and $R_{\geq}$, and some precomputed numbers in the nodes.

5 Reduction to Special Weighted Ancestors

We are to compute $t = \max\{t \leq p\colon \ell_t - m \leq c_t\}$, for a non-irreducible position p. As is seen in Figure 3, this is a kind of geometric "orthogonal range predecessor" problem.

For each irreducible position p' in s, we build a tree $I_{p'}$ with $p''-p'$ nodes, where p'' is the next irreducible position after p' (so that all r with $p' < r < p''$ are not irreducible): Each node of $I_{p'}$ is associated with a unique position from $[p'..p''-1]$ and the root is associated with p'. A node associated with position $r \neq p'$ has weight $w_r = r - p' + c_r$; the root has weight $w_{p'} = +\infty$. The parent of a non-root node associated with r is the node associated with the position $r' = \max\{r' < r\colon w_{r'} > w_r\}$. Thus, the weights strictly increase as one ascends to the root. The tree $I_{p'}$ is easier to explain in terms of the trapezoidal structure drawn in Figure 4: the parent of a node associated with r is its closest predecessor r' whose colored range $b_{r'}[r'..r'+c_{r'}-1]$ goes at least one position farther to the right than $b_r[r..r+c_r-1]$. By Lemma 4, $r - p' + \ell_r = \ell_{p'}$ and, hence, each weight w_r such that $c_r \leq \ell_r$ is upperbounded by $\ell_{p'}$. However, it may happen that $c_r > \ell_r$ and, thus, $w_r > \ell_{p'}$. In this case, the colored range $b_r[r..r+c_r-1]$ stretches beyond the length ℓ_r of b_r (for simplicity, Figure 4 does not have such cases). Such large weights are a source of technical complications in our scheme.

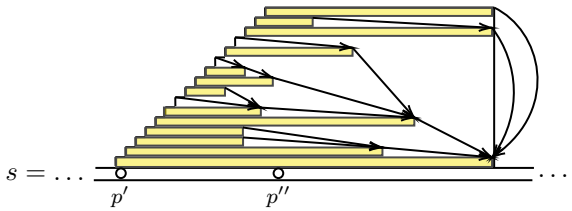


Figure 4 Each stripe of the trapezoid depicts an array b_r, for $r \in [p'..p''-1]$, and its prefix of length c_r (possibly empty) is colored in yellow (not $\min\{c_r+1, \ell_r\}$ like in Figure 3). Each yellow prefix corresponds to a node of $I_{p'}$ and it is connected to the yellow prefix corresponding to its parent in $I_{p'}$. To avoid overloading the picture, gray positions signifying 1s in b_r are not drawn.

The tree can be constructed in linear time inserting consecutively the nodes associated with $p', p'+1, \ldots$ and maintaining a stack that contains all nodes of the path from the last inserted node to the root: to insert a new node with weight w, we pop from the stack all nodes until a node heavier than w is encountered to which the new node is attached.

In terms of the tree $I_{p'}$, the node associated with the sought position t is the nearest ancestor of the node associated with p such that $w_t \geq \ell + p - p'$, where $\ell = q - p + 1$: the condition $w_t \geq \ell + p - p'$ is equivalent to $\ell_t - m \leq c_t$ since $w_t = t - p' + c_t$, $m = \ell_p - \ell$, and $\ell_t - \ell_p = p - t$. Since the threshold $\ell + p - p'$ in the condition does not exceed $\ell_{p'}$, we will be able to ignore differences between weights larger than $\ell_{p'}$ in our algorithm by "cutting" them in a way. Still, even with this restriction, at first glance this looks like quite a general weighted ancestor problem that admits no constant time solution in the space available. However, we will be able to use a structure common to such trees in order to speed up the computation. The idea is to decompose the tree into heavy paths (see below), as is usually done for weighted ancestor queries, and perform fast queries on the paths via the use of more memory; the trick is that, with some care, this space can be shared among "similar" trees and can be "traded" for irreducible LCP values relying again on Lemma 2.

Consider the tree $I_{p'}$. An edge (v, u) connecting a child v to its parent u is called *heavy* if $\text{size}(v) > \text{size}(u)/2$, and *light* otherwise, where $\text{size}(w)$ denotes the number of nodes in the subtree rooted at w. Thus, at most one child can be connected to u by a heavy edge. One can easily show that any leaf–root path contains at most $\log n$ light edges. All nodes of $I_{p'}$ are decomposed into disjoint *heavy paths*, maximal paths with only heavy edges in them. Note that in this version of the heavy-light decomposition [28] the number of heavy edges incident to a given non-leaf node can be zero (it then forms a singleton path). The decomposition can be constructed in linear time in one tree traversal.

The *predecessor search problem*, for a set of increasing numbers $w_1, w_2, \ldots, w_k$ and a given threshold $w \leq w_k$, is to find $\min\{w_i \colon w \leq w_i\}$. Although the problem, in fact, searches for a successor, we call it "predecessor search" as it is essentially an equivalent problem.

▶ **Lemma 6** ([13, Lem. 11]). *Consider a tree on m nodes with weights from $[0..n]$ such that some of the nodes are marked, any leaf–root path contains at most $\mathcal{O}(\log n)$ marked nodes, and the weights on any leaf–root path increase. One can preprocess the tree in $\mathcal{O}(m)$ time so that predecessor search can be performed among the marked ancestors of any node in $\mathcal{O}(1)$ time, provided a table computable in $o(n)$ and independent of the tree is precalculated.*[1]

For each heavy path, we mark the node closest to the root. Then, the data structure of Lemma 6 is built on each tree $I_{p'}$, which takes $\mathcal{O}(n)$ total time for all the trees $I_{p'}$. To answer a weighted ancestor query on $I_{p'}$, we find in $\mathcal{O}(1)$ time the heavy path containing the answer using this data structure and, then, consider the predecessor problem inside the path.

Consider a heavy path whose node weights are $w_{i_1}, w_{i_2}, \ldots, w_{i_k}$ in increasing order. We have to answer a predecessor query on the path for a threshold w such that $w \leq \ell_{p'}$ (recall that only thresholds $\leq \ell_{p'}$ are of interest for us) and it is known that its result is in the path, i.e., $w \leq w_{i_k}$. Let $w_{i_m} = \max\{w_{i_j} \colon w_{i_j} \leq \ell_{p'}\}$. A trivial constant-time solution for such queries is to construct a bit array $a[0..c]$, where $c = w_{i_m} - w_{i_1}$, endowed with a rank data structure such that, for any d, we have $a[d] = 1$ iff $d = w_{i_j} - w_{i_1}$, for some $j \in [1..m]$. Then, the predecessor query with a threshold w such that $w \leq \min\{\ell_{p'}, w_{i_k}\}$ can be answered by counting the number h of 1s in the subarray $a[0..w - w_{i_1} - 1]$, thus giving the result $w_{i_{h+1}} = \min\{w_{i_j} \colon w \leq w_{i_j}\}$. The array a occupies $\mathcal{O}(1 + \ell_{p'}/\log n)$ space since $c \leq \ell_{p'}$.

It turns out that, with minor changes, the arrays a can be shared among many heavy paths in different trees. However, this approach *per se* leads to $\mathcal{O}(n \log n)$ time and space, as will be evident in the sequel. We therefore need a slightly more elaborate solution.

Instead of the array $a[0..c]$, we construct a bit array $\hat{a}[0..\lfloor c/\lfloor \log n \rfloor \rfloor]$ endowed with a rank data structure such that, for any d, $\hat{a}[d] = 1$ iff the subarray $a[d\lfloor \log n \rfloor..(d+1)\lfloor \log n \rfloor - 1]$ contains non-zero values (assuming that $a[i] = 0$, for $i > c$). The bit array $\hat{a}$ packed into $\mathcal{O}(1 + c/\log^2 n)$ machine words of size $\lfloor \log n \rfloor$ bits can be built from the numbers $w_{i_1}, w_{i_2}, \ldots, w_{i_k}$ in one pass in $\mathcal{O}(k + c/\log^2 n)$ time. For each 1 value, $\hat{a}[d] = 1$, we collect all weights w_{i_j} such that $d\lfloor \log n \rfloor \leq w_{i_j} - w_{i_1} < (d+1)\lfloor \log n \rfloor$ into a set S_d. The sets S_d, for all d such that $\hat{a}[d] = 1$, are disjoint and non-empty, and can be assembled in one pass through the weights in $\mathcal{O}(k)$ time. We also store pointers P_h, with $h = 1, 2, \ldots$ $(h \leq m)$, such that P_h refers to a non-empty set S_d such that $\hat{a}[d] = 1$ is the hth 1 value in $\hat{a}$ (i.e., h is the number of 1s in the subarray $\hat{a}[0..d]$). Each set S_d is equipped with the following *fusion heap* data structure.

▶ **Lemma 7** (see [12, 26]). *For any set S of $\mathcal{O}(\log n)$ integers from $[0..n]$, one can build in $\mathcal{O}(|S|)$ time a fusion heap that answers predecessor queries in $\mathcal{O}(1)$ time.*

[1] Although Lemma 11 in [13] does not claim linear construction time, it easily follows from its proof.

With this machinery, a predecessor query with a threshold $w \leq \min\{\ell_{p'}, w_{i_k}\}$ can be answered by first counting the number h of 1s in the subarray $\hat{a}[0..\lfloor (w - w_{i_1})/\lfloor \log n \rfloor \rfloor - 1]$ and, then, finding in $\mathcal{O}(1)$ time the predecessor of w in the fusion heap referred by P_{h+1}.

The array $\hat{a}$ occupies $\mathcal{O}(1 + c/\log^2 n)$ space, which is $\mathcal{O}(1 + \ell_{p'}/\log^2 n)$ since $c \leq \ell_{p'}$. The computation of $\hat{a}$, which takes $\mathcal{O}(k + \ell_{p'}/\log^2 n)$ time, is the most time and space consuming part of the described construction; all other structures take $\mathcal{O}(k)$ time and space. We are to show that the computation of $\hat{a}$ can sometimes be avoided if (almost) the same array was already constructed for a different path. The following lemma is the key for this optimization.

▶ **Lemma 8.** *Given a tree $I_{p'}$ for an irreducible position p', consider its node x associated with a non-irreducible position $p > p'$. Let $s[r..n-1]$ be the closest irreducible lexicographical neighbour of $s[p..n-1]$ and let $c_p = \mathsf{lcp}(r, p)$. Then, the subtree of $I_{p'}$ rooted at x coincides with the tree I_r in which all children of the root with weights $\geq c_p$ are cut, then all weights are increased by $p - p'$, and the weight of the root is set to the weight of x (see Figure 5).*

Proof. Denote by I the subtree rooted at x. Observe that all nodes of I are exactly all nodes of $I_{p'}$ associated with positions from a range $[p..p+i]$, for some $i < c_p$ (see Figure 5). The position $p + i + 1$ is either irreducible or $c_{p+i+1} + i + 1 \geq c_p$. In order to prove the lemma, it suffices to show that (1) all positions from $[r+1..r+i]$ are not irreducible, (2) $c_{p+j} = c_{r+j}$, for any $j \in [1..i]$, and (3) the position $r + i + 1$ is either irreducible or $c_{r+i+1} + i + 1 \geq c_p$.

(1) Suppose, to the contrary, that a position $r + j$, for some $j \in [1..i]$, is irreducible. Then, since $\mathsf{lcp}(r, p) = c_p$ and $j \leq i < c_p$, we have $\mathsf{lcp}(r + j, p + j) = c_p - j$. Therefore, c_{p+j}, the length of the common prefix of $s[p+j..n-1]$ and its closest irreducible lexicographical neighbour, must be at least $c_p - j$ (since it was assumed that $r + j$ is irreducible). But then the weight w_{p+j} of the node associated with $p+j$ is at least $(c_p - j) + (p + j - p') = c_p + p - p'$, which is equal to the weight $w_p = c_p + p - p'$ of x. Thus, x cannot be an ancestor of the node, which is a contradiction.

(2) For $j \in [1..i]$, $s[p+j..n-1]$ and $s[r+j..n-1]$ share a common prefix of length $c_p - j$ since $\mathsf{lcp}(r, p) = c_p$ and $j \leq i < c_p$. Further, $c_{p+j} < c_p - j$ since the weight $w_{p+j} = c_{p+j} + p + j - p'$ is smaller than the weight $w_p = c_p + p - p'$ of its ancestor x. Suppose, without loss of generality, that $s[r+j..n-1]$ is lexicographically smaller than $s[p+j..n-1]$ (the case when it is greater is analogous). Then, neither of the suffixes $s[t..n-1]$ lexicographically lying between $s[r+j..n-1]$ and $s[p+j..n-1]$ can start with an irreducible position, for otherwise $c_{p+j} \geq \mathsf{lcp}(t, p + j) \geq \mathsf{lcp}(r + j, p + j) \geq c_p - j$. Therefore, the closest irreducible lexicographical neighbours of $s[p+j..n-1]$ and $s[r+j..n-1]$ coincide and $c_{p+j} = c_{r+j}$ $(< c_p - j \leq \mathsf{lcp}(p + j, r + j))$.

(3) Suppose that $r + i + 1$ is not irreducible and, by contradiction, $c_{r+i+1} < c_p - i - 1$. The suffixes $s[p+i+1..n-1]$ and $s[r+i+1..n-1]$ have a common prefix of length $c_p - i - 1$ (note that $i + 1 \leq c_p$). The position $p + i + 1$ is not irreducible since otherwise $c_{r+i+1} \geq \mathsf{lcp}(p + i + 1, r + i + 1) \geq c_p - i - 1$. Then, we have $c_{p+i+1} \geq c_p - i - 1$ because otherwise the node corresponding to $p + i + 1$ would have weight $w_{p+i+1} = c_{p+i+1} + p + i + 1 - p'$ that is smaller than the weight $w_p = c_p + p - p'$ of x and, thus, would be a descendant of x. Let $s[t..n-1]$ be the closest irreducible lexicographical neighbour of $s[p+i+1..n-1]$ so that $\mathsf{lcp}(t, p+i+1) = c_{p+i+1}$. Since $c_{p+i+1} \geq c_p - i - 1$ and $\mathsf{lcp}(p+i+1, r+i+1) \geq c_p - i - 1$, we obtain $\mathsf{lcp}(t, r+i+1) \geq c_p - i - 1$. Because t is irreducible, we deduce that $c_{r+i+1} \geq c_p - i - 1$, which is a contradiction. ◀

An array $\hat{a}$ corresponding to a heavy path in a tree $I_{p'}$, for an irreducible position p', occupies $\mathcal{O}(1 + \ell_{p'}/\log^2 n)$ space. Recall that $\ell_{p'} = \mathsf{plcp}[p']$ is an irreducible LCP value since p' is an irreducible position. Therefore, we can afford, for each tree $I_{p'}$, the construction and storage of the array $\hat{a}$ corresponding to the unique heavy path containing the root of $I_{p'}$.

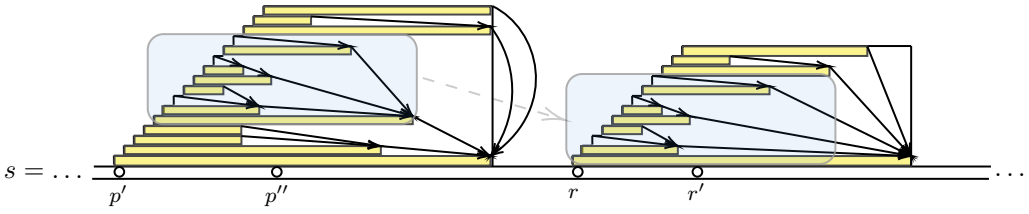


Figure 5 For irreducible positions p' and r, the subtree of $I_{p'}$ highlighted by the left blue rectangle is isomorphic to I_r after cutting the children of the root of I_r outside of the right blue rectangle.

The overall space required for this is $\mathcal{O}(n + \sum_{p'} \ell_{p'} / \log^2 n)$, where the sum is through all irreducible positions p', which is upperbounded by $\mathcal{O}(n + n/\log n) = \mathcal{O}(n)$ due to Lemma 2. Other heavy paths are processed as follows.

Suppose that we are to preprocess a heavy path $x_1 \to x_2 \to \cdots \to x_k$ with node weights $w_{i_1}, w_{i_2}, \ldots, w_{i_k}$ in a tree $I_{p'}$ such that x_k (the node closest to the root) is not the root of $I_{p'}$. We compute sets S_d and pointers P_h corresponding to the path in $\mathcal{O}(k)$ time. As for the array $\hat{a}$, we either construct it for the path from scratch or reuse a suitable array from another path; the details follow. Let $w_{i_m} = \max\{w_{i_j} : w_{i_j} \le \ell_{p'}\}$. As was discussed, the array $\hat{a}$ encodes, in a sense, a predecessor data structure for the weights $w_{i_1}, w_{i_2}, \ldots, w_{i_m}$. In order to have some flexibility for the reuse of arrays, we modify this scheme slightly so that sometimes $\hat{a}$ does not encode the last two weights $w_{i_{m-1}}$ and w_{i_m}. The algorithm that answers a predecessor query for a threshold $w \le \min\{\ell_{p'}, w_{i_k}\}$ on the path is altered accordingly: we first compare w to $w_{i_{m-1}}$ and w_{i_m}, and only then, if necessary, use the array $\hat{a}$ as described above.

Let x_k correspond to a non-irreducible position p and let $s[r..n{-}1]$ be the closest irreducible lexicographical neighbour of $s[p..n{-}1]$. As was shown in Section 4, the position r can be found in $\mathcal{O}(1)$ time. By Lemma 8, the subtree I rooted at x_k is isomorphic to the tree I_r in which all children of the root with weights greater than or equal to c_p are cut, then all weights are increased by $p - p'$, and the root weight is set to w_{i_k}. Denote this isomorphism by ϕ. Note that $\phi(x_k)$ is the root of I_r. The main corollary of Lemma 8 is that in the subtree I any edge $u \to v$ that is not incident to x_k is heavy iff the corresponding edge $\phi(u) \to \phi(v)$ in I_r is heavy. Therefore, all edges in the path $\phi(x_1) \to \phi(x_2) \to \cdots \to \phi(x_k)$ are heavy, except, possibly, the last edge $\phi(x_{k-1}) \to \phi(x_k)$, and no heavy edge enters the node $\phi(x_1)$ in I_r. By Lemma 8, the weights of the nodes $\phi(x_1)$, $\phi(x_2)$, …, $\phi(x_{k-1})$ are $w_{i_1} - (p - p')$, $w_{i_2} - (p - p')$, …, $w_{i_{k-1}} - (p - p')$, respectively. We proceed further in three separate cases.

Heavy $\phi(x_{k-1}) \to \phi(x_k)$ and ℓ_r is large enough. Suppose that the edge $\phi(x_{k-1}) \to \phi(x_k)$ is heavy. Then, $\phi(x_1) \to \phi(x_2) \to \cdots \to \phi(x_k)$ is the unique heavy path of I_r that contains the root. Let $\hat{a}_\phi[0..c_\phi]$ be an array for predecessor queries that was explicitly stored in $\mathcal{O}(1 + \ell_r/\log^2 n)$ space for the path $\phi(x_1) \to \phi(x_2) \to \cdots \to \phi(x_k)$. By definition, for any $d \in [0..c_\phi]$, we have $\hat{a}_\phi[d] = 1$ iff, for some $j \in [1..k]$, the number $(w_{i_j} - (p - p')) - (w_{i_1} - (p - p')) = w_{i_j} - w_{i_1}$ lies in the range $[d\lfloor \log n \rfloor .. (d+1)\lfloor \log n \rfloor - 1]$. Since the latter is also a criterium for $\hat{a}[d] = 1$, the array $\hat{a}_\phi$ can therefore be reused to imitate $\hat{a}$ if $\hat{a}_\phi$ is sufficiently long to "encode" the weights $w_{i_1}, w_{i_2}, \ldots, w_{i_{m-2}}$. More precisely, this is the case iff $\ell_r \ge w_{i_{m-2}} - (p - p')$. Thus, if the edge $\phi(x_{k-1}) \to \phi(x_k)$ is heavy and $\ell_r \ge w_{i_{m-2}} - (p - p')$, then the overall preprocessing of the path $x_1 \to x_2 \to \cdots \to x_k$ takes only $\mathcal{O}(k)$ time since we can store a pointer to the array $\hat{a}_\phi$ and reuse $\hat{a}_\phi$ to imitate the array $\hat{a}$.

Unfortunately, neither of these two conditions necessarily holds in general: the edge $\phi(x_{k-1}) \to \phi(x_k)$ might be light in I_r and ℓ_r might be less than $w_{i_{m-2}} - (p - p')$.

Light $\phi(x_{k-1}) \to \phi(x_k)$ and ℓ_r is large enough. Suppose that the edge $\phi(x_{k-1}) \to \phi(x_k)$ is light in I_r and $\ell_r \geq w_{i_{m-2}} - (p - p')$. By Lemma 8, the edge necessarily becomes heavy if we cut all children of the root of I_r with weights greater than or equal to c_p. We assign numbers $1, 2, \ldots$ to the children of each node in the trees I_r and $I_{p'}$ according to the order in which they were attached during the construction of the trees, i.e., in the increasing order of their associated positions. It follows from the proof of Lemma 8 that if x_{k-1} is the hth child of x_k, then $\phi(x_{k-1})$ is the hth child of the root of I_r. Therefore, the node $\phi(x_{k-1})$ can be located in $\mathcal{O}(1)$ time.

We associate with each child of the root of I_r a pointer, initially set to null. If the pointer in $\phi(x_{k-1})$ is still null at the time we access it while preprocessing the heavy path $x_1 \to x_2 \to \cdots \to x_k$, then we create from scratch a new array $\hat{a}_\phi$ for the heavy path $\phi(x_1) \to \phi(x_2) \to \cdots \to \phi(x_{k-1})$ in $\mathcal{O}(k + \ell_r/\log^2 n)$ time and set the pointer of $\phi(x_{k-1})$ to refer to this array. Since $\ell_r \geq w_{i_{m-2}} - (p - p')$, the array $\hat{a}_\phi$ can be reused to imitate $\hat{a}$ for the path $x_1 \to x_2 \to \cdots \to x_k$ in the same way as was described above. If the pointer in the node $\phi(x_{k-1})$ is not null, then it already refers to a suitable array $\hat{a}_\phi$ that can be reused (note that $\phi(x_1) \to \phi(x_2) \to \cdots \to \phi(x_{k-1})$ is the unique heavy path of the tree I_r that contains the node $\phi(x_{k-1})$). Thus, the preprocessing takes $\mathcal{O}(k)$ time plus, if necessary, $\mathcal{O}(k + \ell_r/\log^2 n)$ time required to construct a new array $\hat{a}_\phi$.

The following lemma shows that a non-null pointer to an array $\hat{a}_\phi$ might be assigned to at most $\log n$ distinct children of the root of I_r (namely, those children that become connected to the root by a heavy edge after a number of children with greater weights were removed). This implies that the overall construction time for all the arrays $\hat{a}_\phi$ throughout the whole algorithm is $\mathcal{O}(n + \sum_r (\ell_r/\log^2 n)\log n) = \mathcal{O}(n)$, where the sum is through all irreducible positions r (so that $\sum_r \ell_r = \mathcal{O}(n \log n)$ by Lemma 2).

▶ **Lemma 9.** *Let S be a tree with at most n nodes whose root r has m children ordered arbitrarily. Suppose that we remove the children of the root (with the subtrees rooted at them) from right to left, one by one, thus producing trees $S_1, S_2, \ldots, S_m$. Let z_i be a node of the tree S_i such that z_i is connected to r by a heavy edge, or z_i is r itself if r has no incident heavy edges in S_i. Then, the set $\{z_1, z_2, \ldots, z_m\}$ contains at most $\log n$ distinct nodes.*

Proof. If $z_i \to r$ is a heavy edge in S_i, then $\mathrm{size}(z_i) > \mathrm{size}(r)/2$ in S_i. Therefore, if we cut any other child of r in S_i, only the number $\mathrm{size}(r)$ decreases and, thus, the edge remains heavy. But when we remove z_i and its subtree from S_i, the number $\mathrm{size}(r)$ decreases by more than half. Such halving may happen at most $\log n - 1$ times, and the result follows. ◀

Small ℓ_r. Suppose that $\ell_r < w_{i_{m-2}} - (p - p')$. Let $\bar{p}$ be the position associated with the node x_{k-1} in the tree $I_{p'}$ and let $s[\bar{r}..n-1]$ be the closest irreducible lexicographical neighbour of $s[\bar{p}..n-1]$. By analogy to the isomorphism ϕ, we define using Lemma 8 an isomorphism ψ that maps the subtree of $I_{p'}$ rooted at the node x_{k-1} onto a "pruned" tree $I_{\bar{r}}$ in which all children of the root with weights greater than or equal to $c_{\bar{p}}$ are cut. Note that $\psi(x_{k-1})$ is the root of $I_{\bar{r}}$. By Lemma 8, the weights of the nodes $\psi(x_1)$, $\psi(x_2), \ldots$, $\psi(x_{k-2})$ are $w_{i_1} - (\bar{p} - p')$, $w_{i_2} - (\bar{p} - p'), \ldots$, $w_{i_{k-2}} - (\bar{p} - p')$, respectively.

It turns out that, instead of imitating the array $\hat{a}$ for the path $x_1 \to x_2 \to \cdots \to x_k$ using an array $\hat{a}_\phi$ from the tree I_r, one can in quite the same way imitate $\hat{a}$ using an appropriate array $\hat{a}_\psi$ from $I_{\bar{r}}$, which must be sufficiently long in the case $\ell_r < w_{i_{m-2}} - (p - p')$. More precisely, we are to prove that $\ell_{\bar{r}} \geq w_{i_{m-2}} - (\bar{p} - p')$, which guarantees that essentially the same approach works well: if the edge $\psi(x_{k-2}) \to \psi(x_{k-1})$ is heavy, then $\hat{a}_\psi$ is an array associated with the unique heavy path containing the root of $I_{\bar{r}}$ and, since $\ell_{\bar{r}} \geq w_{i_{m-2}} - (\bar{p} - p')$, the

array $\hat{a}_\psi$ is long enough to imitate $\hat{a}$; if the edge $\phi(x_{k-2}) \to \psi(x_{k-1})$ is light, then we either reuse a suitable array $\hat{a}_\psi$ that was already stored in the child $\psi(x_{k-2})$ of the root of $I_{\bar{r}}$ (again $\hat{a}_\psi$ can imitate $\hat{a}$ since $\ell_{\bar{r}} \geq w_{i_{m-2}} - (\bar{p} - p')$) or we create this array $\hat{a}_\psi$ from scratch for the path $\psi(x_1) \to \psi(x_2) \to \cdots \to \psi(x_{k-2})$ in $\mathcal{O}(k + \ell_{\bar{r}} / \log^2 n)$ time and store a pointer to it in $\psi(x_{k-2})$. We do not go further into details since they are essentially the same as above.

We actually prove a stronger condition $\ell_{\bar{r}} \geq w_{i_{m-1}} - (\bar{p} - p')$, which implies $\ell_{\bar{r}} \geq w_{i_{m-2}} - (\bar{p} - p')$ since $w_{i_{m-1}} > w_{i_{m-2}}$. Recall that $w_{i_{m-1}} = c_{\bar{p}} + \bar{p} - p'$. Hence, the stronger condition is equivalent to $\ell_{\bar{r}} \geq c_{\bar{p}}$. If the suffix $s[\bar{r}..n-1]$ is lexicographically larger than $s[\bar{p}..n-1]$, then we immediately obtain $\ell_{\bar{r}} = \mathsf{plcp}[\bar{r}] \geq \mathsf{lcp}(\bar{p}, \bar{r}) = c_{\bar{p}}$. Assume that $s[\bar{r}..n-1]$ is lexicographically smaller than $s[\bar{p}..n-1]$. Let us show that this case is actually impossible since it contradicts the condition of "small ℓ_r" that was assumed in the beginning: $\ell_r < w_{i_{m-2}} - (p - p')$. Denote $j = \bar{p} - p$. Since $\mathsf{lcp}(p, r) = c_p$ and $j < c_p$, we obtain $\mathsf{lcp}(\bar{p}, r + j) = c_p - j$. Further, $c_{\bar{p}} < c_p - j$ because the weight $w_{i_{m-1}} = c_{\bar{p}} + \bar{p} - p'$ of the node x_{m-1} is less than the weight $w_{i_k} = c_p + p - p'$ of x_k. Since $\mathsf{lcp}(\bar{p}, \bar{r}) = c_{\bar{p}} < c_p - j = \mathsf{lcp}(\bar{p}, r + j)$, we obtain $\mathsf{lcp}(\bar{r}, r + j) = c_{\bar{p}} < \mathsf{lcp}(\bar{p}, r + j)$ and, further, since we assumed that $s[\bar{r}..n-1]$ is lexicographically smaller than $s[\bar{p}..n-1]$, the suffix $s[\bar{r}..n-1]$ is lexicographically smaller than $s[r+j..n-1]$. Hence, $\ell_{r+j} \geq \mathsf{lcp}(\bar{r}, r + j) = c_{\bar{p}}$. It follows from Lemma 4 that $\ell_r = \ell_{r+j} + j$. Then, $\ell_r \geq c_{\bar{p}} + j$. Conversely, we deduce $\ell_r < w_{i_{m-2}} - (p - p') < w_{i_{m-1}} - (p - p')$ and, substituting $w_{i_{m-1}} = c_{\bar{p}} + p + j - p'$, we obtain $\ell_r < c_{\bar{p}} + j$, which is a contradiction.

To sum up, we spend linear time preprocessing each heavy path in every tree $I_{p'}$ plus $\mathcal{O}(n)$ total time to construct arrays $\hat{a}_\phi$ and $\hat{a}_\psi$, each of which is associated with one of $\log n$ particular children of the root in one of the trees (according to Lemma 9). Therefore, since all the trees $I_{p'}$ contain n nodes in total, the overall time is $\mathcal{O}(n)$.

6 Concluding Remarks

We believe that the presented solution, while certainly still quite complicated and impractical, is more implementable than that of [13]. We expect that, with additional combinatorial insights, one can simplify it even further, perhaps eventually arriving at a practical data structure for the problem. A number of natural and less vaguely formulated problems also arise. For example, can we maintain the weighted ancestor data structure during an online construction of the suffix tree? Is it possible to reduce the space usage and support weighted ancestors in compact suffix trees [24, 27] with $\mathcal{O}(n)$ or $\mathcal{O}(n \log \sigma)$ additional *bits* of space, where σ is the alphabet size?

To the best of our knowledge, the lemma about the sum of irreducible LCP values has found relatively few applications other than in the construction of LCP and PLCP arrays (e.g., see references in [16]): [2, 3, 18, 29] (the application in [18], however, is somewhat spectacular). The techniques developed in our paper might be interesting by themselves and pave the way to more applications of irreducible LCPs in algorithms on strings.

References

1 A. Amir, G.M. Landau, M. Lewenstein, and D. Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms (TALG)*, 3(2):19–31, 2007. doi:10.1145/1240233.1240242.

2 G. Badkobeh, P. Gawrychowski, J. Kärkkäinen, S. J. Puglisi, and B. Zhukova. Tight upper and lower bounds on suffix tree breadth. *Theoretical Computer Science*, 854:63–67, 2021. doi:10.1016/j.tcs.2020.11.037.

3 G. Badkobeh, J. Kärkkäinen, S.J. Puglisi, and B. Zhukova. On suffix tree breadth. In *Proc. 24th Symposium on String Processing and Information Retrieval (SPIRE)*, volume 10508 of *LNCS*, pages 68–73. Springer, 2017. doi:10.1007/978-3-319-67428-5_6.

4 M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *4th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

5 M.A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.

6 O. Berkman and U. Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–230, 1994. doi:10.1016/S0022-0000(05)80002-9.

7 S. Biswas, A. Ganguly, R. Shah, and S.V. Thankachan. Ranked document retrieval with forbidden pattern. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 77–88. Springer, 2015. doi:10.1007/978-3-319-19929-0_7.

8 S. Biswas, A. Ganguly, R. Shah, and S.V. Thankachan. Ranked document retrieval for multiple patterns. *Theoretical Computer Science*, 746:98–111, 2018. doi:10.1016/j.tcs.2018.06.029.

9 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

10 David Clark. *Compact pat trees.* PhD thesis, University of Waterloo, 1997.

11 M. Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *7th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 130–140. Springer, 1996. doi:10.1007/3-540-61258-0_11.

12 M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

13 P. Gawrychowski, M. Lewenstein, and P.K. Nicholson. Weighted ancestors in suffix trees. In *Proc. 22nd Annual European Symposium on Algorithms (ESA)*, pages 455–466. Springer, 2014. doi:10.1007/978-3-662-44777-2_38.

14 D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. doi:10.1137/0213024.

15 G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE, 1989. doi:10.1109/SFCS.1989.63533.

16 J. Kärkkäinen, D. Kempa, and M. Piątkowski. Tighter bounds for the sum of irreducible LCP values. *Theoretical Computer Science*, 656:265–278, 2016. doi:0.1016/j.tcs.2015.12.009.

17 J. Kärkkäinen, G. Manzini, and S.J. Puglisi. Permuted longest-common-prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192. Springer, 2009. doi:10.1007/978-3-642-02441-2_17.

18 D. Kempa and T. Kociumaka. Resolution of the Burrows–Wheeler transform conjecture. *arXiv preprint*, 2019. arXiv:1910.10631.

19 D. Kempa, A. Policriti, N. Prezza, and E. Rotenberg. String attractors: Verification and optimization. In *Proc. 26th Annual European Symposium on Algorithms (ESA)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52:1–52:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ESA.2018.52.

20 T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. A linear-time algorithm for seeds computation. *ACM Transactions on Algorithms (TALG)*, 16(2):1–23, 2020. doi:10.1145/3386369.

21 T. Kopelowitz, G. Kucherov, Y. Nekrich, and T. Starikovskaya. Cross-document pattern matching. *Journal of Discrete Algorithms*, 24:40–47, 2014. doi:10.1016/j.jda.2013.05.002.

22 T. Kopelowitz and M. Lewenstein. Dynamic weighted ancestors. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 565–574. ACM-SIAM, 2007. doi:10.5555/1283383.1283444.

23 U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

24 J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 408–424. SIAM, 2017. doi:10.1137/1.9781611974782.26.

25 M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240. ACM, 2006. doi:10.1145/1132516.1132551.

26 M. Pătraşcu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 166–175. IEEE, 2014. doi:10.1109/FOCS.2014.26.

27 K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x.

28 D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

29 N. Välimäki and S.J. Puglisi. Distributed string mining for high-throughput sequencing data. In *Proc. 12th International Workshop on Algorithms in Bioinformatics (WABI)*, LNCS 7534, pages 441–452. Springer, 2012. doi:10.1007/978-3-642-33122-0_35.

30 P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical systems theory*, 10(1):99–127, 1976. doi:10.1007/BF01683268.

31 D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

Constructing Strings Avoiding Forbidden Substrings

Giulia Bernardini
CWI, Amsterdam, The Netherlands

Alberto Marchetti-Spaccamela
Dept. of Computer, Automatic and Management Engineering, Sapienza University of Rome, Italy
ERABLE Team, Lyon, France

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands
ERABLE Team, Lyon, France

Leen Stougie
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands
ERABLE Team, Lyon, France

Michelle Sweering[1]
CWI, Amsterdam, The Netherlands

Abstract

We consider the problem of constructing strings over an alphabet Σ that start with a given prefix u, end with a given suffix v, and avoid occurrences of a given set of *forbidden substrings*. In the decision version of the problem, given a set S_k of forbidden substrings, each of length k, over Σ, we are asked to decide whether there exists a string x over Σ such that u is a prefix of x, v is a suffix of x, and no $s \in S_k$ occurs in x. Our first result is an $\mathcal{O}(|u| + |v| + k|S_k|)$-time algorithm to decide this problem. In the more general optimization version of the problem, given a set S of forbidden arbitrary-length substrings over Σ, we are asked to construct a shortest string x over Σ such that u is a prefix of x, v is a suffix of x, and no $s \in S$ occurs in x. Our second result is an $\mathcal{O}(|u| + |v| + ||S|| \cdot |\Sigma|)$-time algorithm to solve this problem, where $||S||$ denotes the total length of the elements of S.

Interestingly, our results can be directly applied to solve the reachability and shortest path problems in complete de Bruijn graphs in the presence of forbidden edges or of forbidden paths.

Our algorithms are motivated by data privacy, and in particular, by the data sanitization process. In the context of strings, sanitization consists in hiding forbidden substrings from a given string by introducing the least amount of spurious information. We consider the following problem. Given a string w of length n over Σ, an integer k, and a set S_k of forbidden substrings, each of length k, over Σ, construct a shortest string y over Σ such that no $s \in S_k$ occurs in y and the sequence of all other length-k fragments occurring in w is a subsequence of the sequence of the length-k fragments occurring in y. Our third result is an $\mathcal{O}(nk|S_k| \cdot |\Sigma|)$-time algorithm to solve this problem.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, forbidden strings, de Bruijn graphs, data sanitization

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.9

Funding *Giulia Bernardini*: Netherlands Organisation for Scientific Research (NWO) under project OCENW.GROOT.2019.015 "Optimization for and with Machine Learning (OPTIMAL)"
Leen Stougie: Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003
Michelle Sweering: Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003

[1] Corresponding author

Acknowledgements We wish to thank Gabriele Fici (Università di Palermo) for bringing to our attention the work of Crochemore, Mignosi and Restivo [21].

1 Introduction

We start with some basic definitions and notation from [20]. An *alphabet* Σ is a finite nonempty set of elements called *letters*. We assume throughout an integer alphabet $\Sigma = [1, |\Sigma|]$. A *string* $x = x[1] \dots x[n]$ is a sequence of *length* $|x| = n$ of letters from Σ. The *empty* string, denoted by ε, is the string of length 0. The fragment $x[i \,..\, j]$ is an *occurrence* of the underlying *substring* $s = x[i] \dots x[j]$; s is a *proper* substring of x if $x \neq s$. We also say that s occurs at *position* i in x. A *prefix* of x is a fragment of x of the form $x[1 \,..\, j]$ and a *suffix* of x is a fragment of x of the form $x[i \,..\, n]$. An *infix* of x is a fragment of x that is neither a prefix nor a suffix. The set of all strings over Σ (including ε) is denoted by Σ^*. The set of all length-k strings over Σ is denoted by Σ^k.

We consider the following basic problem on strings.

String Existence Avoiding Forbidden Length-k Substrings (SEFS)
Input: An integer $k > 0$, two strings $u, v \in \Sigma^*$, and a set $S_k \subset \Sigma^k$.
Output: YES if there exists a string $x \in \Sigma^*$ such that u is a prefix of x, v is a suffix of x, and no $s \in S_k$ occurs in x; or NO otherwise.

In what follows we refer to set S_k as the set of *forbidden substrings*.

▶ **Example 1.** Consider $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $k = 4$, $S_k = \{\mathtt{bbbb}, \mathtt{aaba}, \mathtt{abba}\}$, $u = \mathtt{aab}$, and $v = \mathtt{aba}$. SEFS has a positive answer, as there exists, for instance, string $x = \mathtt{aabbbaba}$ with u as a prefix, v as a suffix, and with no occurrence of any $s \in S_k$.

In Section 3, we show the following result.

▶ **Theorem 2.** *Given an integer $k > 1$, two strings $u, v \in \Sigma^*$, and a set $S_k \subset \Sigma^k$, SEFS can be solved in $\mathcal{O}(|u| + |v| + k|S_k|)$ time.*

We also consider the following more general optimization version of the SEFS problem.

Shortest String Avoiding Forbidden Substrings (SSFS)
Input: Two strings $u, v \in \Sigma^*$ and a set $S \subset \Sigma^*$.
Output: A shortest string $x \in \Sigma^*$ such that u is a prefix of x, v is a suffix of x, and no $s \in S$ occurs in x; or FAIL if no such x exists.

Note that in SSFS the set S of forbidden substrings contains strings of arbitrary lengths. In Section 4, we show the following result.

▶ **Theorem 3.** *Given two strings $u, v \in \Sigma^*$, and a set $S \subset \Sigma^*$, SSFS can be solved in $\mathcal{O}(|u| + |v| + ||S|| \cdot |\Sigma|)$ time, where $||S|| = \sum_{s \in S} |s|$, using $\mathcal{O}(|u| + ||S|| \cdot |\Sigma|)$ space.*

Related Work. Crochemore, Mignosi and Restivo [21] showed how to construct a complete deterministic finite automaton (DFA) accepting strings over Σ which do not contain any forbidden substring from a finite anti-factorial language S (see also [8]). This DFA has $\mathcal{O}(||S||)$ states, $\Theta(||S|| \cdot |\Sigma|)$ edges in the worst case, and it can be constructed in $\Theta(||S|| \cdot |\Sigma|)$ time. Thus using this DFA for deciding SEFS would entail a time complexity of $\Omega(|u|+|v|+k|S_k| \cdot |\Sigma|)$ in the worst case. We show a fundamentally different *non-constructive* approach to decide SEFS in $\mathcal{O}(|u| + |v| + k|S_k|)$ time, which is based on combinatorial properties of complete

de Bruijn graphs. For solving the optimization version SSFS, we make use of the DFA complemented with an efficient way to compute the *appropriate* source and sink nodes. This is because u (as a prefix) and v (as a suffix) must occur in string x (possibly overlapping).

Following the definition in [15], a language $L \subseteq \Sigma^*$ is *strictly locally testable* if there exist an integer k and finite sets $F, U, V \subseteq \bigcup_{i=1}^{k-1} \Sigma^i$ and $W \subseteq \Sigma^k$ such that $L = ((U\Sigma^* \cap \Sigma^* V) \setminus \Sigma^* W \Sigma^*) \cup F$. Therefore SEFS can be reduced to determining whether or not some specific strictly locally testable language is nonempty, while SSFS can be reduced to finding a minimum-length string in its corresponding language.

Our Motivation. We are motivated by applications in data privacy, and in particular, in data sanitization. *Data sanitization*, also known as *knowledge hiding*, is a privacy-preserving data mining process, which aims at preventing the mining of confidential knowledge from published datasets. Data sanitization has been an active area of research for the past 25 years [18, 38, 42, 29, 43, 30, 1, 2, 28, 31, 37, 13]. Informally, it is the process of disguising (hiding) confidential information in a given dataset. This process typically incurs some data utility loss that should be minimized. Naturally, privacy constraints and utility objective functions lead to the formulation of combinatorial optimization problems. From a fundamental perspective, it is thus relevant to be able to establish some formal guarantees.

In the context of strings, data sanitization consists in hiding forbidden substrings from a given string by introducing the least amount of spurious information [9, 10, 11, 12]. In previous works [9, 10, 11], we considered various combinatorial optimization problems for string sanitization, all of which receive as input a string w of length n over Σ, an integer k, and a set S_k of forbidden substrings, and conceal the occurrences of forbidden substrings in w through the use of a special letter $\# \notin \Sigma$. In particular, the TFS problem [9] asks to construct a shortest string x such that no string in S_k occurs in x and the order of occurrence of all other length-k substrings over Σ (and thus their frequency) is the same in w and in x. We developed an algorithm that solves the TFS problem in the optimal $\mathcal{O}(n + |x|)$ time [9], assuming that the list of all occurrences of forbidden substrings in w are given at input.

▶ **Example 4.** Let $w =$ abbbbaaabaa, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $k = 4$, and $S_k = \{\mathtt{bbbb}, \mathtt{aaba}, \mathtt{abba}\}$. All occurrences of forbidden substrings are underlined. The solution to the TFS problem is string $x =$ `abbbaaab#abaa`, where $\# \notin \Sigma$: it is the shortest string in which the occurrences of the strings in S_k are concealed and the order of all other length-k substrings over Σ is preserved.

However, as already noted in [9], the occurrences of # in x may reveal the *locations* of the forbidden substrings in w and should therefore be ultimately replaced by letters or strings over Σ in several applications of interest. The problem of replacing the occurrences of # in x with *single letters* of Σ without reintroducing any forbidden substrings and with different optimization criteria has been considered in [9, 10, 12]. Replacing #'s with single letters, though, may be too restrictive, and even "easy" instances may admit no feasible solution.

▶ **Example 5.** Consider the instance from Example 4. The occurrence of # in $x =$ `abbbaaab#abaa` reveals the location in w of the forbidden substring `aaba`. Note that deleting # would reinstate `aaba`; and # cannot be replaced by a single letter from Σ, as both `a` and `b` create occurrences of the forbidden substrings `aaba` and `abba`, respectively.

We thus consider a more general string sanitization problem in which we allow # replacements with strings of *any* length over Σ, so as to widen the set of instances that admit a feasible solution. Given a string w over an alphabet Σ, an integer $k > 1$, and a set $S_k \subset \Sigma^k$, we denote by $\mathcal{S}(w, S_k)$ the sequence over $\Sigma^k \setminus S_k$ in which the ith element (from left to right) is the ith length-k substring occurrence in w that is not in S_k. In other words, $\mathcal{S}(w, S_k)$ is the sequence of non-forbidden length-k fragments of w. This allows us to reformulate SFSS:

Shortest Fully-Sanitized String (SFSS)
Input: A string $w \in \Sigma^n$, an integer $k > 1$, and a set $S_k \subset \Sigma^k$.
Output: A shortest string $y \in \Sigma^*$ such that no $s \in S_k$ occurs in y and $\mathcal{S}(w, S_k)$ is a subsequence of $\mathcal{S}(y, S_k)$; or FAIL if no such y exists.

▶ **Example 6.** Consider again the instance from Example 4. We have $\mathcal{S}(w, S_k) =$ `abbb`, `bbba`, `bbaa`, `baaa`, `aaab`, `abaa`. A solution to the SFSS problem is string $y =$ `abbbaaabbbabaa`. We have that $\mathcal{S}(w, S_k)$ is a subsequence of $\mathcal{S}(y, S_k) =$ `abbb`, `bbba`, `bbaa`, `baaa`, `aaab`, `aabb`, `abbb`, `bbba`, `bbab`, `baba`, `abaa`. Moreover, no $s \in S_k$ occurs in y and y is a shortest such string.

A solution y to the SFSS problem has the following attractive properties, which are related to privacy or utility: (i) no forbidden substring occurs in y (privacy); (ii) y has as a subsequence the sequence of non-forbidden length-k substrings of w (utility); and (iii) y is the shortest possible (utility).

In Section 5, we reduce the SFSS problem to $d \leq n$ special instances of the SSFS problem. Each such special instance can be seen as seeking for a shortest path in the complete de Bruijn graph of order k over Σ [22] in the presence of forbidden edges. The *complete de Bruijn graph* of order k over an alphabet Σ is a directed graph $G_k = (V_k, E_k)$, where the set of nodes $V_k = \Sigma^{k-1}$ is the set of length-$(k-1)$ strings over Σ. There is an edge $(u, v) \in E_k$ if and only if the length-$(k-2)$ suffix of u is the length-$(k-2)$ prefix of v. There is therefore a natural correspondence between an edge (u, v) and the length-k string $u[1]u[2]\dots u[k-1]v[k-1]$. We will thus sometimes abuse notation and write $S_k \subset E_k$. In particular, one such instance asks for constructing a shortest path from node u to node v, with $u, v \in V_k$, avoiding edges $S_k \subset E_k$. We thus finally apply Theorem 3, with $k - 1 = |u| = |v|$ and $S = S_k$, $d \leq n$ times to obtain Theorem 7.

▶ **Theorem 7.** *Given a string w of length n over an alphabet Σ, an integer $k > 1$, and a set $S_k \subset \Sigma^k$, SFSS can be solved in $\mathcal{O}(nk|S_k| \cdot |\Sigma|)$ time.*

Other Related Work. Graph reachability is a classic problem in computer science [34, 39, 19, 40]. It refers to the ability to get from one node to another within a graph. In particular, computing shortest paths is one of the most well-studied algorithmic problems. Graph reachability and shortest path computation in the presence of failing nodes or of failing edges becomes a much more challenging task [14, 7, 6, 4, 17, 5, 16, 3, 32]. One obvious, yet important, application of accommodating such failures is in geographic routing [36].

To the best of our knowledge, reachability and shortest path computation in complete de Bruijn graphs in the presence of failing edges has not been considered before. Theorem 2 and Theorem 3 directly solve the reachability and shortest path versions, respectively. Interestingly, Theorem 3 constructs a shortest path in the presence of arbitrarily-long *failing paths*. Our results, other than in data sanitization, may thus be of independent interest.

2 Algorithmic Toolkit

Let M be a finite nonempty set of strings over Σ of total length m. The *trie* of M, denoted by $\mathsf{TR}(M)$, is a deterministic finite automaton that recognizes M with the following features [20]. Its set of states (nodes) is the set of prefixes of the elements of M; the initial state (root node) is ε; the set of terminal states (leaf nodes) is M; and edges are of the form $(u, \alpha, u\alpha)$, where u and $u\alpha$ are nodes and $\alpha \in \Sigma$. The size of $\mathsf{TR}(M)$ is thus $\mathcal{O}(m)$. The *compacted trie* of M, denoted by $\mathsf{CT}(M)$, contains the root node, the branching nodes, and the leaf nodes of $\mathsf{TR}(M)$. The term compacted refers to the fact that $\mathsf{CT}(M)$ reduces the number of

nodes by replacing each maximal branchless path segment with a single edge, and it uses a fragment of a string $s \in M$ to represent the label of this edge in $\mathcal{O}(1)$ machine words. The size of $\mathsf{CT}(M)$ is thus $\mathcal{O}(|M|)$. When M is the set of suffixes of a string y, then $\mathsf{CT}(M)$ is called the *suffix tree* of y, and we denote it by $\mathsf{ST}(y)$. The suffix tree of a string of length n over an alphabet $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$ can be constructed in $\mathcal{O}(n)$ time [26]. The *generalized suffix tree* of strings $y_1 \ldots, y_k$ over Σ, denoted by $\mathsf{GST}(y_1, \ldots, y_k)$, is the suffix tree of string $y_1\$_1 \ldots y_k\$_k$, where $\$_1, \ldots, \$_k$ are distinct letters not from Σ.

We next recall some basic concepts on randomized algorithms. For an input of size n and an arbitrarily large constant c fixed prior to the execution of a randomized algorithm, the term *with high probability* (whp), or inverse-polynomial probability, means with probability at least $1 - n^{-c}$. When we say that the time complexity of an algorithm holds with high probability, it means that the algorithm terminates in the claimed complexities with probability $1 - n^{-c}$. Such an algorithm is referred to as *Las Vegas whp*. When we say that an algorithm returns a correct answer with high probability, it means that the algorithm returns a correct answer with probability $1 - n^{-c}$. Such an algorithm is referred to as *Monte Carlo whp*.

A *static dictionary* is a data structure that maintains a set K of items that are known in advance. Each item may be associated with some satellite information. A *perfect hash function* for a set K is a hash function that maps the items in K to a set of integers with no collisions. There exists a Las Vegas whp algorithm that constructs a linear-sized static dictionary to maintain S that employs a perfect hash function and supports look-up queries in constant time per query [27]. A *dynamic dictionary* is a data structure that maintains a dynamic set K of items; i.e., a set of items that are not known in advance. Each item may be associated with some satellite information. There exists a Monte Carlo whp algorithm that constructs a linear-sized dynamic dictionary to maintain K that employs a perfect hash function dynamically and supports insert and look-up queries in constant time per query [24].

The *Karp-Rabin fingerpint* (KRF) of a string y over an integer alphabet is defined as $\phi_{q,r}(y) = \sum_{i=1}^{|y|} y[i]r^{|y|-i} \mod q$, where q is a prime number and r is a random integer in $[1, q]$ [35]. A crucial property of KRFs is that, with high probability, no collision occurs among the length-k substrings of a given string. To see this, consider two strings $s \neq t$ each of length k. The polynomial $\phi_{q,r}(s) - \phi_{q,r}(t)$ has at most k roots modulo (prime) q, so the two strings collide, i.e., $\phi_{q,r}(s) = \phi_{q,r}(t)$, with probability no more than $\frac{k}{q-1}$. Thus, for sufficiently large q, we can avoid all possible collisions between the length-k substrings of a string of length n. In particular, for a sufficiently large prime q such that $\log q \in \Theta(\log n)$, $\phi_{q,r}$ is collision-free over all length-k substrings of any fixed string y of length n with high probability. If, however, n is not known in advance, we take $\log q \in \Theta(w)$ instead, where w is the machine word size. We thus work in the word RAM model, where the word size always satisfies $w = \Omega(\log n)$, for any input of size n. We also assume that all standard arithmetic operations between $\mathcal{O}(w)$-bits integers take constant time. The following result is known.

▶ **Lemma 8** ([35]). *For any strings a and b, if we are given $\phi_{q,r}(a)$ and $\phi_{q,r}(b)$, then $\phi_{q,r}(ab)$ can be computed in $\mathcal{O}(1)$ time. If we are given $\phi_{q,r}(ab)$ and $\phi_{q,r}(a)$, then $\phi_{q,r}(b)$ can be computed in $\mathcal{O}(1)$ time.*

Proof. For any a and b, we have $\phi_{q,r}(ab) = (\phi_{q,r}(a)r^{|b|} + \phi_{q,r}(b)) \mod q$ and $\phi_{q,r}(b) = (\phi_{q,r}(ab) - \phi_{q,r}(a)r^{|b|}) \mod q$ by the rules of modular arithmetic. Thus we can compute $\phi_{q,r}(ab)$ from $\phi_{q,r}(a)$ and $\phi_{q,r}(b)$, and $\phi_{q,r}(b)$ from $\phi_{q,r}(ab)$ and $\phi_{q,r}(a)$ in $\mathcal{O}(1)$ time. ◀

3 String Existence Avoiding Forbidden Length-k Substrings

In this section we solve the SEFS problem: is there a string $x \in \Sigma^*$ such that u is a prefix of x, v is a suffix of x, and no $s \in S_k$ occurs in x, where $u, v \in \Sigma^*$ and S_k is a subset of Σ^k?

We start by showing how to solve SEFS efficiently when the strings u, v are of fixed length $k-1$. We will later show how to generalize this result to u and v of any length. We follow a graph-theoretic approach by modelling the problem in terms of complete de Bruijn graphs.

Recall that the complete de Bruijn graph of order k over an alphabet Σ is a directed graph $G_k = (V_k, E_k)$ with $V_k = \Sigma^{k-1}$ and $E_k = \{(u, v) \in V_k \times V_k \mid u[1] \cdot v = u \cdot v[k-1]\}$. A *path* in G_k is a finite sequence of elements from E_k, which joins a sequence of elements from V_k. The *de Bruijn sequence* of order k over Σ is a string in which each element of Σ^k occurs as a substring exactly once [33].

By reachability, we refer to a path in G_k, which starts with a fixed *starting node* u, its infix is a sought (possibly empty) *middle path*, and it ends with a fixed *ending node* v. We consider this notion of reachability in G_k in the presence of *forbidden edges* (or *failing edges*) represented by the set S_k of forbidden length-k substrings over alphabet Σ.

We say that a subgraph $G_k^S = (V_k^S, E_k^S)$ of a complete de Bruijn graph G_k *avoids* $S_k \subset \Sigma^k$ if it consists of all nodes of G_k and all edges of G_k but the ones that correspond to the strings in S_k, that is, if $V_k^S = V_k$ and $E_k^S = E_k \setminus \{(u, v) \in E_k \mid u \cdot v[k-1] \in S_k\}$. Given $u, v \in \Sigma^{k-1}$ and $S_k \subset \Sigma^k$, it can be readily verified that there is a bijection between strings in Σ^n with prefix u and suffix v that do not contain any strings in S_k and paths of length $n-k+1$ that start at u and end at v in G_k^S. In what follows, we show that one can quickly decide whether a node v is reachable from a node u (cf. the SEFS problem) by visiting only a limited portion of G_k^S, even though a shortest such path may be very long.

Our result relies on the notion of the *isoperimetric number* (a.k.a. Cheeger constant or conductance) of a graph. The isoperimetric number measures the "bottleneckedness" of a graph, that is, whether there is a way to partition the nodes into two sets such that the number of edges connecting the two is small compared to the size of the smaller set. More formally, given an undirected graph $G^{\mathrm{u}} = (V, E)$ and a subset of nodes $A \subset V$, the *edge boundary* of A is $\partial_{\mathrm{u}} A = \{\{x, y\} \in E \mid x \in A, y \in V \setminus A\}$: in other words, $\partial_{\mathrm{u}} A$ is the cut-set of the cut $(A, V \setminus A)$. The isoperimetric number of G^{u} is then $h_{\mathrm{u}}(G^{\mathrm{u}}) = \min_{1 \le |A| \le \frac{|V|}{2}} \frac{|\partial_{\mathrm{u}} A|}{|A|}$.[2]

Since we consider *directed* de Bruijn graphs, we will make use of the following notion of isoperimetric number, tailored for the directed case: $h(G) = \min_{1 \le |A| \le \frac{|V|}{2}} \frac{|\partial A|}{|A|}$, where $\partial A = \{(x, y) \in E \mid x \in A, y \in V \setminus A\}$ is the *directed edge boundary* of A, that is, the set of edges outgoing from A. The next lemma, which bounds the isoperimetric number $h(G)$ of a complete directed de Bruijn graph G, was given in [25, Lemma 3.5]. The proof is based on an analogous result on undirected de Bruijn graphs by Delorme and Tillich [23], and we report it here for completeness.

▶ **Lemma 9** ([25]). *Let $G_k = (V_k, E_k)$ be the complete de Bruijn graph of order k over an alphabet Σ. Then*

$$h(G_k) = \min_{1 \le |A| \le \frac{|V_k|}{2}} \frac{|\partial A|}{|A|} \ge \frac{|\Sigma|}{4(k-2)}, \tag{1}$$

where $A \subset V_k$ is a subset of nodes and ∂A is the edge boundary of A.

[2] In the literature, the isoperimetric number of G is also often denoted by $\phi(G)$.

Proof. Let $A \subseteq V_k$ be a cut. The nodes of A have exactly $|\Sigma| \cdot |A|$ ingoing and outgoing edges in total (including self-loops), as in a complete de Bruijn graph each node has exactly $|\Sigma|$ ingoing and $|\Sigma|$ outgoing edges. Let $E[A]$ be the set of edges of which both endpoints are in A. Then $|\partial A| = |\Sigma| \cdot |A| - |E[A]| = |\partial(V_k \setminus A)|$. Note that the undirected edge boundary of A is $\partial_{\mathrm{u}} A = \partial A \cup \partial(V_k \setminus A)$, and $|\partial A| = (|\partial A| + |\partial(V_k \setminus A)|)/2 \geq |\partial A \cup \partial(V_k \setminus A)|/2$. Therefore

$$h(G_k) = \min_{1 \leq |A| \leq \frac{|V_k|}{2}} \frac{|\partial A|}{|A|} \geq \frac{1}{2} \min_{1 \leq |A| \leq \frac{|V_k|}{2}} \frac{|\partial A \cup \partial(V_k \setminus A)|}{|A|} = h_{\mathrm{u}}(G_k^{\mathrm{u}})/2 \geq \frac{|\Sigma|}{4(k-2)},$$

where $h_{\mathrm{u}}(G_k^{\mathrm{u}})$ is the isoperimetric number of the simple, *undirected* version of the complete de Bruijn graph G_k, and the last inequality follows from [23, Theorem 9]. ◀

Main Idea. Lemma 9 states, roughly speaking, that complete de Bruijn graphs have quite a high isoperimetric number, meaning that the edge boundary of any subset of nodes is large compared to the number of nodes in the subset. This implies that eliminating (relatively) few edges from a complete de Bruijn graph is not enough to separate a (relatively) large set of nodes from the rest. Since our goal is to decide whether there exists a path in G_k^S connecting u and v, and S_k has the effect of removing a number of edges from the complete de Bruijn graph G_k, Lemma 9 implies that S_k is large enough to have u and v being separate only if either the set of nodes reachable from u or the set of nodes from which v can be reached is (relatively) small: Figure 1 illustrates this concept.

Our idea is thus to check whether the set of nodes reachable from u or the set of nodes from which v can be reached in G_k^S is small enough to have them separated by S_k or not. If these sets are small enough our algorithm answers NO; otherwise it answers YES. Let us remark that our input consists only of $u \in \Sigma^*$, $v \in \Sigma^*$, $k > 0$ and $S_k \subset \Sigma^k$, and thus G_k^S is not given explicitly: we will generate only the portion of G_k^S that is sufficient to decide SEFS.

The Algorithm

The algorithm relies on the bound given by Lemma 9. Rearranging the factors we get

$$|A| \leq \frac{4(k-2)}{|\Sigma|} |\partial A| \tag{2}$$

for all $A \subset V_k$ with $|A| \leq |V_k|/2$ in the complete de Bruijn graph $G_k = (V_k, E_k)$. Lemma 10 formalizes the main idea and gives a linear-time algorithm for deciding SEFS when $|u| = k-1$ and $|v| = k-1$. We then extend the algorithm for u or v of arbitrary length.

▶ **Lemma 10.** *Given an integer $k > 1$, two strings $u, v \in \Sigma^{k-1}$, and a set $S_k \subset \Sigma^k$, SEFS can be solved in $\mathcal{O}(k|S_k|)$ time.*

Proof. We start by generating the portion of $G_k^S = (V_k^S, E_k^S)$ that can be reached from u in a breadth-first fashion, thus generating an edge only if the corresponding length-k string is not in S_k. We stop when we cannot reach any new nodes or if we have reached $\lfloor \frac{4(k-2)|S_k|}{|\Sigma|} \rfloor + 1$ distinct nodes. Let A be this set of *reachable* nodes from u. If $|A| < \left\lfloor \frac{4(k-2)|S_k|}{|\Sigma|} \right\rfloor + 1$ then A contains all and only the nodes which can be reached from u. In this case there exists a path from u to v if and only if $v \in A$, i.e., v is one of the reachable nodes. Otherwise, we repeat the same procedure, this time using v as the starting point and traversing the edges backwards: let B be set of nodes we reached backwards from v. Again, if $|B| < \lfloor \frac{4(k-2)|S_k|}{|\Sigma|} \rfloor + 1$, then there exists a path from u to v if and only if $u \in B$.

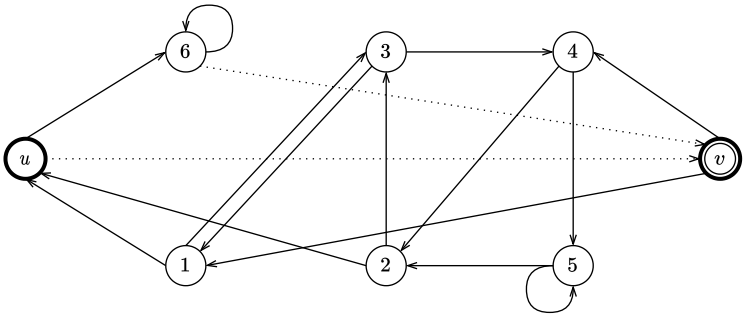


Figure 1 A schematic representation of the complete de Bruijn graph of order 4 over an alphabet of size 2. The dotted edges correspond to a set S_4 of size 2, showing that its size is enough to separate a set of up to two nodes from the rest, and thus, in this example, to make v unreachable from u. It can be easily verified that $|S_4| = 2$ is not enough to separate any 3 nodes from the rest.

Suppose however that $|A| = |B| = \lfloor \frac{4(k-2)|S_k|}{|\Sigma|} \rfloor + 1$, and thus A and B do not contain all the nodes that are reachable from u and from which can reach v, respectively. We claim that in this case v is always reachable from u. In fact, if A and B have a nonempty intersection, then clearly there exists a path from u to v. Otherwise, suppose for a contradiction that there does not exist any path from u to v in the whole G_k^S. Let $V_k \supseteq A' \supseteq A$ be set of nodes which are reachable from u in the whole G_k^S and let $V_k \supseteq B' \supseteq B$ be the set of nodes which can reach v. Since there is no path from u to v in G_k^S, the sets A' and B' are disjoint. Therefore one of them contains at most half the nodes, that is, $\lfloor \frac{4(k-2)|S_k|}{|\Sigma|} \rfloor + 1 \leq \min\{|A'|, |B'|\} \leq |V_k|/2$. Applying Equation (2) to the smallest of the two, we get $|\partial A'| > |S_k|$ or $|\partial B'| > |S_k|$. However note that, since there does not exist a path from u to v in G_k^S, $\partial A' \cup \partial B'^C \subseteq E_k \setminus E_k^S$, where $\partial B'^C$ denotes the directed edge boundary of the complement of B' in V_k. Since in a complete de Bruijn graph each node has the same number of incoming and outgoing edges, it holds $|\partial B'| = |\partial B'^C|$. This is a contradiction. Therefore, we conclude that there exits a path from u to v whenever $|A| = |B| = \lfloor \frac{4(k-2)|S_k|}{|\Sigma|} \rfloor + 1$.

Now that we have bounded the total number of nodes of G_k^S that are needed to decide SEFS, let us describe how we can generate them efficiently. We start by computing the KRF of every forbidden substring, and maintain them in a static dictionary $\mathsf{F}(S_k)$ [27]. This can be done in $\mathcal{O}(k|S_k|)$ time. During the whole process, we also maintain a dynamic dictionary GN [24], where we insert the KRFs of the generated nodes. From each generated node $w = w[1\,..\,k-1]$, we thus need to (i) compute the KRF of string $w\alpha$ corresponding to an outgoing edge in the complete de Bruijn graph G_k, for all $\alpha \in \Sigma$; (ii) check whether the KRF of string $w\alpha$, for all $\alpha \in \Sigma$, is in $\mathsf{F}(S_k)$; (iii) check which of the non-forbidden edges $w\alpha$ lead to a node $w[2\,..\,k-1]\alpha$ whose KRF is not in GN; and (iv) update GN by adding the KRFs of the latter nodes $w[2\,..\,k-1]\alpha$ to GN. Since the portion of G_k^S that we generate is connected, the strings of which we compute the KRFs at any step of this procedure can be obtained by appending α to w, and then by chopping off the first letter of w to obtain $w[2\,..\,k]\alpha$, for all $\alpha \in \Sigma$. By using Lemma 8, we can compute each such KRF in $\mathcal{O}(1)$ time per α, except for the first node, where we spend $\mathcal{O}(k)$ time. Since in G_k there are $|\Sigma|$ edges outgoing from each node, we can compute all KRFs at w and look them up in $\mathsf{F}(S_k)$ and in GN in $\mathcal{O}(|\Sigma|)$ time in total. The new nodes can be inserted in GN in $\mathcal{O}(|\Sigma|)$ time in total as well.

Since we generate $\mathcal{O}(k|S_k|/|\Sigma|)$ nodes in total, and since we spend $\mathcal{O}(|\Sigma|)$ time to process each such node, the overall time required by the above algorithm is $\mathcal{O}(k|S_k|)$. ◀

Let us now discuss the case where u or v are not of fixed length $k-1$: note that we still consider forbidden substrings of fixed length k, which determines the order of G_k. In particular, we consider the case where u or v are of length smaller than $k-1$ (Case 1) and the case where u or v are longer than $k-1$ (Case 2). We finally combine all possible cases.

Case 1: $|u| < k-1$ or $|v| < k-1$. When $|u| < k-1$ (resp. $|v| < k-1$) we should add to the set of reachable nodes A (resp. to B) all the nodes that have u as a suffix (resp. v as a prefix), and then generate the portion of G_k^S reachable from each of them (resp. that can reach each of them), adding the new reached nodes to A (resp. to B) until either its size exceeds the bound given by Lemma 9 or we cannot find any new nodes, again as we described in the proof of Lemma 10.

To show that this can be done efficiently, let us start by observing that the number of nodes from which we start generating the graph increases when the length of u (resp. of v) decreases. As a consequence, for u or v short enough we can decide the problem in constant time. Indeed, if $\max(|u|,|v|) < (k-1) - \log_{|\Sigma|}\left(\frac{4(k-1)|S_k|}{|\Sigma|}\right)$, then the answer is always positive. This is because of a simple counting argument: let us focus on u, as the argument for v is the same. The number of nodes of G_k of which u is a suffix is $|\Sigma|^{(k-1)-|u|}$, which is greater than $\frac{4(k-1)|S_k|}{|\Sigma|}$ whenever $|u| < (k-1) - \log_{|\Sigma|}\left(\frac{4(k-1)|S_k|}{|\Sigma|}\right)$, implying the existence of a path from u to v.

We thus only need to consider the case where u (resp. v) is of length between $(k-1) - \log_{|\Sigma|}\left(\frac{4(k-1)|S_k|}{|\Sigma|}\right)$ and $k-2$. Let us focus on u, as the procedure for v is entirely analogous. Let $d = (k-1) - |u|$. The starting nodes that we need to generate are all and only the length-$(k-1)$ strings over Σ of the form pu, where p is one of the $|\Sigma|^d$ strings of length $k-1-|u|$ over Σ. To obtain them and compute their KRFs efficiently we proceed as follows. We first compute the KRF of u in $\mathcal{O}(|u|) = \mathcal{O}(k)$ time. We then construct the de Bruijn sequence of order d over Σ in time $\mathcal{O}(|\Sigma|^d)$ [33], which is in $\mathcal{O}(k|S_k|/|\Sigma|)$ as we are considering $|u| > (k-1) - \log_{|\Sigma|}\left(\frac{4(k-1)|S_k|}{|\Sigma|}\right)$. We then use a sliding window of size d over the de Bruijn sequence to compute the KRFs of its length-d fragments in overall $\mathcal{O}(|\Sigma|^d) = \mathcal{O}(k|S_k|/|\Sigma|)$ time using Lemma 8. For each length-d fragment p, we apply Lemma 8 again to compute the KRF of node pu in $\mathcal{O}(1)$ time from the KRFs of u and p. Therefore the whole algorithm takes $\mathcal{O}(k|S_k|/|\Sigma|)$ time to generate the starting nodes and compute their KRFs, plus $\mathcal{O}(k|S_k|)$ time to apply the algorithm described in the proof of Lemma 10 from these starting nodes.

Case 2: $|u| > k-1$ or $|v| > k-1$. When $|u| > k-1$ (resp. $|v| > k-1$) it suffices to construct the path which spells u (resp. v) in G_k^S and run the algorithm described in the proof of Lemma 10 from the last node of the path (resp. from the first node). While constructing the path, we compute the KRFs of the length-k substrings of u (resp. v) and check on the dictionary of forbidden substrings whether they are forbidden. If any substring of u (resp. v) is forbidden, then the answer to SEFS is clearly NO. This process requires $\mathcal{O}(|u|)$ (resp. $\mathcal{O}(|v|)$) time in addition to the time required by Lemma 10.

The correctness of the algorithm follows by Lemma 9. Then combining Lemma 10, Case 1, and Case 2 leads to the main result of this section.

▶ **Theorem 2.** *Given an integer $k > 1$, two strings $u, v \in \Sigma^*$, and a set $S_k \subset \Sigma^k$, SEFS can be solved in $\mathcal{O}(|u| + |v| + k|S_k|)$ time.*

▶ Remark 11. The algorithm for obtaining Theorem 2 is Monte Carlo whp due to the use of KRFs and dynamic dictionaries.

Reachability in Complete de Bruijn Graphs

Note that since G_k is complete, it can be specified by k and Σ in $\mathcal{O}(1)$ machine words. Let us now formally define the following reachability problem on complete de Bruijn graphs.

Reachability in de Bruijn Graphs Avoiding Forbidden Edges (RFE)
Input: The complete de Bruijn graph $G_k = (V_k, E_k)$ of order $k > 1$ over an alphabet Σ, nodes $u, v \in V_k$, and a set $S_k \subset E_k$.
Output: YES if there exists a path from u to v avoiding any $e \in S_k$; or NO otherwise.

Lemma 10 directly translates to the following corollary.

▶ **Corollary 12.** *Given the complete de Bruijn graph $G_k = (V_k, E_k)$ of order k over an alphabet Σ, nodes $u, v \in V_k$, and a set $S_k \subset E_k$, RFE can be solved in $\mathcal{O}(k|S_k|)$ time.*

4 Shortest String Avoiding Forbidden Substrings

In this section we solve the SSFS problem: construct a shortest $x \in \Sigma^*$ such that u is a prefix of x, v is a suffix of x, and no $s \in S$ occurs in x, where $u, v \in \Sigma^*$ and $S \subset \Sigma^*$.

Let $||S|| = \sum_{s \in S} |s|$. We make the standard assumption that Σ is a subset of $[1, |u| + |v| + ||S|| + 1]$. If this is not the case, we use a static dictionary [27] to do so in $\mathcal{O}(|u| + |v| + ||S||)$ time. Note that all letters of Σ which are neither in u or in v nor in one of the strings in S are interchangeable. They can therefore all be replaced by a single new letter, reducing the alphabet to a size of at most $|u| + |v| + ||S|| + 1$. We will henceforth assume that Σ is such a reduced alphabet. Further note that the input size of the SSFS instance is $(||S|| + |u| + |v|) \log |\Sigma|$ bits or $||S|| + |u| + |v|$ machine words. We further assume that set S is *anti-factorial*, i.e., no $s_1 \in S$ is a proper substring of another element $s_2 \in S$. If that is not the case, we take the set without such s_2 elements to be S. This can be done in $\mathcal{O}(||S||)$ time by constructing the generalized suffix tree of the original S after reducing Σ [26].

Main Idea. We say that a string y is *S-dangerous* if $y = \varepsilon$ or y is a proper prefix of an $s \in S$; we drop S from S-dangerous when this is clear from the context. We aim to construct a labeled directed graph $G(D, E)$ as follows. The set of nodes is the set D of dangerous strings. There exists a directed edge labeled with $\alpha \in \Sigma$ in the set E of edges from node w_1 to node w_2, if $w_1\alpha$ is not in S and w_2 is the longest dangerous suffix of $w_1\alpha$.

Recall that u and v must be a prefix and a suffix of the string x we need to construct, respectively. We set the longest dangerous string in D that is a suffix of u to be the *source* node. We set every node w, such that wv does not contain a string of S, to be a *sink* node. A shortest path from the source node to any sink node corresponds to a shortest such x, where u and v are not allowed to overlap. The overlap case is treated separately.

The Algorithm

The algorithm has two main stages. In the first stage, we construct the graph $G(D, E)$. In the second stage, we find the source and the sinks, and we construct a shortest string x.

Crochemore, Mignosi and Restivo [21] showed how to construct a complete DFA accepting strings over Σ, which do not contain any forbidden substring from S. This is precisely the directed graph $G(D, E)$.[3] We show that this DFA has $\Theta(||S||)$ states and $\Theta(||S|| \cdot |\Sigma|)$ edges

[3] In what follows, we use the term graph and automaton interchangeably, depending on the context.

in the worst case. If we take this DFA and multiply it with the automaton accepting strings of the form uwv, with $w \in \Sigma^*$, we get another automaton accepting all strings of length at least $|u| + |v|$ starting with u, ending with v and not containing any element $s \in S$ as a substring. Since this product automaton would have $\mathcal{O}((|u| + |v|)||S||)$ nodes, we will instead show an efficient way to compute the appropriate source and sink nodes on $G(D, E)$ in $\mathcal{O}(|u|+|v|+||S||)$ time resulting in a total time cost of $\mathcal{O}(|u|+|v|+||S|| \cdot |\Sigma|)$ (Theorem 3). We start by showing how $G(D, E)$ can be constructed efficiently for completeness (see also [21]).

Constructing the Graph

First, we construct the trie of the strings in S. This takes $\mathcal{O}(||S||)$ time [20]. We merge the leaf nodes, which correspond to the strings in S, into one forbidden node s'. Note that all other nodes correspond to dangerous strings. We can therefore identify the set of nodes with $D' = D \cup \{s'\}$. We turn this into an automaton by computing a transition function $\delta : D' \times \Sigma \to D'$, which sends each pair $(w, \alpha) \in D \times \Sigma$ to the longest dangerous or forbidden suffix of $w\alpha$ and $(s', \alpha) \in \{s'\} \times \Sigma$ to s'. We can then draw the edges corresponding to the transitions to obtain the graph $G(D, E)$. To help constructing this transition function, we also define a failure function $f : D \to D$ that sends each dangerous string to its longest proper dangerous suffix, which is well-defined because the empty string ε is always dangerous.

In the trie, we already have the edges corresponding to $\delta(w, \alpha) = w\alpha$ if $w\alpha \in D'$. We first add $\delta(s', \alpha) = s'$, for all $\alpha \in \Sigma$. For the failure function, note that $f(\varepsilon) = f(\alpha) = \varepsilon$.

To find the remaining values, we traverse the trie in a breadth first search manner. Let w be an internal node of the trie, that is, a dangerous string of length $\ell > 0$. Then

$$f(w) = \delta(f(w[1\,..\,\ell-1]), w[\ell]) \text{ and } \delta(w, \alpha) = \begin{cases} w\alpha & \text{if } w\alpha \in D' \\ \delta(f(w), \alpha) & \text{if } w\alpha \notin D' \end{cases}.$$

Note that this is well defined, because $w[1\,..\,\ell-1]$ and $f(w)$ are dangerous strings shorter than w, so the corresponding function values are already known.

Once we have computed the transition function and created the corresponding automaton, we delete s' and all its edges thus obtaining $G(D, E)$. To ensure that we can access the node $\delta(w, \alpha)$ in constant time we use a static dictionary on the nodes of $G(D, E)$ [27]. We can alternatively implement the transition functions by arrays in $\Theta(|\Sigma|)$ space per array.

Observe that we need to traverse $|D|$ nodes and compute $|\Sigma| + 1$ function values at each node (one value for f and $|\Sigma|$ values for δ). Every function value is computed in constant time, and therefore the total time complexity of the construction step is $\mathcal{O}(||S|| + |D| \cdot |\Sigma|)$.

▶ **Lemma 13.** *$G(D, E)$ has $\Omega(||S||)$ states and $\Omega(||S|| \cdot |\Sigma|)$ edges in the worst case. $G(D, E)$ can be constructed in the optimal $\mathcal{O}(||S|| \cdot |\Sigma|)$ time.*

Proof. For the first part, consider the instance where S consists of all strings of the form ww with $w \in \Sigma^{k'}$. Then the input size is $\|S\| = 2k'|\Sigma|^{k'}$, while there are more than $k'|\Sigma|^{k'}$ states and $k'|\Sigma|^{k'+1}$ edges. The second part follows from the above discussion (see also [21]). ◀

Constructing a Shortest String

To find the source node, that is, the longest dangerous string that is a suffix of u, we start at the node of $G(D, E)$ that used to be the root of the trie, which corresponds to ε, and follow the edges labeled with the letters of u one by one. This takes $\mathcal{O}(|u|)$ time. Finding the sink nodes directly is more challenging. The trick is to compute the *non-sink nodes* first instead. The non-sink nodes are those dangerous strings $d \in D$ such that the string dv has a

forbidden string $s \in S$ as a substring. To this end, we construct the generalized suffix tree of the strings in S. Recall that this is the compressed trie containing all suffixes of all forbidden strings in S. This takes $\mathcal{O}(||S||)$ time [26]: let us remark that this step has to be done only once. In order to access the children of an explicit suffix tree node by the first letter of their edge label a static dictionary is used [27]. We then find, for each nonempty prefix p of v, all suffixes of all forbidden substrings that are equal to p. We do that by spelling v from the root of the suffix tree of S. There are no more than $||S||$ such suffixes in total, thus the whole process takes $\mathcal{O}(|v| + ||S||)$ time. For each such suffix p, we set the prefix q of the corresponding forbidden substring to be a non-sink node, i.e., we have that $qp \in S$, q is a non-sink, and p is a prefix of v. Recall that all proper prefixes of the elements of S are nodes of $G(D, E)$, and so this is well defined. Any other node is set to be a sink node.

We next consider two cases: $|x| \leq |u| + |v|$ (Case 1) and $|x| \geq |u| + |v|$ (Case 2).

Case 1: $|x| \leq |u| + |v|$. In this case, u and v have a suffix/prefix overlap: a nonempty suffix of u is a prefix of v. We can compute the lengths of all possible suffix/prefix overlaps in $\mathcal{O}(|u| + |v|)$ time and $\mathcal{O}(|u|)$ space by, for instance, constructing the suffix tree of u and spelling the prefixes of v from the root. In order to access the children of an explicit suffix tree node by the first letter of their edge label, a static dictionary is used [27]. We still have to check whether the strings created by such suffix/prefix overlaps contain any forbidden substrings. We do that by starting at ε in $G(D, E)$ and following the edges corresponding to u one by one. If we are at a sink node after following i edges and we have that $u[i + 1 \,..\, |u|] = v[1 \,..\, |u| - i]$, then we output $x = u[1 \,..\, i]v$ and halt. Processing all these edges takes $\mathcal{O}(|u|)$ time.

Case 2: $|x| \geq |u| + |v|$. Suppose that Case 1 did not return any feasible path. We then use breadth first search on $G(D, E)$ from the source node to the *nearest* sink node to find a path. If we are at a sink node after following a path spelling string h, then we output $x = uhv$ and halt. In the worst case, we traverse the whole $G(D, E)$. It takes $\mathcal{O}(|E|) = \mathcal{O}(|D| \cdot |\Sigma|)$ time.

In case no feasible path is found in $G(D, E)$, we report FAIL.

Correctness

The paths we find in the algorithm correspond exactly to strings p such that $x = pv$ has u as a prefix and x does not contain any forbidden substrings. Since we search for these paths in order of increasing length, the algorithm will find the shortest p and hence the shortest x, if it exists or report FAIL otherwise.

Complexities

Constructing $G(D, E)$ takes $\mathcal{O}(||S|| + |D| \cdot |\Sigma|)$ time (see also [21]). Finding the source and all sink nodes takes $\mathcal{O}(|u| + |v| + ||S||)$ time. Checking Case 1 takes $\mathcal{O}(|u| + |v|)$ time. Checking Case 2 takes $\mathcal{O}(|D| \cdot |\Sigma|)$ time. It should also be clear that the following bound on the size of the output holds: $|x| \leq |u| + |v| + |D|$. The total time complexity of the algorithm is thus

$$\mathcal{O}(||S|| + |u| + |v| + |D| \cdot |\Sigma|) = \mathcal{O}(|u| + |v| + ||S|| \cdot |\Sigma|).$$

▶ Remark 14. By symmetry, we can obtain a time complexity of $\mathcal{O}(||S|| + |u| + |v| + |D_s| \cdot |\Sigma|)$, where D_s is the set including ε and the proper suffixes of forbidden substrings.

The algorithm uses $\mathcal{O}(||S|| \cdot |\Sigma| + |u|)$ working space, which is the space occupied by $G(D, E)$ and the suffix tree of u.

We obtain the main result of this section.

▶ **Theorem 3.** *Given two strings $u, v \in \Sigma^*$, and a set $S \subset \Sigma^*$, SSFS can be solved in $\mathcal{O}(|u| + |v| + ||S|| \cdot |\Sigma|)$ time, where $||S|| = \sum_{s \in S} |s|$, using $\mathcal{O}(|u| + ||S|| \cdot |\Sigma|)$ space.*

▶ Remark 15. The algorithm for obtaining Theorem 3 is Las Vegas whp due to the use of static dictionaries, which is a standard assumption in algorithms with large alphabets. If $|\Sigma|$ is polynomially bounded in the input size, it can be made deterministic at no extra cost.

A Full Example

In Figure 2, we illustrate the difference between the de Bruijn graph perspective and the automaton perspective. Let $u =$ `ab`, $v =$ `ca`, and $S = \{$`bc`$\}$. We start at node ε of the automaton. After processing u we are at source node `b`. The suffix tree of S contains suffixes `c` and `bc`. Since `c` is a prefix of v, the complementary prefix `b` of the forbidden substring `bc` is a non-sink and thus ε a sink. Note that u and v do not have any suffix/prefix overlap. Hence we use breadth first search (Case 2). The shortest path from source `b` to sink ε is `a`. Therefore $x =$ `abaca` is a shortest string with prefix u and suffix v not containing any substring from S.

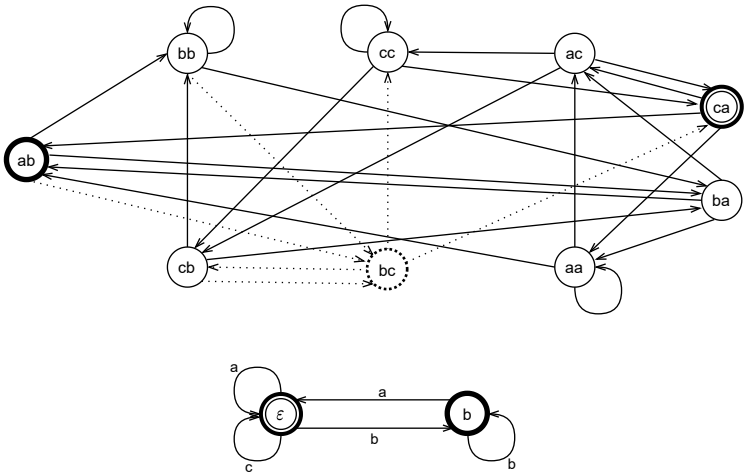


Figure 2 Recall that we have one forbidden substring, `bc`, of length 2. One could then construct the complete de Bruijn graph of order 3 over alphabet $\Sigma =$ `{a,b,c}` (*Top*); and find the sequence of nodes `ab` $\rightarrow$ `ba` $\rightarrow$ `ac` $\rightarrow$ `ca`, which gives a shortest path starting from node `ab`, ending at node `ca`, and avoiding the forbidden node `bc`. (*Bottom*) the graph $G(D, E)$ after we have computed the source node `b` and the only sink node ε. The shortest path is then `a` which gives $x =$ `ab` $\cdot$ `a` $\cdot$ `ca`.

Shortest Path in Complete de Bruijn Graphs

We also consider the following more general optimization version of the RFE problem, the reachability problem in complete de Bruijn graphs:

Shortest Path in de Bruijn Graphs Avoiding Forbidden Edges (SPFE)
Input: The complete de Bruijn graph $G_k = (V_k, E_k)$ of order $k > 1$ over an alphabet Σ, nodes $u, v \in V_k$, and a set $S_k \subset E_k$.
Output: A shortest path from u to v avoiding any $e \in S_k$; or FAIL if no such path exists.

Note that SPFE is a special case of SSFS. Theorem 3 yields the following corollary.

▶ **Corollary 16.** *Given the complete de Bruijn graph $G_k = (V_k, E_k)$ of order k over an alphabet Σ, nodes $u, v \in V_k$, and a set $S_k \subset E_k$, SPFE can be solved in $\mathcal{O}(k|S_k| \cdot |\Sigma|)$ time.*

5 Shortest Fully-Sanitized String

In this section, we show how to solve the SFSS problem: given $w \in \Sigma^n$ and $S_k \subset \Sigma^k$, construct a shortest $y \in \Sigma^*$ such that no $s \in S_k$ occurs in y and the sequence of non-forbidden length-k substrings of w is a subsequence of the sequence of the length-k substrings of y.

We show that Corollary 16 can be applied on the output of the TFS problem, which we denote by string x, to solve the SFSS problem. Recall from the introduction that x is a shortest string such that no string in S_k occurs in x and the order of all other length-k fragments over Σ is the same in w and in x. In [9], we showed that x is unique and it is always of the form $x = x_0\#_1x_1\#_2\cdots\#_dx_d$, with $x_i \in \Sigma^*$ and $|x_i| \geq k$. It is easy to see why: if we had an occurrence of $\#_ix_i\#_{i+1}$ with $|x_i| \leq k-1$ in x then we could have deleted $\#_ix_i$ to obtain a shorter string x, which is a contradiction. Let us summarize the results related to string x from [9].

▶ **Theorem 17** ([9])**.** *Let x be a solution to the TFS problem. Then x is unique, it is of the form $x = x_0\#_1x_1\#_2\cdots\#_dx_d$, with $x_i \in \Sigma^*$, $|x_i| \geq k$, and $d \leq n$, it can be constructed in the optimal $\mathcal{O}(n + |x|)$ time, and $|x| = \Theta(nk)$ in the worst case.*

Since $|x_i| \geq k$, each # replacement in x with a letter from Σ can be treated *separately*. In particular, an instance $x_i\#_{i+1}x_{i+1}$ of this problem can be formulated as a shortest path problem in the complete de Bruijn graph of order k over alphabet Σ in the presence of forbidden edges. Corollary 16 can thus be applied d times on $x = x_0\#_1x_1\#_2\cdots\#_dx_d$ to replace the d occurrences of # in x and obtain a final string over Σ: given an instance $x_i\#_{i+1}x_{i+1}$, we set u to be the length-$(k-1)$ suffix of x_i and v to be the length-$(k-1)$ prefix of x_{i+1}. Let us denote by y the string obtained by this algorithm.

▶ **Example 18.** Let $w = \texttt{abbbbaaabaa}$, $\Sigma = \{\texttt{a}, \texttt{b}\}$, $k = 4$, and $S_k = \{\texttt{bbbb}, \texttt{aaba}, \texttt{abba}\}$ (the instance from Example 4), and the solution $x = \texttt{abbbaaab\#abaa}$ of the TFS problem. By setting $u = \texttt{aab}$ and $v = \texttt{aba}$ in the SPFE problem, we obtain as output the path corresponding to string $p = \texttt{aabbbaba}$. The prefix $\texttt{aab}$ of p corresponds to the starting node u, its infix $\texttt{bb}$ corresponds to the middle path found, and its suffix $\texttt{aba}$ corresponds to the ending node v. We use p to replace $\texttt{aab\#aba}$ and obtain the final string $y = \texttt{abbbaaabbbabaa}$.

However, to prove that y is a solution to the SFSS problem, we further need to prove that $\mathcal{S}(w, S_k)$ is a subsequence of $\mathcal{S}(y, S_k)$, and that y is a shortest possible such string.

▶ **Lemma 19.** *Let $x = x_0\#_1x_1\#_2\cdots\#_dx_d$, with $x_i \in \Sigma^*$ and $|x_i| \geq k$, be a solution to the TFS problem on a string w, and let y be the string obtained by applying Corollary 16 d times on x to replace the occurrences of #. String y is a shortest string over Σ such that $\mathcal{S}(w, S_k)$ is a subsequence of $\mathcal{S}(y, S_k)$ and no $s \in S_k$ occurs in y.*

Proof. No $s \in S_k$ occurs in y by construction. With a slight abuse of notation, let $\mathcal{S}(x, S_k)$ be the sequence of length-k substrings over Σ occurring in x from left to right. Since x is a solution to the TFS problem, we have that $\mathcal{S}(x, S_k) = \mathcal{S}(w, S_k)$. Since $x_0, x_1, \ldots, x_d$ occur in y in the same order as in x by construction, it follows that $\mathcal{S}(x, S_k) = \mathcal{S}(w, S_k)$ is a subsequence of $\mathcal{S}(y, S_k)$. We now need to show that there does not exist another string y' shorter than y such that $\mathcal{S}(w, S_k)$ is a subsequence of $\mathcal{S}(y', S_k)$ and no $s \in S_k$ occurs in y'. Suppose for a contradiction that such a shorter string y' does exist. Since x is the shortest string such that $\mathcal{S}(x, S_k) = \mathcal{S}(w, S_k)$ and no $s \in S_k$ occurs in x, all the length-k substrings of $x_0, x_1, \ldots, x_d$ are also a subsequence of $\mathcal{S}(y', S_k)$ by hypothesis.

Let $y[\ell_i \,..\, r_i]$ and $y'[\ell'_i \,..\, r'_i]$ be the shortest substrings of y and y', respectively, where the length-k substrings of x_i and x_{i+1} appear and such that $|y[\ell_i \,..\, r_i]| > |y'[\ell'_i \,..\, r'_i]|$ (there must be an i such that this is the case, as we supposed $|y| > |y'|$). Since $y[\ell_i \,..\, r_i]$ is obtained by applying Corollary 16 to the length-$(k-1)$ suffix of x_i and the length-$(k-1)$ prefix of x_{i+1}, it is a shortest string that has x_i as a prefix and x_{i+1} as a suffix, implying that $y'[\ell'_i \,..\, r'_i]$ can only be shorter if it is not of the same form: have x_i as a prefix and x_{i+1} as a suffix. Suppose then that x_i is not a prefix of $y'[\ell'_i \,..\, r'_i]$, and thus there exist two consecutive length-k substrings of x_i that are not consecutive in $y'[\ell'_i \,..\, r'_i]$. But then it is always possible to remove any letters between the two in $y'[\ell'_i \,..\, r'_i]$ to make them consecutive and obtain a string shorter than $y'[\ell'_i \,..\, r'_i]$. This operation does not introduce any occurrences of some $s \in S_k$, as the two length-k substrings are consecutive in x_i which, in turn, does not contain any $s \in S_k$. By repeating this reasoning on any two length-k substrings of x_i and x_{i+1}, we obtain a string y''_i that has x_i as a prefix, x_{i+1} as a suffix and such that $|y''_i| < |y'[\ell'_i \,..\, r'_i]| < |y[\ell_i \,..\, r_i]|$. This is a contradiction, as $y[\ell_i \,..\, r_i]$ is a shortest string that has x_i as a prefix and x_{i+1} as a suffix. ◀

By Theorem 17, Corollary 16, and Lemma 19 we obtain the main result of this section.

▶ **Theorem 7.** *Given a string w of length n over an alphabet Σ, an integer $k > 1$, and a set $S_k \subset \Sigma^k$, SFSS can be solved in $\mathcal{O}(nk|S_k| \cdot |\Sigma|)$ time.*

▶ Remark 20. The algorithm for obtaining Theorem 7 is Las Vegas whp due to the use of Corollary 16 (which relies on Theorem 3). If $|\Sigma|$ is polynomially bounded in the size of the input, the algorithm can be made deterministic at no extra cost.

We stress that the fact that y is the shortest possible is important for utility. Let $G(x)[v]$ denote the total number of occurrences of string v in string x. The *k-gram profile* of x is the vector $G_k(x) = (G(x)[v]), v \in \Sigma^k$. The *$k$-gram distance* between two strings is defined as the L_1-norm of the difference of their k-gram profiles. The k-gram distance is a pseudo-metric that is widely used (especially in bioinformatics), because it can be computed in linear time in the sum of the lengths of the two strings [41]. It is now straightforward to see that the k-gram distance between strings w (input of SFSS) and y (output of SFSS), such that $\mathcal{S}(w, S_k)$ is a subsequence of $\mathcal{S}(y, S_k)$ and no $s \in S_k$ occurs in y, is minimal. Thus, conceptually, SFSS introduces in y the least amount of spurious information.

6 Open Questions

We leave the following questions unanswered:

1. Can the SEFS problem be solved deterministically in $\mathcal{O}(|u| + |v| + k|S_k|)$ time? One could investigate whether the use of KRFs and dynamic dictionaries can be avoided.

2. Can the SSFS problem be solved faster than $\mathcal{O}(|u| + |v| + ||S|| \cdot |\Sigma|)$ time? One should perhaps design a fundamentally different technique that avoids the DFA construction, because, as we have shown, the latter has $\Omega(||S||)$ states and $\Omega(||S|| \cdot |\Sigma|)$ edges.
3. Sometimes we may want to solve many instances of the SSFS problem having the same set of forbidden substrings but different u and v; for example, in the SFSS problem (see Section 5). Can we solve q such instances faster than applying Theorem 3 q times?

References

1 Osman Abul, Francesco Bonchi, and Fosca Giannotti. Hiding sequential and spatiotemporal patterns. *IEEE Trans. Knowl. Data Eng.*, 22(12):1709–1723, 2010. doi:10.1109/TKDE.2009.213.

2 Osman Abul and Harun Gökçe. Knowledge hiding from tree and graph databases. *Data Knowl. Eng.*, 72:148–171, 2012. doi:10.1016/j.datak.2011.10.002.

3 Surender Baswana, Keerti Choudhary, Moazzam Hussain, and Liam Roditty. Approximate single-source fault tolerant shortest path. *ACM Trans. Algorithms*, 16(4):44:1–44:22, 2020. doi:10.1145/3397532.

4 Surender Baswana, Keerti Choudhary, and Liam Roditty. Fault tolerant reachability for directed graphs. In *DISC*, pages 528–543, 2015. doi:10.1007/978-3-662-48653-5_35.

5 Surender Baswana, Keerti Choudhary, and Liam Roditty. Fault-tolerant subgraph for single-source reachability: General and optimal. *SIAM J. Comput.*, 47(1):80–95, 2018. doi:10.1137/16M1087643.

6 Surender Baswana and Neelesh Khanna. Approximate shortest paths avoiding a failed vertex: Near optimal data structures for undirected unweighted graphs. *Algorithmica*, 66(1):18–50, 2013. doi:10.1007/s00453-012-9621-y.

7 Surender Baswana, Utkarsh Lath, and Anuradha S. Mehta. Single source distance oracle for planar digraphs avoiding a failed node or link. In *SODA*, pages 223–232, 2012. doi:10.1137/1.9781611973099.20.

8 Marie-Pierre Béal, Maxime Crochemore, Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Computing forbidden words of regular languages. *Fundam. Informaticae*, 56(1-2):121–135, 2003. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi56-1-2-08.

9 Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. String sanitization: A combinatorial approach. In *ECML PKDD*, volume 11906, pages 627–644, 2019. doi:10.1007/978-3-030-46150-8_37.

10 Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giovanna Rosone, and Michelle Sweering. Combinatorial algorithms for string sanitization. *ACM Trans. Knowl. Discov. Data*, 15(1):8:1–8:34, 2020. doi:10.1145/3418683.

11 Giulia Bernardini, Huiping Chen, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Leen Stougie, and Michelle Sweering. String sanitization under edit distance. In *CPM*, pages 7:1–7:14, 2020. doi:10.4230/LIPIcs.CPM.2020.7.

12 Giulia Bernardini, Alessio Conte, Garance Gourdel, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon Pissis, Giulia Punzi, Leen Stougie, and Michelle Sweering. Hide and mine in strings: Hardness and algorithms. In *ICDM*, pages 924–929, 2020. doi:10.1109/ICDM50108.2020.00103.

13 Luca Bonomi, Liyue Fan, and Hongxia Jin. An information-theoretic approach to individual sequential data sanitization. In *WSDM*, pages 337–346, 2016. doi:10.1145/2835776.2835828.

14 Andrei Z. Broder, Danny Dolev, Michael J. Fischer, and Barbara Simons. Efficient fault-tolerant routings in networks. *Inf. Comput.*, 75(1):52–64, 1987. doi:10.1016/0890-5401(87)90063-0.

15 Pascal Caron. Families of locally testable languages. *Theoretical Computer Science*, 242(1):361–376, 2000. doi:10.1016/S0304-3975(98)00332-6.

16 Panagiotis Charalampopoulos, Shay Mozes, and Benjamin Tebeka. Exact distance oracles for planar graphs with failing vertices. In *SODA*, pages 2110–2123, 2019. doi:10.1137/1.9781611975482.127.

17 Keerti Choudhary. An optimal dual fault tolerant reachability oracle. In *ICALP*, pages 130:1–130:13, 2016. doi:10.4230/LIPIcs.ICALP.2016.130.

18 Chris Clifton and Don Marks. Security and privacy implications of data mining. In *SIGMOD*, pages 15–19, 1996.

19 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

20 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

21 Maxime Crochemore, Filippo Mignosi, and Antonio Restivo. Automata and forbidden words. *Inf. Process. Lett.*, 67(3):111–117, 1998. doi:10.1016/S0020-0190(98)00104-5.

22 Nicolaas Govert de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie V. Wetenschappen*, 49:758–764, 1946.

23 C. Delorme and J.-P. Tillich. The spectrum of de Bruijn and Kautz graphs. *European Journal of Combinatorics*, 19(3):307–319, 1998. doi:10.1006/eujc.1997.0183.

24 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *ICALP*, pages 6–19, 1990. doi:10.1007/BFb0032018.

25 Igor Dolinka. On free spectra of locally testable semigroup varieties. *Glasgow Mathematical Journal*, 53(3):623–629, 2011. doi:10.1017/S0017089511000188.

26 Martin Farach. Optimal suffix tree construction with large alphabets. In *FOCS*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.

27 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

28 Aris Gkoulalas-Divanis and Grigorios Loukides. Revisiting sequential pattern hiding to enhance utility. In *SIGKDD*, pages 1316–1324, 2011. doi:10.1145/2020408.2020605.

29 Aris Gkoulalas-Divanis and Vassilios S. Verykios. An integer programming approach for frequent itemset hiding. In *CIKM*, pages 748–757, 2006. doi:10.1145/1183614.1183721.

30 Aris Gkoulalas-Divanis and Vassilios S. Verykios. Exact knowledge hiding through database extension. *IEEE Trans. Knowl. Data Eng.*, 21(5):699–713, 2009. doi:10.1109/TKDE.2008.199.

31 Robert Gwadera, Aris Gkoulalas-Divanis, and Grigorios Loukides. Permutation-based sequential pattern hiding. In *ICDM*, pages 241–250, 2013. doi:10.1109/ICDM.2013.57.

32 Giuseppe F. Italiano, Adam Karczmarz, and Nikos Parotsidis. Planar reachability under single vertex or edge failures, 2021. doi:10.1137/1.9781611976465.163.

33 L.Ro Ford Jr. A cyclic arrangement of m-tuples. Technical Report Report P-1071, Rand Corporation, 1957.

34 Tiko Kameda. On the vector representation of the reachability in planar directed graphs. *Inf. Process. Lett.*, 3(3):75–77, 1975. doi:10.1016/0020-0190(75)90019-8.

35 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.

36 Young-Jin Kim, Ramesh Govindan, Brad Karp, and Scott Shenker. Geographic routing made practical. In *NSDI*, 2005. URL: http://www.usenix.org/events/nsdi05/tech/kim.html.

37 Grigorios Loukides and Robert Gwadera. Optimal event sequence sanitization. In *ICDM*, pages 775–783, 2015. doi:10.1137/1.9781611974010.87.

38 Stanley R. M. Oliveira and Osmar R. Zaïane. Protecting sensitive knowledge by data sanitization. In *ICDM*, pages 613–616, 2003. doi:10.1109/ICDM.2003.1250990.

39 Steven Skiena. *The Algorithm Design Manual, Third Edition*. Texts in Computer Science. Springer, 2020. doi:10.1007/978-3-030-54256-6.

40 Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004. doi:10.1145/1039488.1039493.

41 Esko Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, 1992. doi:10.1016/0304-3975(92)90143-4.

42 Vassilios S. Verykios, Ahmed K. Elmagarmid, Elisa Bertino, Yücel Saygin, and Elena Dasseni. Association rule hiding. *IEEE Trans. Knowl. Data Eng.*, 16(4):434–447, 2004. doi:10.1109/TKDE.2004.1269668.

43 Yi-Hung Wu, Chia-Ming Chiang, and Arbee L. P. Chen. Hiding sensitive association rules with limited side effects. *IEEE Trans. Knowl. Data Eng.*, 19(1):29–42, 2007. doi:10.1109/TKDE.2007.250583.

Gapped Indexing for Consecutive Occurrences

Philip Bille
Technical University of Denmark, DTU Compute, Lyngby, Denmark

Inge Li Gørtz
Technical University of Denmark, DTU Compute, Lyngby, Denmark

Max Rishøj Pedersen
Technical University of Denmark, DTU Compute, Lyngby, Denmark

Teresa Anna Steiner
Technical University of Denmark, DTU Compute, Lyngby, Denmark

Abstract

The classic string indexing problem is to preprocess a string S into a compact data structure that supports efficient pattern matching 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* (return the number of occurrences of the pattern). In this paper we consider a variant of string indexing, where the goal is to compactly represent the string such that given two patterns P_1 and P_2 and a gap range $[\alpha, \beta]$ we can quickly find the consecutive occurrences of P_1 and P_2 with distance in $[\alpha, \beta]$, i.e., pairs of subsequent occurrences with distance within the range. We present data structures that use $\widetilde{O}(n)$ space and query time $\widetilde{O}(|P_1| + |P_2| + n^{2/3})$ for existence and counting and $\widetilde{O}(|P_1| + |P_2| + n^{2/3}\text{occ}^{1/3})$ for reporting. We complement this with a conditional lower bound based on the set intersection problem showing that any solution using $\widetilde{O}(n)$ space must use $\widetilde{\Omega}(|P_1| + |P_2| + \sqrt{n})$ query time. To obtain our results we develop new techniques and ideas of independent interest including a new suffix tree decomposition and hardness of a variant of the set intersection problem.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Pattern matching

Keywords and phrases String indexing, two patterns, consecutive occurrences, conditional lower bound

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.10

Related Version *Full Version*: https://arxiv.org/abs/2102.02505

Funding *Philip Bille*: Supported by the Danish Research Council grant DFF-8021-002498.
Inge Li Gørtz: Supported by the Danish Research Council grant DFF-8021-002498.
Max Rishøj Pedersen: Supported by the Danish Research Council grant DFF-8021-002498.

1 Introduction

The classic string indexing problem is to preprocess a string S into a compact data structure that supports efficient pattern matching 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* (return the number of occurrences of the pattern). An important variant of this problem is the *gapped string indexing problem* [6, 8, 10, 14, 27, 28, 31]. Here, the goal is to compactly represent the string such that given two patterns P_1 and P_2 and a *gap range* $[\alpha, \beta]$ we can quickly find occurrences of P_1 and P_2 with distance in $[\alpha, \beta]$. Searching and indexing with gaps is frequently used in computational biology applications [6, 11, 13, 14, 19, 21, 22, 32, 35, 38].

Another variant is *string indexing for consecutive occurrences* [9, 40]. Here, the goal is to compactly represent the string such that given a pattern P and a gap range $[\alpha, \beta]$ we can quickly find *consecutive occurrences* of P with distance in $[\alpha, \beta]$, i.e., pairs of subsequent occurrences with distance within the range.

In this paper, we consider the natural combination of these variants that we call *gapped indexing for consecutive occurrences*. Here, the goal is to compactly represent the string such that given two patterns P_1 and P_2 and a gap range $[\alpha, \beta]$ we can quickly find the consecutive occurrences of P_1 and P_2 with distance in $[\alpha, \beta]$.

We can apply standard techniques to obtain several simple solutions to the problem. To state the bounds, let n be the size of S. If we store the suffix tree for S, we can answer queries by searching for both query strings, merging the results, and removing all non-consecutive occurrences. This leads to a solution using $O(n)$ space and $\widetilde{O}(|P_1|+|P_2|+\mathrm{occ}_{P_1}+\mathrm{occ}_{P_2})$ query time, where occ_{P_1} and occ_{P_2} denote the number of occurrences of P_1 and P_2, respectively[1]. However, $\mathrm{occ}_{P_1} + \mathrm{occ}_{P_2}$ may be as large as $\Omega(n)$ and much larger than the size of the output.

Alternatively, we can obtain a fast query time in terms of the output at the cost of increasing the space to $\Omega(n^2)$. To do so, store for each node v in the suffix tree the set of all consecutive occurrences (i, j) where i is the suffix number of a leaf below v in a standard 2D range searching data structure organized by the lexicographic order of j and the distance of the consecutive occurrence. To answer a query, we then perform a 2D range search in the structure corresponding to the locus of P_1 using the lexicographic range in the suffix tree defined by P_2 and the gap range. This leads to a solution for reporting queries using $\widetilde{O}(n^2)$ space and $\widetilde{O}(|P_1| + |P_2| + \mathrm{occ})$ time, where occ is the size of the output. For existence and counting, we obtain the same bound without the occ term.

In this paper, we introduce new solutions that significantly improve the above time-space trade-offs. Specifically, we present data structures that use $\widetilde{O}(n)$ space and query time $\widetilde{O}(|P_1| + |P_2| + n^{2/3})$ for existence and counting and $\widetilde{O}(|P_1| + |P_2| + n^{2/3}\mathrm{occ}^{1/3})$ for reporting. We complement this with a conditional lower bound based on the set intersection problem showing that any solution using $\widetilde{O}(n)$ space must use $\widetilde{\Omega}(|P_1| + |P_2| + \sqrt{n})$ query time. To obtain our results we develop new techniques and ideas of independent interest including a new suffix tree decomposition and hardness of a variant of the set intersection problem.

1.1 Setup and Results

Throughout the paper, let S be a string of length n. Given two patterns P_1 and P_2 a *consecutive occurrence* in S is a pair of occurrences (i, j), $0 \leq i < j < |S|$ where i is an occurrence of P_1 and j an occurrence of P_2, such that no other occurrences of either P_1 or P_2 occurs in between. The *distance* of a consecutive occurrence (i, j) is $j - i$. Our goal is to preprocess S into a compact data structure that given pattern strings P_1 and P_2 and a gap range $[\alpha, \beta]$ supports the following queries:

- $\mathsf{Exists}(P_1, P_2, \alpha, \beta)$: determine if there is a consecutive occurrence of P_1 and P_2 with distance within the range $[\alpha, \beta]$.
- $\mathsf{Count}(P_1, P_2, \alpha, \beta)$: return the number of consecutive occurrences of P_1 and P_2 with distance within the range $[\alpha, \beta]$.
- $\mathsf{Report}(P_1, P_2, \alpha, \beta)$: report all consecutive occurrences of P_1 and P_2 with distance within the range $[\alpha, \beta]$.

[1] $\widetilde{O}$ and $\widetilde{\Omega}$ ignores polylogarithmic factors.

We present new data structures with the following bounds:

▶ **Theorem 1.** *Given a string of length n, we can*

(i) *construct an $O(n)$ space data structure that supports* Exists$(P_1, P_2, \alpha, \beta)$ *and* Count$(P_1, P_2, \alpha, \beta)$ *queries in $O(|P_1| + |P_2| + n^{2/3} \log^{\epsilon} n)$ time for constant $\epsilon > 0$, or*

(ii) *construct an $O(n \log n)$ space data structure that supports* Report$(P_1, P_2, \alpha, \beta)$ *queries in $O(|P_1| + |P_2| + n^{2/3}\text{occ}^{1/3} \log n \log \log n)$ time, where* occ *is the size of the output.*

Hence, ignoring polylogarithmic factors, Theorem 1 achieves $\widetilde{O}(n)$ space and query time $\widetilde{O}(|P_1| + |P_2| + n^{2/3})$ for existence and counting and $\widetilde{O}(|P_1| + |P_2| + n^{2/3}\text{occ}^{1/3})$ for reporting. Compared to the above mentioned simple suffix tree approach that finds all occurrences of the query strings and merges them, we match the $\widetilde{O}(n)$ space bound, while reducing the dependency on n in the query time from worst-case $\Omega(|P_1| + |P_2| + n)$ to $\widetilde{O}(|P_1| + |P_2| + n^{2/3})$ for Exists and Count queries and $\widetilde{O}(|P_1| + |P_2| + n^{2/3}\text{occ}^{1/3})$ for Report queries.

We complement Theorem 1 with a conditional lower bound based on the set intersection problem. Specifically, we use the Strong SetDisjointness Conjecture from [20] to obtain the following result:

▶ **Theorem 2.** *Assuming the Strong SetDisjointness Conjecture, any data structure on a string S of length n that supports* Exists *queries in $O(n^{\delta} + |P_1| + |P_2|)$ time, for $\delta \in [0, 1/2]$, requires $\widetilde{\Omega}\left(n^{2-2\delta-o(1)}\right)$ space. This bound also holds if we limit the queries to only support ranges of the form $[0, \beta]$, and even if the bound β is known at preprocessing time.*

With $\delta = 1/2$, Theorem 2 implies that any near linear space solution must have query time $\widetilde{\Omega}(|P_1| + |P_2| + \sqrt{n})$. Thus, Theorem 1 is optimal within a factor roughly $n^{1/6}$. On the other hand, with $\delta = 0$, Theorem 2 implies that any solution with optimal $\widetilde{O}(|P_1| + |P_2|)$ query time must use $\widetilde{\Omega}(n^{2-o(1)})$ space. Note that this matches the trade-off achieved by the above mentioned simple solution that combines suffix trees with two-dimensional range searching data structures.

Finally, note that Theorem 2 holds even when the gap range is of the form $[0, \beta]$. As a simple extension of our techniques, in the appendix we show how to improve our solution from Theorem 1 to match Theorem 2 in this special case.

1.2 Techniques

To obtain our results we develop new techniques and show new interesting properties of consecutive occurrences. We first consider Exists and Count queries. The key idea is to split gap ranges into large and small distances. For large distances there can only be a limited number of consecutive occurrences and we show how these can be efficiently handled using a segmentation of the string. For small distances, we cluster the suffix tree and store precomputed answers for selected pairs of nodes. Since the number of distinct distances is small we obtain an efficient bound on the space.

We extend our solution for Exists and Count queries to handle Report queries. To do so we develop a new decomposition of suffix trees, called the *induced suffix tree decomposition* that recursively divides the suffix tree in half by index in the string. Hence, the decomposition is a balanced binary tree, where every node stores the suffix tree of a substring of S. We show how to traverse this structure to efficiently recover the consecutive occurrences.

For our conditional lower bound we show a reduction based on the set intersection problem. Along the way we show that set intersection remains hard even if all elements in the instance have the same frequency.

1.3 Related Work

As mentioned, string indexing for gaps and consecutive occurrences are the most closely related lines of work to this paper. Another related area is *document indexing*, where the goal is to preprocess a collection of strings, called *documents*, to report those documents that contain patterns subject to various constraints. For a comprehensive overview of this area see the survey by Navarro [36].

A well studied line of work within document indexing is *document indexing for top-k queries* [12, 23, 24, 25, 26, 33, 34, 37, 39, 42, 43]. The goal is to efficiently report the top-k documents of smallest weight, where the weight is a function of the query. Specifically, the weight can be the distance of a pair of occurrences of the same or two different query patterns [25, 33, 37, 42]. The techniques for top-k indexing (see e.g. Hon et al. [25]) can be adapted to efficiently solve gapped indexing for consecutive occurrences in the special case when the gap range is of the form $[0, \beta]$. However, since these techniques heavily exploit that the goal is to find the top-k *closest occurrences*, they do not generalize to general gap ranges.

There are several results on conditional lower bounds for pattern matching and string indexing [4, 5, 20, 29, 30]. Notably, Ferragina et al. [16] and Cohen and Porat [15] reduce the *two dimensional substring indexing problem* to set intersection (though the goal was to prove an upper, not a lower bound). In the two dimensional substring indexing problem the goal is to preprocess pairs of strings such that given two patterns we can output the pairs that contain a pattern each. Larsen et al. [30] prove a conditional lower bound for the document version of indexing for two patterns, i.e., finding all documents containing both of the two query patterns. Goldstein et al. [20] show that similar lower bounds can be achieved via conjectured hardness of set intersection. Thus, there are several results linking indexing for two patterns and set intersection. Our reduction is still quite different, since we need a translation from intersection to distance.

1.4 Outline

The paper is organized as follows. In Section 2 we define notation and recall some useful results. In Section 3 we show how to answer Exists and Count queries, proving Theorem 1(i). In Section 4 we show how to answer Report queries, proving Theorem 1(ii). In Section 5 we prove the lower bound, proving Theorem 2. In Appendix A we apply our techniques to solve the variant where $\alpha = 0$.

2 Preliminaries

Strings

A *string* S of length n is a sequence $S[0]S[1]\dots S[n-1]$ of characters from an alphabet Σ. A contiguous subsequence $S[i,j] = S[i]S[i+1]\dots S[j]$ is a *substring* of S. The substrings of the form $S[i, n-1]$ are the *suffixes* of S. The *suffix tree* [44] is a compact trie of all suffixes of $S\$$, where \$ is a symbol not in the alphabet, and is lexicographically smaller than any letter in the alphabet. Each leaf is labelled with the index i of the suffix $S[i, n-1]$ it corresponds to. Using perfect hashing [18], the suffix tree can be stored in $O(n)$ space and solve the string indexing problem (i.e., find and report all occurrences of a pattern P) in $O(m + \text{occ})$ time, where m is the length of P and occ is the number of times P occurs in S.

For any node v in the suffix tree, we define $\text{str}(v)$ to be the string found by concatenating all labels on the path from the root to v. The *locus* of a string P, denoted $\text{locus}(P)$, is the minimum depth node v such that P is a prefix of $\text{str}(v)$. The *suffix array* stores the suffix

indices of $S\$$ in lexicographic order. We identify each leaf in the suffix tree with the suffix index it represents. The suffix tree has the property that the leaves below any node represent suffixes that appear in consecutive order in the suffix array. For any node v in the suffix tree, range(v) denotes the range that v spans in the suffix array. The *inverse suffix array* is the inverse permutation of the suffix array, that is, an array where the ith element is the index of suffix i in the suffix array.

Orthogonal range successor

The *orthogonal range successor problem* is to preprocess an array $A[0, \ldots, n-1]$ into a data structure that efficiently supports the following queries:

- RangeSuccessor(a, b, x): return the successor of x in $A[a, \ldots, b]$, that is, the minimum $y > x$ such that there is an $i \in [a, b]$ with $A[i] = y$.
- RangePredecessor(a, b, x): return the predecessor of x in $A[a, \ldots, b]$, that is, the maximum $y < x$ such that there is an $i \in [a, b]$ with $A[i] = y$.

3 Existence and Counting

In this section we give a data structure that can answer Exists and Count queries. The main idea is to split the query interval into "large" and "small" distances. For large distances we exploit that there can only be a small number of consecutive occurrences and we check them with a simple segmentation of S. For small distances we cluster the suffix tree and precompute answers for selected pairs of nodes.

We first show how to use orthogonal range successor queries to find consecutive occurrences. Then we define the clustering scheme used for the suffix tree and give the complete data structure.

3.1 Using Orthogonal Range Successor to Find Consecutive Occurrences

Assume we have found the loci of P_1 and P_2 in the suffix tree. Then we can answer the following queries in a constant number of orthogonal range successor queries on the suffix array:

- FindConsecutive$_{P_2}(i)$: given an occurrence i of P_1, return the consecutive occurrence (i, j) of P_1 and P_2, if it exists, and No otherwise.
- FindConsecutive$_{P_1}(j)$: given an occurrence j of P_2, return the consecutive occurrence (i, j) of P_1 and P_2, if it exists, and No otherwise.

Given a query FindConsecutive$_{P_2}(i)$, we answer as follows. First, we compute the index $j =$ RangeSuccessor(range(locus(P_2)), i) to get the closest occurrence of P_2 after i. Then, we compute $i' =$ RangePredecessor(range(locus(P_1)), j) to get the closest occurrence of P_1 before j. If $i = i'$ then no other occurrence of P_1 exists between i and j and they are consecutive. In that case we return (i, j). Otherwise, we return No.

Similarly, we can answer FindConsecutive$_{P_1}(j)$ by first doing a RangePredecessor and then a RangeSuccessor query. Thus, given the loci of both patterns and a specific occurrence of either P_1 or P_2, we can in a constant number of RangeSuccessor and RangePredecessor queries find the corresponding consecutive occurrence, if it exists.

3.2 Data Structure

To build the data structure we will use a cluster decomposition of the suffix tree.

Cluster Decomposition

A cluster decomposition of a tree T is defined as follows: For a connected subgraph $C \subseteq T$, a *boundary node* v is a node $v \in C$ such that either v is the root of T, or v has an edge leaving C – that is, there exists an edge (v, u) in the tree T such that $u \in T \setminus C$. A *cluster* is a connected subgraph C of T with at most two boundary nodes. A cluster with one boundary node is called a *leaf cluster*. A cluster with two boundary nodes is called a *path cluster*. For a path cluster C, the two boundary nodes are connected by a unique path. We call this path the *spine* of C. A *cluster partition* is a partition of T into clusters, i.e. a set CP of clusters such that $\bigcup_{C \in CP} V(C) = V(T)$ and $\bigcup_{C \in CP} E(C) = E(T)$ and no two clusters in CP share any edges. Here, $E(G)$ and $V(G)$ denote the edge and vertex set of a (sub)graph G, respectively. We need the next lemma which follows from well-known tree decompositions [1, 2, 3, 17] (see Bille and Gørtz [7] for a direct proof).

▶ **Lemma 3.** *Given a tree T with n nodes and a parameter τ, there exists a cluster partition CP such that $|CP| = O(n/\tau)$ and every $C \in CP$ has at most τ nodes. Furthermore, such a partition can be computed in $O(n)$ time.*

Data Structure

We build a clustering of the suffix tree of S as in Lemma 3, with cluster size at most τ, where τ is some parameter satisfying $0 < \tau \leq n$. Then the counting data structure consists of:

- The suffix tree of S, with some additional information for each node. For each node v we store:
  - The range v spans in the suffix array, i.e., range(v).
  - A bit that indicates if v is on a spine.
  - If v is on a spine, a pointer to the lower boundary node of the spine.
  - If v is a leaf, the local rank of v. That is, the rank of v in the text order of the leaves in the cluster that contains v. Note that this is at most τ.
- The inverse suffix array of S.
- A range successor data structure on the suffix array of S.
- An array $M(u, v)$ of length $\lfloor \frac{n}{\tau} \rfloor + 1$ for every pair of boundary nodes (u, v). For $1 \leq x \leq \lfloor \frac{n}{\tau} \rfloor$, $M(u, v)[x]$ is the number of consecutive occurrences (i, j) of str(u) and str(v) with distance at most x. We set $M(u, v)[0] = 0$.

Denote $M(u, v)[\alpha, \beta] = M(u, v)[\beta] - M(u, v)[\alpha - 1]$, that is, $M(u, v)[\alpha, \beta]$ is the number of consecutive occurrences of str(u) and str(v) with a distance in $[\alpha, \beta]$.

Space Analysis

We store a constant amount of words per node in the suffix tree. The suffix tree and inverse suffix array occupy $O(n)$ space. For the orthogonal range successor data structure we use the data structure of Nekrich and Navarro [41] which uses $O(n)$ space and $O(\log^{\epsilon} n)$ time, for constant $\epsilon > 0$. There are $O\left(n^2/\tau^2\right)$ pairs of boundary nodes and for each pair we store an array of length $O\left(n/\tau\right)$. Therefore the total space consumption is $O\left(n + n^3/\tau^3\right)$.

3.3 Query Algorithm

We now show how to count the consecutive occurrences (i, j) with a distance in the interval, i.e. $\alpha \leq j - i \leq \beta$. We call each such pair a *valid occurrence*.

To answer a query we split the query interval $[\alpha, \beta]$ into two: $[\alpha, \lfloor \frac{n}{\tau} \rfloor]$ and $[\lfloor \frac{n}{\tau} \rfloor + 1, \beta]$, and handle these separately.

3.3.1 Handling Distances $> \frac{n}{\tau}$

We start by finding the loci of P_1 and P_2 in the suffix tree. As shown in Section 3.1, this allows us to find the consecutive occurrence containing a given occurrence of either P_1 or P_2. We implicitly partition the string S into segments of (at most) $\lfloor n/\tau \rfloor$ characters by calculating τ segment boundaries. Segment i, for $0 \leq i < \tau$, contains characters $S[i \cdot \lfloor \frac{n}{\tau} \rfloor, (i+1) \cdot \lfloor \frac{n}{\tau} \rfloor - 1]$ and segment τ (if it exists) contains the characters $S[\tau \cdot \lfloor \frac{n}{\tau} \rfloor, n-1]$. We find the last occurrence of P_1 in each segment by performing a series of RangePredecessor queries, starting from the beginning of the last segment. Each time an occurrence i is found we perform the next query from the segment boundary to the left of i, continuing until the start of the string is reached. For each occurrence i of P_1 found in this way, we use $\mathsf{FindConsecutive}_{P_2}(i)$ to find the consecutive occurrence (i, j) if it exists. We check each of them, discard any with distance $\leq \frac{n}{\tau}$ and count how many are valid.

3.3.2 Handling Distances $\leq \frac{n}{\tau}$

In this part, we only count valid occurrences with distance $\leq \frac{n}{\tau}$. Consider the loci of P_1 and P_2 in the suffix tree. Let C_i denote the cluster that contains $\text{locus}(P_i)$ for $i = 1, 2$. There are two main cases.

At least one locus is not on a spine

If either locus is in a small subtree hanging off a spine in a cluster or in a leaf cluster, we directly find all consecutive occurrences as follows: If $\text{locus}(P_1)$ is in a small subtree then we use $\mathsf{FindConsecutive}_{P_2}(i)$ on each leaf i below $\text{locus}(P_1)$ to find all consecutive occurrences, count the valid occurrences and terminate. If only $\text{locus}(P_2)$ is in a small subtree then we use $\mathsf{FindConsecutive}_{P_1}(j)$ for each leaf j below $\text{locus}(P_2)$, count the valid occurrences and terminate.

Both loci are on the spine

If neither locus is in a small subtree then both are on spines. Let (b_1, b_2) denote the lower boundary nodes of the clusters C_1 and C_2, respectively. There are two types of consecutive occurrences (i, j):

(i) Occurrences where either i or j are inside C_1 resp. C_2.
(ii) Occurrences below the boundary nodes, that is, i is below b_1 and j is below b_2.

See Figure 1(a). We describe how to count the different types of occurrences next.

Type (i) occurrences. To find the valid occurrences (i, j) where either $i \in C_1$ or $j \in C_2$ we do as follows. First we find all the consecutive occurrences (i, j) where i is a leaf in C_1 by computing $\mathsf{FindConsecutive}_{P_2}(i)$ for all leaves i below $\text{locus}(P_1)$ in C_1. We count all valid occurrences we find in this way. Then we find all remaining consecutive occurrences (i, j) where j is a leaf in C_2 by computing $\mathsf{FindConsecutive}_{P_1}(j)$ for all leaves j below $\text{locus}(P_2)$ in C_2. If $\mathsf{FindConsecutive}_{P_1}(j)$ returns a valid occurrence (i, j) we use the inverse suffix array to check if the leaf i is below b_1. This can be done by checking whether i's position in the suffix array is in $\text{range}(b_1)$. If i is below b_1 we count the occurrence, otherwise we discard it.

Type (ii) occurrences. Next, we count the consecutive occurrences (i, j), where both i and j are below b_1 and b_2, respectively. We will use the precomputed table, but we have to be a careful not to overcount. By its construction, $M(b_1, b_2)[\alpha, \min(\lfloor \frac{n}{\tau} \rfloor, \beta)]$ is the number

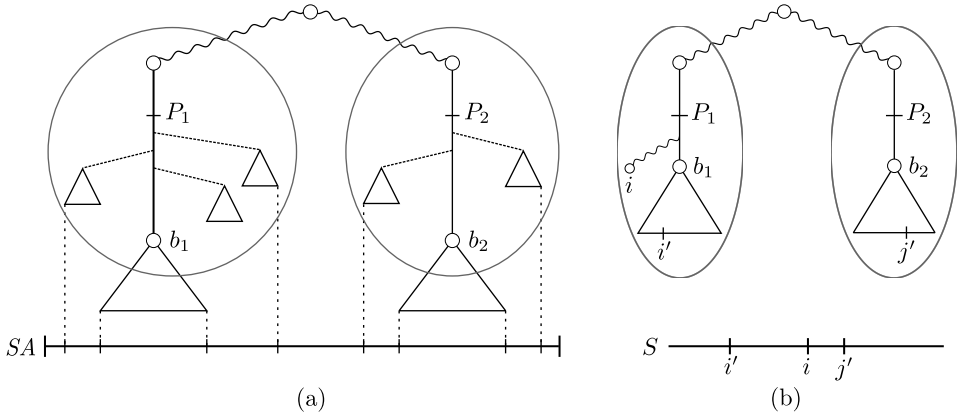


Figure 1 (a) Any consecutive occurrences (i, j) of P_1 and P_2 is either also a consecutive occurrence of $\mathrm{str}(b_1)$ and $\mathrm{str}(b_2)$, or i or j are within the respective cluster. The suffix array is shown in the bottom with the corresponding ranges marked. (b) Example of a false occurrence. Here (i', j') is a consecutive occurrence of $\mathrm{str}(b_1)$ and $\mathrm{str}(b_2)$, but not a consecutive occurrence of P_1 and P_2 due to i. The string S is shown in bottom with the positions of the occurrences marked.

of consecutive occurrences (i', j') of $\mathrm{str}(b_1)$ and $\mathrm{str}(b_2)$, where $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$. However, not all of these occurrence (i', j') are necessarily *consecutive* occurrences of P_1 and P_2, as there could be an occurrence of P_1 in C_1 or P_2 in C_2 which is between i' and j'. We call such a pair (i', j') a *false occurrence*. See Figure 1(b). We proceed as follows.

1. Set $c = M(b_1, b_2)[\alpha, \min(\lfloor \frac{n}{\tau} \rfloor, \beta)]$.
2. Construct the lists L_i containing the leaves in C_i that are below $\mathrm{locus}(P_i)$ sorted by text order for $i = 1, 2$. We can obtain the lists as follows. Let $[a, b]$ be the range of $\mathrm{locus}(P_i)$ and $[a', b'] = \mathrm{range}(b_i)$. Sort the leaves in $[a, a' - 1] \cup [b' + 1, b]$ using their local rank.
3. Until both lists are empty iteratively pick and remove the smallest element e from the start of either list. There are two cases.
   - e is an element of L_1.
     - Compute $j' = \mathsf{RangeSuccessor}(\mathrm{range}(b_2), e)$ to get the closest occurrence of $\mathrm{str}(b_2)$ after e.
     - Compute $i' = \mathsf{RangePredecessor}(\mathrm{range}(b_1), j')$ to get the closest occurrence of $\mathrm{str}(b_1)$ before j'.
   - e is an element of L_2.
     - Compute $i' = \mathsf{RangePredecessor}(\mathrm{range}(b_2), e)$ to get the previous occurrence i' of $\mathrm{str}(b_1)$.
     - Compute $j' = \mathsf{RangeSuccessor}(\mathrm{range}(b_1), j')$ to get the following occurrence j' of $\mathrm{str}(b_2)$.

   If $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$ and $i' < e < j'$ decrement c by one. We skip any subsequent occurrences that are also inside (i', j'). As the lists are sorted by text order, all occurrences that are within the same consecutive occurrence (i', j') are handled in sequence.

   Finally, we add the counts of the different type of occurrences.

Correctness

Consider a consecutive occurrence (i, j) where $j - i > \frac{n}{\tau}$. Such a pair must span a segment boundary, i.e., i and j cannot be in the same segment. As (i, j) is a *consecutive* occurrence, i is the last occurrence of P_1 in its segment and j is the first occurrence of P_2 in its segment.

With the RangePredecessor queries we find all occurrences of P_1 that are the last in their segment. We thus check and count all valid occurrences of large distance in the initial pass of the segments.

If either locus is in a small subtree we use $\mathsf{FindConsecutive}_{P_2}(.)$ or $\mathsf{FindConsecutive}_{P_1}(.)$ on the leaves below that locus, which by the arguments in Section 3.1 will find all consecutive occurrences.

Otherwise, both loci are on a spine. To count occurrences of type (i), we first compute $\mathsf{FindConsecutive}_{P_2}(i)$ for all leaves i below $\text{locus}(P_1)$ in C_1 and then $\mathsf{FindConsecutive}_{P_1}(j)$ for all leaves j below $\text{locus}(P_2)$ in C_2. However, any valid occurrence (i, j) where both $i \in C_1$ and $j \in C_2$ is found by both operations. Therefore, whenever we find a valid occurrence (i, j) via $i = \mathsf{FindConsecutive}_{P_1}(j)$ for $j \in C_2$, we only count the occurrence if i is below b_1. Thus we count all type (i) occurrences exactly once.

To count type (ii) occurrences we start with $c = M(b_1, b_2)[\alpha, \min(\lfloor \frac{n}{\tau} \rfloor, \beta)]$, which is the number of consecutive occurrences (i', j') of $\text{str}(b_1)$ and $\text{str}(b_2)$, where $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$. Each (i', j') is either also a consecutive occurrence of P_1 and P_2, or there exists an occurrence of P_1 or P_2 between i' and j'. Let (i', j') be a false occurrence and let w.l.o.g. i be an occurrence of P_1 with $i' < i < j'$. Then i is a leaf in C_1, since (i', j') is a *consecutive* occurrence of $\text{str}(b_1)$ and $\text{str}(b_2)$. In step 3 we check for each leaf inside the clusters below the loci, if it is between a consecutive occurrence (i', j') of $\text{str}(b_1)$ and $\text{str}(b_2)$ and if $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$. In that case (i', j') is a false occurrence and we adjust the count c. As (i', j') can have multiple occurrences of P_1 and P_2 inside it, we skip subsequent occurrences inside (i', j'). After adjusting for false occurrences, c is the number of type (ii) occurrences.

Time Analysis

We find the loci in $O(|P_1| + |P_2|)$ time. Then we perform a number of range successor and FindConsecutive queries. The time for a FindConsecutive query is bounded by the time to do a constant number of range successor queries. To count the large distances we check at most τ segment boundaries and thus perform $O(\tau)$ range successor and FindConsecutive queries.

For small distances, if either locus is not on a spine we check the leaves below that locus. There are at most τ such leaves due to the clustering. To count type (i) occurrences we check the leaves below the loci that are inside the clusters. There are at most 2τ such leaves in total. To count type (ii) occurrences we check two lists constructed from the leaves inside the clusters below the loci. There are again at most 2τ such leaves in total. For each of these $O(\tau)$ leaves we use a constant number of range successor and FindConsecutive queries. Thus the time for this part is bounded by the time to perform $O(\tau)$ range successor queries.

Using the data structure of Nekrich and Navarro [41], each range successor query takes $O(\log^\epsilon n)$ time so the total time for these queries is $O(\tau \log^\epsilon n)$. For type (ii) occurrences we sort two lists of size at most τ from a universe of size τ, which we can do in $O(\tau)$ time. Thus, the total query time is $O(|P_1| + |P_2| + \tau \log^\epsilon n)$.

Setting $\tau = \Theta(n^{2/3})$ we get a data structure that uses $O\left(n + n^3/\tau^3\right) = O(n)$ space and has query time $O(|P_1| + |P_2| + \tau \log^\epsilon n) = O(|P_1| + |P_2| + n^{2/3} \log^\epsilon n)$, for constant $\epsilon > 0$. We answer an Exists query with a Count query, terminating when the first valid occurrence is found. This concludes the proof of Theorem 1(i).

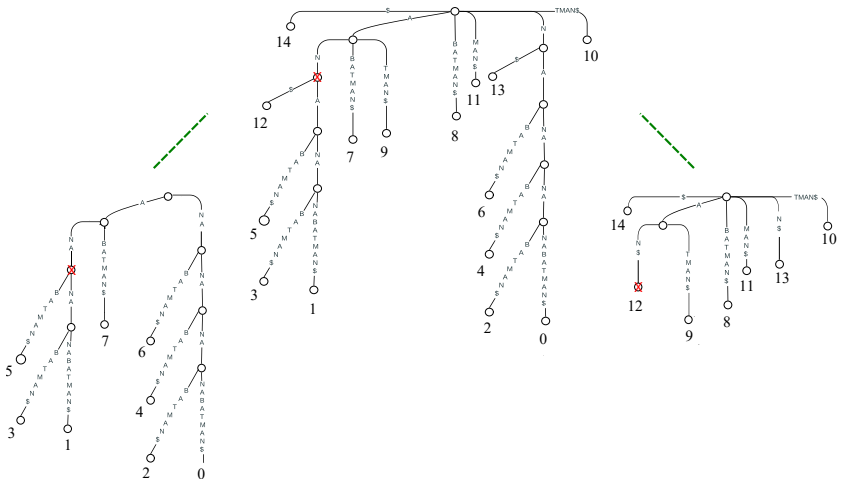


Figure 2 The suffix tree of `NANANANABATMAN$` together with its children trees $T[0, 7]$ and $T[8, 14]$. The red crosses show a node in the parent tree and and its successor nodes in the two children trees.

4 Reporting

In this section, we describe our data structure for reporting queries. Note that in Section 3, we explicitly find all valid occurrences *except* for type (ii) occurrences, where we use the precomputed values. In this section, we describe how we can use a recursive scheme to report these.

The main idea, inspired by fast set intersection by Cohen and Porat [15], is to build a recursive binary structure which allows us to recursively divide the problem into subproblems of half the size. Intuitively, the subdivision is a binary tree where every node contains the suffix tree of a substring of S. We use this structure to find type (ii) occurrences by recursing on smaller trees. We define the binary decomposition of the suffix tree next. The details of the full solution follow after that.

4.1 Induced Suffix Tree Decomposition

Let T be a suffix tree of a string S of length n. For an interval $[a, b]$ of *text positions*, we define $T[a, b]$ to be the subtree of T *induced* by the leaves in $[a, b]$: That is, we consider the subtree consisting of leaves in $[a, b]$ together with their ancestors. We then delete each node that has only one child in the subtree and contract its ingoing and outgoing edge. See Figure 2.

The *induced suffix tree decomposition of* T now consists of a higher level binary tree structure, the *decomposition tree*, where each node corresponds to an induced subtree of the suffix tree. The root corresponds to $T[0, n-1]$, and whenever we move down in the decomposition tree, the interval splits in half. We also associate a level with each of the induced subtrees, which is their depth in the decomposition tree. In more detail, the decomposition tree is a binary tree such that:

- The root of the decomposition tree corresponds to $T[0, n-1]$ and has level 0.
- For each $T[a, b]$ of level i in the decomposition, if $b - a > 1$, its two children in the decomposition tree are $T[a, c]$ and $T[c+1, b]$ where $c = \lfloor \frac{a+b}{2} \rfloor$; we will sometimes refer to these as "children trees" to differentiate from children in the suffix tree.

The decomposition tree is a balanced binary tree and the total size of the induced subtrees in the decomposition is $O(n \log n)$: There are at most 2^i decomposition tree nodes on level i, each of which corresponds to an induced subtree of size $O\left(\frac{n}{2^i}\right)$, and thus the total size of the trees on each of the $O(\log n)$ levels is $O(n)$.

For each node v in $T[a, b]$, we define the *successor node* of v in each of the children trees of $T[a, b]$ in the following way: If v exists in the child tree, the successor node is v. Else, it is the closest descendant which is present. Note that from the way the induced subtrees are constructed, v has at most one successor node in each child tree.

The induced suffix tree decomposition of S consists of:

- Each $T[a, b]$ stored as a compact trie.
- For each $T[a, b]$ we store a sparse suffix array $SA_{[a,b]}$, that is, the suffix array of $S[a, b]$ with the original indices within S.
- For each node v in $T[a, b]$ we store a pointer from v to its successor nodes in each child tree, if it exists, and the interval in $SA_{[a,b]}$ that corresponds to the leaves below v.

Since we store only constant information per node in any $T[a, b]$, the total space usage of this is $O(n \log n)$.

4.2 Data Structure

The reporting data structure consists of:

- The induced suffix tree decomposition for S,
- An orthogonal range successor data structure on the suffix array, and
- The data structure from Section 3 for each $T[a, b]$ in the induced suffix tree decomposition with parameters n_i and τ_i, where $n_i = \lfloor \frac{n}{2^i} \rfloor$ and $\tau_i = \Theta(n_i^{2/3})$, such that $n_i/\tau_i = \lfloor n_i^{1/3} \rfloor$. The only change is that we do not store an orthogonal range successor data structure for each of the induced subtrees.

Space Analysis

We use the $O(n \log \log n)$ space and $O(\log \log n)$ time orthogonal range successor structure of Zhou [45]. The data structure from Section 3 for each $T[a, b]$ of level i is linear in n_i. Thus, by the arguments of Section 4.1, the total space is $O(n \log n)$.

4.3 Query Algorithm

The main idea behind the algorithm is the following: For large distances, as in Section 3, we implicitly segment S to find all consecutive occurrences of at least a certain distance. For small distances, we are going to use the cluster decomposition and counting arrays to decide whether valid occurrences exist. That is, if one of the loci is in a small subtree, we use $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$ to find all consecutive occurrences. Else, we perform a query as in Section 3 to decide whether any valid occurrences exist, and if yes, we recurse on smaller subtrees.

The idea here is, that in the induced suffix tree decomposition, the trees are divided in half by *text position* - therefore, a *consecutive* occurrence either will be fully contained in the left child tree, fully contained in the right child tree, or have the property that the occurrence of P_1 is the maximum occurrence in the left child tree and the occurrence of P_2 is the minimum occurrence in the right child tree. We will check the border case each time when we recurse.

In detail, we do the following: We find the loci of P_1 and P_2 in the suffix tree. As in the previous section, we check τ_0 segment boundaries with $\tau_0 = \Theta(n^{2/3})$ to find all consecutive occurrences with distance within $[\max(\alpha, \lfloor n^{1/3} \rfloor), \beta]$. Now, we only have to find consecutive occurrences of distance within $[\alpha, \min(\beta, \lfloor n^{1/3} \rfloor)]$ in $T = T[0, n-1]$. In general, let $n_i = \lfloor \frac{n}{2^i} \rfloor$ and $\beta_i = \min(\beta, \lfloor n_i^{1/3} \rfloor)$ and let $T[a, b]$ be an induced subtree of level i.

To find all consecutive occurrences with distance within $[\alpha, \beta_i]$ in $T[a, b]$ of level i, given the loci of P_1 and P_2 in $T[a, b]$, recursively do the following:

- If any of the loci is not on a spine of a cluster, we find all consecutive occurrences using $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$ and check for each of them if they are valid; we report all such, then terminate.
- Else, we use the query algorithm for small distances from Section 3 to decide whether a valid occurrence with distance within $[\alpha, \beta_i]$ exists in $T[a, b]$.
  If such a valid occurrence exists, we recurse; that is, set $c = \lfloor \frac{a+b}{2} \rfloor$. We use RangePredecessor to find the last occurrence of P_1 before and including c, and RangeSuccessor to find the first occurrence of P_2 after c. Then we check if they are consecutive (again using RangePredecessor and RangeSuccessor), and if it is a valid occurrence. If yes, we add it to the output. Then, for both $S[a, c]$ and $S[c+1, b]$, we implicitly partition them into segments of size $\lfloor n_{i+1}^{1/3} \rfloor$ and find and output all valid occurrences of distance $> n_{i+1}^{1/3}$. Then we follow pointers to the successor nodes of the current loci to find the loci of P_1 and P_2 in the children trees $T[a, c]$ and $T[c+1, b]$ and recurse on those trees to find all consecutive occurrences of distance within $[\alpha, \beta_{i+1}]$

Correctness

At any point before we recurse on level i, we check all consecutive occurrences of distance $> n_{i+1}^{1/3}$ by segmenting the current substring of S. By the arguments of the previous section, we will find all such valid occurrences. Thus, on the subtrees of level $i+1$, we need only care about consecutive occurrences with distance in $[\alpha, \beta_{i+1}]$.

By the properties of the induced suffix tree decomposition, a consecutive occurrence of P_1 and P_2 that is present in $T[a, b]$ will either be fully contained in $T[a, c]$, or in $T[c+1, b]$, or the occurrence of P_1 is the last occurrence before and including c and the occurrence of P_2 is the first occurrence after c. We check the border case each time we recurse. Thus, no consecutive occurrences get lost when we recurse. If we stop the recursion, it is either because one of the loci was in a small subtree or that no valid occurrences with distance within $[\alpha, \beta_i]$ exists in $T[a, b]$. In the first case we found all valid occurrences with distance within $[\alpha, \beta_i]$ in $T[a, b]$ by the same arguments as in Section 3. Thus, we find all valid occurrences of P_1 and P_2.

Time Analysis

For finding the loci, we first spend $O(|P_1| + |P_2|)$ time in the initial suffix tree $T[0, n-1]$; after that, we spend constant time each time we recurse to follow pointers. The rest of the time consumption is dominated by the number of queries to the orthogonal range successor data structure, which we will count next.

Consider the recursion part of the algorithm as a traversal of the decomposition tree, and consider the subtree of the decomposition tree we traverse. Each leaf of that subtree is a node where we stop recursing. Since we only recurse if we know there is an occurrence to be found, there are at most $O(\mathrm{occ})$ leaves. Thus, we traverse at most $O(\mathrm{occ} \log n)$ nodes.

Each time we recurse, we spend a constant number of RangeSuccessor and RangePredecessor queries to check the border cases. Additionally, we spend $O(n_i^{2/3})$ such queries on each node of level i that we visit in the decomposition tree: For finding the "large" occurrences, and

additionally either for reporting everything within a small subtree or doing an existence query. For finding large occurrences, there are $O(n_i^{2/3})$ segments to check. The number of orthogonal range successor queries used for existence queries or reporting within a small subtree is bounded by the number of leaves within a cluster, which is also $O(n_i^{2/3})$.

Now, let x be the number of decomposition tree nodes we traverse and let l_i, $i = 1, \ldots, x$, be the level of each such node. The goal is to bound $\sum_{i=1}^{x} \left(\frac{n}{2^{l_i}}\right)^{2/3}$. By the argument above, $x = O(\text{occ} \log n)$. Note that because the decomposition tree is binary we have that $\sum_{i=1}^{x} \frac{1}{2^{l_i}} \leq \log n$. The number of queries to the orthogonal range successor data structure is thus asymptotically bounded by:

$$\begin{aligned}
\sum_{i=1}^{x} \left(\frac{n}{2^{l_i}}\right)^{2/3} &= n^{2/3} \sum_{i=1}^{x} \left(\frac{1}{2^{l_i}}\right)^{2/3} \cdot 1 \\
&\leq n^{2/3} \left(\sum_{i=1}^{x} \left(\frac{1}{2^{l_i}}\right)^{\frac{2}{3}\cdot\frac{3}{2}}\right)^{2/3} \left(\sum_{i=1}^{x} 1^3\right)^{1/3} \\
&= n^{2/3} \left(\sum_{i=1}^{x} \frac{1}{2^{l_i}}\right)^{2/3} x^{1/3} \\
&= O(n^{2/3} \text{occ}^{1/3} \log n)
\end{aligned}$$

For the inequality, we use Hölder's inequality, which holds for all $(x_1, \ldots, x_k) \in \mathbb{R}^k$ and $(y_1, \ldots, y_k) \in \mathbb{R}^k$ and p and q both in $(1, \infty)$ such that $1/p + 1/q = 1$:

$$\sum_{i=1}^{k} |x_i y_i| \leq \left(\sum_{i=1}^{k} |x_i|^p\right)^{1/p} \left(\sum_{i=1}^{k} |y_i|^q\right)^{1/q} \tag{1}$$

We apply (1) with $p = 3/2$ and $q = 3$.

Since the data structure of Zhou [45] uses $O(\log \log n)$ time per query, the total running time of the algorithm is $O(|P_1| + |P_2| + n^{2/3}\text{occ}^{1/3} \log n \log \log n)$. This concludes the proof of Theorem 1(ii).

5 Lower Bound

We now prove the conditional lower bound from Theorem 2 based on set intersection. We use the framework and conjectures as stated in Goldstein et al. [20]. Throughout the section, let $\mathcal{I} = S_1, , \ldots, S_m$ be a collection of m sets of total size N from a universe U. The *SetDisjointness problem* is to preprocess $\mathcal{I}$ into a compact data structure, such that given any pair of sets S_i and S_j, we can quickly determine if $S_i \cap S_j = \emptyset$. We use the following conjecture.

▶ **Conjecture 4** (Strong SetDisjointness Conjecture). *Any data structure that can answer SetDisjointness queries in t query time must use $\widetilde{\Omega}\left(\frac{N^2}{t^2}\right)$ space.*

5.1 SetDisjointness with Fixed Frequency

We define a weaker variant of the SetDisjointness problem: the *f-FrequencySetDisjointness problem* is the SetDisjointness problem where every element occurs in precisely f sets. We now show that any solution to the f-FrequencySetDisjointness problem implies a solution to SetDisjointness, matching the complexities up to polylogarithmic factors.

▶ **Lemma 5.** *Assuming the Strong SetDisjointness Conjecture, every data structure that can answer f-FrequencySetDisjointness queries in time $O(N^\delta)$, for $\delta \in [0, 1/2]$, must use $\widetilde{\Omega}\left(N^{2-2\delta-o(1)}\right)$ space.*

Proof. Assume there is a data structure D solving the f-FrequencySetDisjointness problem in time $O(N^\delta)$ and space $O\left(N^{2-2\delta-\epsilon}\right)$ for constant ϵ with $0 < \epsilon < 1$. Let $\mathcal{I} = S_1, \ldots, S_m$ be a given instance of SetDisjointness, where each S_i is a set of elements from universe U, and assume w.l.o.g. that m is a power of two.

Define the *frequency* of an element, f_e, as the number of sets in $\mathcal{I}$ that contain e. We construct $\log m$ instances $\mathcal{I}_1, \ldots, \mathcal{I}_{\log m}$ of the f-FrequencySetDisjointness problem. For each j, $1 \leq j \leq \log m$, the instance $\mathcal{I}_j$ contains the following sets:

- For each $i \in [1, m]$ a set S_i^j containing all $e \in S_i$ that satisfy $2^{j-1} \leq f_e < 2^j$;
- 2^{j-1} "dummy sets", which contain extra copies of elements to make sure that all elements have the same frequency. That is, we add every element with $2^{j-1} \leq f_e < 2^j$ to the first $2^j - f_e$ dummy sets. These sets will not be queried in the reduction.

Instance $\mathcal{I}_j$ has $O(m)$ sets and every element occurs exactly 2^j times. Further, the total number of elements is at most $2N$. We now build f-FrequencySetDisjointness data structures $D_j = D(\mathcal{I}_j)$ for each of the $\log m$ instances.

To answer a SetDisjointness query for two sets S_{i_1} and S_{i_2}, we query D_j for the sets $S_{i_1}^j$ and $S_{i_2}^j$, for each $1 \leq j \leq \log m$. If there exists a j such that $S_{i_1}^j$ and $S_{i_2}^j$ are not disjoint, we output that S_i and S_j are not disjoint. Else, we output that they are disjoint.

If there exists $e \in S_{i_1} \cap S_{i_2}$, let j be such that $2^{j-1} \leq f_e < 2^j$. Then $e \in S_{i_1}^j \cap S_{i_2}^j$, and we will correctly output that the sets are not disjoint. If S_{i_1} and S_{i_2} are disjoint, then, since $S_{i_1}^j$ is a subset of S_{i_1} and $S_{i_2}^j$ is a subset of S_{i_2}, the queried sets are disjoint in every instance. Thus we also answer correctly in this case.

Let N_j denote the total number of elements in $\mathcal{I}_j$. For each j, we have $N_j \leq 2N$ and thus $N_j^{2-2\delta-\epsilon} \leq (2N)^{2-2\delta-\epsilon}$. Thus, the space complexity is asymptotically bounded by

$$\sum_{j=1}^{\lceil \log m \rceil} N_j^{2-2\delta-\epsilon} = O(N^{2-2\delta-\epsilon} \log m).$$

Similarly, we have $N_j^\delta = O(N^\delta)$ and so the time complexity is asymptotically bounded by

$$\sum_{j=1}^{\lceil \log m \rceil} N_j^{\delta} = O(N^{\delta} \log m).$$

This is a contradiction to Conjecture 4. ◀

5.2 Reduction to Gapped Indexing

We can reduce the f-FrequencySetDisjointness problem to Exists queries of the gapped indexing problem: Assume we are given an instance of the f-FrequencySetDisjointness problem with a total of N elements. Each distinct element occurs f times. Assume again w.l.o.g. that the number of sets m is a power of two. Assign to each set S_i in the instance a unique binary string w_i of length $\log m$. Build a string S as follows: Consider an arbitrary ordering $e_1, e_2, \ldots$ of the distinct elements present in the f-FrequencySetDisjointness instance. Let \$ be an extra letter not in the alphabet. The first $B = f \cdot \log m + f$ letters are a

$$S_1 = \{1, 2\} \qquad w_1 = 00$$
$$S_2 = \{3, 4\} \qquad w_2 = 01$$
$$S_3 = \{1, 3\} \qquad w_3 = 10$$
$$S_4 = \{2, 4\} \qquad w_4 = 11$$

$$S = \underbrace{00\$10\$}_{1}\ \$\$\$\$\$\$\ \underbrace{00\$11\$}_{2}\ \$\$\$\$\$\$\ \underbrace{01\$10\$}_{3}\ \$\$\$\$\$\$\ \underbrace{01\$11\$}_{4}\ \$\$\$\$\$\$$$

Figure 3 Instance of f-FrequencySetDisjointness problem reduced to Exists. Alphabet $\Sigma = \{0, 1\}$ and fixed frequency $f = 2$, resulting in block size $B = 2 \cdot 2 + 2 = 6$.

concatenation of $w_i\$$ of all sets S_i that e_1 is contained in, sorted by i. This block is followed by B copies of \$. Then, we have B symbols consisting of the strings for each set that e_2 is contained in, again followed by B copies of \$, and so on. See Figure 3 for an example.

For a query for two sets S_i and S_j, where $i < j$, we set $P_1 = w_i$ and $P_2 = w_j$, $\alpha = 0$, and $\beta = B$. If the sets are disjoint, then there are no occurrences which are at most B apart. Otherwise w_i and w_j occur in the same block, and w_j comes after w_i. The length of the string S is $2N \log m + 2N$: In the block for each element, we have $\log m + 1$ letters for each of its occurrences, and it is followed by a \$ block of the same length.

This means that if we can solve Exists queries in $s(n)$ space and $t(n) + O(|P_1| + |P_2|)$ time, where n is the length of the string, we can solve the f-FrequencySetDisjointness problem in $s(2N \log m + 2N)$ space and $t(2N \log m + 2N) + O(\log m)$ time. Together with Lemma 5, Theorem 2 follows.

6 Conclusion

We have considered the problem of gapped indexing for consecutive occurrences. We have given a linear space data structure that can count the number of such occurrences. For the reporting problem, we have given a near-linear space data structure. The running time for both includes an $O(n^{2/3})$ term, which forms a gap of $O(n^{1/6})$ to the conditional lower bound of $O(\sqrt{n})$. Thus, the most obvious open question is whether we can close this gap, either by improving the data structure or finding a stronger lower bound.

Further, we have used the property that there can only be few consecutive occurrences of large distances. Thus, our solution cannot be easily extended to finding *all* pairs of occurrences with distance within the query interval. An open question is if it is possible to get similar results for that problem. Lastly, document versions of similar problems have concerned themselves with finding all documents that contain P_1 and P_2 or the top-k of smallest distance; conditional lower bounds for these problems are also known. It would be interesting to see if any of these results be extended to finding all documents that contain a (consecutive) occurrence of P_1 and P_2 that has a distance within a query interval.

References

1 Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th ICALP*, pages 270–280, 1997.

2 Stephen Alstrup, Jacob Holm, and Mikkel Thorup. Maintaining center and median in dynamic trees. In *Proc. 7th SWAT*, pages 46–56, 2000.

3 Stephen Alstrup and Theis Rauhe. Improved labeling scheme for ancestor queries. In *Proc. 13th SODA*, pages 947–953, 2002.

4 Amihood Amir, Timothy M. Chan, Moshe Lewenstein, and Noa Lewenstein. On hardness of jumbled indexing. In *Proc. 41st ICALP*, pages 114–125, 2014.

5 Amihood Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. Mind the gap: Essentially optimal algorithms for online dictionary matching with one gap. In *Proc. 27th ISAAC*, pages 12:1–12:12, 2016.

6 Johannes Bader, Simon Gog, and Matthias Petri. Practical variable length gap pattern matching. In *Proc. 15th SEA*, pages 1–16, 2016.

7 Philip Bille and Inge Li Gørtz. The tree inclusion problem: In linear space and faster. *ACM Trans. Algorithms*, 7(3):1–47, 2011.

8 Philip Bille and Inge Li Gørtz. Substring range reporting. *Algorithmica*, 69(2):384–396, 2014.

9 Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, Eva Rotenberg, and Teresa Anna Steiner. String Indexing for Top-k Close Consecutive Occurrences. In *Proc. 40th FSTTCS*, volume 182, pages 14:1–14:17, 2020.

10 Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014.

11 Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind. String matching with variable length gaps. *Theoret. Comput. Sci.*, 443, 2012. Announced at SPIRE 2010.

12 Sudip Biswas, Arnab Ganguly, Rahul Shah, and Sharma V Thankachan. Ranked document retrieval for multiple patterns. *Theor. Comput. Sci.*, 746:98–111, 2018.

13 P Bucher and A Bairoch. A generalized profile syntax for biomolecular sequence motifs and its function in automatic sequence interpretation. In *Proc. 2nd ISMB*, pages 53–61, 1994.

14 Manuel Cáceres, Simon J Puglisi, and Bella Zhukova. Fast indexes for gapped pattern matching. In *Proc. 46th SOFSEM*, pages 493–504, 2020.

15 Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theor. Comput. Sci.*, 411(40-42):3795–3800, 2010.

16 Paolo Ferragina, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Two-dimensional substring indexing. *J. Comput. Syst. Sci.*, 66(4):763–774, 2003.

17 Greg N Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, 1997.

18 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.

19 Kimmo Fredriksson and Szymon Grabowski. Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. *Inf. Retr.*, 11(4):335–357, 2008.

20 Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional Lower Bounds for Space/Time Tradeoffs. In *Proc. 15th WADS*, pages 421–436. Springer, 2017.

21 Tuukka Haapasalo, Panu Silvasti, Seppo Sippu, and Eljas Soisalon-Soininen. Online dictionary matching with variable-length gaps. In *Proc. 10th SEA*, pages 76–87, 2011.

22 K Hofmann, P Bucher, L Falquet, and A Bairoch. The PROSITE database, its status in 1999. *Nucleic Acids Res*, 27(1):215–219, 1999.

23 Wing-Kai Hon, Manish Patil, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Indexes for document retrieval with relevance. In *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 351–362, 2013.

24 Wing-Kai Hon, Manish Patil, Rahul Shah, and Shih-Bin Wu. Efficient index for retrieving top-k most frequent documents. *J. Discrete Algorithms*, 8(4):402–417, 2010.

25 Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-k string retrieval. *J. ACM*, 61(2):1–36, 2014. Announced at 50th FOCS.

26 Wing-Kai Hon, Sharma V. Thankachan, Rahul Shah, and Jeffrey Scott Vitter. Faster compressed top-k document retrieval. In *Proc. 23rd DCC*, pages 341–350, 2013.

27 Costas S Iliopoulos and M Sohel Rahman. Indexing factors with gaps. *Algorithmica*, 55(1):60–70, 2009.

28 Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. Range non-overlapping indexing and successive list indexing. In *Proc. 11th WADS*, pages 625–636, 2007.

29 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proc. 27th SODA*, pages 1272–1287, 2016.

30 Kasper Green Larsen, J Ian Munro, Jesper Sindahl Nielsen, and Sharma V Thankachan. On hardness of several string indexing problems. *Theoret. Comput. Sci.*, 582:74–82, 2015.

31 Moshe Lewenstein. Indexing with gaps. In *Proc. 18th SPIRE*, pages 135–143, 2011.

32 Gerhard Mehldau and Gene Myers. A system for pattern matching applications on biosequences. *Bioinformatics*, 9(3):299–314, 1993.

33 J. Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V. Thankachan. Top-k term-proximity in succinct space. *Algorithmica*, 78(2):379–393, 2017. Announced at 25th ISAAC.

34 J. Ian Munro, Gonzalo Navarro, Rahul Shah, and Sharma V. Thankachan. Ranked document selection. *Theor. Comput. Sci.*, 812:149–159, 2020.

35 Eugene W. Myers. Approximate matching of network expressions with spacers. *J. Comput. Bio.*, 3(1):33–51, 1992.

36 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):1–47, 2014.

37 Gonzalo Navarro and Yakov Nekrich. Time-optimal top-k document retrieval. *SIAM J. Comput.*, 46(1):80–113, 2017. Announced at 23rd SODA.

38 Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *J. Comput. Bio.*, 10(6):903–923, 2003.

39 Gonzalo Navarro and Sharma V. Thankachan. New space/time tradeoffs for top-k document retrieval on sequences. *Theor. Comput. Sci.*, 542:83–97, 2014. Announced at 20th SPIRE.

40 Gonzalo Navarro and Sharma V. Thankachan. Reporting consecutive substring occurrences under bounded gap constraints. *Theor. Comput. Sci.*, 638:108–111, 2016. Announced at 26th CPM.

41 Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *Proc 13th SWAT*, pages 271–282, 2012.

42 Rahul Shah, Cheng Sheng, Sharma V. Thankachan, and Jeffrey Scott Vitter. Top-k document retrieval in external memory. In *Proc. 21st ESA*, pages 803–814, 2013.

43 Dekel Tsur. Top-k document retrieval in optimal space. *Inf. Process. Lett.*, 113(12):440–443, 2013.

44 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th FOCS*, pages 1–11, 1973.

45 Gelin Zhou. Two-dimensional range successor in optimal time and almost linear space. *Inf. Process. Lett.*, 116(2):171–174, 2016.

A Gapped Indexing for $[0, \beta]$ Gaps

In this section, we consider the special case where the queries are one sided intervals of the form $[0, \beta]$. We give a data structure supporting the following tradeoffs:

▶ **Theorem 6.** *Given a string of length n, we can*

(i) *construct an $O(n)$ space data structure that supports* Exists$(P_1, P_2, 0, \beta)$ *queries in $O(|P_1| + |P_2| + \sqrt{n}\log^{\epsilon} n)$ time for constant $\epsilon > 0$, or*

(ii) *construct an $O(n \log n)$ space data structure that supports* Count$(P_1, P_2, 0, \beta)$ *and* Report$(P_1, P_2, 0, \beta)$ *queries in $O(|P_1| + |P_2| + (\sqrt{n \cdot \mathrm{occ}}) \log\log n)$ time, where* occ *is the size of the output.*

Note that since the results match (up to log factors) the best known results for set intersection, this is about as good as we can hope for. We mention here that for this specific problem, a similar tradeoff follows from the strategies used by Hon et al. [25]. The results from

that paper include (among others) a data structure for documents such that given a query of two patterns P_1 and P_2 and a number k, one can output the k documents with the closest occurrences of P_1 and P_2. Thus, the problem is slightly different, however, with some adjustments, the results from Theorem 6 follow (up to a log factor). We show a simple, direct solution.

The data structure is a simpler version of the data structure considered in the previous sections. The main idea is that for each pair of boundary nodes u and v, we do not have to store an array of distances, but only one number that carries all the information: the smallest distance of a consecutive occurrence of $\text{str}(u)$ and $\text{str}(v)$. Thus, for existence, we can cluster with $\tau = \sqrt{n}$ to achieve linear space, and we do not need to check large distances separately. For the reporting solution, we store the decomposition from Section 4.1, and use the matrix M to decide where to recurse. In the following we will describe the details.

Existence data structure

For solving Exists queries in this setting, we cluster the suffix tree with parameter $\tau = \sqrt{n}$. Again, we store the linear space orthogonal range successor data structure by Nekrich and Navarro [41] on the suffix array. For each pair of boundary nodes (u, v), we store at $M(u, v)$ the minimum distance of a consecutive occurrence of $\text{str}(u)$ and $\text{str}(v)$. The total space is linear. To query, we proceed similarly as in Section 3 for the "small distances": We find the loci of P_1 and P_2. If any of the loci is not on the spine, we check all consecutive occurrences using $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$. If both loci are on the spine, denote $b_1,\ b_2$ the lower boundary nodes of the respective clusters. Find $M(b_1, b_2)$. If $M(b_1, b_2) \leq \beta$, we can immediately return Yes: If a valid occurrence (i', j') of $\text{str}(b_1)$ and $\text{str}(b_2)$ exists, then either (i', j') is a consecutive occurrence of P_1 and P_2, or there exists a consecutive occurrence of smaller distance. Otherwise, that is if $M(b_1, b_2) > \beta$, all valid occurrences (i, j) have the property that either i is in the cluster of $\text{locus}(P_1)$ or j is in the cluster of $\text{locus}(P_2)$, and we check all such pairs using $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$. The running time is $O(|P_1| + |P_2| + \sqrt{n}\log^{\epsilon} n)$.

Reporting data structure

For the reporting data structure, we store the decomposition of the suffix tree as described in Section 4.1 and the $O(n \log n)$ space orthogonal range successor data structure by Zhou [45] on the suffix array. For each induced subtree of level i in the decomposition, we store the existence data structure we just described.

Reporting algorithm

The algorithm follows a similar, but simpler, recursive structure as in Section 4. We begin by finding the loci of P_1 and P_2. If either of the loci is not on a spine, we find all consecutive occurrences using $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$, check if they are valid, report these, and terminate. If both loci are on a spine, we check $M(b_1, b_2)$ for the lower boundary nodes b_1 and b_2. If $M(b_1, b_2) > \beta$, all valid occurrences (i, j) have the property that either i is in the cluster of $\text{locus}(P_1)$ or j is in the cluster of $\text{locus}(P_2)$. We check all such pairs using $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$, report the valid occurrences, and terminate. If $M(b_1, b_2) \leq \beta$, we recurse on the children trees. That is, we check the border case and follow pointers to the loci in the children trees.

Analysis

The space is $O(n \log n)$, just as in Section 4.

For time analysis, we spend $O(\sqrt{\frac{n}{2^{l_i}}})$ orthogonal range successor queries on the nodes in the decomposition tree of level l_i where we stop the recursion. For all other nodes we visit in the tree traversal, we only spend a constant number of queries. In total, we visit $O(\text{occ} \log(n/\text{occ}) + \text{occ})$ decomposition tree nodes (by following the analysis in [15]), and we spend $O(\sqrt{\frac{n}{2^{l_i}}})$ orthogonal range successor queries on $O(\text{occ})$ many such nodes.

We use the same notation as in Section 4. By $x = O(\text{occ})$ we now denote the number of nodes where we stop the algorithm and output. Since each such node can be seen as a leaf in a binary tree, $\sum_{i=1}^{x} \frac{1}{2^{l_i}} \leq 1$. We use the Cauchy-Schwarz inequality (which is a special case of Hölders with $p = q = 2$). We get as an asymptotic bound for the number of orthogonal range successor queries:

$$\begin{aligned}\sum_{i=1}^{x} \sqrt{\frac{n}{2^{l_i}}} &= \sqrt{n} \sum_{i=1}^{x} \sqrt{\frac{1}{2^{l_i}}} \cdot 1 \\ &\leq \sqrt{n} \sqrt{\sum_{i=1}^{x} \frac{1}{2^{l_i}}} \sqrt{\sum_{i=1}^{x} 1} \\ &\leq \sqrt{nx} = O(\sqrt{n \cdot \text{occ}}).\end{aligned}$$

Note that since $\text{occ} \log(n/\text{occ}) = O(\text{occ}\sqrt{n/\text{occ}}) = O(\sqrt{n \cdot \text{occ}})$, this brings the total number of orthogonal range successor queries to $O(\text{occ} + \sqrt{n \cdot \text{occ}})$. Using the data structure by Zhou [45], the time bound from Theorem 6 follows.

Disorders and Permutations

Laurent Bulteau
LIGM, Univ Gustave Eiffel, CNRS, F-77454 Marne-la-Vallée, France

Samuele Giraudo
LIGM, Univ Gustave Eiffel, CNRS, F-77454 Marne-la-Vallée, France

Stéphane Vialette
LIGM, Univ Gustave Eiffel, CNRS, F-77454 Marne-la-Vallée, France

Abstract

The *additive x-disorder* of a permutation is the sum of the absolute differences of all pairs of consecutive elements. We show that the additive x-disorder of a permutation of $S(n)$, $n \geq 2$, ranges from $n-1$ to $\lfloor n^2/2 \rfloor - 1$, and we give a complete characterization of permutations having extreme such values. Moreover, for any positive integers n and d such that $n \geq 2$ and $n-1 \leq d \leq \lfloor n^2/2 \rfloor - 1$, we propose a linear-time algorithm to compute a permutation $\pi \in S(n)$ with additive x-disorder d.

2012 ACM Subject Classification Mathematics of computing → Permutations and combinations

Keywords and phrases Permutation, Algorithm

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.11

1 Introduction

Here we follow a young researcher in computer science who is about to pass an audition for a permanent position in a prestigious university. As she arrived early in the main building of the university, she decides to use one of the elevators to change her mind before reaching the audition room on time. The chosen elevator has n buttons to move to the floor 1, 2, ..., n of the building. To move from a floor a to a floor b, the elevator takes $|b-a|$ seconds, regardless of whether it goes up or down. Our candidate, loving the challenge herself, decides to visit all floors once and only once each. Knowing that she arrived d seconds early, how can she propose a route that takes exactly that long? And for which values d is there at least one solution? It is assumed that the candidate can reach the initial floor of her ballad instantly from the university entry hall and reach the dreaded audition room instantly from the last visited floor. Fig. 1 shows an example.

We tackle this combinatorial problem by studying additive disorders of permutations. Let $\pi \in S(n)$ be a permutation of size $n \geq 2$. The *x-difference sequence* of π is the $(n-1)$-sequence constructed by considering the absolute difference of all pairs of adjacent letters of π, and its *y-difference sequence* is constructed by considering all distances between two consecutive values in π. Moreover, the *additive x-disorder* of π is the sum of the integers in its x-difference sequence and the *additive y-disorder* of π is the sum of the integers in its y-difference sequence. For example, the x-difference sequence of $\pi = 514263 \in S(6)$ is $(4, 3, 2, 4, 3)$, its y-difference sequence is $(2, 2, 3, 2, 4)$, its additive x-disorder is 16, and its additive y-disorder is 13.

These values associated with permutations are actually statistics: they are maps from combinatorial objects to integers. The literature in algorithmic and combinatorics abounds with examples and studies of similar statistics on permutations. One can cite for instance the major index [5], the inversion number [4], the total displacement [4] (Problem 5.1.1.28), the descent number [2], and the number of cycles [6] of permutations. The present paper is intended to be a first study of these just described disorder statistics.

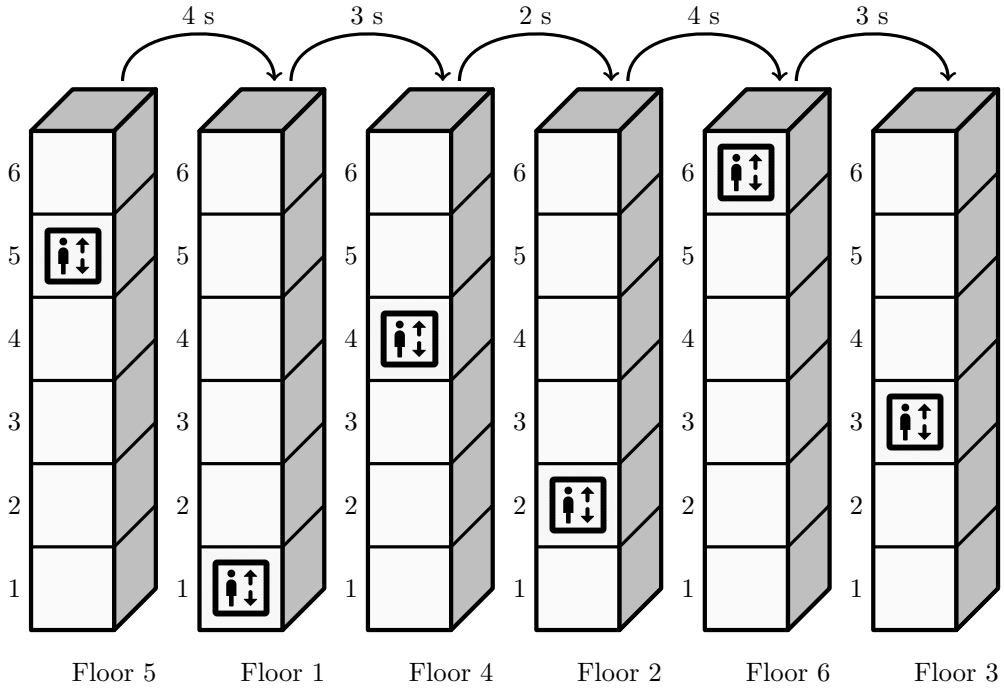


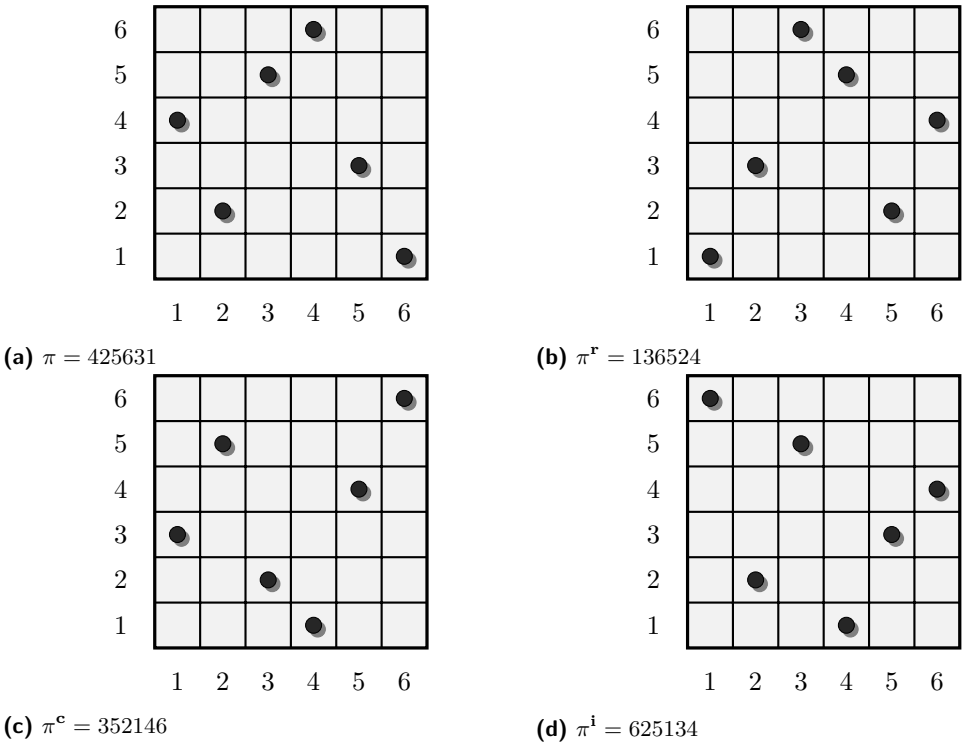
Figure 1 The candidate has to visit exactly each of the six floors of the building. She visits them in following order: 5, 1, 4, 2, 6, and 3. The total duration is 16 s. The candidate would have achieved the maximum duration by visiting floors in the following order: 4, 1, 5, 2, 6, and 3. In this case, the total duration is 17 s (the solution is not unique as the order 3, 5, 1, 6, 2 and 4 provides another solution with total duration 17 s).

The maximum additive x-disorder of a permutation in $S(n)$, $n \geq 2$, is given by Sequence A047838 of the OEIS[1]. More precisely, this sequence is concerned with maximum additive y-disorder, but as we will show soon, the maximum additive x-disorder and the maximum additive y-disorder of permutations in $S(n)$ coincide. It is conjectured[2] that the maximum additive x-disorder of a permutation in $S(n)$ is $\lfloor n^2/2 \rfloor - 1$. We prove that the conjecture is correct.

Given an $(n-1)$-sequence of positive integers D, it is shown in [3] that deciding whether there exists some permutation $\pi \in S(n)$ such that D is the x-difference sequence of π is NP-complete. Pursuing this line of research, we complement [3] by showing that for any integer d with $n - 1 \leq d \leq \lfloor n^2/2 \rfloor - 1$, there exists a permutation $\pi \in S(n)$ with additive x-disorder d. The proof is constructive. Note that, given an n-sequence of positive integers $D = (d_1, d_2, \ldots, d_n)$, deciding whether there exist two permutations $\pi, \sigma \in S(n)$ such that $d_i = \pi(i) + \sigma(i)$ for $1 \leq i \leq n$ is NP-complete [7].

This paper is organized as follows. Section 2 gives concise background and notation for the disorder setting. We prove that the maximum additive x-disorder of a permutation in $S(n)$ is $\lfloor n^2/2 \rfloor - 1$ in Section 3, and in Section 3.4 that there exists a permutation that achieves any legal additive x-disorder (our approach is constructive).

[1] `https://oeis.org/A047838`
[2] More precisely, the upper bound relies on correctness of Sequence A007590 of the OEIS.



(a) $\pi = 425631$ **(b)** $\pi^{\mathbf{r}} = 136524$

(c) $\pi^{\mathbf{c}} = 352146$ **(d)** $\pi^{\mathbf{i}} = 625134$

Figure 2 Trivial bijections of $\pi = 425631$.

2 Definitions

For any non-negative integer n, we let $[n]$ stand for the set $\{1, 2, \ldots, n\}$. A *permutation* of size n is a one-to-one mapping $[n] \to [n]$. The set of all permutations of size n is denoted by $S(n)$. For a permutation $\pi \in S(n)$, we write $\pi(i)$ for the integer at position i, $i \in [n]$.

Let $\pi \in S(n)$. The *reverse* of π is the permutation $\pi^{\mathbf{r}}$ defined by $\pi^{\mathbf{r}}(i) = \pi(n - i + 1)$ for every $i \in [n]$. The *complement* of π is the permutation $\pi^{\mathbf{c}}$ defined by $\pi^{\mathbf{c}}(i) = n - \pi(i) + 1$ for every $i \in [n]$. The *inverse* is the regular group theoretical inverse on permutations, $\pi^{\mathbf{i}}$ is defined by $\pi^{\mathbf{i}}(i) = j$ if and only if $\pi(j) = i$ for every $i \in [n]$. See Fig. 2. The reverse, complement, and inverse are called the *trivial bijections* from $S(n)$ to itself [1].

Let $\pi \in S(n)$, $n \geq 2$. The *x-difference sequence* [3] of π, denoted $\Delta_x(\pi)$, is the $(n-1)$-sequence defined by

$$\Delta_x(\pi) = (|\pi(2) - \pi(1)|, |\pi(3) - \pi(2)|, \ldots, |\pi(n) - \pi(n-1)|)\,.$$

The *y-difference sequence* of π, denoted $\Delta_y(\pi)$, is the $(n-1)$-sequence defined by $\Delta_y(\pi) = \Delta_x(\pi^{\mathbf{i}})$. See Fig. 3 for an illustration.

The *additive x-disorder* (resp. *additive y-disorder*) of π, denoted by $\delta_x^+(\pi)$ (resp. $\delta_y^+(\pi)$), is defined by $\delta_x^+(\pi) = \sum_{d \in \Delta_x(\pi)} d$ (resp. $\delta_y^+(\pi) = \sum_{d \in \Delta_y(\pi)} d$).

▶ **Example 1.** See Fig. 3 for two examples. Besides, by setting $\pi = 251463$, we have $\Delta_x(\pi) = (3, 4, 3, 2, 3)$, $\Delta_y(\pi) = (2, 5, 2, 2, 3)$, $\delta_x^+(\pi) = 3 + 4 + 3 + 2 + 3 = 15$ and $\delta_y^+(\pi) = 2 + 5 + 2 + 2 + 3 = 14$.

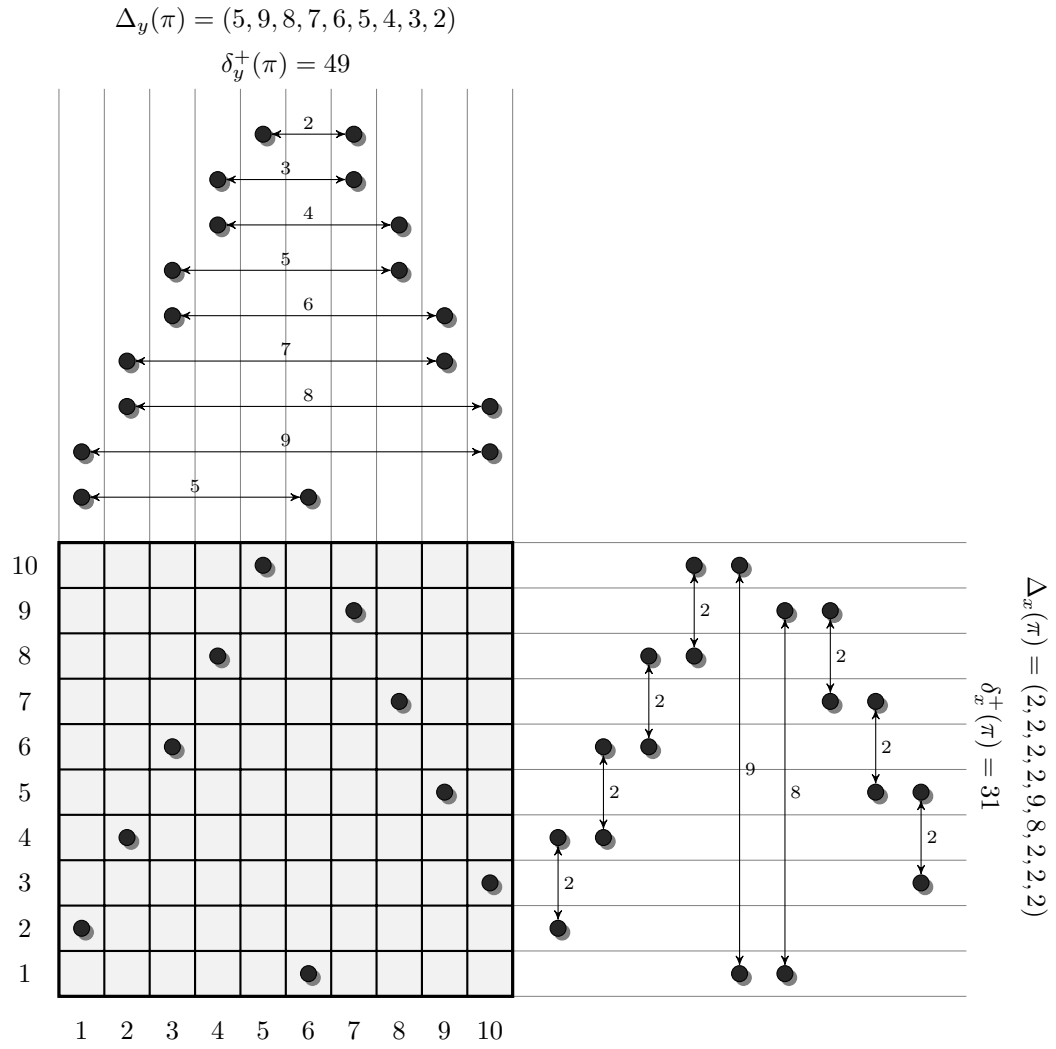


Figure 3 The x-difference sequence, the y-difference sequence, the additive x-disorder, and the additive y-disorder of the permutation $\pi = 2468A19753$ ("A" stands for "10").

3 Bounds on Additive Disorder

In this section, we show that the additive x-disorder and y-disorder of a permutation of $S(n)$ ranges from $n-1$ to $\lfloor n^2/2 \rfloor - 1$. More precisely, we give a complete characterization of permutations having extreme such values (Theorems 3 and 8), and show that every value in this range is the additive x-disorder or y-disorder of some permutation (Theorem 10).

3.1 Basic properties

▶ **Lemma 2.** *For every $\pi \in S(n)$, $n \geq 2$, the four following assertions hold:*

1. $(\Delta_x(\pi))^{\mathbf{r}} = \Delta_x(\pi^{\mathbf{r}})$;
2. $\Delta_x(\pi) = \Delta_x(\pi^{\mathbf{c}})$;
3. $\delta_x^+(\pi) = \delta_y^+(\pi^{\mathbf{i}})$;
4. $\delta_y^+(\pi) = \delta_x^+(\pi^{\mathbf{i}})$.

Proof. The equality $(\Delta_x(\pi))^{\mathbf{r}} = \Delta_x(\pi^{\mathbf{r}})$ is obvious. As for $\Delta_x(\pi) = \Delta_x(\pi^{\mathbf{c}})$, it is enough to observe that, for every $1 \leq i < n$,

$$\begin{aligned}|\pi(i+1) - \pi(i)| &= |\pi(i+1) - (n+1) - \pi(i) + (n+1)| \\ &= |(n - \pi(i+1) + 1) - (n - \pi(i) + 1)| \\ &= |\pi^{\mathbf{c}}(i+1) - \pi^{\mathbf{c}}(i)|\,.\end{aligned}$$

The last two assertions are direct consequences of the definition of the y-difference sequence of π as the x-difference sequence of the inverse of π. ◀

The last two assertions of Lemma 2 imply that all results about additive x-disorders of permutations can be rephrased in terms of additive y-disorders and conversely. For this reason, in what follows we shall focus on additive x-disorder and refer to it simply as *additive disorder*.

3.2 Minimum disorder

▶ **Theorem 3.** *The minimum possible additive disorder of a permutation of $S(n)$ is $n-1$. It is attained exactly by the identity permutation and its reverse.*

Proof. In any permutation $\pi \in S(n)$, $|\pi(i+1) - \pi(i)| \geq 1$, so $\delta_x^+(\pi) \geq n-1$. The bound is reached if $\pi(i+1) \in \{\pi(i+1), \pi(i-1)\}$ for all i. In particular, if $\pi(i) \in \{1, n\}$, then $i \in \{1, n\}$ as well (otherwise one of $\pi(i-1), \pi(i+1)$ would not be at distance 1 from $\pi(i)$). Assume $\pi(1) = 1$, then for each j, $\pi(j) = j$ (by induction, $\pi(j+1) \in \{\pi(j)+1, \pi(j)-1\}$, and $\pi(j+1) \neq \pi(j-1) = \pi(j) - 1$). So π is the identity. Similarly if $\pi(1) = n$, then π is the reverse of the identity. ◀

3.3 Maximum disorder

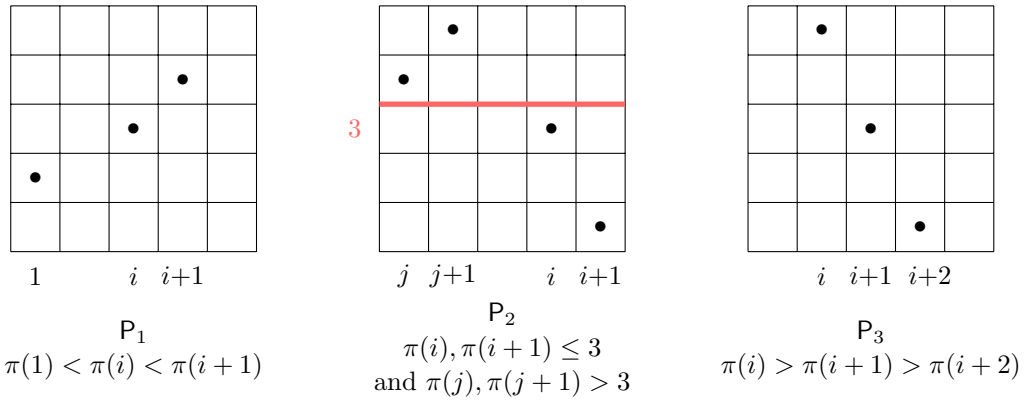
A permutation $\pi \in S(n)$ is *bipartite with threshold k* if $k \in \{\lfloor n/2 \rfloor, \lceil n/2 \rceil\}$ and for every $i \in [n-1]$, either $\pi(i) \leq k$ and $\pi(i+1) > k$, or $\pi(i) > k$ and $\pi(i+1) \leq k$. Such a permutation has *centered endpoints* if $\{\pi(1), \pi(n)\}$ is either $\{\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1\}$ (if $k = \lceil n/2 \rceil$) or $\{\lceil n/2 \rceil, \lceil n/2 \rceil + 1\}$ (if $k = \lfloor n/2 \rfloor$).

▶ **Example 4.** The permutation $\pi = 25371648$ of $S(8)$ is bipartite with threshold 4 and has no centered endpoints. The permutation $\pi = 46172835$ of $S(8)$ is bipartite with threshold 4 and has centered endpoints. The permutation $\pi = 41523$ of $S(5)$ is bipartite with threshold 2 and has centered endpoints. The permutation $\pi = 34152$ of $S(5)$ is bipartite with threshold 3 and has centered endpoints.

We say that permutation π has pattern P_1, P_2, or P_3 if it satisfies the following properties, respectively (see Fig. 4, we then show that as forbidden patterns they characterize permutations with maximal disorder):

- Pattern P_1 (*extreme endpoint*). There is $j \in [n-1]$ such that
  1. $\pi(1) < \pi(j) < \pi(j+1)$,
  2. or $\pi(1) > \pi(j) > \pi(j+1)$,
  3. or $\pi(j) < \pi(j+1) < \pi(n)$,
  4. or $\pi(j) > \pi(j+1) > \pi(n)$.
- Pattern P_2 (*two separated pairs*). There are $i, j \in [n-1]$ and $k \in [n]$ such that
  1. $\pi(i), \pi(i+1) \leq k$,
  2. and $\pi(j), \pi(j+1) > k$.

- Pattern P_3 (*three in a row*). There is $j \in [n-2]$ such that
  1. $\pi(j) < \pi(j+1) < \pi(j+2)$,
  2. or $\pi(j) > \pi(j+1) > \pi(j+2)$.



P_1
$\pi(1) < \pi(i) < \pi(i+1)$

P_2
$\pi(i), \pi(i+1) \leq 3$
and $\pi(j), \pi(j+1) > 3$

P_3
$\pi(i) > \pi(i+1) > \pi(i+2)$

Figure 4 Forbidden patterns in maximal disorder permutations.

▶ **Lemma 5.** *A permutation that does not have patterns* P_1, P_2, *and* P_3 *is bipartite and has centered endpoints.*

Proof. Let $a = \max_{i\in[n-1]}(\min\{\pi(i), \pi(i+1)\})$ and $b = \min_{i\in[n-1]}(\max\{\pi(i), \pi(i+1)\})$.

If $b < a$, then for some i, j we have $\pi(i), \pi(i+1) \leq b < a \leq \pi(j), \pi(j+1)$, *i.e.*, π has pattern P_2.

If $a = b$, let j such that $\pi(j) = a = b$. Then one of $\pi(j-1)$, $\pi(j+1)$ must be larger than $\pi(j)$ (by definition of a), and the other must be smaller than $\pi(j)$ (by definition of b). In particular, $j \neq 1, n$ and π has pattern P_3.

If $a > b$. Let $A = \{h \mid \pi(h) \leq a\}$ and $B = \{h \mid \pi(h) \geq b\}$. Then A and B are disjoint, $a = |A|$ and $b = n + 1 - |B|$. Moreover, each set $\{i, i+1\}$ contains one element in A and one in B, so A, B is a partition of $[n]$ (in other words, $b = a + 1$), and $a = |A| \in \{\lfloor n/2 \rfloor, \lceil n/2 \rceil\}$. Overall for every $i \neq n$, $\max\{\pi(i), \pi(i+1)\} > a$ and $\min\{\pi(i), \pi(i+1)\} \leq a$, so π is bipartite with threshold a.

To show that endpoints are centered, first note that if n is even, then $\{1, n\}$ has one element in A, the other in B. If n is odd, either $a = \lceil n/2 \rceil$ and $\{1, n\} \subseteq A$, or $a = \lfloor n/2 \rfloor$ and $\{1, n\} \subseteq B$.

If $\pi(1) < a$ let h be any position such that $\pi(1) < \pi(h) \leq a$. Then $h = n$ (otherwise, by definition of a, $\pi(1) < \pi(h) < \pi(h+1)$ and π has pattern P_1 version 1). So in particular, there can be only one such value of h, so $\pi(1) = a - 1$. Furthermore, $\{1, n\} \subseteq A$ so n is odd and $\{\pi(1), \pi(n)\} = \{a, a-1\} = \{\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1\}$, so π has centered endpoints.

Similarly if $\pi(1) > b$, we have $\pi(1) = b + 1$, $\pi(n) = b$ and n is odd with $\{\pi(1), \pi(n)\} = \{\lceil n/2 \rceil, \lceil n/2 \rceil + 1\}$ and π has centered endpoints.

The same arguments apply if $\pi(n) < a$ or $\pi(n) > b$, so the only case left is $\{\pi(1), \pi(n)\} = \{a, b\}$, which yields that n is even and $\{\pi(1), \pi(n)\} = \{\frac{n}{2}, \frac{n}{2} + 1\}$, so π has centered endpoints. ◀

▶ **Lemma 6.** *If* π *has one of patterns* P_1, P_2, *or* P_3, *then there exists* π' *such that* $\delta_x^+(\pi') > \delta_x^+(\pi)$.

Proof. Pattern P_1 version 1: for some j, $\pi(1) < \pi(j) < \pi(j+1)$. Let π' be the permutation obtained from π by reversing positions 1 to j. Then $\delta_x^+(\pi') = \delta_x^+(\pi) - (\pi(j+1) - \pi(j)) + (\pi(j+1) - \pi(1)) = \delta_x^+(\pi) + \pi(j) - \pi(1) > \delta_x^+(\pi)$. Patterns P_1 versions 2, 3, and 4 are symmetrical.

Pattern P_2: Depending on the relative order of i and j, of $\pi(i)$ and $\pi(i+1)$, and of $\pi(j)$ and $\pi(j+1)$ we have a total of eight cases to check. Assuming $i < j$ leaves the four following alternatives, which correspond to two distinct patterns, up to symmetry:

- Pattern P_2' (*monotonous pairs*). There are $i < j \in [n-1]$ such that
  1. $\pi(i) < \pi(i+1) < \pi(j) < \pi(j+1)$
  2. or $\pi(i+1) < \pi(i) < \pi(j+1) < \pi(j)$.
- Pattern P_2'' (*non-monotonous pairs*).There are $i < j \in [n-1]$ such that
  1. $\pi(i) < \pi(i+1) < \pi(j+1) < \pi(j)$
  2. or $\pi(i+1) < \pi(i) < \pi(j) < \pi(j+1)$.

Pattern P_2' version 1: for some $i < j$, $\pi(i) < \pi(i+1) < \pi(j) < \pi(j+1)$. Let π' be the permutation obtained from π by reversing positions $i+1$ to j. Then $\delta_x^+(\pi') = \delta_x^+(\pi) - (\pi(i+1) - \pi(i)) - (\pi(j+1) - \pi(j)) + (\pi(j) - \pi(i)) + (\pi(j+1) - \pi(i+1)) = \delta_x^+(\pi) + 2(\pi(j) - \pi(i+1)) > \delta_x^+(\pi)$.

Pattern P_2'' version 1: for some $i < j$, $\pi(i) < \pi(i+1) < \pi(j+1) < \pi(j)$. Let π' be the permutation obtained from π by reversing positions $i+1$ to j. Then $\delta_x^+(\pi') = \delta_x^+(\pi) - (\pi(i+1) - \pi(i)) - (\pi(j) - \pi(j+1)) + (\pi(j) - \pi(i)) + (\pi(j+1) - \pi(i+1)) = \delta_x^+(\pi) + 2(\pi(j+1) - \pi(i)) > \delta_x^+(\pi)$.

This completes the proof of P_2.

Pattern P_3 version 1: for some j, $\pi(j) < \pi(j+1) < \pi(j+2)$. Let π' be the permutation obtained from π by moving $\pi(j+1)$ to position 1. Then $\delta_x^+(\pi') = \delta_x^+(\pi) - (\pi(j+1) - \pi(j)) - (\pi(j+2) - \pi(j+1)) + (\pi(j+2) - \pi(j)) + |\pi(j+1) - \pi(1)| = \delta_x^+(\pi) + |\pi(j+1) - \pi(1)| > \delta_x^+(\pi)$. Pattern P_3 version 2 is symmetrical. ◀

▶ **Lemma 7.** *The additive disorder of a bipartite permutation $\pi \in S(n)$ is*

$$\delta_x^+(\pi) = \lfloor n^2/2 \rfloor - |\pi(1) - \lceil n/2 \rceil| - |\pi(n) - \lceil n/2 \rceil| .$$

Proof. Let $m = \lfloor n/2 \rfloor$, and k be a threshold for which π is bipartite. If n is even then $\lfloor n^2/2 \rfloor - 1 = 2m^2 - 1$, and for $n = 2m+1$, $\lfloor n^2/2 \rfloor - 1 = \frac{1}{2}((2m+1)^2 - 1) - 1 = 2m^2 + 2m - 1$

By the definition of bipartite, for any i, $|\pi(i+1) - \pi(i)| = |\pi(i) - k| + |\pi(i+1) - k|$. Thus,

$$\begin{aligned}\delta_x^+(\pi) + |\pi(1) - k| + |\pi(n) - k| &= |\pi(1) - k| + \left(\sum_{i=1}^{n-1} |\pi(i+1) - \pi(i)| \right) + |\pi(n) - k| \\ &= 2 \sum_{i=1}^{n} |\pi(i) - k|.\end{aligned}$$

We introduce the partition $H \cup L$ of $[1, n]$ as $L = \{i \mid \pi(i) \leq k\}$ and $H = \{i \mid \pi(i) > k\}$ (in particular, $|L| = k$ and $|H| = n - k$). Note that $i \to \pi(i) - k$ is a bijection between H and $[1, n-k]$, and that $i \to k - \pi(i)$ is a bijection between L and $[0, k-1]$. We have

$$\sum_{i \in H} |\pi(i) - k| = \sum_{i \in H} (\pi(i) - k) = \sum_{j=1}^{n-k} j = (n-k)(n-k+1)$$

and

$$\sum_{i\in L} |\pi(i) - k| = \sum_{i\in L} (k - \pi(i)) = \sum_{j=0}^{k-1} j = (k-1)k.$$

So overall

$$\begin{aligned} 2\sum_{i=1}^{n} |\pi(i) - k| &= (n-k)(n-k+1) + (k-1)k \\ &= (n-k)^2 + k^2 + n - 2k. \end{aligned}$$

If n is even ($n = 2m$), then $k = n - k = m$ and $n - 2k = 0$.

$$2\sum_{i=1}^{n} |\pi(i) - k| = 2m^2 = \frac{n^2}{2}.$$

Also, $|\pi(i) - k| = |\pi(i) - \lceil n/2 \rceil|$ for $i = 1$ and $i = n$, so this concludes the proof when n is even.

If n is odd ($n = 2m + 1$), k can be m or $m + 1$. If $k = m$, then $n - k = m + 1$ and $n - 2k = 1$.

$$2\sum_{i=1}^{n} |\pi(i) - k| = (m+1)^2 + m^2 + 1.$$

Also, $\pi(1)$ and $\pi(n)$ are both greater than k and $k = \lceil n/2 \rceil - 1$, so

$$|\pi(1) - k| + |\pi(n) - k| = |\pi(1) - \lceil n/2 \rceil| + |\pi(n) - \lceil n/2 \rceil| + 2.$$

This gives the following disorder:

$$\delta_x^+(\pi) = m^2 + (m+1)^2 - |\pi(1) - \lceil n/2 \rceil| - |\pi(n) - \lceil n/2 \rceil| - 1.$$

Otherwise, if $k = m + 1$, then $n - k = m$ and $n - 2k = -1$.

$$2\sum_{i=1}^{n} |\pi(i) - k| = m^2 + (m+1)^2 - 1.$$

Also, $k = \lceil n/2 \rceil$, so

$$|\pi(1) - k| + |\pi(n) - k| = |\pi(1) - \lceil n/2 \rceil| + |\pi(n) - \lceil n/2 \rceil|.$$

This gives the same formula for the additive disorder:

$$\delta_x^+(\pi) = m^2 + (m+1)^2 - |\pi(1) - \lceil n/2 \rceil| - |\pi(n) - \lceil n/2 \rceil| - 1.$$

Note that $m^2 + (m+1)^2 - 1 = 2m^2 + 2m = \frac{1}{2}((2m+1)^2 - 1) = \lfloor n^2/2 \rfloor$, so this completes the proof when n is odd. ◀

▶ **Theorem 8.** *The maximum possible additive disorder of a permutation of $S(n)$ is $\lfloor n^2/2 \rfloor - 1$. It is attained exactly by bipartite permutations with centered endpoints.*

Proof. First, note that according to Lemma 7, any bipartite permutation with centered endpoints has disorder $\lfloor n^2/2 \rfloor - 1$.

Conversely, let π be a permutation with maximal disorder. It may not have any of the patterns P_1, P_2, or P_3 by Lemma 6, hence it is bipartite with centered endpoints by Lemma 5. ◀

Algorithm 1 Given positive integers n and d, the algorithm returns a permutation $\pi \in S(n)$ such that $\delta_x^+(\pi) = d$.

```
1  Realization(n, d)
2  if d < n − 1 or d > ⌊n²/2⌋ − 1 then
3      return error
4  else if d = n − 1 then
5      return 12...n
6  else if d ≤ ⌊(n − 1)²/2⌋ + 1 then
7      π ← Realization(n − 1, d − 2)
8      i ← max(2, Position(π, n − 1))
9      return Insertion(π, i, n)
10 else
11     d′ ← ⌊n²/2⌋ − d
12     π ← 12...⌈n/2⌉
13     σ ← n(n − 1)...(⌈n/2⌉ + 1)
14     if d′ ≤ ⌊n/2⌋ then
15         i ← ⌊n/2⌋ + 1
16         j ← i − d′
17     else
18         i ← ⌊n/2⌋ + 1 + (−1)^(n mod 2)(d′ − ⌊n/2⌋)
19         j ← 1
20     π ← PutFirst(π, j)
21     if i ∈ π then
22         π ← PutLast(π, i)
23     else
24         σ ← PutLast(σ, i)
25     return Interleave(π, σ)
```

3.4 All disorders in the range can be achieved

To state the upcoming algorithm, let us set some definitions. For any word u of length n, any word v of length m, any $i \in [n]$, and any letter a, let

- `Position(`u`,` a`)` be the position of a in u when a occurs in u;
- `Insertion(`u`,` i`,` a`)` be the word $u(1)\dots u(i-1)au(i)\dots u(n)$;
- `PutFirst(`u`,` a`)` (resp. `PutLast(`u`,` a`)`), where a is at position i in u, be the word $au(1)\dots u(i-1)u(i+1)\dots u(n)$ (resp. $u(1)\dots u(i-1)u(i+1)\dots u(n)a$);
- `Interleave(`u`,` v`)` be the word $u(1)v(1)u(2)v(2)\dots u(k)v(k)w$ where $k = \min\{n, m\}$ and w is the suffix of u of length $n - m$ if $n - m \geq 0$ or the suffix of v of length $m - n$ otherwise.

Let us now consider the algorithm `Realization`, taking as inputs a value $n \geq 2$ and an integer d, and outputting when this is possible a permutation of $S(n)$ having d as additive disorder (see Algorithm 1).

▶ **Example 9.** Table 1 shows some permutations built by `Realization(`n`,` d`)`.

Table 1 The permutations built by `Realization`(n, d), where $2 \leq n \leq 7$ and $1 \leq d \leq 23$.

	n						
d	1	2	3	4	5	6	7
1	_	12	_	_	_	_	_
2	_	_	123	_	_	_	_
3	_	_	132	1234	_	_	_
4	_	_	_	1243	12345	_	_
5	_	_	_	1432	12354	123456	_
6	_	_	_	1423	12543	123465	1234567
7	_	_	_	2413	15432	123654	1234576
8	_	_	_	_	15423	126543	1234765
9	_	_	_	_	25413	165432	1237654
10	_	_	_	_	15243	165423	1276543
11	_	_	_	_	25143	265413	1765432
12	_	_	_	_	_	165243	1765423
13	_	_	_	_	_	265143	2765413
14	_	_	_	_	_	162435	1765243
15	_	_	_	_	_	162534	2765143
16	_	_	_	_	_	261534	1762435
17	_	_	_	_	_	361524	1762534
18	_	_	_	_	_	_	2761534
19	_	_	_	_	_	_	3761524
20	_	_	_	_	_	_	1726453
21	_	_	_	_	_	_	1726354
22	_	_	_	_	_	_	2716354
23	_	_	_	_	_	_	3716254

We note that Algorithm 1 runs in polynomial time in n and d. In fact, provided the data structure used for permutations allows constant-time insertions of elements before and after n, then it is actually linear. To this end, double-ended queues with a pointer to the highest value are a solution.

▶ **Theorem 10.** *For any n and any $n-1 \leq d \leq \lfloor n^2/2 \rfloor - 1$, Algorithm 1 yields a permutation $\pi \in S(n)$ with $\delta_x^+(\pi) = d$ in linear-time w.r.t. n.*

Proof. The proof is by induction on n, assume that the theorem is true for $n-1$. We write π^* for the permutation returned by `Realization`(n, d). We distinguish three cases depending on the value of d.

If $d = n - 1$, then π^* is the identity permutation and $\delta_x^+(\pi^*) = d$.

If $n \leq d \leq \lfloor (n-1)^2/2 \rfloor + 1$, then by induction π (line 7) is a permutation of $[n-1]$ with $\delta_x^+(\pi') = d-2$ (since $n-2 \leq d-2 \leq \lfloor (n-1)^2/2 \rfloor - 1$). By the choice of i, we have $2 \leq i \leq n$ and $\{\pi_{i-1}, \pi_i\} = \{n-1, x\}$ for some $1 \leq x < n-1$. Then $\pi^* = (\pi_1, \ldots, \pi_{i-1}, n, \pi_i, \ldots, \pi_n)$, so

$$\begin{aligned}\delta_x^+(\pi^*) &= \delta_x^+(\pi) - |\pi_{i-1} - \pi_i| + |n - \pi_i| + |n - \pi_{i-1}| \\ &= \delta_x^+(\pi) - (n-1-x) + (n-(n-1)) + (n-x) \\ &= (d-2) + 2 = d.\end{aligned}$$

Finally, if $\lfloor (n-1)^2/2 \rfloor + 2 \leq d \leq \lfloor n^2/2 \rfloor - 1$. We have $d' = \lfloor n^2/2 \rfloor - d$ (line 11). The value of d' is bounded as follows

$$1 \leq d' \leq \lfloor n^2/2 \rfloor - \lfloor (n-1)^2/2 \rfloor - 2 = \lfloor n^2/2 \rfloor - \lfloor (n^2+1)/2 \rfloor + n - 2 \leq n - 2.$$

Note that π^* is built as the interleaving of π (containing $\{1, \ldots, \lceil n/2 \rceil\}$) and σ (containing $\{\lceil n/2 \rceil + 1, \ldots, n\}$), so it is a bipartite permutation. By Lemma 7, it suffices to verify that $|\pi^*(1) - \lceil n/2 \rceil| + |\pi^*(n) - \lceil n/2 \rceil| = d'$. Values i and j are defined lines 14 to 19. First remark, using $d' \leq n-2$, that $1 \leq j \leq n$, $1 \leq i \leq n$, and $i \neq j$. We show that (i) $\pi^*(1) = j$, (ii) $\pi^*(n) = i$, and (iii) $|j - \lceil n/2 \rceil| + |i - \lceil n/2 \rceil| = d'$. Property (i) is clear since $1 \leq j \leq \lfloor n/2 \rfloor$ by construction, so $\pi(1) = j$ after line 20, and finally $\pi^*(1) = j$. Towards (ii), note that i is the last element of either π or σ after line 24, so it suffices to show that $i \leq \lceil n/2 \rceil$ if and only if n is odd. We now discuss specific cases depending on the value of d' and the parity of n.

If $d' \leq \lfloor n/2 \rfloor$, we have $i = \lfloor n/2 \rfloor + 1$, so $i \leq \lceil n/2 \rceil$ iff n is odd (so $\pi^*(n) = i$). Furthermore, $j \leq \lceil n/2 \rceil \leq i$, so $|j - \lceil n/2 \rceil| + |i - \lceil n/2 \rceil| = i - j = d'$.

If $d' > \lfloor n/2 \rfloor$, we have $j = 1$ and $i = \lfloor n/2 \rfloor + 1 + (-1)^{n \mod 2}(d' - \lfloor n/2 \rfloor)$. So $|j - \lceil n/2 \rceil| = \lceil n/2 \rceil - 1$. If n is odd, $i < \lceil n/2 \rceil$ and $|i - \lceil n/2 \rceil| = d' - \lceil n/2 \rceil + 1$ and $|i - \lceil n/2 \rceil| + |j - \lceil n/2 \rceil| = d'$. If n is even, $i = d' + 1 \geq \lceil n/2 \rceil$, and $|i - \lceil n/2 \rceil| + |j - \lceil n/2 \rceil| = d'$.

The linearity of the algorithm w.r.t. n is clear. Indeed, the only trick consists, in the case starting at line 7, in having a constant-time insertion of the letter n in the permutation π returned by the recursive call. Since n is always inserted adjacent to the letter $n-1$, it is enough to store the position of the last letter to achieve the claimed complexity. ◀

4 Concluding remarks

There are many questions left open in this paper. Below we briefly discuss three directions for further research.

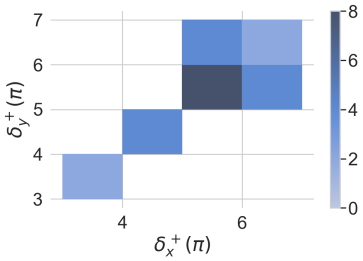
1. Sure enough, our candidate that arrived d seconds early has to start at some given floor i to reach the audition at some another floor j. How can she propose a route that starts at floor i, ends at floor j and takes exactly that long? And for which values d is there at least one solution? Note that for $n = 4$, if one focus on permutations that start with 1 and end with 4, we have $\delta_x^+(1234) = 3$, $\delta_x^+(1324) = 5$ but no permutation $\pi \in S(4)$ starting with 1 and ending with 4 achieves $\delta_x^+(\pi) = 4$.
2. Given an $(n-1)$-sequence D_x, it is NP-complete to decide whether there exists a permutation $\pi \in S(n)$ such that $\Delta_x(\pi) = D_x$. This was proved by M. De Biasi [3]. It is natural to ask for the following extension: Given two $(n-1)$-sequences D_x and D_y, how hard is the problem to decide whether there exists a permutation $\pi \in S(n)$ such that $\Delta_x(\pi) = D_x$ and $\Delta_y(\pi) = D_y$? What about the case $D_x = D_y$? See Table 2 for the landscape of $S(4)$.
3. We have shown that for any positive integer d_x, $n - 1 \leq d_x \leq \lfloor n^2/2 \rfloor - 1$, one can construct in linear-time a permutation $\pi \in S(n)$ such that $d_x = \delta_x^+(\pi)$. The most natural question to ask is: Given two positive integers d_x and d_y, how hard is the problem to decide whether there exists a permutation π such that $d_x = \delta_x^+(\pi)$ and $d_y = \delta_y^+(\pi)$? Again, what about the case $d_x = d_y$? See Table 3 for the landscape of $S(4)$ and refer to Fig. 5 for visualizing the distribution of points $(\delta_x^+(\pi), \delta_y^+(\pi))$ for all permutations $\pi \in S(n)$, $4 \leq n \leq 11$. More generally, towards a better understanding of the important aspects of differences in large permutations, the study of the distribution of the points $(\delta_x^+(\pi), \delta_y^+(\pi))$ for $\pi \in S(n)$ is likely to be a promising direction (see Fig. 6 and Fig. 5).

Table 2 Permutations of $S(4)$ with given difference sequences.

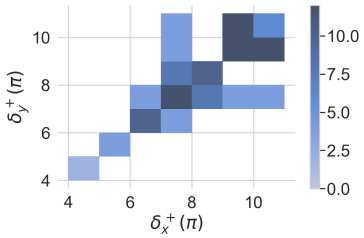
D_x	D_y	$\pi \in S(4)$ with $\Delta_x(\pi) = D_x$ and $\Delta_y(\pi) = D_y$
$(1,1,1)$	$(1,1,1)$	1234, 4321
$(1,1,3)$	$(1,1,3)$	3214
$(1,1,3)$	$(3,1,1)$	2341
$(1,2,1)$	$(1,2,1)$	1243, 2134, 3421, 4312
$(1,2,3)$	$(2,1,2)$	2314, 3241
$(1,3,1)$	$(1,3,1)$	2143, 3412
$(2,1,2)$	$(1,2,3)$	3124, 4213
$(2,1,2)$	$(2,1,2)$	1324, 4231
$(2,1,2)$	$(3,2,1)$	1342, 2431
$(2,3,2)$	$(2,3,2)$	2413, 3142
$(3,1,1)$	$(1,1,3)$	4123
$(3,1,1)$	$(3,1,1)$	1432
$(3,2,1)$	$(2,1,2)$	1423, 4132

Table 3 Permutations of $S(4)$ with given disorders.

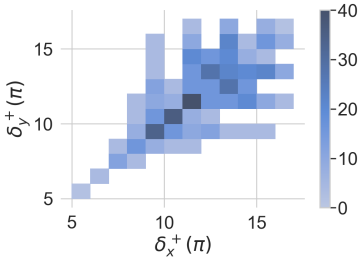
d_x	d_y	$\pi \in S(4)$ with $\delta_x^+(\pi) = d_x$ and $\delta_x^+(\pi) = d_y$
3	3	1234, 4321
4	4	1243, 2134, 3421, 4312
5	5	1324, 1432, 2143, 2341, 3214, 3412, 4123, 4231
5	6	1342, 2431, 3124, 4213
6	5	1423, 2314, 3241, 4132
7	7	2413, 3142



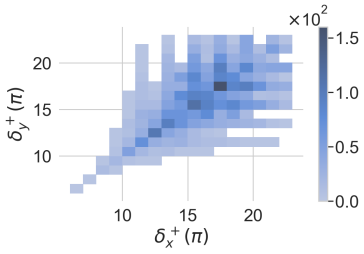
(a) $\pi \in S(4)$



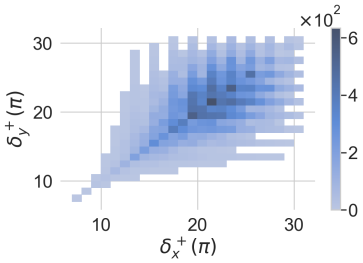
(b) $\pi \in S(5)$



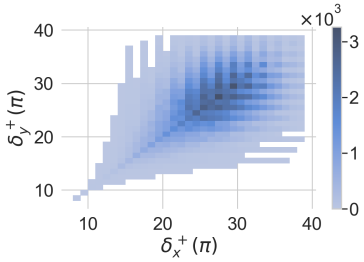
(c) $\pi \in S(6)$



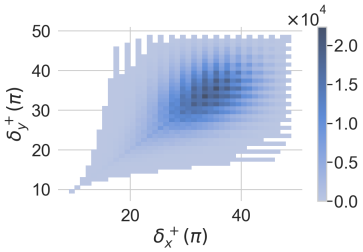
(d) $\pi \in S(7)$



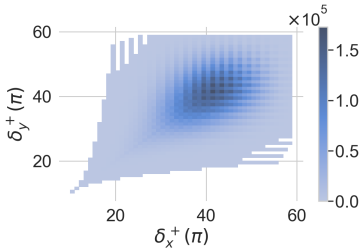
(e) $\pi \in S(8)$



(f) $\pi \in S(9)$

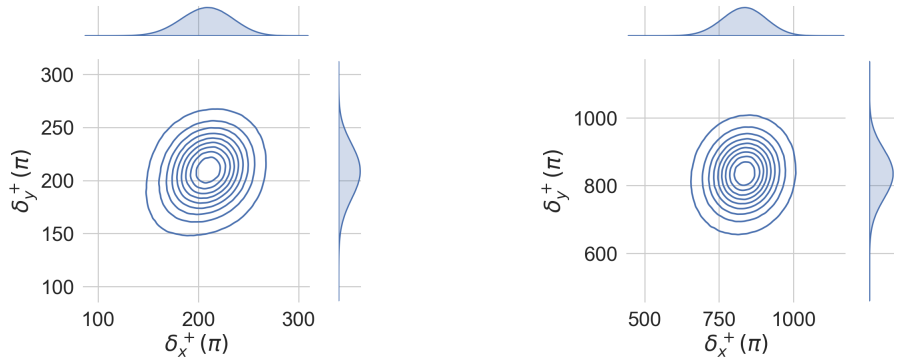


(g) $\pi \in S(10)$

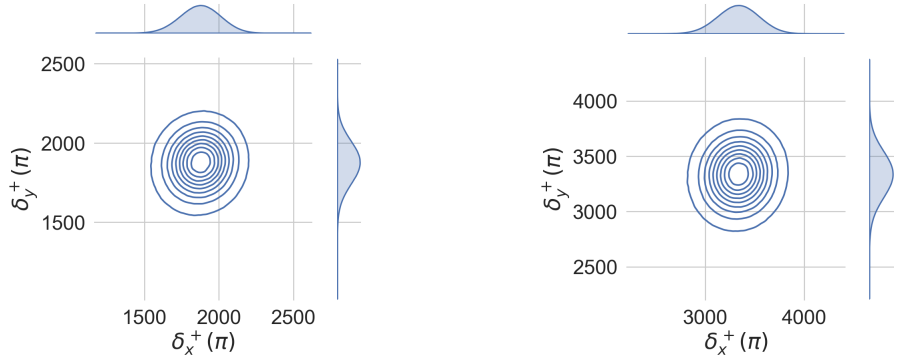


(h) $\pi \in S(11)$

Figure 5 Bivariate histograms of pairs $(\delta_x^+(\pi), \delta_y^+(\pi))$ for all permutations $\pi \in S(n)$, $4 \leq n \leq 11$.



(a) $\pi \in S(25)$ **(b)** $\pi \in S(50)$



(c) $\pi \in S(75)$ **(d)** $\pi \in S(100)$

Figure 6 Kernel Density Estimate (KDE) of pairs $(\delta_x^+(\pi), \delta_y^+(\pi))$ for 10^7 random permutations $\pi \in S(n)$, $n \in \{25, 50, 75, 100\}$.

References

1 M. Bóna. *Combinatorics of Permutations, Second Edition.* Discrete mathematics and its applications. CRC Press, 2012.

2 P. Brändén and A. Claesson. Mesh patterns and the expansion of permutation statistics as sums of permutation patterns. *Electron. J. Combin.*, 18(2):Paper 5, 14, 2011.

3 M. De Biasi. Permutation Reconstruction from Differences. *Electron. J. Combin.*, 1(4):4.3, 2014.

4 D. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching.* Addison Wesley Longman, 1998.

5 P. A. MacMahon. The Indices of Permutations and the Derivation Therefrom of Functions of a Single Variable Associated with the Permutations of any Assemblage of Objects. *Amer. J. Math.*, 35(3):281–322, 1913.

6 R. P. Stanley. *Enumerative Combinatorics*, volume 1. Cambridge University Press, 2nd edition edition, 2011.

7 W. Yu, H. Hoogeveen, and J. K. Lenstra. Minimizing Makespan in a Two-Machine Flow Shop with Delays and Unit-Time Operations is NP-Hard. *J. Sched.*, 7(5):333–348, 2004.

Computing Covers of 2D-Strings

Panagiotis Charalampopoulos
The Interdisciplinary Center Herzliya, Israel

Jakub Radoszewski
University of Warsaw, Poland

Wojciech Rytter
University of Warsaw, Poland

Tomasz Waleń
University of Warsaw, Poland

Wiktor Zuba
University of Warsaw, Poland

Abstract

We consider two notions of covers of a two-dimensional string T. A (rectangular) subarray P of T is a 2D-cover of T if each position of T is in an occurrence of P in T. A one-dimensional string S is a 1D-cover of T if its vertical and horizontal occurrences in T cover all positions of T. We show how to compute the smallest-area 2D-cover of an $m \times n$ array T in the optimal $\mathcal{O}(N)$ time, where $N = mn$, all aperiodic 2D-covers of T in $\mathcal{O}(N \log N)$ time, and all 2D-covers of T in $N^{4/3} \cdot \log^{\mathcal{O}(1)} N$ time. Further, we show how to compute all 1D-covers in the optimal $\mathcal{O}(N)$ time. Along the way, we show that the Klee's measure of a set of rectangles, each of width and height at least $\sqrt{n}$, on an $n \times n$ grid can be maintained in $\sqrt{n} \cdot \log^{\mathcal{O}(1)} n$ time per insertion or deletion of a rectangle, a result which could be of independent interest.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases 2D-string, cover, dynamic Klee's measure problem

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.12

Funding *Panagiotis Charalampopoulos*: Supported by the Israel Science Foundation grant 592/17.
Jakub Radoszewski: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.
Tomasz Waleń: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.
Wiktor Zuba: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.

1 Introduction

We say that a string C is a *cover* of a string S if each position of S is inside an occurrence of C in S. In other words, S can be obtained from several copies of C by concatenations with possible overlaps. As an example, string *abaababaabaaba* has proper covers *aba* and *abaaba*. The shortest cover and all covers of a string can be computed in linear time; see [2, 7] and [16, 17], respectively.

We consider two notions of covers of 2D-strings, 1D-covers and 2D-covers. We say that a 2D-string C is a *2D-cover* of a 2D-string T if each position of T is inside an occurrence of C in T. For an example, see Figure 1. Let T be an $m \times n$ 2D-string with $N = m \cdot n$. An $\mathcal{O}(N^2)$-time algorithm for computing all 2D-covers of T and an $\mathcal{O}(N)$ *average*-time algorithm for computing the smallest-area 2D-cover of T were shown in [19]. They also present applications of the 2D-covers problem. The problem was also mentioned recently in the context of string recovery in [1].

An $\mathcal{O}(N)$-time algorithm for computing the 2D-covers of T that are of a square shape $n \times n$, in the case that T is a square matrix itself, was shown in [11]. Let us note that there is a big difference between square-shaped 2D-covers and rectangular 2D-covers: we have only $O(n)$ square-shaped covers of an $n \times n$ 2D-string, while there may be $\Omega(n^2)$ distinct rectangular 2D-covers. This makes the rectangular case much harder.

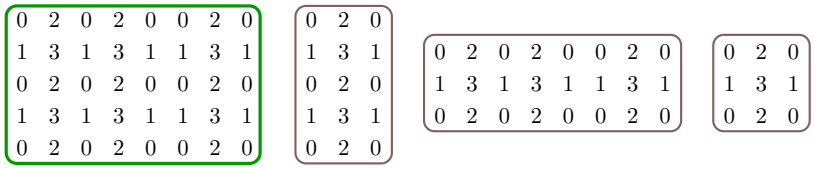


Figure 1 A 2D-string and its proper 2D-covers (the first one is vertically periodic, the two others are aperiodic).

We further say that a (1D) string C is a *1D-cover* of a 2D-string T if each position of T is inside an occurrence of C in a row, read from left to right, or in a column of T, read top-down. For an example, see Figure 2.

Let us define the two types of covers of 2D-strings more formally. A subarray $T[i\,..\,j, i'\,..\,j']$ is called a *2D-substring* of T. For an $m' \times n'$ 2D-string S, we denote:

$$Occ(S,T) = \{(i,j) \,:\, T[i\,..\,i+m'-1, j\,..\,j+n'-1] = S\}, \text{ and}$$
$$Cov(S,T) = \bigcup\{[i\,..\,i+m'-1] \times [j\,..\,j+n'-1] \,:\, (i,j) \in Occ(S,T)\}.$$

▶ **Definition 1** (2D-cover). *A 2D-string S is a 2D-cover of an $m \times n$ 2D-string T if $Cov(S,T) = [1\,..\,m] \times [1\,..\,n]$.*

A 1D-string (or simply a string) can be considered as a 2D-string of height 1. We denote by $\mathsf{trans}(S)$ the transpose of a 2D-string S. If S is a 1D-string, then $\mathsf{trans}(S)$ is a single column. We say that a position (i,j) of a 2D-string T is *horizontally covered* by a 1D-string S if $(i,j) \in Cov(S,T)$, and it is *vertically covered* by S if $(i,j) \in Cov(\mathsf{trans}(S),T)$.

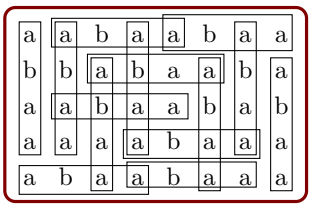


Figure 2 The string *abaa* is a 1D-cover of this 2D-string, i.e., its occurrences as a horizontal strip, read left-to-right, and as a vertical strip, read top-down, cover the 2D-string. Note that *aaba* is not a 1D-cover here.

▶ **Definition 2** (1D-cover). *A 1D-string S is a 1D-cover of a 2D-string T if each position of T is covered horizontally or vertically by S.*

When restricted to 2D-strings T of height 1, i.e. 1D-strings, both definitions yield the standard definition of a cover of a string.

Our Results. For an $m \times n$ 2D-string T of size $N = mn$ over an integer alphabet, we show the following:

- The smallest-area 2D-cover of T can be computed in $\mathcal{O}(N)$ time.
- All aperiodic 2D-covers of T can be computed in $\mathcal{O}(N \log N)$ time.
- All 2D-covers of T can be computed in $\tilde{\mathcal{O}}(N^{4/3})$ time, or in $\tilde{\mathcal{O}}(N\sqrt{\max(m,n)})$ time.[1]
- All 1D-covers of T can be computed in $\mathcal{O}(N)$ time.

The results concerning 2D- and 1D-covers can be found in Sections 3 and 5 and in Section 6, respectively. First, in Section 2 we show several connections between covers and periodicity in 2D. In the intermediate Section 4 we show a solution to an auxiliary geometric problem that employs a solution to dynamic Klee's measure for *fat* rectangles in a grid of size N and is the cornerstone of our algorithm for computing all 2D-covers.

2 Preliminaries

Let T be an $m \times n$ 2D-string of height m and width n, $T = T[1\,..\,m, 1\,..\,n]$. We write $m = \mathsf{height}(T)$ and $n = \mathsf{width}(T)$ and say that $N = mn$ is the *size* of T, which we denote as $\mathsf{size}(T) = N$. We assume that the 2D-string T for which covers are to be computed is over an integer alphabet $[1\,..\,N^{\mathcal{O}(1)}]$.

For an $m \times n$ 2D-string T, by $\mathsf{hstr}(T)$ we denote a length-n 1D-string over alphabet $[1\,..\,n]$ such that $\mathsf{hstr}(T)[i] = \mathsf{hstr}(T)[j]$ if and only if the i-th and j-th columns of T are equal. Similarly we define $\mathsf{vstr}(T)$, a length-m vertical string over alphabet $[1\,..\,m]$, by taking rows instead of columns. Let us denote by $T^{(h)}$, $T^{<h>}$, $T_{(w)}$ the 2D-substrings consisting of the first h rows, the last h rows, and the first w columns of T, respectively.

▶ **Lemma 3.** *For an $m \times n$ 2D-string T, strings $\mathsf{hstr}(T^{(h)})$, $\mathsf{hstr}(T^{<h>})$, $\mathsf{vstr}(T_{(w)})$ for all h, w can be computed in $\mathcal{O}(N)$ time.*

Proof. We consider computing all strings $\mathsf{hstr}(T^{(h)})$ for $h = 1, \ldots, m$; the other cases are symmetric. First of all, we renumber consistently the characters in each row of T separately so that they are in $[1\,..\,n]$. This can be done in $\mathcal{O}(N)$ time using one global radix sort. Thus we have already computed $\mathsf{hstr}(T^{(1)})$. Assume now we have computed $\mathsf{hstr}(T^{(h-1)})$. Then we compose a 2D-string T' of height 2, the first row is $\mathsf{hstr}(T^{(h-1)})$, the second one is the h-th row of T. It is enough now to encode consistently the columns of T', which can be done in $\mathcal{O}(n)$ time using radix sort as each column of T' is a pair of integers in $[1\,..\,n]$. ◀

Let us introduce a function $\mathrm{Is2DCover}(X, Y)$ which tests if a 2D-string X is a 2D-cover of a 2D-string Y. In [19], a 2D pattern matching algorithm [3, 6] and 2D dynamic programming were used to show the following result.

▶ **Lemma 4** ([19]). $\mathrm{Is2DCover}(X, Y)$ *can be computed in $\mathcal{O}(\mathsf{size}(Y))$ time.*

A string B is a *border* of a string S if and only if B is both a prefix and a suffix of S. It is readily verified that each cover of S is a border of S. A similar relation holds for covers of 2D-strings. A *2D-border* of a 2D-string T is a 2D-string U that occurs in each of the four corners of T. A *1D-border* of a 2D-string is a 1D-string which is a prefix of the first row or the first column of T and also a suffix of the last row or the last column of T; see Figure 3.

▶ **Observation 5.** *A 2D-/1D-cover of a 2D-string T is also a 2D-/1D-border of T.*

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses polylogarithmic factors.

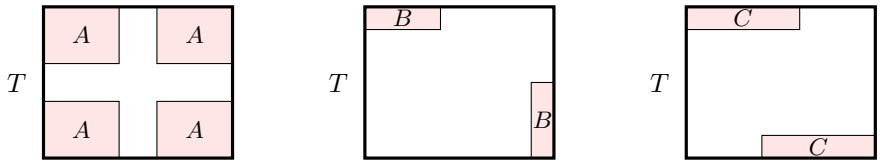


Figure 3 A is a 2D-border of T and B, C are 1D-borders of T (the cases when the top-left corner is covered horizontally; there are two other cases).

All 1D-borders of a 2D-string can be computed in linear time using the Knuth-Morris-Pratt (KMP) algorithm [14] as borders of strings of the form $X\#Y$, for a sentinel letter $\#$ that is not in the alphabet, where X and Y are the first row or column and the last row or column of T, respectively. Moreover, all 2D-borders of a 2D-string can be computed in linear time, as shown in the following lemma (which works for any alphabet with constant-time letter comparisons).

▶ **Lemma 6** ([11, Theorem 3.2]). *All 2D-borders of a 2D-string T of size N can be computed in $\mathcal{O}(N)$ time.*

We say that a string S of length $|S|$ has period p if $S[i] = S[i+p]$ for all $i = 1, \ldots, |S|-p$. By $\mathsf{per}(S)$ we denote the smallest period of S. String S is called *periodic* if $2 \cdot \mathsf{per}(S) \leq |S|$ and *aperiodic* otherwise. Let us state the periodicity lemma, one of the most classical results in combinatorics on strings.

▶ **Lemma 7** (Periodicity Lemma (weak version) [13]). *If a string S has periods p and q such that $p+q \leq |S|$, then $\gcd(p,q)$ is also a period of S.*

We say that p is a vertical (resp. horizontal) period of a 2D-string T if it is a period of each column (resp. row) of T. We denote the smallest vertical and horizontal periods of T by $\mathsf{vper}(T)$ and $\mathsf{hper}(T)$, respectively. A 2D-string T is called *periodic* if $2 \cdot \mathsf{vper}(T) \leq \mathsf{height}(T)$ or $2 \cdot \mathsf{hper}(T) \leq \mathsf{width}(T)$, and *aperiodic* otherwise.

The shortest cover of a string is aperiodic [2]. A similar observation can be made in 2D.

▶ **Observation 8.** *The shortest 1D-cover and the smallest-area 2D-cover of a 2D-string are aperiodic.*

▶ **Lemma 9.** *For two 2D-borders of T of widths w and w' with $w < w' \leq \frac{3}{2}w$, the one with the smaller height is horizontally periodic. Symmetrically, for two 2D-borders of heights h and h' with $h < h' \leq \frac{3}{2}h$, the one with smaller width is vertically periodic.*

Proof. Both 2D-borders occur in the top-left and the top-right corners. This means that, for h equal to the minimum of the heights of the two 2D-borders, $\mathsf{hstr}(T^{(h)})[1\,..\,w]$ appears both as prefix and as suffix of $\mathsf{hstr}(T^{(h)})[1\,..\,w']$.

Hence, $p = w'-w \leq \frac{w}{2}$ is a horizontal period of $T[1\,..\,h, 1\,..\,w']$. Then, p is also a horizontal period of the 2D-border of height h, as this 2D-border is a 2D-substring of $T[1\,..\,h, 1\,..\,w']$. For the symmetric statement, it suffices to transpose all involved 2D-strings. ◀

3 Computing aperiodic 2D-Covers

We first consider computation of the smallest 2D-cover, which is also (automatically) aperiodic and of all aperiodic 2D-covers. Later, in Section 5, we consider the more involved computation of all 2D-covers.

3.1 Computing Smallest 2D-Cover in Linear Time

A 2D-substring of height h and width w is called a *candidate* if it is the smallest-area 2D-cover of height h of both $T^{(h)}$ and $T^{<h>}$, and the smallest-area 2D-cover of width w of $T_{(w)}$; see Figure 4. Note that there is at most one candidate for each height h: the smallest-area 2D-cover of $T^{(h)}$ of height h. Further, note that each candidate is a 2D-border of T.

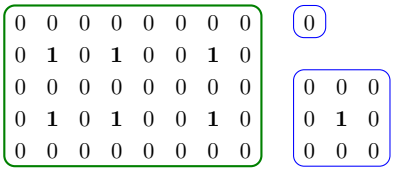


Figure 4 This 2D-string has two candidates, shown on the right. The 3×3 candidate is the smallest-area 2D-cover.

Correctness of the algorithm is based on the following lemma.

▶ **Lemma 10.** *If X, Y are candidates and $\mathsf{height}(X) < \mathsf{height}(Y)$, then*
(a) *$\mathsf{width}(X) \leq \mathsf{width}(Y)$;* **(b)** *$\mathsf{height}(Y) > \frac{3}{2}\mathsf{height}(X)$;*
(c) *if X is a 2D-cover of T, then it is also a 2D-cover of Y.*

Proof. If we had $\mathsf{width}(Y) < \mathsf{width}(X)$, then $T^{(\mathsf{height}(X))}$ would have a 2D-cover of height $\mathsf{height}(X)$ and width $\mathsf{width}(Y)$ (composed of the first $\mathsf{height}(X)$ rows of Y), so X would not be a candidate for its height. This proves part (a).

Now, recall that X and Y are 2D-borders of T. If we had $\mathsf{height}(Y) \leq \frac{3}{2}\mathsf{height}(X)$, then point (a) and Lemma 9 would imply that X is vertically periodic. Hence, X would not be the smallest-area 2D-cover of $T_{(\mathsf{width}(X))}$ of width $\mathsf{width}(X)$ (Observation 8).

Part (c) follows from the fact that Y is a 2D-border of T, similarly to the fact that a cover C of a 1D-string S is a cover of every border B of S that satisfies $|C| \leq |B|$; see [2]. ◀

▶ Remark 11. Lemma 10 (a) implies that there is exactly one smallest-area 2D-cover of T. In particular, [19] also considered the problem of computing an $h \times w$ 2D-cover that is minimal in terms of $h + w$ or $\max(h, w)$, and our algorithm below solves these variants as well.

Algorithm 1 Smallest 2D-cover.

```
CAND := the set of candidates;
X := the element of CAND of smallest height;
remove X from CAND;
foreach Y in CAND, in increasing order of heights do
    if not Is2DCover(X, Y) then X := Y;
return X;
```

▶ **Theorem 12.** *The smallest-area 2D-cover of a 2D-string T of size N can be computed in $\mathcal{O}(N)$ time.*

Proof. We linearize in $\mathcal{O}(N)$ time all 2D-strings $T^{(h)}$, $T^{<h>}$, $T_{(w)}$ using Lemma 3, and compute their shortest covers in the sense of 1D-strings. Then $T[1\,..\,h, 1\,..\,w]$ is a candidate if and only if the shortest covers of $\mathsf{hstr}(T^{(h)})$, $\mathsf{hstr}(T^{<h>})$ are of length w and the shortest cover of $\mathsf{vstr}(T_{(w)})$ is of length h.

We proceed as in Algorithm 1. The most expensive part of the algorithm is the operation Is2DCover(X, Y). It costs $\mathcal{O}(\mathsf{size}(Y))$ time. The sum of these costs is linear since the sizes of Y are geometrically growing (in each step by a factor at least 3/2) due to points (a) and (b) of Lemma 10. The largest among them has size $\mathcal{O}(N)$.

The correctness of the algorithm follows from point (c) of Lemma 10. More precisely, an invariant holds that among all the candidates that were considered in the foreach-loop up to a given point, only X can be the smallest-area 2D-cover of T. Indeed, if Is2DCover(X, Y) is true, then Y is not the smallest-area 2D-cover because it has a 2D-cover itself, and otherwise X cannot be a 2D-cover of T by point (c) of Lemma 10. ◀

3.2 Computing All Aperiodic 2D-Covers

We start with a tight asymptotic bound on the number of aperiodic 2D-covers.

▶ **Lemma 13.** *A 2D-string T of size N has $\mathcal{O}(\log^2 N)$ distinct aperiodic 2D-covers. Moreover, there is an infinite family of binary 2D-strings with $\Omega(\log^2 N)$ distinct aperiodic 2D-covers.*

Proof. Lemma 9 implies that there are $\mathcal{O}(\log^2 N)$ distinct aperiodic 2D-borders of T: at most one of height in $[(\frac{3}{2})^i \,..\, (\frac{3}{2})^{i+1}]$ and width in $[(\frac{3}{2})^j \,..\, (\frac{3}{2})^{j+1}]$ for each pair of non-negative integers i, j. The same bound applies to 2D-covers due to Observation 5.

For $i = 1, 2$, let $\mathcal{C}_i$ be the set of lengths of covers of a string S_i of length n_i over alphabet Σ_i. Then, an $n_1 \times n_2$ 2D-string T over alphabet $\Sigma_1 \times \Sigma_2$ defined as $T[i, j] = (S_1[i], S_2[j])$ has 2D-covers of all dimensions in $\mathcal{C}_1 \times \mathcal{C}_2$.

A binary Fibonacci string of length n has $\Theta(\log n)$ aperiodic covers [10]. Hence, with the above construction we can obtain a 2D-string of size $N = n^2$ over the alphabet with 4 letters with $\Omega(\log^2 N)$ aperiodic 2D-covers. We can encode each letter by its 2-digit binary representation, obtaining a binary 2D-string with $\Omega(\log^2 N)$ aperiodic 2D-covers. ◀

A direct application of the bound from Lemma 13 and the Is2DCover routine would yield an $\mathcal{O}(N \log^2 N)$-time algorithm. The theorem below, whose proof can be found in Appendix B, shaves a $\log N$ factor from this complexity. Let us note that it uses the same order of inspecting candidates as the algorithm Simple-All-2D-covers below.

▶ **Theorem 14.** *All aperiodic 2D-covers of a 2D-string of size N can be computed in $\mathcal{O}(N \log N)$ time.*

4 Rectangle Cover Problem

In the Klee's measure problem in 2D, we are given M rectangles in the plane and are to output the area of the union of these rectangles. This problem can be solved in $\mathcal{O}(M \log M)$ time in the offline setting [4, 8]. In the dynamic variant of the problem, where rectangles can be inserted and deleted, Klee's measure can be maintained in $\tilde{\mathcal{O}}(\sqrt{M})$ time per update [18]. Below, we consider a special version of the Klee's measure problem with *fat* rectangles in a grid of size bounded by N and use it to solve an auxiliary problem called Rectangle Cover. Details omitted due to space constraints can be found in Appendix A.

For a set $\mathcal{F}$ of rectangles let us denote by $\mathcal{F}_{h,w}$ the subset of rectangles with height at least h and width at least w.

Rectangle Cover
Input: A set $\mathcal{F}$ consisting of M rectangles in $[0\,..\,m] \times [0\,..\,n]$, where $N = m \cdot n$, $n \geq m$.
Output: All pairs (h, w) (called *good* pairs) for which $\lvert \bigcup \mathcal{F}_{h,w} \rvert = N$.

We will reduce Rectangle Cover to a restricted variant of the following problem.

Dynamic Klee's Measure
Input: An initially empty set of rectangles $\mathcal{R}$ in $[0\,..\,m] \times [0\,..\,n]$; $N = mn$, $n \geq m$.
Updates: Insert or delete a rectangle to $\mathcal{R}$ and return the area of $\bigcup \mathcal{R}$.

We start by a direct reduction that we then refine in Lemma 16.

▶ **Lemma 15.** *The Rectangle Cover problem reduces in $\mathcal{O}(M+N)$ time to the Dynamic Klee's Measure problem with a grid of the same dimensions and $\mathcal{O}(M)$ updates.*

Proof. Note that if (h, w) is a good pair, then $(h-1, w)$ and $(h, w-1)$ are good pairs as well. Hence, we will compute for each h, the maximum w for which (h, w) is good, as shown in Algorithm 2. We use the following equality and a symmetric one.

$$\mathcal{F}_{h,w} = \mathcal{F}_{h,w+1} \cup \{X \in \mathcal{F} : \mathsf{width}(X) = w, \mathsf{height}(X) \geq h\}.$$

Algorithm 2 Rectangle Cover via Dynamic Klee's Measure.

```
w := n + 1; R := ∅;
for h := 1 to m do
    while w > 0 and |⋃ R| ≠ N do
        // Invariant: R = F_{h,w}
        w := w − 1;
        R := R ∪ {X ∈ F : width(X) = w, height(X) ≥ h};
    Report (h, 1), ..., (h, w);
    R := R \ {X ∈ F : height(X) = h};
```

Let us note that each rectangle from $\mathcal{F}$ is inserted at most once to $\mathcal{R}$ and deleted at most once from $\mathcal{R}$. The condition in the while-loop can be checked by computing $\mathcal{R}$'s Klee's measure. Rectangles that are to be inserted to $\mathcal{R}$ and deleted from $\mathcal{R}$ in subsequent steps can be easily determined if all rectangles in $\mathcal{F}$ are pre-sorted by their widths and heights, via bucket sort, in $\mathcal{O}(M+N)$ time. ◀

A rectangle will be called a **fat rectangle** if both its width *and* height are at least $\sqrt{n}$.

▶ **Lemma 16.** *The Rectangle Cover problem reduces in $\tilde{\mathcal{O}}(M \min\{m, \sqrt{n}\} + N)$ time to the Dynamic Klee's Measure problem on a grid of the same dimensions and $\mathcal{O}(M)$ insertions and deletions of fat rectangles.*

Proof. For every $h \leq \min\{m, \sqrt{n}\}$, we compute the maximum w such that (h, w) is a good pair using binary search. In order to test a candidate (h, w), we compute the area of the union of rectangles from $\mathcal{F}_{w,h}$ (i.e., solve static Klee's measure problem in 2D) using one of the classic $\mathcal{O}(M \log M)$-time approaches [4, 8]. If $m > \sqrt{n}$, the same approach is then used for every $w \leq \sqrt{n}$. Finally, for $h, w > \sqrt{n}$ we use Algorithm 2 from the proof of Lemma 15. More precisely, we start the for-loop with $h = \lceil \sqrt{n} \rceil$ and break when w drops below $\lceil \sqrt{n} \rceil$. Thus, the set $\mathcal{R}$ contains only fat rectangles throughout the execution of the algorithm. ◀

A proof of the following lemma can be found in Appendix A.

▶ **Lemma 17.** *The Dynamic Klee's Measure problem with fat rectangles can be solved in $\tilde{\mathcal{O}}(\sqrt{n})$ time per operation, after $\tilde{\mathcal{O}}(N)$-time preprocessing.*

▶ **Lemma 18.** *The* Rectangle Cover *problem can be solved in* $\tilde{\mathcal{O}}(M \cdot \min\{m, \sqrt{n}\} + N)$ *time.*

Proof. We use Lemma 16 to reduce the problem to Dynamic Klee's Measure problem on fat rectangles with $\mathcal{O}(M)$ updates in $\tilde{\mathcal{O}}(M \cdot \min\{m, \sqrt{n}\} + N)$ time, and Lemma 17 to solve the latter problem with $\tilde{\mathcal{O}}(\sqrt{n})$ update time. Note that the Dynamic Klee's Measure problem is void if $m \leq \sqrt{n}$ (no fat rectangle exists). ◀

▶ Remark 19. To solve the Rectangle Cover problem it is not necessary to compute the exact area of union of the fat rectangles but just to check if they cover the whole grid. This would slightly simplify the solution, as shown in Appendix A.1, but the main idea would stay intact. We decided to state the auxiliary problem as a variant of dynamic Klee's measure since it could find other applications.

5 Computing All 2D-Covers

Let us start with simple but less efficient algorithms. We say that (h, w) is a weak candidate if $T[1\,..\,h, 1\,..\,w]$ is a 2D-cover of $T^{(h)}$ and of $T_{(w)}$. Let us denote by L_h the sorted list of widths of weak candidates of height h.

For a list L of integers we denote by $cut(L, y)$ the list of elements $x \in L$ such that $x \leq y$. We next present two preliminary algorithms for computing all 2D-covers of T.

Algorithm 3 Simple-All-2D-covers.

Result := $\emptyset$;
for $h := 1$ **to** m **do**
  foreach $z \in L_h$, *in decreasing order* **do**
    if Is2DCover($T[1\,..\,h, 1\,..\,z]$, T) **then**
      Result := *Result* $\cup \{(h, w) : w \in cut(L_h, z)\}$; **break**;
    else remove z from all lists L_t;
return *Result*;

Algorithm 4 Binary-Search-All-2D-covers.

Result := $\emptyset$;
for $h := 1$ **to** m **do**
  // Binary search with $\mathcal{O}(\log n)$ instances of Is2DCover
  $z := \max(\{w \in L_h : \text{Is2DCover}(T[1\,..\,h, 1\,..\,w], T)\} \cup \{0\})$;
  if $z > 0$ **then** *Result* := *Result* $\cup \{(h, w) : w \in cut(L_h, z)\}$;
return *Result*;

▶ **Proposition 20.** *The algorithm Simple-All-2D-covers outputs all 2D-covers of* T *in* $\mathcal{O}(N^{3/2})$ *time if* T *is a square matrix, and* $\mathcal{O}(N \max(m, n))$ *time in general. The algorithm Binary-Search-All-2D-covers computes all 2D-covers in* $\mathcal{O}(N^{3/2} \log N)$ *time.*

Proof. In the algorithm Simple-All-2D-covers in each column we make at most one negative test of Is2DCover and in each row at most one positive test. Hence, the time complexity is $\mathcal{O}(N \max(m, n))$.

We run the algorithm Binary-Search-All-2D-covers for T if $m \leq n$ and otherwise for T transposed. For each h we execute $\mathcal{O}(\log n)$ instances of Is2DCover, which require time $\mathcal{O}(N \log n)$ in total for a given h. Summing over all heights, this yields $\mathcal{O}(Nm \log n) = \mathcal{O}(N^{3/2} \log N)$ time. ◀

We proceed with faster computation of all 2D-covers. Henceforth let us assume w.l.o.g. that $n \geq m$. Let T be an $m \times n$ 2D-string of size $N = mn$. We say that a 2D-string U is an (x, y)-array if

$$\mathsf{height}(U) \in [\left(\tfrac{3}{2}\right)^x .. \left(\tfrac{3}{2}\right)^{x+1}) \text{ and } \mathsf{width}(U) \in [\left(\tfrac{3}{2}\right)^y .. \left(\tfrac{3}{2}\right)^{y+1}).$$

We call a 2D-border of T that is an (x, y)-array an (x, y)-border of T.

We say that a 2D-string U is of *periodic type* (p, q, a, b) if $\mathsf{vper}(U) = p$, $\mathsf{hper}(U) = q$, $\mathsf{height}(U) \bmod p = a$, and $\mathsf{width}(U) \bmod q = b$.

▶ **Lemma 21.** *For given x, y, all (x, y)-borders of T are of the same periodic type.*

Proof. Assume there are two (x, y)-borders U and V of T. Let us show that $\mathsf{vper}(U) = \mathsf{vper}(V)$. If $\mathsf{height}(U) = \mathsf{height}(V)$, then each column of U occurs as a column of V and vice versa. Hence, $\mathsf{vper}(U) = \mathsf{vper}(V)$. If $\mathsf{height}(U) \neq \mathsf{height}(V)$, then $p := |\mathsf{height}(U) - \mathsf{height}(V)| \in [1 .. \frac{1}{2} \cdot (\frac{3}{2})^x)$ is a vertical period of both U and V, so both are vertically periodic with period p (c.f. Lemma 9). Let $p' = \mathsf{vper}(U)$ and $p'' = \mathsf{vper}(V)$. Then $p' \leq p$ and both p and p' are periods of each column of U. Hence, by the periodicity lemma (Lemma 7), $p' \mid p$. Similarly for p''. Hence, p' (p'') is the least common multiple of smallest periods of the set of length-p prefixes of columns of U (resp. V). The sets of length-p prefixes of columns of U and V are the same. We thus have $\mathsf{vper}(U) = p' = p'' = \mathsf{vper}(V)$. Moreover, p' divides $|\mathsf{height}(U) - \mathsf{height}(V)|$, which shows that $\mathsf{height}(U) \bmod p = \mathsf{height}(V) \bmod p$. The proof for the horizontal period is symmetric. ◀

Let us recall that all 2D-borders of T can be computed in $\mathcal{O}(N)$ time (Lemma 6). For each non-empty group of (x, y)-borders, we can compute the periodic type of one of its representatives U in $\mathcal{O}((\frac{3}{2})^{x+y})$ time using the KMP algorithm for each column and row. This gives $\mathcal{O}(N)$ time in total. Below we show how to compute all 2D-covers of T of a given periodic type (p, q, a, b) in $\tilde{\mathcal{O}}(N \cdot \min\{m, \sqrt{n}\})$ time.

Let us henceforth fix a periodic type (p, q, a, b) and denote $X = T[1 .. p + a, 1 .. q + b]$. We call X the *root* of all 2D-covers of this periodic type. We denote by V_X the set of top-left corners of occurrences of X in T. It can be computed using 2D pattern matching for X.

A grid of points $\{(u + ip, v + jq) : 0 \leq i < \alpha, 0 \leq j < \beta\}$ for any u, v will be called an (α, β)-*r-grid*, or simply an *r-grid*. If a 2D-string U of height h and width w has a periodic type (p, q, a, b), then its occurrence in T *implies* a $(\lfloor h/p \rfloor, \lfloor w/q \rfloor)$-grid subset of V_X. See Figure 5. Moreover, such an r-grid in V_X always *generates* an occurrence of U in T. An (α, β)-r-grid D in V_X is called *maximal* if there is no (α', β')-r-grid D' in V_X such that D is a proper subset of D'. The next lemma follows from [9, Section 5].

▶ **Lemma 22.** *The set of maximal r-grids in V_X can be computed in $\mathcal{O}(N)$ time.*

Every r-grid in V_X can be extended to a maximal r-grid in V_X. This suggests the following important observation.

▶ **Observation 23.** *A 2D-string of height h and width w and of periodic type (p, q, a, b) is a 2D-cover of T if and only if T is covered by substrings of T that are generated by maximal (α, β)-r-grids satisfying $\alpha \geq \lfloor h/p \rfloor, \beta \geq \lfloor w/q \rfloor$.*

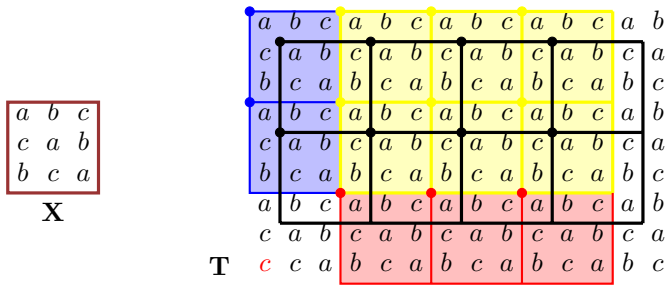


Figure 5 Let $(p, q, a, b) = (3, 3, 0, 0)$. Some (not all) occurrences of X in T are shown, with the corresponding elements of V_X drawn as dots. The $(2, 4)$-r-grid in the black frame is maximal, whereas the yellow $(2, 3)$-r-grid is not; it is a part of two maximal r-grids: blue+yellow and red+yellow.

From now on we will treat 2D-substrings generated by maximal r-grids in T as rectangles. The following lemma follows directly from the above discussion.

▶ **Lemma 24.** *The problem of computing all 2D-covers of T of a given periodic type reduces in $\mathcal{O}(N)$ time to an instance of* Rectangle Cover *with $M = \mathcal{O}(N)$ and the same N.*

▶ **Theorem 25.** *All 2D-covers of a 2D-string of size $N = m \times n$, $n \geq m$, can be computed in $\tilde{\mathcal{O}}(N \cdot \min\{m, \sqrt{n}\})$ time or $\tilde{\mathcal{O}}(N^{4/3})$ time.*

Proof. For each of the $\tilde{\mathcal{O}}(1)$ groups of (x, y)-borders, we apply Lemma 24 to reduce the problem in scope to Rectangle Cover in $\mathcal{O}(N)$ time, then we apply the algorithm from Lemma 18. In total we obtain $\tilde{\mathcal{O}}(N \cdot \min\{m, \sqrt{n}\})$ time, and we can use the fact that $\min\{m, \sqrt{n}\} \leq (m \cdot n)^{1/3} = N^{1/3}$. ◀

6 Computing 1D-Covers

Let us recall that a 1D-cover of an $m \times n$ 2D-string T must be a 1D-border of T (see Observation 5 and Figure 3). We will only show how to compute 1D-covers of T which are prefixes of the first row and suffixes of the last row. The three other cases can be treated analogously. Henceforth we denote $W = T[1, 1\,..\,n]\#T[m, 1\,..\,n]$, where $\#$ is some sentinel letter not in the alphabet.

6.1 An $\mathcal{O}(N \log N)$-Time Algorithm Computing All 1D-Covers

We will partition the 1D-borders in scope in $\mathcal{O}(\log n)$ groups based on their periods. Then, we will show how to process each group in $\mathcal{O}(N)$ time, relying on the following property: each element of a group G can be covered by each of the elements of G that are shorter than it. Hence, it will suffice to compute the longest element of each group that covers T.

For a (1D-)string S, let us denote by $\mathbf{B}(S)$ all borders of S which are either aperiodic or are the longest ones with a given smallest period. Let us denote by $\mathbf{A}(S)$ the set of all aperiodic borders of S; $\mathbf{A}(S) \subseteq \mathbf{B}(S)$.

▶ **Fact 26** ([12]). *The size of $\mathbf{B}(S)$ is $\mathcal{O}(\log |S|)$.*

By a *big border* of a string X, we mean X itself or any border Z of X such that $|Z| \geq 2 \cdot \mathsf{per}(X)$. Given a string S of length n, due to Fact 26, we can partition its borders into $\mathcal{O}(\log n)$ groups, such that the following holds for each group G: G is the set of big

borders of its longest element $X \in \mathbf{B}(S)$. We denote such a group by $\mathsf{Group}(X)$ and call $\mathsf{per}(X)$ the *period of the group*. In particular, for a string $X \in \mathbf{B}(S)$ of the form U^kV with $k \geq 2$, $\mathsf{per}(X) = |U|$ and $0 \leq |V| < \mathsf{per}(X)$, we have $\mathsf{Group}(X) = \{U^jV : j \in [2\,..\,k]\}$.

▶ **Observation 27.** *The defined groups form a partition of the set of borders of S.*

▶ **Example 28.** For $S = (abaab)^5a$, we have $\mathbf{B}(S) = \{a, aba, abaaba, S\}$ and $\mathsf{Group}(S) = \{(abaab)^ia : i = 2, \ldots, 5\}$. The groups of the remaining elements of $\mathbf{B}(S)$ are singletons.

The following lemma will be used in Section 6.3, but we state it here as the required definitions have been already introduced. Its proof can be found in Appendix B.

▶ **Lemma 29.** *For $X, Y \in \mathbf{B}(S)$ with $|X| < |Y|$, we have $\mathsf{per}(Y) \geq \mathsf{per}(X) \cdot |\mathsf{Group}(X)|$. Moreover, for $X, Y, Z, W \in \mathbf{B}(S)$ with $|X| < |Y| < |Z| < |W|$, we have $\mathsf{per}(W) \geq \frac{9}{8}\mathsf{per}(X)$.*

Every 1D-cover of T that is a prefix of the first row of T and a suffix of the last row of T is in $\mathsf{Group}(X)$ for some $X \in \mathbf{B}(W)$. The following lemma gives us the monotonicity property that was mentioned before, which will allow us to handle each group efficiently.

▶ **Lemma 30.** *If some $Z \in \mathsf{Group}(X)$ for $X \in \mathbf{B}(W)$ is a 1D-cover of T, then all elements of $\mathsf{Group}(X)$ that are shorter than Z are also 1D-covers of T.*

Proof. If $|\mathsf{Group}(X)| > 1$, then X is periodic. Thus, the elements of $\mathsf{Group}(X)$ are of the form U^jV, where V is a (possibly empty) prefix of U and $j \geq 2$. It is then readily verified that $Z = U^kV$, and hence each of its occurrences in T, can be covered by U^jV for all $j \in [2\,..\,k)$. ◀

Let us introduce two arrays that are building blocks of the algorithm. For a family $\mathcal{F}$ of subintervals of $[1\,..\,n]$, for each $k \in [1\,..\,n]$, let us denote by $\mathcal{F}\text{-}Cov[k]$ the length of the longest interval in $\mathcal{F}$ containing k. If there is no such interval for a given k, then $\mathcal{F}\text{-}Cov[k] = 0$. The table $\mathcal{F}\text{-}Cov$ can be computed in $\mathcal{O}(|\mathcal{F}| + n)$ time by sorting the intervals using radix sort, and then performing a standard line sweeping algorithm.

For two strings X and Y, we define the table $LBB_{X,Y}$ such that $LBB_{X,Y}[i]$ is the length of the longest big border in $\mathsf{Group}(X)$ which starts at position i in Y; $LBB_{X,Y}[i] = 0$ if there is no such big border.

Algorithm 5 1D-Covers from a group.

Input: A 2D-string T and $X \in \mathbf{B}(W)$, where $W = T[1, 1\,..\,n]\#T[m, 1\,..\,n]$.
Output: The elements of $\mathsf{Group}(X)$ that cover T.

processing rows:

for $i := 1$ **to** m **do**
  $Y :=$ i-th row of T; $\mathcal{F} := \{[\,j\,..\,j + LBB_{X,Y}[j] - 1\,] \;:\; j \in [1\,..\,n]\}$;
  for $j := 1$ **to** n **do** $maxCov[i, j] := \mathcal{F}\text{-}Cov[j]$;

processing columns:

for $j := 1$ **to** n **do**
  $Y :=$ j-th column of T; $\mathcal{F} := \{[\,i\,..\,i + LBB_{X,Y}[i] - 1\,] \;:\; i \in [1\,..\,m]\}$;
  for $i := 1$ **to** m **do** $maxCov[i, j] := \max(maxCov[i, j], \mathcal{F}\text{-}Cov[i])$;

$min := \min\{maxCov[i, j] \;:\; 1 \leq i \leq m, 1 \leq j \leq n\}$;

return $\{Z \in \mathsf{Group}(X) \;:\; |Z| \leq min\}$;

▶ **Lemma 31.** *Given strings X and Y and integers $|X|$ and $\mathsf{per}(X)$, $LBB_{X,Y}$ can be computed in time $\mathcal{O}(|Y|)$.*

Proof. If X is not periodic, $\mathsf{Group}(X) = \{X\}$ and it suffices to find all occurrences of X in Y using the KMP algorithm. Otherwise, let $p = \mathsf{per}(X)$. We use the KMP algorithm to find all occurrences of $U := X[1\,..\,2p + (|X| \bmod p)]$ in Y. We compute the $LBB_{X,Y}$ array right-to-left. If there is an occurrence of U in Y starting at position i, then $LBB_{X,Y}[i] = \max(|U|, p + LBB_{X,Y}[i+p])$. Otherwise, $LBB_{X,Y}[i] = 0$. ◀

The algorithm processing a single group is specified in the pseudocode above. Each row and column of T is processed separately. The goal is to compute, for each position of T, the longest element of $\mathsf{Group}(X)$ that covers it, eventually stored in $maxCov[i,j]$. Then, by taking the minimum over all positions of T, due to Lemma 30, we can identify the elements of $\mathsf{Group}(X)$ that cover T. Each row/column Y is processed according to the following observation, in time proportional to its length due to efficient computation of $\mathcal{F}$-Cov and LBB arrays (Lemma 31).

▶ **Observation 32.** *Let X and Y be two strings and k be an integer, such that $X \in \mathbf{B}(Y)$ and $k \in [i\,..\,i + LBB_{X,Y}[i] - 1]$. Then, all elements of $\{Z \in \mathsf{Group}(X) : |Z| \leq LBB_{X,Y}[i]\}$ cover the position k in Y. If $[i\,..\,i + LBB_{X,Y}[i] - 1]$ is the maximal (over all i's) fragment in Y containing position k, then $LBB_{X,Y}[i]$ is the length of the longest element of $\mathsf{Group}(X)$ covering position k in Y.*

▶ **Lemma 33.** *Algorithm 5 computes all the 1D-borders from a single group $\mathsf{Group}(X)$ that cover T in $\mathcal{O}(N)$ time.*

We thus obtain an $\mathcal{O}(N \log N)$-time algorithm for the problem in scope. We first compute all borders of W and organize them in groups $\mathsf{Group}(X)$ for $X \in \mathbf{B}(W)$ in $\mathcal{O}(n)$ time. Then, we process each group separately, obtaining the following preliminary result.

▶ **Proposition 34.** *All 1D-covers of a 2D-string of size N can be computed in $\mathcal{O}(N \log N)$ time.*

6.2 A Linear-Time Algorithm Computing Aperiodic 1D-Covers

We will now show how to compute in linear time all aperiodic 1D-covers of T. The developed tools will also be useful in obtaining a general algorithm that computes all 1D-covers. A shortest 1D-cover is aperiodic (Observation 8), so in this section we also obtain a linear-time algorithm for computing all shortest 1D-covers.

We first prove a lemma that will allow us to efficiently process aperiodic borders of W.

▶ **Lemma 35.** *We have $\sum_{B \in \mathbf{A}(W)} |Occ(B,T)| = \mathcal{O}(N)$.*

Proof. Let $B_1, \ldots, B_k$ be all aperiodic 1D-borders of W. First, two occurrences of an aperiodic string S can overlap by up to $|S|/2 - 1$ letters. Hence, $|Occ(B_i,T)| = \mathcal{O}(N/|B_i|)$. Further, for each $i \in [1\,..\,k)$, B_i is a border of B_{i+1} and hence, since the two occurrences of B_i overlap by less than $|B_i|/2$ letters, we have $|B_{i+1}| > 3|B_i|/2$. This implies the lemma. ◀

By the next lemma, we can compute all sets $Occ(B,T)$, for $B \in \mathbf{A}(W)$, in time $\mathcal{O}(N)$. The lemma can be proved using the $PREF$ array [12], as shown in Appendix B, or the more heavyweight internal pattern matching queries [15].

▶ **Lemma 36.** *Let $S_1, \ldots, S_k$ be prefixes of W. Then we can compute all the sets $Occ(S_i,T)$ in $\mathcal{O}(N + \sum_{i=1}^{k} |Occ(S_i,T)|)$ time.*

We reduce our problem to the following auxiliary problem.

COLOURED STRIPS PROBLEM
Input: $\mathcal{O}(N)$ horizontal/vertical line segments on an $m \times n$ grid, where $N = mn$. Each line segment has a colour in $1, \ldots, \lfloor \log N \rfloor$.
Output: For each colour answer YES iff segments of this colour cover the whole grid.

▶ **Lemma 37.** *Computing all aperiodic 1D-covers of a 2D-string of size $N = mn$ can be reduced in $\mathcal{O}(N)$ time to an instance of COLOURED STRIPS PROBLEM with the same m, n.*

Proof. We construct an instance of the COLOURED STRIPS PROBLEM by choosing line segments of colour $i \in [1\,..\,k]$ to be inclusion-maximal fragments of rows/columns of T covered by B_i, where $\mathbf{A}(W) = \{B_1, \ldots, B_k\}$.

The set of horizontal line segments of colour i is $\{[x, y\,..\,y+|B_i|-1] : (x, y) \in Occ(B_i, T)\}$, and the set of vertical line segments is $\{[x\,..\,x + |B_i| - 1, y] : (x, y) \in Occ(\mathsf{trans}(B_i), T)\}$. Lemmas 35 and 36 show that the total number of line segments is indeed $\mathcal{O}(N)$ and that they can be computed in linear time, respectively. ◀

▶ **Lemma 38.** *The COLOURED STRIPS PROBLEM can be solved in $\mathcal{O}(N)$ time.*

Proof. We first use radix sort to merge horizontal line segments of the same colour as long as any two intersect. We then repeat this for vertical line segments. Now no two line segments of the same direction and colour intersect.

We will assign in $\mathcal{O}(N)$ time to each grid cell (i, j) a bitmask $hcol(i, j)$, such that its kth bit is set iff cell (i, j) is covered by a horizontal line segment of colour k. We explicitly store $hcol(i, j)$ in $\mathcal{O}(1)$ words of space. We process each row using a line sweeping algorithm, updating the maintained bitmask whenever we encounter the endpoint of a horizontal line segment. We similarly compute an analogously defined bitmask $vcol(i, j)$ for vertical line segments, for each cell (i, j).

In the end, we return all colours in the bitmask $\bigwedge_{i,j} (hcol(i, j) \vee vcol(i, j))$. ◀

The last two lemmas imply immediately the following preliminary result.

▶ **Proposition 39.** *All aperiodic 1D-covers of a 2D-string of size N can be computed in $\mathcal{O}(N)$ time.*

Next, we will present a more involved algorithm that computes *all* 1D-covers in linear time. It will rely on several additional technical concepts.

6.3 A Linear-Time Algorithm Computing All 1D-Covers

We will first show how to efficiently process all groups with large periods, using the fact that a pattern P in a text Y has $\mathcal{O}(|Y|/\mathsf{per}(P))$ occurrences. For each group with period p greater than $\log N$, we will compute all occurrences of its shortest element in each row/column in $\mathcal{O}(N/p)$ total time, after a global $\mathcal{O}(N)$-time preprocessing, using Lemma 36. Then, using this representation, we will be able to compute the elements of the group that cover T in $\mathcal{O}(N \log N/p)$ time. The periods grow geometrically by Lemma 29 and are at least $\log N$, so this yields $\mathcal{O}(N)$ time in total.

Then, we could employ Lemma 33 for each group with period at most $\log N$. There are only $\mathcal{O}(\log \log N)$ such groups, as the period of each group is at least a constant factor larger than the period of the previous group. Thus, we could process these groups in $\mathcal{O}(N \log \log N)$ time in total.

We will conclude this section by showing how to process in linear time all groups with periods not larger than $\log N$ using a variant of the Coloured Strips Problem, thus obtaining a linear-time algorithm for the problem in scope.

6.3.1 Handling Groups with Large Periods

We will rely on the fact that the total number of essential intervals in the families $\mathcal{F}$ in Algorithm 5 is of linear size.

For each group $\mathsf{Group}(X)$ with period greater than $\log N$, we first invoke Lemma 36 for the shortest element $S \in \mathsf{Group}(X)$. Then we perform the following *strip-merging routine*: In each row/column, we merge any two occurrences of S that are exactly $\mathsf{per}(S)$ positions apart as long as we can. If occurrences are treated as segments, this produces $\mathcal{O}(N/\mathsf{per}(S))$ horizontal/vertical line segments; each segment stores a weight equal to its length. We have thus reduced the problem in scope to the following problem.

Restricted 2D Manhattan Skyline Problem
Input: M horizontal/vertical line segments with positive weights in an $m \times n$ grid.
Output: A point of the grid with minimum weight; the weight of a point is equal to the maximum weight of a line segment that covers it or 0 if the point is not covered by any segment.

▶ **Lemma 40.** *The Restricted 2D Manhattan Skyline Problem can be solved in time* $\mathcal{O}(M \log M)$.

Proof. We first sort the endpoints of line segments in $\mathcal{O}(M \log M)$ time; first by their x coordinate and then by their y coordinate. We now present a top-to-bottom line sweeping algorithm. The broom stores, for each point in $[1\,..\,n]$, the maximum weight of a vertical segment that contains it provided that the weight is positive. We implement the broom as a balanced BST. When processing a row with k horizontal segments, $[1\,..\,n]$ can be split into $\mathcal{O}(k)$ pairwise-disjoint intervals with equal maximum weight of a horizontal segment covering them. Then, by asking a query to the broom for each of the $\mathcal{O}(k)$ pairwise-disjoint intervals, we are able to compute the maximum weight of a vertical segment that intersects each of the intervals; consequently, a point of this row with minimum weight. Each query to the balanced BST is answered in $\mathcal{O}(\log M)$ time, so a row with k horizontal line segments is processed in $\mathcal{O}(k \log M)$ time. The total number of updates to the broom is upper bounded by the number of endpoints of vertical line segments, and each of them is processed in $\mathcal{O}(\log M)$ time. The stated complexity follows. ◀

▶ **Lemma 41.** *All 1D-covers with period greater than* $\log N$ *of a 2D-string* T *of size* N *can be computed in time* $\mathcal{O}(N)$.

Proof. We have $\mathcal{O}(\log n)$ groups to process, each with period greater than $\log N$. We showed that we can reduce, in $\mathcal{O}(N)$ time, the problem in scope to $\mathcal{O}(\log n)$ instances of the Restricted 2D Manhattan Skyline Problem; for each group with period p an instance with $M = \mathcal{O}(N/p)$. The periods grow geometrically (Lemma 29) and are at least $\log N$, so the time needed to solve all these instances is $\mathcal{O}(N/\log N \cdot \log N) = \mathcal{O}(N)$ by Lemma 40. ◀

6.3.2 Handling Groups with Small Periods

Let us consider all groups with periods at most $\log N$. If any of the considered groups contains at least $\log n$ strings, we treat it in $\mathcal{O}(N)$ time, invoking Algorithm 5 (see Lemma 33). Note that, due to Lemma 29, we can have at most one such group.

▶ **Lemma 42.** *The remaining groups with periods at most* $\log N$ *have* $\mathcal{O}(\log N)$ *elements in total.*

Proof. The number of groups is $\mathcal{O}(\log \log N)$ since their periods are geometrically increasing by Lemma 29. Moreover, if $n_1, \ldots, n_k$ are sizes of the non-singleton groups, then by Lemma 29 we have $\prod_{i=1}^{k} n_i = \mathcal{O}(\log N)$, which implies that $\sum_{i=1}^{k} n_i = \mathcal{O}(\log N)$. ◀

We now have only $\mathcal{O}(\log N)$ strings to test. Unfortunately, we cannot use the Coloured Strips Problem directly, because the reduction of Lemma 37 could yield $\omega(N)$ line segments, as now we take several strings from each group. However, we can treat the elements of each group as a batch. Let us consider the following variant of the Coloured Strips Problem.

Coloured Strips Problem with Shades
Input: $\mathcal{O}(N)$ horizontal/vertical line segments on an $m \times n$ grid, where $N = mn$. Each line segment has a colour i and a shade in $[1\,..\,s_i]$, such that $\sum_i s_i = \mathcal{O}(\log N)$. The shades of a colour i are sorted from the lightest (1) to the darkest (s_i). No three line segments of the same colour intersect and none is contained in another line segment of the same colour.
Output: For each colour i, its darkest shade x such that line segments of colour i and shades non-darker than x cover the grid, if any.

Note that the above problem with one colour is a variant of the Restricted 2D Manhattan Skyline Problem.

For each group that we are to process, we invoke Lemma 36 for its shortest element, and then perform the strip-merging routine, outlined in Section 6.2 – the group number is the colour of the line segment and the length of a line segment is now its shade. Over all groups this takes $\mathcal{O}(N)$ time. We summarize the above discussion in the following statement.

▶ **Lemma 43.** *Computing all 1D-covers of a 2D-string of size* $N = m \times n$ *with period not larger than* $\log N$ *can be reduced in* $\mathcal{O}(N)$ *time to an instance of Coloured Strips Problem with Shades with the same* m, n.

The Coloured Strips Problem with Shades could be solved in $\mathcal{O}(N)$ time even without the assumptions on the intersections of segments of the same colour, but their presence makes the solution simpler.

▶ **Lemma 44.** *The Coloured Strips Problem with Shades can be solved in time* $\mathcal{O}(N)$.

Proof. The proof mimics that of Lemma 38, with a few differences. We compute for each grid cell (i, j) a bitmask $hcol(i, j)$ of size $\sum_i s_i$ that specifies, for each colour k and shade a, if the cell is covered by a horizontal line segment of colour k and a shade of darkness at least a. Now a horizontal line segment of shade a sets all bits of shades $1, \ldots, a$ in the bitmask, which can be done in $\mathcal{O}(1)$ time with standard word-RAM operations.

With the presence of shades, we can no longer merge intersecting horizontal line segments of the same colour. However, when processing each row using a line sweeping algorithm, the simplifying assumptions in the problem guarantee that we may have at most two active line segments of the same colour. We maintain their shades explicitly, which lets us update the maintained bitmask.

We then compute *vcol*-bitmasks and return the darkest shade of each colour whose corresponding bit is set in the bitmask $\bigwedge_{i,j} (hcol(i, j) \vee vcol(i, j))$. ◀

By combining Lemmas 41, 43 and 44 we arrive at the main result of this section.

▶ **Theorem 45.** *All 1D-covers of a 2D-string of size* N *can be computed in time* $\mathcal{O}(N)$.

References

1 Amihood Amir, Ayelet Butman, Eitan Kondratovsky, Avivit Levy, and Dina Sokol. Multidimensional period recovery. In *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020*, volume 12303 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2020. doi:10.1007/978-3-030-59212-7_9.

2 Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.

3 Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7(4):533–541, 1978. doi:10.1137/0207043.

4 Jon Louis Bentley. Algorithms for Klee's rectangle problems. Unpublished notes, Computer Science Department, Carnegie Mellon University, 1977.

5 Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980. doi:10.1145/358841.358850.

6 Richard S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977. doi:10.1016/0020-0190(77)90017-5.

7 Dany Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.

8 Timothy M. Chan. Klee's measure problem made easy. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013*, pages 410–419. IEEE Computer Society, 2013. doi:10.1109/FOCS.2013.51.

9 Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. The number of repetitions in 2D-strings. In *28th Annual European Symposium on Algorithms, ESA 2020*, volume 173 of *LIPIcs*, pages 32:1–32:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.32.

10 Michalis Christou, Maxime Crochemore, and Costas S. Iliopoulos. Quasiperiodicities in Fibonacci strings. *Ars Combinatoria*, 129:211–225, 2016.

11 Maxime Crochemore, Costas S. Iliopoulos, and Maureen Korda. Two-dimensional prefix string matching and covering on square matrices. *Algorithmica*, 20(4):353–373, 1998. doi:10.1007/PL00009200.

12 Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific, 2002. doi:10.1142/4838.

13 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.2307/2034009.

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

15 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

16 Dennis W. G. Moore and William F. Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994. doi:10.1016/0020-0190(94)00045-X.

17 Dennis W. G. Moore and William F. Smyth. A correction to "An optimal algorithm to compute all the covers of a string". *Information Processing Letters*, 54(2):101–103, 1995. doi:10.1016/0020-0190(94)00235-Q.

18 Mark H. Overmars and Chee-Keng Yap. New upper bounds in Klee's measure problem. *SIAM Journal on Computing*, 20(6):1034–1045, 1991. doi:10.1137/0220065.

19 Alexandru Popa and Andrei Tanasescu. An output-sensitive algorithm for the minimization of 2-dimensional string covers. In *Theory and Applications of Models of Computation - 15th Annual Conference, TAMC 2019*, volume 11436 of *Lecture Notes in Computer Science*, pages 536–549. Springer, 2019. doi:10.1007/978-3-030-14812-6_33.

A Special Case of Klee's Measure Problem

In this section we show a solution to a special case of a dynamic version Klee's measure problem. Let us start with an auxiliary lemma.

We say that a rectangle is a *corner rectangle* in a grid if one of its corners coincides with a corner of the grid. We say that a rectangle is a *side rectangle* of a grid if its width or height is equal to k and one of its sides coincides with one of the sides of the grid. Our solution is based on the following key lemma.

▶ **Lemma 46.** *Dynamic Klee's Measure problem on a $k \times k$ grid with only corner and side rectangles inserted to $\mathcal{R}$ can be solved in $\tilde{\mathcal{O}}(k)$ time per insertion/deletion of a corner rectangle and $\tilde{\mathcal{O}}(1)$ time per insertion/deletion of a side rectangle.*

Proof. Let us first consider operations on corner rectangles. Each corner rectangle covers a top or bottom interval of positions in each grid column. Hence, the total area that is *not* covered by corner rectangles consists of at most one interval of positions in each column; see Figure 6. This representation of the non-covered area can be computed in $\mathcal{O}(k)$ time if, for each type of a corner rectangle (top-left, top-right etc.) we store the maximum-height corner rectangle of each possible width. These values can be updated in $\tilde{\mathcal{O}}(k)$ time per insertion or deletion of a corner rectangle if all corner rectangles are stored in a balanced binary search tree (BST).

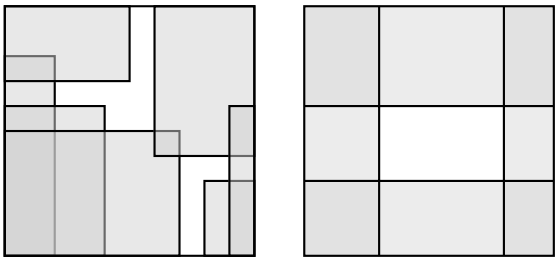


Figure 6 The intersection of the region that is not covered by corner rectangles with each column consists of at most one interval (left). The region that is not covered by side rectangles is a rectangle (right). The non-covered regions are shown in white.

The area not covered by side rectangles is a rectangle that only depends on the two highest side rectangles that contain the horizontal sides of grid and the two widest side rectangles that contain the vertical sides; see Figure 6 again. The rectangle can be updated in $\tilde{\mathcal{O}}(1)$ time after an insertion or a deletion of a side rectangle if all side rectangles are stored in a balanced BST. We construct a data structure that is recomputed from scratch after each insertion or deletion of a corner rectangle and allows us to compute the area of the intersection of the region that is not covered by corner rectangles with a query rectangle.

The area of each column covered by corners can be represented by (at most) two disjoint vertical strips (i.e., width-1 rectangles), each adjacent to the boundary of the grid. Let us focus on handling vertical strips with opposite corners $(x, 0)$ and $(x + 1, y)$, as vertical strips with corners (x, y) and $(x + 1, k)$ can be handled symmetrically. No two vertical strips in our collection intersect, and hence we can indeed handle each case separately.

We will maintain a set of (weighted) points $\mathcal{P}$ in 2D, such that a vertical strip with corners $(x, 0)$ and $(x + 1, y)$ will be represented by point (x, y) with weight y. We will store a 2D range query data structure over $\mathcal{P}$, capable of returning the number and the total weight of points inside any queried axes-parallel rectangle. Consider such a rectangle R with corners (i, a) and (j, b), $i \leq j$, $a \leq b$.

We have the following cases for the intersection of a vertical strip β (as above) and R.

- Case I: $b \leq y$. In this case, the intersection of β and R has area $b - a$. The total area of R covered by such vertical strips is equal to the product of $b - a$ and the number of points of $\mathcal{P}$ in $[i\,..\,j-1] \times [b\,..\,\infty)$.
- Case II: $a < y < b$. In this case, the intersection of β and R has area $y - a$. The total area of R covered by such vertical strips is equal to the total weight of points of $\mathcal{P}$ in $[i\,..\,j-1] \times [a\,..\,b)$ minus the product of their number and a.

We can implement the 2D range query data structures with $\tilde{\mathcal{O}}(1)$ query time and $\tilde{\mathcal{O}}(k)$ space and construction time using range trees [5]. ◀

Let us recall that a rectangle is called a fat rectangle if its width *and* height are at least $\sqrt{n}$.

▶ **Lemma 17.** *The* Dynamic Klee's Measure *problem with fat rectangles can be solved in* $\tilde{\mathcal{O}}(\sqrt{n})$ *time per operation, after* $\tilde{\mathcal{O}}(N)$*-time preprocessing.*

Proof. Let us partition the grid into unit squares of height and width $\lceil\sqrt{n}\rceil$. Each fat rectangle in the grid is thus partitioned into: corner rectangles in at most 4 unit squares, side rectangles in $\mathcal{O}((n+m)/\sqrt{n}) = \mathcal{O}(\sqrt{n})$ unit squares, and several consecutive full unit squares in each column; see Figure 7.

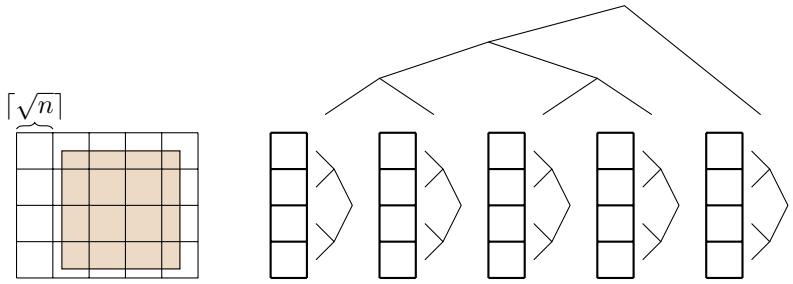


Figure 7 To the left: a fat rectangle decomposed into 4 corner rectangles, 8 side rectangles and 4 full unit squares. To the right: the structure of range trees that forms a kd-tree.

Each unit square stores the data structure of Lemma 46 that returns the covered area in this unit square. We also build range trees over each column of unit squares, and one range tree over the set of columns (see Figure 7 again), in order to support intervals of full unit squares in rectangle decompositions. We can use the range tree by Bentley [4] that allows insertions and deletions of intervals and counting the length of the union of intervals in $\mathcal{O}(\log N)$ time. This construction of range trees is similar to the kd-tree used in [18]. Let us mention that the range trees need to be slightly adjusted in order to account for the values returned by the data structure of Lemma 46 for unit squares that are not covered in full by any rectangle. ◀

A.1 Checking if the Whole Grid is Covered

If we are only interested in checking if the union of all rectangles in the dynamic problem covers the whole grid, which is the case in our application, the approach can be simplified as follows.

In Lemma 17, the data structure for each unit square returns a single bit of information, so simply no modification is needed in the range tree by Bentley [4].

The 2D range query data structure from the proof of Lemma 46 can be substituted with range minimum queries on a 1D array as follows. The complement of the union of

corner rectangles consists of horizontal and vertical strips, at most one vertical strip in a column and at most one horizontal strip in a row. Consequently, the complement of union of corner rectangles can be described in $\mathcal{O}(k)$ space using $dist_N$, $dist_E$, $dist_S$, $dist_W$ arrays that measure the distance (respectively in the direction *North, South, East, West*) from each half-integral point in the corresponding side of the whole grid to the first non-covered position in a given direction (see Figure 8).

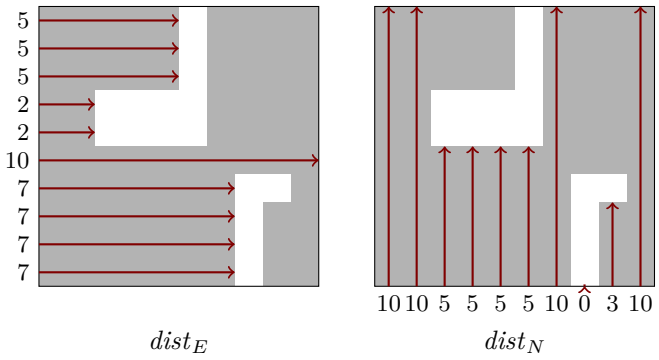


Figure 8 Representation of the regions not covered by corner rectangles from Figure 6. The sequences of numbers in the left/bottom sides are the arrays $dist_E, dist_N$. The (similar) arrays $dist_S, dist_W$ are not shown here.

Using four range minimum queries over *dist* arrays we can check in $\mathcal{O}(1)$ time if there exists some non-covered grid cell in a query rectangle. More precisely, a rectangle with corners (i, a) and (j, b), $i \leq j$, $a \leq b$, does *not* contain any non-covered cell if and only if at least one of the following conditions is satisfied:

- $\min dist_E[i\,..\,j-1] \geq b$,
- $\min dist_W[i\,..\,j-1] \geq k-a$,
- $\min dist_N[a\,..\,b-1] \geq j$,
- $\min dist_S[a\,..\,b-1] \geq k-i$.

B Remaining Proofs

▶ **Theorem 14.** *All aperiodic 2D-covers of a 2D-string of size N can be computed in $\mathcal{O}(N \log N)$ time.*

Proof. By Lemma 9, aperiodic 2D-borders have only $\mathcal{O}(\log m)$ and $\mathcal{O}(\log n)$ possible heights and widths, respectively. All periodic 2D-borders can be filtered out in $\mathcal{O}(N)$ time using the border arrays of $\mathsf{hstr}(T^{(h)})$ and $\mathsf{vstr}(T_{(w)})$ for all h, w. Let us leave only those aperiodic 2D-borders as candidates which cover both $T^{(h)}$ and $T_{(w)}$ for their respective height h and width w. They can be identified using an algorithm for finding all covers in a 1D-string [16, 17] applied to $\mathsf{hstr}(T^{(h)})$ and $\mathsf{vstr}(T_{(w)})$.

If a candidate $T[1\,..\,h, 1\,..\,w]$ is a 2D-cover of T, then candidates $T[1\,..\,h', 1\,..\,w]$ and $T[1\,..\,h, 1\,..\,w']$ for $h' \leq h$, $w' \leq w$ are also 2D-covers of T. We check the candidates sorted by non-increasing height (and by non-decreasing width in case of draws), using the Is2DCover(C, T) routine for each candidate C. If the candidate in question turns out to be a 2D-cover, then we approve all the remaining ones with the same width as well, and if it is not, then we can remove all the remaining ones of the same height. Hence, we can charge each test to a unique width or height, which means that we perform $\mathcal{O}(\log n + \log m) = \mathcal{O}(\log N)$ tests in total. Each test can be performed in $\mathcal{O}(N)$ time due to Lemma 4. ◀

▶ **Lemma 29.** *For $X, Y \in \mathbf{B}(S)$ with $|X| < |Y|$, we have $\mathit{per}(Y) \geq \mathit{per}(X) \cdot |\mathit{Group}(X)|$. Moreover, for $X, Y, Z, W \in \mathbf{B}(S)$ with $|X| < |Y| < |Z| < |W|$, we have $\mathit{per}(W) \geq \frac{9}{8}\mathit{per}(X)$.*

Proof. We first state a definition and a few facts. A string U is called *primitive* if it cannot be expressed as V^k for a string V and an integer $k > 1$. The *synchronization property* states that a non-empty string U is primitive if and only if it occurs only twice in UU: as a prefix and as a suffix. Moreover, any $|\mathsf{per}(U)|$-length fragment of a string U is primitive [12].

Let us now prove the first statement of the lemma. If X is aperiodic, then the conclusion is clear. Henceforth let us assume that X is periodic. In this case we have $\mathsf{per}(Y) > \mathsf{per}(X)$. Towards a contradiction, suppose that

$$\mathsf{per}(Y) < \mathsf{per}(X) \cdot |\mathsf{Group}(X)| \leq |X| - \mathsf{per}(X).$$

First, it cannot be that $\mathsf{per}(X)$ divides $\mathsf{per}(Y)$, since this would contradict the primitivity of $S[1\,..\,\mathsf{per}(Y)]$.

In the complementary case that $\mathsf{per}(X)$ does not divide $\mathsf{per}(Y)$, we have that

$$\mathsf{per}(Y) + \mathsf{per}(X) < |X| < |Y| \text{ implies } [1\,..\,\mathsf{per}(X)] = S[\mathsf{per}(Y) + 1\,..\,\mathsf{per}(Y) + \mathsf{per}(X)].$$

We thus have an occurrence of $U = S[1\,..\,\mathsf{per}(X)]$ in X that starts in a position that is not a multiple of $\mathsf{per}(X)$. This contradicts the primitivity of U, due to the synchronization property.

We now move to the proof of the second statement of the lemma. First, let us note that for $U, V \in \mathbf{B}(S)$ with $|U| < |V|$ we have $|V| \geq \frac{3}{2}|U|$. Let us distinguish between two cases.

- **Case I:** $\mathsf{per}(W) = \mathsf{per}(Z)$.
  In this case, Z must be aperiodic by the definition of $\mathbf{B}(S)$, i.e. $\mathsf{per}(Z) > |Z|/2$. Then, Z's longest border cannot be longer than $|Z| - \mathsf{per}(Z) < \mathsf{per}(Z)$. As Y must be a border of Z, we have $|Y| < \mathsf{per}(Z)$. Then, the fact that $\frac{3}{2}\mathsf{per}(X) \leq \frac{3}{2}|X| \leq |Y|$ proves the statement.
- **Case II:** $\mathsf{per}(W) > \mathsf{per}(Z)$.
  In this case, by the periodicity lemma (Lemma 7) we have $|Z| \leq 2\mathsf{per}(W)$. We thus have $\frac{9}{8}\mathsf{per}(X) \leq \frac{9}{8}|X| \leq \frac{3}{4}|Y| \leq |Z|/2 \leq \mathsf{per}(W)$. ◀

▶ **Lemma 36.** *Let $S_1, \ldots, S_k$ be prefixes of W. Then we can compute all the sets $Occ(S_i, T)$ in $\mathcal{O}(N + \sum_{i=1}^{k} |Occ(S_i, T)|)$ time.*

Proof. Let us recall the *PREF* array of a string S that stores, for each $i \in [2\,..\,|S|]$, the length of the longest common prefix of $S[i\,..\,|S|]$ and S. This array can be computed in linear time [12]. We use the *PREF* array to compute the sets $Occ(S_i, T)$ in time linear in their total size plus $\mathcal{O}(N)$ as follows. Let us assume that S_i are sorted by increasing lengths. We store a doubly-linked list L of pairs, initially containing all pairs from $[1\,..\,m] \times [1\,..\,n]$. Moreover, each pair stores a pointer to its corresponding element in the list L, if such an element exists. For each row r, we compute the table $PREF_r$ as the *PREF* array of $T[1, 1\,..\,n]\#T[r, 1\,..\,n]$. We construct n buckets and store in the j-th bucket all pairs (r, c) such that $PREF_r[n + 1 + c] = j$. Then, for each $j = 1, \ldots, n$, we report L as $Occ(S_i, T)$ if $j = |S_i|$ and then remove from L all pairs from the j-th bucket. ◀

A Fast and Small Subsampled R-Index

Dustin Cobas
CeBiB – Center for Biotechnology and Bioengineering, Santiago, Chile
Dept. of Computer Science, University of Chile, Santiago, Chile

Travis Gagie
CeBiB – Center for Biotechnology and Bioengineering, Santiago, Chile
Dalhousie University, Halifax, Canada

Gonzalo Navarro
CeBiB – Center for Biotechnology and Bioengineering, Santiago, Chile
Dept. of Computer Science, University of Chile, Santiago, Chile

Abstract

The *r*-index (Gagie et al., JACM 2020) represented a breakthrough in compressed indexing of repetitive text collections, outperforming its alternatives by orders of magnitude. Its space usage, $\mathcal{O}(r)$ where r is the number of runs in the Burrows–Wheeler Transform of the text, is however larger than Lempel–Ziv and grammar-based indexes, and makes it uninteresting in various real-life scenarios of milder repetitiveness. In this paper we introduce the *sr*-index, a variant that limits a large fraction of the space to $\mathcal{O}(\min(r, n/s))$ for a text of length n and a given parameter s, at the expense of multiplying by s the time per occurrence reported. The *sr*-index is obtained by carefully subsampling the text positions indexed by the *r*-index, in a way that we prove is still able to support pattern matching with guaranteed performance. Our experiments demonstrate that the *sr*-index sharply outperforms virtually every other compressed index on repetitive texts, both in time and space, even matching the performance of the *r*-index while using 1.5–3.0 times less space. Only some Lempel–Ziv-based indexes achieve better compression than the *sr*-index, using about half the space, but they are an order of magnitude slower.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Pattern matching, r-index, compressed text indexing, repetitive text collections

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.13

Supplementary Material *Software (Source Code)*: `https://github.com/duscob/sr-index` archived at `swh:1:dir:edbc683faf60b1c3604e211cb464d6183822bb0a`

Funding *Dustin Cobas*: ANID/Scholarship Program/DOCTORADO BECAS CHILE/2020-21200906, Chile. Basal Funds FB0001, ANID, Chile.
Travis Gagie: NSERC Discovery Grant RGPIN-07185-2020. Basal Funds FB0001, ANID, Chile.
Gonzalo Navarro: Fondecyt grant 1-200038, ANID, Chile. Basal Funds FB0001, ANID, Chile.

1 Introduction

The rapid surge of massive repetitive text collections, like genome and sequence read sets and versioned document and software repositories, has raised the interest in text indexing techniques that exploit repetitiveness to obtain orders-of-magnitude space reductions, while supporting pattern matching directly on the compressed text representations [10, 21].

Traditional compressed indexes rely on statistical compression [22], but this is ineffective to capture repetitiveness [15]. A new wave of repetitiveness-aware indexes [21] build on other compression mechanisms like Lempel–Ziv [16] or grammar compression [14]. A particularly useful index of this kind is the rlfm-index [18, 19], because it emulates the classical suffix array [20] and this simplifies translating suffix-array based algorithms to run on it [17].

The rlfm-index represents the Burrows–Wheeler Transform (BWT) [3] of the text in run-length compressed form, because the number r of maximal equal-letter runs in the BWT is known to be small on repetitive texts. A problem with the rlfm-index is that, although it can count the number of occurrences of a pattern using $\mathcal{O}(r)$ space, it needs to sample the text at every sth position, for a parameter s, in order to locate each of those occurrences in time proportional to s. The $\mathcal{O}(n/s)$ additional space incurred on a text of length n ruins the compression on very repetitive collections, where $r \ll n$. The recent *r*-index [11] closed the long-standing problem of efficiently locating the occurrences within $\mathcal{O}(r)$ space, offering pattern matching time orders of magnitude faster than previous repetitiveness-aware indexes.

In terms of space, however, the *r*-index is considerably larger than Lempel–Ziv based indexes of size $\mathcal{O}(z)$, where z is the number of phrases in the Lempel–Ziv parse. Gagie et al. [11] show that, on extremely repetitive text collections where $n/r = 500$–$10{,}000$, r is around $3z$ and the *r*-index size is 0.06–0.2 bits per symbol (bps), about twice that of the lz-index [15], a baseline Lempel–Ziv based index. However, r degrades faster than z as repetitiveness drops: in an experiment on bacterial genomes in the same article, where $n/r \approx 100$, the *r*-index space approaches 0.9 bps, 4 times that of the lz-index; r also approaches $4z$. Experiments on other datasets show that the *r*-index tends to be considerably larger [23, 5, 6, 1].Indeed, in some realistic cases n/r can be over 1,500, but in most cases it is well below: 40–160 on versioned software and document collections and fully assembled human chromosomes, 7.5–50 on virus and bacterial genomes (with r in the range $4z$–$7z$), and 4–9 on sequencing reads; see Section 5. An *r*-index on such a small n/r ratio easily becomes larger than the plain sequence data.

In this paper we tackle the problem of the (relatively) large space usage of the *r*-index. This index manages to locate the pattern occurrences by sampling r text positions (corresponding to the ends of BWT runs). We show that one can remove some carefully chosen samples so that, given a parameter s, the index stores only $\mathcal{O}(\min(r, n/s))$ samples while its locating machinery can still be used to guarantee that every pattern occurrence is located within $\mathcal{O}(s)$ steps. We call the resulting index the *subsampled* r-*index*, or *sr*-index. The worst-case time to locate the occ occurrences of a pattern of length m on an alphabet of size σ then rises from $\mathcal{O}((m + occ)\log(\sigma + n/r))$ in the implemented *r*-index to $\mathcal{O}((m + s \cdot occ)\log(\sigma + n/r))$ in the *sr*-index, which matches the search cost of the rlfm-index.

The *sr*-index can then be seen as a hybrid between the *r*-index (matching it when $s = 1$) and the rlfm-index (obtaining its time with less space; the spaces become similar when repetitiveness drops). In practice, however, the *sr*-index performs much better than both on repetitive texts, sharply dominating the rlfm-index, the best grammar-based index [5], and in most cases the lz-index, both in space and time. The *sr*-index can also get as fast as the *r*-index while using 1.5–4.0 times less space. Its only remaining competitor is a hybrid between a Lempel–Ziv based and a statistical index [7]. This index can use up to half the space of the *sr*-index, but it is an order of magnitude slower. Overall, the *sr*-index stays orders of magnitude faster than all the alternatives while using practical amounts of space in a wide range of repetitiveness scenarios.

2 Background

The *suffix array* [20] $\mathsf{SA}[1..n]$ of a string $\mathcal{T}[1..n]$ over alphabet $[1..\sigma]$ is a permutation of the starting positions of all the suffixes of $\mathcal{T}$ in lexicographic order, $\mathcal{T}[\mathsf{SA}[i]..n] < \mathcal{T}[\mathsf{SA}[i+1]..n]$ for all $1 \le i < n$. The suffix array can be binary searched in time $\mathcal{O}(m \log n)$ to obtain the range $\mathsf{SA}[sp..ep]$ of all the suffixes prefixed by a search pattern $P[1..m]$ (which then occurs $occ = ep - sp + 1$ times in $\mathcal{T}$). Once they are *counted* (i.e., their suffix array range is

determined), those occurrences are *located* in time $\mathcal{O}(occ)$ by simply listing their starting positions, $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$. The suffix array can then be stored in $n\lceil \lg n \rceil$ bits (plus the $n\lceil \lg \sigma \rceil$ bits to store $\mathcal{T}$) and searches for P in $\mathcal{T}$ in total time $\mathcal{O}(m \log n + occ)$.

Compressed suffix arrays (CSAs) [22] are space-efficient representations of both the suffix array (SA) and the text ($\mathcal{T}$). They can find the interval $\mathsf{SA}[sp..ep]$ corresponding to $P[1..m]$ in time $\mathrm{t}_{\mathrm{search}}(m)$ and access any cell $\mathsf{SA}[j]$ in time $\mathrm{t}_{\mathrm{lookup}}(n)$, so they can be used to search for P in time $\mathcal{O}(\mathrm{t}_{\mathrm{search}}(m) + occ\, \mathrm{t}_{\mathrm{lookup}}(n))$. Most CSAs need to store sampled SA values to compute any $\mathsf{SA}[j]$ in order to support the locate operation, inducing the tradeoff of using $\mathcal{O}((n/s)\log n)$ extra bits to obtain time $\mathrm{t}_{\mathrm{lookup}}(n)$ proportional to a parameter s.

The *Burrows–Wheeler Transform* [3] of $\mathcal{T}$ is a permutation $\mathsf{BWT}[1..n]$ of $\mathcal{T}[1..n]$ defined as $\mathsf{BWT}[i] = \mathcal{T}[\mathsf{SA}[i] - 1]$ (and $\mathcal{T}[n]$ if $\mathsf{SA}[i] = 1$), which boosts the compressibility of $\mathcal{T}$. The fm-index [8, 9] is a CSA that represents SA and $\mathcal{T}$ within the *statistical entropy* of $\mathcal{T}$, by exploiting the connection between the BWT and SA. For counting, the fm-index resorts to *backward search*, which successively finds the suffix array ranges $\mathsf{SA}[sp_i..ep_i]$ of $P[i..m]$, for $i = m$ to 1, starting from $\mathsf{SA}[sp_{m+1}..ep_{m+1}] = [1..n]$ and then

$$\begin{aligned} sp_i &= C[c] + \mathtt{rank}_c(\mathsf{BWT}, sp_{i+1} - 1) + 1, \\ ep_i &= C[c] + \mathtt{rank}_c(\mathsf{BWT}, ep_{i+1}), \end{aligned}$$

where $c = P[i]$, $C[c]$ is the number of occurrences of symbols smaller than c in $\mathcal{T}$, and $\mathtt{rank}_c(\mathsf{BWT}, j)$ is the number of times c occurs in $\mathsf{BWT}[1..j]$. Thus, $[sp, ep] = [sp_1, ep_1]$ if $sp_i \le ep_i$ holds for all $1 \le i \le m$.

For locating the occurrences $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$, the fm-index uses SA sampling as described: it stores sampled values of SA at regularly spaced text positions, say multiples of s. This is done via the so-called *LF-steps*: The BWT allows one to efficiently compute, given j such that $\mathsf{SA}[j] = i$, the value j' such that $\mathsf{SA}[j'] = i - 1$, called $j' = \mathsf{LF}(j)$. The formula is

$$\mathsf{LF}(i) = C[c] + \mathtt{rank}_c(\mathsf{BWT}, i),$$

where $c = \mathsf{BWT}[i]$. Note that the LF-steps virtually traverse the text backwards. By marking with 1s in a bitvector $B[1..n]$ the positions j^* such that $\mathsf{SA}[j^*]$ is a multiple of s, we can start from any j and, in $k < s$ LF-steps, find some sampled position $j^* = \mathsf{LF}^k(j)$ where $B[j^*] = 1$. By storing those values $\mathsf{SA}[j^*]$ explicitly, we have $\mathsf{SA}[j] = \mathsf{SA}[j^*] + k$.

By implementing BWT with a wavelet tree, for example, access and $\mathtt{rank}_c$ on BWT can be supported in time $\mathcal{O}(\log \sigma)$, and the fm-index searches in time $\mathcal{O}((m + s \cdot occ)\log \sigma)$ [9].

Since the statistical entropy is insensitive to repetitiveness [15], however, the fm-index is not adequate for repetitive datasets. The *Run-Length FM-index*, rlfm-index (and its variant rlcsa) [18, 19], is a modification of the fm-index aimed at repetitive texts. Say that the $\mathsf{BWT}[1..n]$ is formed by r maximal runs of equal symbols, then r is relatively small in repetitive collections (in particular, $r = \mathcal{O}(z \log^2 n)$, where z is the number of phrases of the Lempel–Ziv parse of $\mathcal{T}$ [13]). The rlfm-index supports counting within $\mathcal{O}(r \log n)$ bits, by implementing the backward search over alternative data structures. In particular, it marks in a bitvector $\mathtt{Start}[1..n]$ with 1s the positions j starting BWT runs, that is, where $j = 1$ or $\mathsf{BWT}[j] \neq \mathsf{BWT}[j-1]$. The first letter of each run is collected in an array $\mathtt{Letter}[1..r]$. Since $\mathtt{Start}$ has only r 1s, it can be represented within $r \lg(n/r) + \mathcal{O}(r)$ bits. Within this space, one can access any bit $\mathtt{Start}[j]$ and support operation $\mathtt{rank}_1(\mathtt{Start}, j)$, which counts the number of 1s in $\mathtt{Start}[1..j]$, in time $\mathcal{O}(\log(n/r))$ [25]. Therefore, we simulate $\mathsf{BWT}[j] = \mathtt{Letter}[\mathtt{rank}_1(\mathtt{Start}, j)]$ in $\mathcal{O}(r \log n)$ bits. The backward search formula can be efficiently simulated as well, leading to $\mathcal{O}((m + s \cdot occ)\log(\sigma + n/r))$ search time. However, the rlfm-index still uses SA samples to locate, and when $r \ll n$ (i.e., on repetitive texts), the $\mathcal{O}((n/s)\log n)$ added bits ruin the $\mathcal{O}(r \log n)$-bit space (s is typically $\mathcal{O}(\log n)$ or close).

The *r*-index [11] closed the long-standing problem of efficiently locating the occurrences of a pattern in a text using $\mathcal{O}(r \log n)$-bit space. The experiments showed that the *r*-index outperforms all the other implemented indexes by orders of magnitude in space or in time to locate pattern occurrences on highly repetitive datasets. However, other experiments on more typical repetitiveness scenarios [23, 5, 6, 1] showed that the space of the *r*-index degrades very quickly as repetitiveness decreases. For example, a grammar-based index (which can be of size $g = \mathcal{O}(z \log(n/z))$) is usually slower but significantly smaller [5], and an even slower Lempel–Ziv based index of size $O(z)$ [15] is even smaller. Some later proposals [24] further speed up the *r*-index by increasing the constant accompanying the $\mathcal{O}(r \log n)$-bit space. The unmatched time performance of the *r*-index comes then with a very high price in space on all but the most highly repetitive text collections, which makes it of little use in many relevant application scenarios. This is the problem we address in this paper.

3 The *r*-index Sampling Mechanism

Gagie et al. [11] provide an $\mathcal{O}(r \log n)$-bits data structure that not only finds the range $\mathsf{SA}[sp..ep]$ of the occurrences of P in $\mathcal{T}$, but also gives the value $\mathsf{SA}[ep]$, that is, the text position of the last occurrence in the range. They then provide a second $\mathcal{O}(r \log n)$-bits data structure that, given $\mathsf{SA}[j]$, efficiently finds $\mathsf{SA}[j-1]$. This suffices to efficiently find all the occurrences of P, in time $\mathcal{O}((m + occ) \log \log(\sigma + n/r))$ in their theoretical version.

In addition to the theoretical design, Gagie et al. and Boucher et al. [11, 2] provided a carefully engineered *r*-index implementation. The counting data structures (which find the range $\mathsf{SA}[sp..ep]$) require, for any small constant $\epsilon > 0$, $r \cdot ((1+\epsilon)\lg(n/r) + \lg\sigma + \mathcal{O}(1))$ bits (largely dominated by the described arrays `Start` and `Letter`), whereas the locating data structures (which obtain $\mathsf{SA}[ep]$, and $\mathsf{SA}[j-1]$ given $\mathsf{SA}[j]$), require $r \cdot (2\lg n + \mathcal{O}(1))$ further bits. The locating structures are then significantly heavier in practice, especially when n/r is not that large. Together, the structures use $r \cdot ((1+\epsilon)\lg(n/r) + 2\lg n + \lg\sigma + \mathcal{O}(1))$ bits of space and perform backward search steps and LF-steps in time $\mathcal{O}(\frac{1}{\epsilon}\log(\sigma + n/r))$, so they search for P in time $\mathcal{O}(\frac{1}{\epsilon}(m + occ)\log(\sigma + n/r))$.

For conciseness we do not describe the counting data structures of the *r*-index, which are the same of the rlfm-index and which we do not modify in our index. The *r*-index locating structures, which we do modify, are formed by the following components:

- `First`$[1..n]$: a bitvector marking with 1s the *text* positions of the letters that are the first in a BWT run. That is, if $j = 1$ or $\mathsf{BWT}[j] \neq \mathsf{BWT}[j-1]$, then $\texttt{First}[\mathsf{SA}[j]-1] = 1$. Since `First` has only r 1s, it is represented in compressed form using $r\lg(n/r) + \mathcal{O}(r)$ bits, while supporting $\texttt{rank}_1$ in time $\mathcal{O}(\log(n/r))$ and, in $\mathcal{O}(1)$ time, the operation $\texttt{select}_1(\texttt{First}, j)$ (the position of the jth 1 in `First`) [25]. This allows one find the rightmost 1 up to position i, $\texttt{pred}(\texttt{First}, i) = \texttt{select}_1(\texttt{First}, \texttt{rank}_1(\texttt{First}, i))$.
- `FirstToRun`$[1..r]$: a vector of integers (using $r\lceil\lg r\rceil$ bits) mapping each letter marked in `First` to the BWT run where it lies. That is, if the pth BWT run starts at $\mathsf{BWT}[j]$, and $\texttt{First}[i] = 1$ for $i = \mathsf{SA}[j] - 1$, then $\texttt{FirstToRun}[\texttt{rank}_1(\texttt{First}, i)] = p$.
- `Samples`$[1..r]$: a vector of $\lceil\lg n\rceil$-bit integers storing samples of SA, so that $\texttt{Samples}[p]$ is the text position $\mathsf{SA}[j]-1$ corresponding to the last letter $\mathsf{BWT}[j]$ in the pth BWT run.

These structures are used in the following way in the *r*-index implementation [11]:

Problem 1: When computing the ranges $\mathsf{SA}[sp..ep]$ along the backward search, we must also produce the value $\mathsf{SA}[ep]$. They actually compute all the values $\mathsf{SA}[ep_i]$. This is stored for $\mathsf{SA}[ep_{m+1}] = \mathsf{SA}[n]$ and then, if $\mathsf{BWT}[ep_{i+1}] = P[i]$, we know that $ep_i = \mathsf{LF}(ep_{i+1})$ and thus $\mathsf{SA}[ep_i] = \mathsf{SA}[ep_{i+1}] - 1$. Otherwise, $ep_i = \mathsf{LF}(j)$ and $\mathsf{SA}[ep_i] = \mathsf{SA}[j] - 1$, where

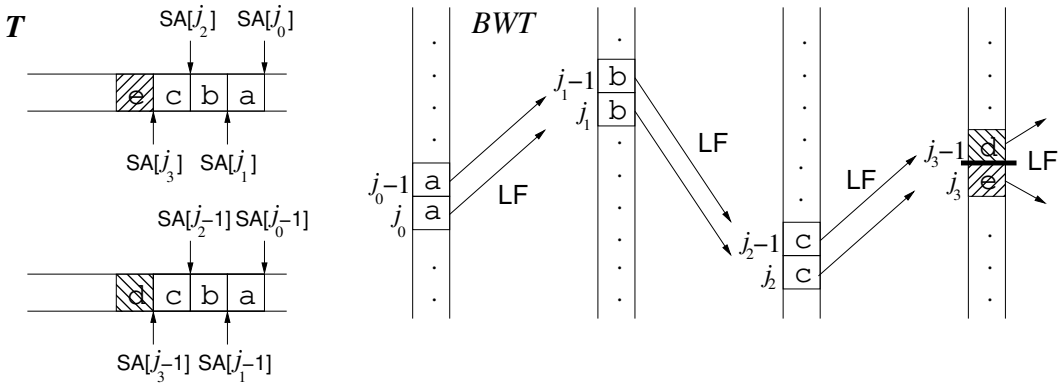


Figure 1 Schematic example of the sampling mechanism of the *r*-index. There is a run border between $j_3 - 1$ and j_3.

$j \in [sp_{i+1}..ep_{i+1}]$ is the largest position with $\mathsf{BWT}[j] = P[i]$. The position j is efficiently found with their counting data structures, and the remaining problem is how to compute $\mathsf{SA}[j]$. Since j must be an end of run, however, this is simply computed as $\texttt{Samples}[p] + 1$, where $p = \texttt{rank}_1(\texttt{Start}, j)$ is the run where j belongs.

Problem 2: When locating we must find $\mathsf{SA}[j-1]$ from $i = \mathsf{SA}[j] - 1$. There are two cases:

- $j - 1$ ends a BWT run, that is, $\texttt{Start}[j] = 1$, and then $\mathsf{SA}[j-1] = \texttt{Samples}[p-1] + 1$, where p is as in Problem 1;
- $j - 1$ is in the same BWT run of j, in which case they compute $\mathsf{SA}[j-1] = \phi(i)$, where[1]

$$\phi(i) = \texttt{Samples}[\texttt{FirstToRun}[\texttt{rank}_1(\texttt{First}, i)] - 1] + 1 + (i - \texttt{pred}(\texttt{First}, i)). \quad (1)$$

This formula works because, when j and $j-1$ are in the same BWT run, it holds that $\mathsf{LF}(j-1) = \mathsf{LF}(j) - 1$ [8]. Figure 1 explains why this property makes the formula work. Consider two BWT positions, $j = j_0$ and $j' = j - 1 = j_0 - 1$, that belong to the same run. The LF formula will map them to consecutive positions, j_1 and $j'_1 = j_1 - 1$. If j_1 and $j_1 - 1$ still belong to the same run, LF will map them to consecutive positions again, j_2 and $j'_2 = j_2 - 1$, and once again, j_3 and $j'_3 = j_3 - 1$. Say that j_3 and $j_3 - 1$ do not belong to the same run. This means that $j_3 - 1$ ends a run (and thus it is stored in $\texttt{Samples}$) and j_3 starts a run (and thus $\mathsf{SA}[j_3] - 1$ is marked in $\texttt{First}$). To the left of the BWT positions we show the areas of $\mathcal{T}$ virtually traversed as we perform consecutive LF-steps. Therefore, if we know $i = \mathsf{SA}[j] - 1 = \mathsf{SA}[j_0] - 1$, the nearest 1 in $\texttt{First}$ to the left is at $\texttt{pred}(\texttt{First}, i) = \mathsf{SA}[j_3] - 1$ (where there is an $\texttt{e}$ in $\mathcal{T}$) and $p = \texttt{FirstToRun}[\texttt{rank}(i)]$ is the number of the BWT run that starts at j_3. If we subtract 1, we have the BWT run ending at $j_3 - 1$, and then $\texttt{Samples}[p-1]$ is the position preceding $\mathsf{SA}[j_3 - 1]$ (where there is a $\texttt{d}$ in $\mathcal{T}$). We add $1 + (i - \texttt{pred}(\texttt{First}, i)) = 4$ to obtain $\mathsf{SA}[j_0 - 1] = \mathsf{SA}[j-1]$.

These components make up, effectively, a sampling mechanism of $\mathcal{O}(r \log n)$ bits (i.e., sampling the end of runs), instead of the traditional one of $\mathcal{O}((n/s) \log n)$ bits (i.e., sampling every sth text position).

[1] The special case where $\texttt{rank}_1(\texttt{First}, i) = 0$ is handled separately.

4 Our Subsampled *r*-index

Despite its good performance on highly repetitive texts, the sampling mechanism introduced by the *r*-index is excessive in areas where the BWT runs are short, because those induce oversampled ranges on the text. In this section we describe an *r*-index variant we dub *subsampled* r-*index*, or sr-*index*, which can be seen as a hybrid between the *r*-index and the rlfm-index. The *sr*-index samples the text at end of runs (like the *r*-index), but in oversampled areas it removes some samples to ensure that no three consecutive samples lie within distance s (roughly as in the rlfm-index). It then handles text areas with denser and sparser sampling in different ways.

4.1 Subsampling

The *sr*-index subsampling process removes *r*-index samples in oversampled areas. Concretely, let $t'_1 < \cdots < t'_r$ be the text positions of the last letters in BWT runs, that is, the sorted values in array `Samples`. For any $1 < i < r$, we remove the sample t'_i if $t'_{i+1} - t'_{i-1} \leq s$, where s is a parameter. This condition is tested and applied sequentially for $i = 2, \ldots, r-1$ (that is, if we removed t'_2 because $t'_3 - t'_1 \leq s$, then we next remove t'_3 if $t'_4 - t'_1 \leq s$; otherwise we remove t'_3 if $t'_4 - t'_2 \leq s$). Let us call $t_1, t_2, \ldots$ the sequence of the remaining samples.

The arrays `First`, `FirstToRun`, and `Samples` are built on the samples t_i only. That is, if we remove the sample $\texttt{Samples}[p] = t'$, we also remove the 1 in `First` corresponding to the first letter of the $(p+1)$th BWT run, which is the one Eq. (1) would have handled with $\texttt{Samples}[p]$. We also remove the corresponding entry of `FirstToRun`. Note that, if j is the first position of the $(p+1)$th run and $j-1$ the last of the pth run, then if we remove $\texttt{Samples}[p] = \mathsf{SA}[j-1] - 1$, we remove the corresponding 1 at position $\mathsf{SA}[j] - 1$ in `First`. Finally, note that `FirstToRun` must be adapted to point to the corresponding entry of `Samples`, once some entries of the latter are removed.

It is not hard to see that subsampling avoids the excessive space usage when r is not small enough, reducing it from $\mathcal{O}(r)$ to $\mathcal{O}(\min(r, n/s))$ entries for the locating structures.

▶ **Lemma 1.** *The subsampled structures* `First`*,* `FirstToRun`*, and* `Samples` *use* $\min(r, 2\lceil n/(s+1)\rceil) \cdot (2 \lg n + \mathcal{O}(1))$ *bits of space.*

Proof. This is the same space as in the implemented *r*-index, with the number of samples reduced from r to $\min(r, 2\lceil n/(s+1)\rceil)$. We start with r samples and remove some, so there are at most r. By construction, any remaining sample t_i satisfies $t_{i+1} - t_{i-1} > s$, so if we cut the text into blocks of length $s+1$, no block can contain more than 2 samples. ◀

Our index adds the following small structure on top of the above ones, so as to mark the removed samples:

`Removed`$[1..r]$: A bitvector telling which of the original samples have been removed, that is, $\texttt{Removed}[p] = 1$ iff the sample at the end of the pth BWT run was removed. We can compute any $\texttt{rank}_1(\texttt{Removed}, p)$ in constant time using $r + o(r)$ bits [4].

It is easy to see that, once the *r*-index structures are built, the *sr*-index subsampling, as well as building and updating the associated structures, are lightweight tasks, easily carried out in $\mathcal{O}(r)$ space and $\mathcal{O}(r \log r)$ time. It is also possible to build the subsampled structures directly without building the full *sr*-index sampling first, in $\mathcal{O}(n \log(\sigma + n/r))$ time: we simulate a backward text traversal using LF-steps, so that we can build bitvector `Removed`. A second similar traversal fills the 1s in `First` and the entries in `FirstToRun` and `Samples` for the runs whose sample was not removed.

4.2 Solving Problem 1

For Problem 1, we must compute $\mathsf{SA}[j]$, where j is the end of the pth run, with $p = \mathtt{rank}_1(\mathtt{Start}, j)$. This position is sampled in the *r*-index, where the problem is thus trivial: $\mathsf{SA}[j] = \mathtt{Samples}[p] + 1$. However, in the *sr*-index it might be that $\mathtt{Removed}[p] = 1$, which means that the subsampling process removed $\mathsf{SA}[j]$. In this case, we compute $j_k = \mathsf{LF}^k(j)$ for $k = 1, 2, \ldots$ until finding a sampled value $\mathsf{SA}[j_k]$ (i.e., $j_k = n$ or $\mathtt{Start}[j_k + 1] = 1$) that is not removed (i.e., $q = \mathtt{rank}_1(\mathtt{Start}, j_k)$ and $\mathtt{Removed}[q] = 0$). We then compute $q' = q - \mathtt{rank}_1(\mathtt{Removed}, q)$, and $\mathsf{SA}[j] = \mathtt{Samples}[q'] + k + 1$.

The next lemma shows that we find a nonremoved sample for some $k < s$.

▶ **Lemma 2.** *If there is a removed sample t'_j such that $t_i < t'_j < t_{i+1}$, then $t_{i+1} - t_i \leq s$.*

Proof. Since our subsampling process removes samples left to right, by the time we removed t'_j, the current sample t_i was already the nearest remaining sample to the left of t'_j. If the sample following t'_j was the current t_{i+1}, then we removed t'_j because $t_{i+1} - t_i \leq s$, and we are done. Otherwise, there were other samples to the right of t'_j, say $t'_{j+1}, t'_{j+2}, \ldots, t'_{j+k}$, that were consecutively removed until reaching the current sample t_{i+1}. We removed t'_j because $t'_{j+1} - t_i \leq s$. Then, for $1 \leq l < k$, we removed t'_{j+l} (after having removed $t'_j, t'_{j+1}, \ldots, t'_{j+l-1}$) because $t'_{j+l+1} - t_i \leq s$. Finally, we removed t'_{j+k} because $t_{i+1} - t_i \leq s$. ◀

This implies that, from a removed sample $\mathtt{Samples}[p] = t'$, surrounded by the remaining samples $t_i < t' < t_{i+1}$, we can perform only $k = t' - t_i < s$ LF-steps until $j_k = \mathsf{LF}^{(k)}(j)$ satisfies $\mathsf{SA}[j_k] - 1 = t_i$ and thus it is stored in $\mathtt{Samples}[q]$ and not removed.

If we followed verbatim the modified backward search of the *r*-index, finding every $\mathsf{SA}[ep_i]$, we would perform $\mathcal{O}(m \cdot s)$ steps on the *sr*-index. We now reduce this to $\mathcal{O}(m + s)$ steps by noting that the only value we need is $\mathsf{SA}[ep] = \mathsf{SA}[ep_1]$. Further, we need to know $\mathsf{SA}[ep_{i+1}]$ to compute $\mathsf{SA}[ep_i]$ only in the easy case where $\mathsf{BWT}[ep_{i+1}] = P[i]$ and so $\mathsf{SA}[ep_i] = \mathsf{SA}[ep_{i+1}] - 1$. Otherwise, the value $\mathsf{SA}[ep_i]$ is computed afresh.

We then proceed as follows. We do not compute any value $\mathsf{SA}[ep_i]$ during backward search; we only remember the last (i.e., smallest) value i' of i where the computation was not easy, that is, where $\mathsf{BWT}[ep_{i'+1}] \neq P[i']$. Then, $\mathsf{SA}[ep_1] = \mathsf{SA}[ep_{i'}] - (i' - 1)$ and we need to apply the procedure described above only once: we compute $\mathsf{SA}[j]$, where j is the largest position in $[sp_{i'+1}..ep_{i'+1}]$ where $\mathsf{BWT}[j] = P[i']$, and then $\mathsf{SA}[ep_{i'}] = \mathsf{SA}[j] - 1$.

Algorithm 1 gives the complete pseudocode that solves Problem 1. Note that, if P does not occur in $\mathcal{T}$ (i.e., $occ = 0$) we realize this after the $\mathcal{O}(m)$ backward steps because some $sp_i > ep_i$, and thus we do not spend the $\mathcal{O}(s)$ extra steps.

4.3 Solving Problem 2

For Problem 2, finding $\mathsf{SA}[j-1]$ from $i = \mathsf{SA}[j] - 1$, we first proceed as in Problem 1, from $j-1$. We compute $j'_k = \mathsf{LF}^k(j-1)$ for $k = 0, \ldots, s-1$. If any of those j'_k is the last symbol of its run (i.e., $j'_k = n$ or $\mathtt{Start}[j'_k + 1] = 1$), and the sample corresponding to this run was not removed (i.e., $\mathtt{Removed}[q] = 0$, with $q = \mathtt{rank}_1(\mathtt{Start}, j'_k)$), then we can obtain immediately $\mathsf{SA}[j'_k] = \mathtt{Samples}[q'] + 1$, where $q' = q - \mathtt{rank}_1(\mathtt{Removed}, q)$, and thus $\mathsf{SA}[j-1] = \mathsf{SA}[j'_k] + k$.

Unlike in Problem 1, $\mathsf{SA}[j-1]$ is not necessarily an end of run, and therefore we are not guaranteed to find a solution for $0 \leq k < s$. However, the following property shows that, if there were some end of runs j'_k, it is not possible that all were removed from $\mathtt{Samples}$.

▶ **Lemma 3.** *If there are no remaining samples in $\mathsf{SA}[j-1] - s, \ldots, \mathsf{SA}[j-1] - 1$, then no sample was removed between $\mathsf{SA}[j-1] - 1$ and its preceding remaining sample.*

Algorithm 1 Counting pattern occurrences on the *sr*-index.

Input : Search pattern $P[1..m]$.
Output: Returns suffix array range $[sp, ep]$ for P and $\mathsf{SA}[ep]$.

```
1  sp ← 1; ep ← n + 1
2  i ← m; i' ← m + 1
3  while i ≥ 1 and sp ≤ ep do
4      p ← rank_1(Start, ep)
5      if Letter[p] ≠ P[i] then
6          i' ← i; p' ← p
7      c ← P[i]
8      sp ← C[c] + rank_c(BWT, sp − 1) + 1
9      ep ← C[c] + rank_c(BWT, ep)
10 if sp > ep then return "P does not occur in T"
11 if i' = m + 1 then return [sp, ep] and SA[ep] = SA[n] − m (SA[n] is stored)
12 c ← P[i']
13 q ← select_c(Letter, rank_c(Letter, p')) (supported by the rlfm-index/r-index)
14 j ← select_1(Start, q + 1) − 1
15 k ← 0
16 while (j < n and Start[j + 1] = 0) or Removed[q] = 1 do
17     j ← LF(j)
18     q ← rank_1(Start, j)
19     k ← k + 1
20 return [sp, ep] and SA[ep] = Samples[q − rank_1(Removed, q)] + k + 1 − (i' − 1)
```

Proof. Let $t_i < \mathsf{SA}[j-1]-1 < t_{i+1}$ be the samples surrounding $\mathsf{SA}[j-1]-1$, so the remaining sample preceding $\mathsf{SA}[j-1]-1$ is t_i. Since $t_i < \mathsf{SA}[j-1]-s$, it follows that $t_{i+1}-t_i > s$ and thus, by Lemma 2, no samples were removed between t_i and t_{i+1}. ◀

This means that, if the process above fails to find an answer, then we can directly use Eq. (1), as we prove next.

▶ **Lemma 4.** *If there are no remaining samples in* $\mathsf{SA}[j-1]-s, \ldots, \mathsf{SA}[j-1]-1$*, then subsampling removed no* 1*s in* `First` *between positions* $i = \mathsf{SA}[j]-1$ *and* $\texttt{pred}(\texttt{First}, i)$.

Proof. Let $t_i < \mathsf{SA}[j-1]-1 < t_{i+1}$ be the samples surrounding $\mathsf{SA}[j-1]-1$, and $k = \mathsf{SA}[j-1]-1-t_i$. Lemma 3 implies that no sample existed between $\mathsf{SA}[j-1]-1$ and $\mathsf{SA}[j-1]-k = t_i+1$, and there exists one at t_i. Consequently, no 1 existed in `First` between positions $\mathsf{SA}[j]-1$ and $\mathsf{SA}[j]-k$ (inclusively), and there exists one in $\mathsf{SA}[j]-1-k$. Indeed, $\texttt{pred}(\texttt{First}, i) = \mathsf{SA}[j]-1-k$. ◀

A final twist, which does not change the worst-case complexity but improves performance in practice, is to reuse work among successive occurrences. Let $\mathsf{BWT}[sm..em]$ be a maximal run inside $\mathsf{BWT}[sp..ep]$. For every $sm \le j \le em$, the first LF-step will lead us to $\mathsf{LF}(j) = \mathsf{LF}(sm) + (j - sm)$; therefore we can obtain them all with only one computation of LF. Therefore, instead of finding $\mathsf{SA}[sp], \ldots, \mathsf{SA}[ep]$ one by one, we report $\mathsf{SA}[ep]$ (which we know) and cut $\mathsf{BWT}[sp..ep-1]$ into maximal runs using bitvector `Start`. Then, for each maximal run $\mathsf{BWT}[sm..em]$, if the end of run $\mathsf{BWT}[em]$ is sampled, we report its position and continue

Algorithm 2 Locating pattern occurrences on the *sr*-index.

Input : Global array $Res[1..occ]$ of results, range $[sp, ep]$ to report, $\mathsf{SA}[ep]$.
Output : Fills $Res[i] = \mathsf{SA}[sp-1+i]$ for all $1 \leq i \leq occ$.

```
1  Res[ep − sp + 1] ← SA[ep] (known from backward search)
2  if sp < ep then locate(sp, ep − 1, 0)
3  Proc locate(sm, em, k)
4      if k = s then
5          for im = em, ..., sm do
6              i ← Res[im − sp + 2] − 1
7              Res[im − sp + 1] ← φ(i) (Eq. (1))
8      else
9          if Start[em + 1] = 1 then
10             q ← rank_1(Start, em)
11             if Removed[q] = 0 then
12                 Res[em − sp + 1] ← Samples[q − rank_1(Removed, q)] + 1 + k
13                 em ← em − 1
14         q ← rank_1(Start, sm)
15         while sm ≤ em do
16             im ← select_1(Start, q + 1)
17             if im − 1 > em then im ← em + 1
18             locate(sm, im − 1, k + 1)
19             sm ← im
20             q ← q + 1
```

recursively reporting $\mathsf{SA}[\mathsf{LF}(sm)..\mathsf{LF}(sm)+(em-sm)-1]$; otherwise we continue recursively reporting $\mathsf{SA}[\mathsf{LF}(sm)..\mathsf{LF}(sm)+(em-sm)]$. Note that we must add k to the results reported at level k of the recursion. By Lemma 2, every end of run found in the way has been reported before level $k = s$. When $k = s$, then, we use Eq. (1) to obtain $\mathsf{SA}[em], \ldots, \mathsf{SA}[sm]$ consecutively from $\mathsf{SA}[em+1]$, which must have been reported because it is ep or was an end of run at some level of the recursion.

Algorithm 2 gives the complete procedure to solve Problem 2.

4.4 The basic index, *sr*-index$_0$

We have just described our most space-efficient index, which we call *sr*-index$_0$. Its space and time complexity is established in the next theorem.

▶ **Theorem 5.** *The* sr-*index*$_0$ *uses* $r\cdot((1+\epsilon)\lg(n/r)+\lg\sigma+\mathcal{O}(1))+\min(r, 2\lceil n/(s+1)\rceil)\cdot 2\lg n$ *bits of space, for any constant* $\epsilon > 0$*, and finds all the* occ *occurrences of* $P[1..m]$ *in* $\mathcal{T}$ *in time* $\mathcal{O}(\frac{1}{\epsilon}(m + s\cdot occ)\log(\sigma + n/r))$.

Proof. The space is the sum of the counting structures of the *r*-index and our modified locating structures, according to Lemma 1. The space of bitvector `Removed` is $\mathcal{O}(r)$ bits, which is accounted for in the formula.

As for the time, we have seen that the modified backward search requires $\mathcal{O}(m)$ steps if $occ = 0$ and $\mathcal{O}(m + s)$ otherwise (Problem 1). Each occurrence is then located in $\mathcal{O}(s)$ steps (Problem 2). In total, we complete the search with $\mathcal{O}(m + s\cdot occ)$ steps.

Each step involves $\mathcal{O}(\frac{1}{\epsilon}\log(\sigma + n/r))$ time in the basic *r*-index implementation, including Eq. (1). Our index includes additional `rank`s on `Start` and other constant-time operations, which are all in $\mathcal{O}(\log(n/r))$. Since the `First` now has $\mathcal{O}(\min(r, n/s))$ 1s, however, operation $\texttt{rank}_1$ on it takes time $\mathcal{O}(\log(n/\min(r, n/s))) = \mathcal{O}(\log\max(n/r, s)) = \mathcal{O}(\log(n/r + s))$. Yet, this `rank` is computed only once per occurrence reported, when using Eq. (1), so the total time per occurrence is still $\mathcal{O}(\log(n/r + s) + s \cdot \log(\sigma + n/r)) = \mathcal{O}(s \cdot \log(\sigma + n/r))$. ◀

Note that, in asymptotic terms, the *sr*-index is never worse than the rlfm-index with the same value of s and, with $s = 1$, it boils down to the *r*-index. Using predecessor data structures of the same asymptotic space of our lighter sparse bitvectors, the logarithmic times can be reduced to loglogarithmic [11], but our focus is on low practical space usage.

Note also that this theorem can be obtained by simply choosing the smallest between the *r*-index and the rlfm-index. In practice, however, the *sr*-index performs much better than both extremes, providing a smooth transition that retains sparsely indexed areas of $\mathcal{T}$ while removing redundancy in oversampled areas. This will be demonstrated in Section 5.

4.5 A faster and larger index, *sr*-index$_1$

The *sr*-index$_0$ guarantees locating time proportional to s and uses almost no extra space. On the other hand, on Problem 2 it performs up to s LF-steps for *every* occurrence, even when this turns out to be useless. The variant *sr*-index$_1$ adds a new component, also small, to speed up some cases:

`Valid`: a bitvector storing one bit per (remaining) sample in text order, so that $\texttt{Valid}[q] = 0$ iff there were removed samples between the qth and the $(q+1)$th 1s of `First`.

With this bitvector, if we have $i = \mathsf{SA}[j] - 1$ and $\texttt{Valid}[\texttt{rank}_1(\texttt{First}, i)] = 1$, we know that there were no removed samples between i and $\texttt{pred}(\texttt{First}, i)$ (even if they are less than s positions apart). In this case we can skip the computation of $\mathsf{LF}^k(j-1)$ of *sr*-index$_0$, and directly use Eq. (1). Otherwise, we must proceed exactly as in *sr*-index$_0$ (where it is still possible that we compute all the LF-steps unnecessarily). More precisely, this can be tested for every value between sm and em so as to report some further cells before recursing on the remaining ones, in lines 14–19 of Algorithm 2.

The space and worst-case complexities of Theorem 5 are preserved in *sr*-index$_1$.

4.6 Even faster and larger, *sr*-index$_2$

Our final variant, *sr*-index$_2$, adds a second and significantly larger structure:

`ValidArea`: an array whose cells are associated with the 0s in `Valid`. If $\texttt{Valid}[q] = 0$, then $d = \texttt{ValidArea}[q - \texttt{rank}_1(\texttt{Valid}, q)]$ is the distance from the qth 1 in `First` to the next removed sample. Each entry in `ValidArea` requires $\lceil \lg s \rceil$ bits, because removed samples must be at distance less than s from their preceding sample, by Lemma 2.

If $\texttt{Valid}[\texttt{rank}_1(\texttt{First}, i)] = 0$, then there was a removed sample at $\texttt{pred}(\texttt{First}, i) + d$, but not before. So, if $i < \texttt{pred}(\texttt{First}, i) + d$, we can still use Eq. (1); otherwise we must compute the LF-steps $\mathsf{LF}^k(j-1)$ and we are guaranteed to succeed in less than s steps. This improves performance considerably in practice, though the worst-case time complexity stays as in Theorem 5 and the space increases by at most $r \lg s$ bits.

5 Experimental Results

We implemented the *sr*-index in C++14, on top of the SDSL library[2], and made it available at `https://github.com/duscob/sr-index`.

We benchmarked the *sr*-index against available implementations for the *r*-index, the rlfm-index, and several other indexes for repetitive text collections.

Our experiments ran on a hardware with two Intel(R) Xeon(R) CPU E5-2407 processors at 2.40 GHz and 250 GB RAM. The operating system was Debian Linux kernel `4.9.0-14-amd64`. We compiled with full optimization and no multithreading.

Our reported times are the average user time over 1000 searches for patterns of length $m = 10$ obtained at random from the texts. We give space in bits per symbol (bps) and times in microseconds per occurrence (μs/occ). Indexes that could not be built on some collection, or that are out of scale in space or time, are omitted in the corresponding plots.

5.1 Tested indexes

We included the following indexes in our benchmark; their space decrease as s grows:

sr-index: Our index, including the three variants, with sampling values $s = 4, 8, 16, 32, 64$.
r-index: The *r*-index implementation we build on.[3]
rlcsa: An implementation of the run-length CSA [19], which outperforms the actual rlfm-index implementation.[4] We use text sampling values $s = n/r \times f/8$, with $f = 8, 10, 12, 14, 16$.
csa: An implementation of the CSA [28], which outperforms in practice the fm-index [8, 9]. This index, obtained from SDSL, acts as a control baseline that is not designed for repetitive collections.We use text sampling parameter $s = 16, 32, 64, 128$.
g-index: The best grammar-based index implementation we are aware of [5].[5] We use Patricia trees sampling values $s = 4, 16, 64$.
lz-index and lze-index: Two variants of the Lempel–Ziv based index [15].[6]
hyb-index: A hybrid between a Lempel–Ziv and a BWT-based index [7].[7] We build it with parameters $M = 8, 16$, the best for this case.

5.2 Collections

We benchmark various repetitive text collections; Table 1 gives some basic measures on them.

PizzaChili: A generic collection of real-life texts of various sorts and repetitiveness levels, which we use to obtain a general idea of how the indexes compare. We use 4 collections of microorganism genomes (influenza, cere, para, and escherichia) and 4 versioned document collections (the English version of einstein, kernel, worldleaders, coreutils).[8]
Synthetic DNA: A 100KB DNA text from PizzaChili, replicated 1,000 times and each copied symbol mutated with a probability from 0.001 (DNA-001, analogous to human assembled genomes) to 0.03 (DNA-030, analogous to sequence reads). We use this collection to study how the indexes evolve as repetitiveness decreases.

[2] From `https://github.com/simongog/sdsl-lite`.
[3] From `https://github.com/nicolaprezza/r-index`.
[4] From `https://github.com/adamnovak/rlcsa`.
[5] From `https://github.com/apachecom/grammar_improved_index`.
[6] From `https://github.com/migumar2/uiHRDC`.
[7] From `https://github.com/hferrada/HydridSelfIndex`.
[8] From `http://pizzachili.dcc.uchile.cl/repcorpus/real`.

Table 1 Basic characteristics of the repetitive texts used in our benchmark. Size is given in MB.

Collection	Size	n/r	Collection	Size	n/r
influenza	147.6	51.2	DNA-001	100.0	142.4
cere	439.9	39.9	DNA-003	100.0	58.3
para	409.4	27.4	DNA-010	100.0	26.0
escherichia	107.5	7.5	DNA-030	100.0	11.6
einstein	447.7	1611.2	HLA	53.7	161.4
kernel	238.0	92.4	Chr19	2,819.3	89.2
worldleaders	44.7	81.9	Salmonella	3,840.5	43.9
coreutils	195.8	43.8	Reads	2,565.5	8.9

Real DNA: Some real DNA collections to study other aspects:

HLA: A dataset with copies of the short arm (p arm) of human chromosome 6 [27].[9] This arm contains about 60 million base pairs (Mbp) and it includes the 3 Mbp HLA region. That region is known to be highly variable, so the *r*-index sampling should be sparse for most of the arm and oversample the HLA region.

Chr19 and Salmonella: Human and bacterial assembled genome collections, respectively, of a few billion base pairs. We include them to study how the indexes behave on more massive data. Chr19 is the set of 50 human chromosome 19 genomes taken from the 1000 Genomes Project [30], whereas Salmonella is the set of 815 Salmonella genomes from the GenomeTrakr project [29].

Reads: A large collection of sequence reads, which tend to be considerably less repetitive than assembled genomes.[10] We include this collection to study the behavior of the indexes on a popular kind of bioinformatic collection with mild repetitiveness. In Reads the sequencing errors have been corrected, and thus its $n/r \approx 9$ is higher than the $n/r \approx 4$ reported on crude reads [6].

5.3 Results

Figures 2 and 3 show the space taken by all the indexes and their search time.

A first conclusion is that *sr*-index$_2$ always dominates *sr*-index$_0$ and *sr*-index$_1$, so we will refer to it simply as *sr*-index from now on. The plots show that the extra information we associate to the samples makes a modest difference in space, while time improves considerably. This *sr*-index can be almost as fast as the *r*-index, and an order of magnitude faster than all the others, while using 1.5–4.0 less space than the *r*-index. Therefore, as promised, we are able to remove a significant degree of redundancy in the *r*-index without affecting its outstanding time performance.

In all the PizzaChili collections, the *sr*-index dominates almost every other index, outperforming them both in time and space. The only other index on the Pareto curve is the hyb-index, which can use as little as a half of the space of the sweet spot of the *sr*-index, but still at the price of being an order of magnitude slower. This holds even on escherichia, where n/r is well below 10, and both the rlcsa and the csa become closer to the *sr*-index.

In general, in all the collections with sufficient repetitiveness, say n/r over 25, the *sr*-index sharply dominates as described. As repetitiveness decreases, with n/r reaching around 10, the rlcsa and the csa approach the *sr*-index and outperform every other repetitiveness-aware index, as expected. This happens on escherichia (as mentioned) and Reads (where the *sr*-index,

[9] From `ftp://ftp.ebi.ac.uk/pub/databases/ipd/imgt/hla/fasta/hla_gen.fasta`.
[10] From `https://trace.ncbi.nlm.nih.gov/Traces/sra/?run=ERR008613`.

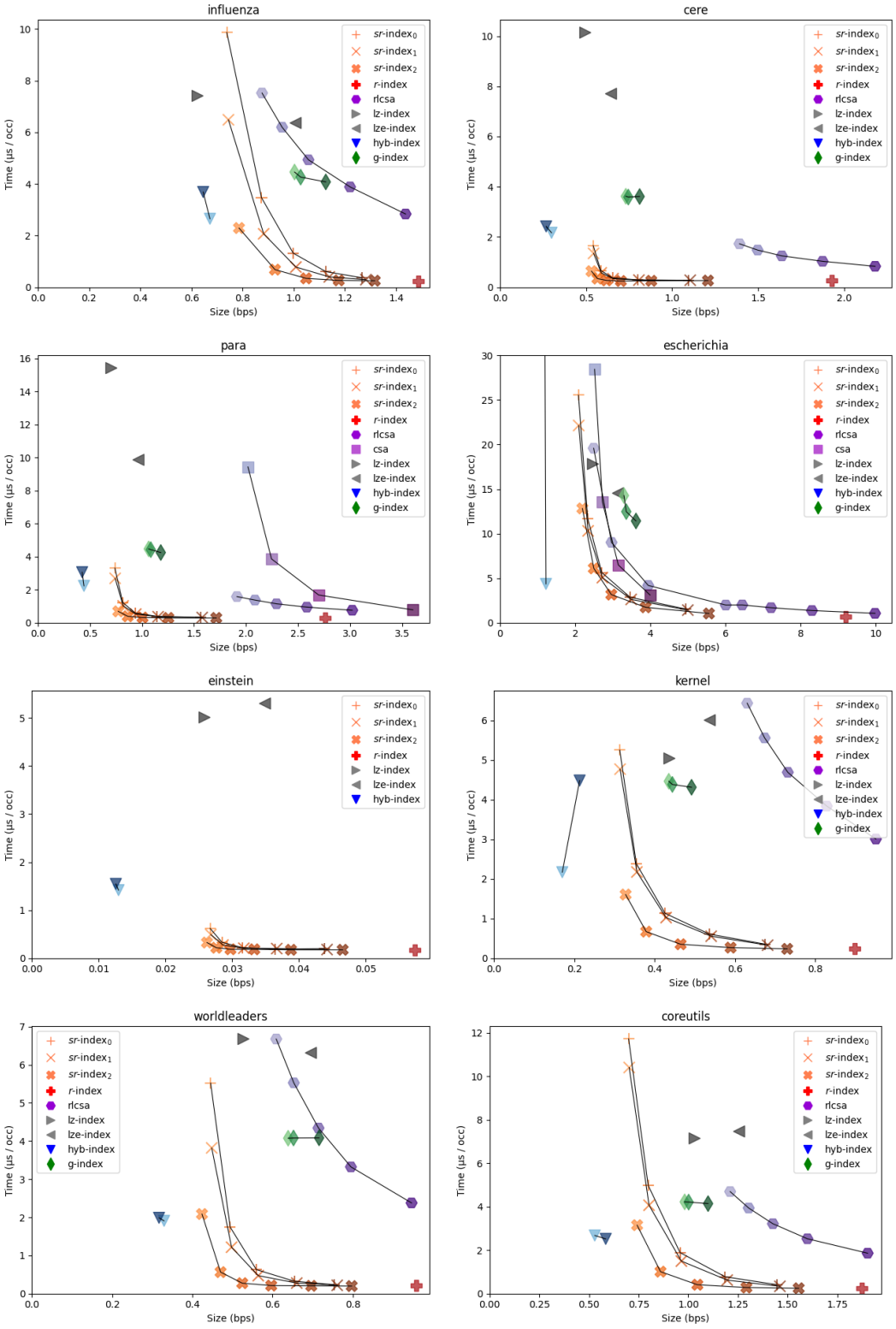


Figure 2 Space-time tradeoffs for the PizzaChili collections.

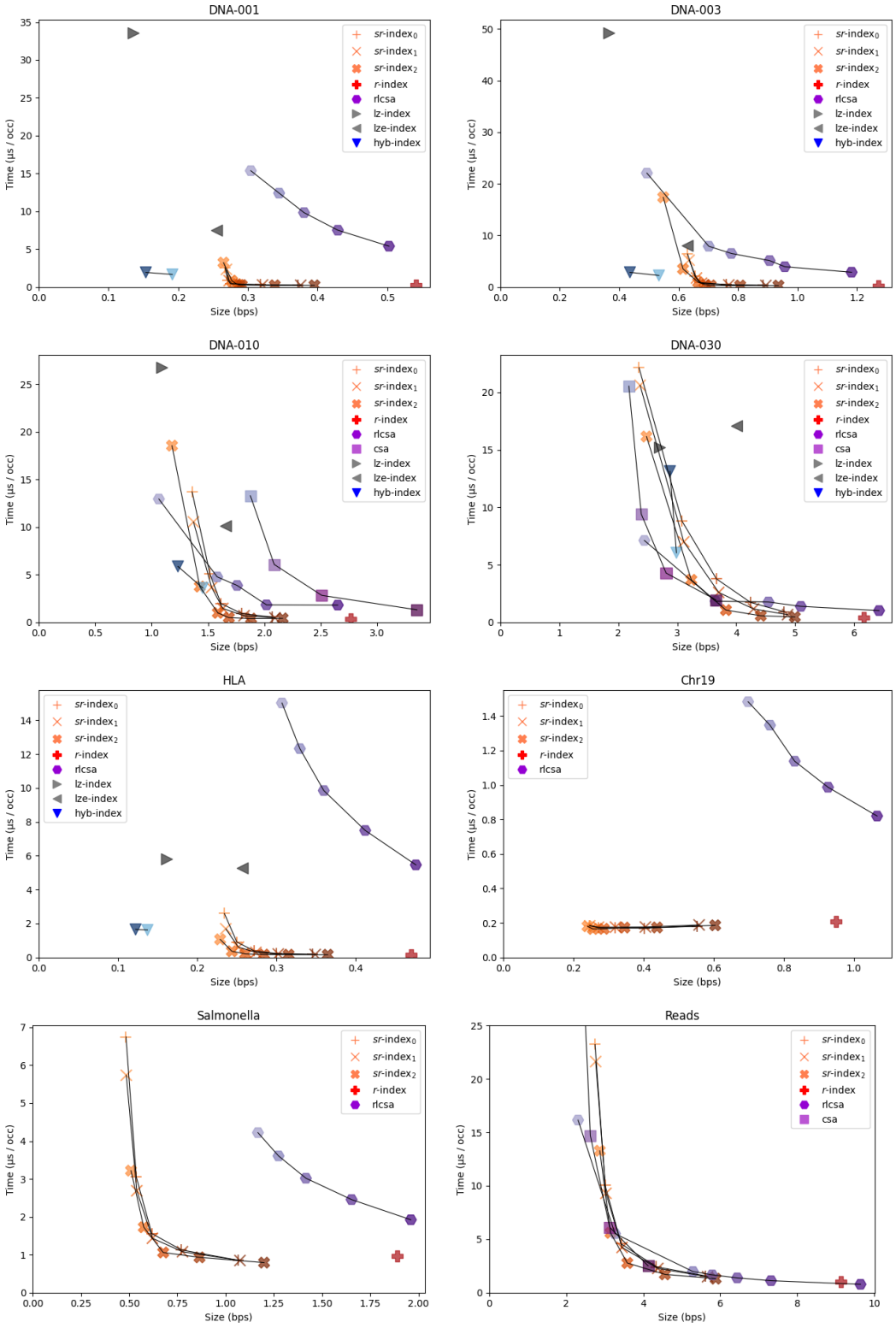


Figure 3 Space-time tradeoffs for the synthetic and real DNA datasets.

the rlcsa, and the csa behave similarly). This is also the case on the least repetitive synthetic DNA collection, DNA-030, where the mutation rate reaches 3%. In this collection, the repetitiveness-unaware csa largely dominates all the space-time map.

We expected the *sr*-index to have a bigger advantage over the *r*-index on the HLA dataset because its oversampling is concentrated, but the results are similar to those on randomly mutated DNA with about the same n/r value (DNA-001). In general, the bps used by the *sr*-index can be roughly predicted from n/r; for example the sweet spot often uses around $40r$ total bits, although it takes $20r$–$30r$ bits in some cases. The *r*-index uses $70r$–$90r$ bits.

The bigger collections (Chr19, Salmonella, Reads), on which we could build the BWT-related indexes only, show that the same observed trends scale to gigabyte-sized collections of various repetitiveness levels.

6 Conclusions

We have introduced the *sr*-index, an *r*-index variant that solves the problem of its relatively bloated space while retaining its high search performance. The *sr*-index is orders of magnitude faster than the other repetitiveness-aware indexes, while outperforming most of them in space as well. It matches the time performance of the *r*-index while using 1.5–4.0 less space.

Unlike the *r*-index, the *sr*-index uses little space even in milder repetitiveness scenarios, which makes it usable in a wider range of bioinformatic applications. For example, it uses 0.25–0.60 bits per symbol (bps) while reporting each occurrence within a microsecond on gigabyte-sized human and bacterial genomes, where the original *r*-index uses 0.95–1.90 bps. In general, the *sr*-index outperforms classic compressed indexes on collections with repetitiveness levels n/r over as little as 7 in some cases, though in general it is reached by repetitiveness-unaware indexes when n/r approaches 10, which is equivalent to a DNA mutation rate around 3%.

Compared to the rlfm-index, which for pattern searching is dominated by the *sr*-index, the former can use its regular text sampling to compute any entry of the suffix array or its inverse in time proportional to the sampling step s. Obtaining an analogous result on the *sr*-index, for example to implement compressed suffix trees, is still a challenge. Other proposals for accessing the suffix array faster than the rlfm-index [12, 26] illustrate this difficulty: they require even more space than the *r*-index.

References

1 Christina Boucher, Ondrej Cvacho, Travis Gagie, Jan Holub, Giovanni Manzini, Gonzalo Navarro, and Massimiliano Rossi. PFP compressed suffix trees. In *Proc. 23rd Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–72, 2021.

2 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14(1):13:1–13:15, 2019.

3 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

4 David R. Clark. *Compact PAT Trees.* PhD thesis, University of Waterloo, Canada, 1996.

5 Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 2021.

6 Diego Díaz-Domínguez and Gonzalo Navarro. A grammar compressor for collections of reads with applications to the construction of the BWT. In *Proc. 31st Data Compression Conference (DCC)*, pages 93–102, 2021.

7 Héctor Ferrada, Dominik Kempa, and Simon J. Puglisi. Hybrid indexing revisited. In *Proc. 20th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–8, 2018.

8 Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005.

9 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed Representations of Sequences and Full-text Indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

10 Travis Gagie and Gonzalo Navarro. *Compressed Indexes for Repetitive Textual Datasets*, pages 475–480. Springer, 2019.

11 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):article 2, 2020.

12 Rodrigo González, Gonzalo Navarro, and Héctor Ferrada. Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmics*, 19(1):article 1, 2014.

13 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows–Wheeler transform conjecture. In *Proc. 61st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1013, 2020.

14 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

15 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

16 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

17 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

18 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

19 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

20 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

21 Gonzalo Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

22 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

23 Gonzalo Navarro and Víctor Sepúlveda. Practical indexing of repetitive collections using Relative Lempel–Ziv. In *Proc. 29th Data Compression Conference (DCC)*, pages 201–210, 2019.

24 Takaaki Nishimoto and Yasuo Tabei. Faster queries on BWT-runs compressed indexes. *CoRR*, 2006.05104, 2020. `arXiv:2006.05104`.

25 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

26 Simon J. Puglisi and Bella Zhukova. Relative Lempel–Ziv compression of suffix arrays. In *Proc. 27th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 89–96, 2020.

27 James Robinson, Dominic J. Barker, Xenia Georgiou, Michael A. Cooper, Paul Flicek, and Steven G. E. Marsh. IPD-IMGT/HLA Database. *Nucleic Acids Research*, 48(D1):D948–D955, 10 2019.

28 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

29 Eric L. Stevens, Ruth Timme, Eric W. Brown, Marc W. Allard, Errol Strain, Kelly Bunning, and Steven Musser. The public health impact of a publically available, environmental database of microbial genomes. *Frontiers in Microbiology*, 8:808, 2017.

30 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.

The Longest Run Subsequence Problem: Further Complexity Results

Riccardo Dondi
Università degli Studi di Bergamo, Bergamo, Italy

Florian Sikora
Université Paris-Dauphine, PSL University, CNRS, LAMSADE, 75016 Paris, France

Abstract

Longest Run Subsequence is a problem introduced recently in the context of the scaffolding phase of genome assembly (Schrinner et al., WABI 2020). The problem asks for a maximum length subsequence of a given string that contains at most one run for each symbol (a run is a maximum substring of consecutive identical symbols). The problem has been shown to be NP-hard and to be fixed-parameter tractable when the parameter is the size of the alphabet on which the input string is defined. In this paper we further investigate the complexity of the problem and we show that it is fixed-parameter tractable when it is parameterized by the number of runs in a solution, a smaller parameter. Moreover, we investigate the kernelization complexity of Longest Run Subsequence and we prove that it does not admit a polynomial kernel when parameterized by the size of the alphabet or by the number of runs. Finally, we consider the restriction of Longest Run Subsequence when each symbol has at most two occurrences in the input string and we show that it is APX-hard.

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability; Theory of computation → Approximation algorithms analysis; Theory of computation → Graph algorithms analysis

Keywords and phrases Parameterized complexity, Kernelization, Approximation Hardness, Longest Subsequence

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.14

Funding *Florian Sikora*: Partially funded by ESIGMA (ANR-17-CE23-0010).

1 Introduction

A fundamental problem in computational genomics is genome assembly, whose goal is reconstructing a genome given a set of reads (a read is a sequence of base pairs) [2, 5]. After the generation of initial assemblies, called *contigs*, they have to be ordered correctly, in a phase called *scaffolding*. One of the commonly used approaches for scaffolding is to consider two (or more) incomplete assemblies of related samples, thus allowing the alignment of contigs based on their similarities [7]. However, the presence of genomic repeats and structural differences may lead to misleading connections between contigs.

Consider two sets X, Y of contigs, such that the order of contigs in Y has to be inferred using the contigs in X. Each contig in X is divided into equal size bins and each bin is mapped to a contig in Y (based on best matches). As a consequence, each bin in X can be partitioned based on the mapping to contigs of Y. However this mapping of bins to contigs, due to errors (in the sequencing or in the mapping process) or mutations, may present some inconsistencies, in particular bins can be mapped to scattered contigs, thus leading to an inconsistent partition of X, as shown in Fig. 1. In order to infer the most likely partition of X (and then distinguish between the transition from one contig to the other and errors in the mapping), the method proposed in [10] asks for a longest subsequence of the contig matches in X such that each contig run occurs at most once (see Fig. 1 for an example).

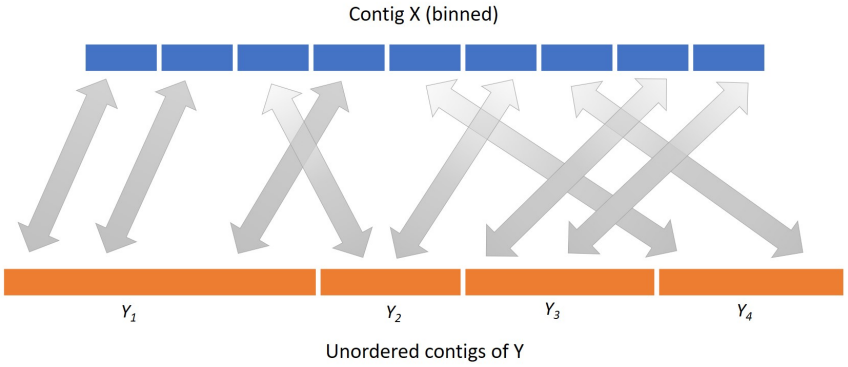


Figure 1 An example of matching a binned contig (X) with the unordered contigs of Y. The string inferred from this matching is $S = y_1\ y_1\ y_2\ y_1\ y_4\ y_2\ y_4\ y_3\ y_3$. Notice that S induces an inconsistent partition of the bins of X, for example for the mapping of Y_1 and Y_2. Indeed, Y_1 is mapped in the first, second and fourth bin of X, while Y_2 is mapped in the third and sixth bin of X. A longest run subsequence R of S is $R = y_1\ y_1\ y_1\ y_4\ y_4\ y_3\ y_3$, that induces a partition of some bins of X.

This problem, called Longest Run Subsequence, has been recently introduced and studied by Schrinner et al. [10]. Longest Run Subsequence has been shown to be NP-hard [10] and fixed-parameter tractable when the parameter is the size of the alphabet on which the input string is defined [10]. Furthermore, an integer linear program has been given for the problem [10]. Schrinner et al. let as future work approximability and parameterized complexity results on the problem [10]. Note that this problem could be seen as close to the "run-length encoded" string problems in the string literature, where a string is described as a sequence of symbols followed by the number of its consecutive occurrences, i.e. a sequence of runs where only its symbol and its length is stored (see for example [3]). While finding the longest common subsequence between two of such strings is a polynomial task, our problem is, to the best of our knowledge, not studied in literature before the work of Schrinner et al. [10].

In this paper we further investigate the complexity of the Longest Run Subsequence problem. We start in Section 2 by introducing some definitions and by giving the formal definition of the problem. Then in Section 3 we give a randomized fixed-parameter algorithm, where the parameter is the number of runs in a solution, based on the multilinear detection technique. In Section 4, we investigate the kernelization complexity of Longest Run Subsequence and we prove that it does not admit a polynomial kernel when parameterized by the size of the alphabet or by the number of runs. Notice that the problem admits a polynomial kernel when parameterized by the length of the solution (see Observation 6). Finally, in Section 5 we consider the restriction of Longest Run Subsequence when each symbol has at most two occurrences in the input string and we show that it is APX-hard. We conclude the paper with some open problems.

2 Definitions

In this section we introduce the main definitions we need in the rest of the paper.

Problem Definition. Given a string S, $|S|$ denotes the length of the string; $S[i]$, with $1 \leqslant i \leqslant |S|$, denotes the symbol of S in position i, $S[i, j]$, with $1 \leqslant i \leqslant j \leqslant |S|$, denotes the substring of S that starts in position i and ends in position j. Notice that if $i = j$, then $S[i, i]$ is the symbol $S[i]$. Given a symbol a, we denote by a^p, for some integer $p \geqslant 1$, a string consisting of the concatenation of p occurrences of symbol a.

A *run* in S is a substring $S[i,j]$, with $1 \leqslant i \leqslant j \leqslant |S|$, such that $S[z] = a$, for each $i \leqslant z \leqslant j$, with $a \in \Sigma$. Given $a \in \Sigma$, an a-run is a run in S consisting of repetitions of symbol a. Given a string S on alphabet Σ, a *run subsequence* S' of S is a subsequence that contains *at most one run* for each symbol $a \in \Sigma$.

Now, we are ready to define the Longest Run Subsequence problem.

Longest Run Subsequence
- **Input**: A string S on alphabet Σ, an integer k.
- **Output**: Does there exist a run subsequence R of length k?

A string $S[i,j]$ contains an a-run with $a \in \Sigma$, if it contains a run subsequence which is an a-run. A run subsequence of S which is an a-run, with $a \in \Sigma$, is maximal if it contains all the occurrences of symbol a in S. Note that an optimal solution may not take maximal runs. For example, consider

$S = abacaabbab$

an optimal run subsequence in S is

$R = aaaabbb$

Note that no run in R is maximal and even that some symbol of Σ may not be in an optimal solution of Longest Run Subsequence, in the example no c-run belongs to R.

Graph Definitions. Given a graph $G = (V, E)$, we denote by $N(v) = \{u : \{u, v\} \in E\}$, the neighbourhood of v. The closed neighbourhood of v is $N[v] = N(v) \cup \{v\}$. $V' \subset V$ is an independent set when $\{u, v\} \notin E$ for each $u, v \in V'$. We recall that a graph $G = (V, E)$ is *cubic* when $|N(v)| = 3$ for each $v \in V$.

Parameterized Complexity. A *parameterized problem* is a decision problem specified together with a *parameter*, that is, an integer k depending on the instance. A problem is *fixed-parameter tractable* (FPT for short) if it can be solved in time $f(k) \cdot |I|^c$ (often briefly referred to as FPT-*time*) for an instance I of size $|I|$ with parameter k, where f is a computable function and c is a constant. Given a parameterized problem P, a *kernel* is a polynomial-time computable function which associates with each instance of P an equivalent instance of P whose size is bounded by a function h of the parameter. When h is a polynomial, the kernel is said to be *polynomial*. See the book [6] for more details.

In order to prove that such polynomial kernel is unlikely, we need additional definitions and results.

▶ **Definition 1** (Cross-Composition [4]). *We say that a problem L* cross-composes *to a parameterized problem Q if there is a polynomial equivalence relation $\mathcal{R}$ and an algorithm which given t instances $x_1, x_2, \ldots, x_t$ of L belonging to the same equivalence class $\mathcal{R}$, computes an instance (x^*, k^*) of Q in time polynomial in $\sum_{i=1}^{t} |x_i|$ such that (i) $(x^*, k^*) \in Q \iff x_i \in L$ for some i and (ii) k^* is bounded by a polynomial in $(\max_i |x_i| + \log t)$.*

This definition is useful for the following result, which we will use to prove that a polynomial kernel for Longest Run Subsequence with parameter $|\Sigma|$ is unlikely.

▶ **Theorem 2** ([4]). *If an NP-hard problem L has a cross-composition into a parameterized problem Q and Q has a polynomial kernel, then NP $\subseteq$ coNP/poly.*

For our FPT algorithm, we will reduce our problem to another problem, called k-Multilinear Detection problem (k-MLD), which can be solved efficiently. In this problem, we are given a polynomial over a set of variables X, represented as an arithmetic circuit $\mathcal{C}$, and the question is to decide if this polynomial contains a multilinear term of degree exactly k. A polynomial is a sum of monomials. The *degree* of a monomial is the sum of its variables degrees and a monomial is *multilinear* if the degree of all its variables is equal to 1 (therefore, a multilinear monomial of degree k contains k different variables). For example, $x_1^2x_2 + x_1x_2x_3$ is a polynomial over 3 variables, both monomials are of degree 3 but only the second one is multilinear.

Note that the size of the polynomial could be exponentially large in $|X|$ and thus we cannot just check each monomial. We will therefore encode the polynomial in a compressed form: the circuit $\mathcal{C}$ is represented as a Directed Acyclic Graph (DAG), where leaves are variables X and internal nodes are multiplications or additions. The following result is fundamental for k-MLD.

▶ **Theorem 3** ([8, 11]). *There exists a randomized algorithm solving k-MLD in time $O(2^k|C|)$ and $O(|C|)$ space.*

Approximation. In Section 5, we prove the APX-hardness of Longest Run Subsequence with at most two occurrences for each symbol in Σ, by designing an L-reduction from Maximum Independent Set on cubic graphs. We recall here the definition of L-reduction. Notice that, given a solution S of a problem (A or B in the definition), we denote by $val(S)$ the value of S (for example, in our problem, the length of a run subsequence).

▶ **Definition 4** (L-reduction [9]). *Let A and B be two optimization problems. Then A is said to be L-reducible to B if there are two constants $\alpha, \beta > 0$ and two polynomial-time computable functions f, g such that: (i) f maps an instance I of A into an instance I' of B such that $opt_B(I') \leqslant \alpha \cdot opt_A(I)$, (ii) g maps each solution S' of I' into a solution S of I such that $|val(S) - opt_A(I)| \leqslant \beta \cdot |val(S') - opt_B(I')|$.*

L-reductions are useful in order to apply the following theorem.

▶ **Theorem 5** ([9]). *Let A and B be two optimization problems. If A is L-reducible to B and B has a PTAS, then A has a PTAS.*

Parameterized Complexity Status of the Problem

In the paper, we consider the parameterized complexity of Longest Run Subsequence under the different parameterizations. We consider the following parameters:

- The length k of the solution of Longest Run Subsequence
- The size $|\Sigma|$ of the alphabet
- The number r of runs in a solution of Longest Run Subsequence

Notice that $r \leqslant |\Sigma| \leqslant k$. Indeed, there always exists a solution consisting of one occurrence for each symbol in Σ, hence we can assume that $|\Sigma| \leqslant k$. Clearly, $r \leqslant |\Sigma|$, since each run in a solution of Longest Run Subsequence is associated with a distinct symbol of Σ.

In Table 1, we present the status of the parameterized complexity of Longest Run Subsequence for these parameters.

It is easy to see that Longest Run Subsequence has a polynomial kernel for parameter k.

Table 1 Parameterized complexity status for the three different parameters considered in this paper. Since these parameters are in decreasing value order, note that positive results propagate upwards, while negative results propagate downwards. In bold the new results we present in this paper.

	FPT	Poly Kernel
k	Yes (Obs. 6)	**Yes** (Obs. 6)
$\|\Sigma\|$	Yes [10]	**No** (Th. 9)
r	**Yes & Poly Space** (Th. 8)	**No** (Cor. 10)

▶ **Observation 6.** Longest Run Subsequence *has a* k^2 *kernel.*

Proof. First, notice that if there exists an a-run R' of length at least k, for some $a \in \Sigma$, then R' is a solution of Longest Run Subsequence. Also note that if $|\Sigma| \geqslant k$, let R^+ be a subsequence of S consisting of one occurrence of each symbol of Σ (notice that it is always possible to define such a solution). Then R^+ is a solution of Longest Run Subsequence of sufficient size.

Therefore, we can assume that S is defined over an alphabet $|\Sigma| < k$ and that each symbol has less then k occurrences (otherwise there exists an a-run of length at least k for some $a \in \Sigma$). Hence Longest Run Subsequence has a kernel of size k^2. ◀

Schrinner et al. prove that Longest Run Subsequence is in FPT for parameter $|\Sigma|$, using exponential space [10]. Due to a folklore result [6], this also implies that there is a kernel for this parameter. We will prove that there is no polynomial kernel for this parameter in Section 4.

3 An FPT Algorithm for Parameter Number of Runs

In this section, we consider Longest Run Subsequence when parameterized by the number of different runs, denoted by r, in the solution, that is whether there exists a solution of Longest Run Subsequence consisting of exactly r runs such that it has length at least k. We present a randomized fixed-parameter algorithm for Longest Run Subsequence based on multilinear monomial detection.

The algorithm we present is for a variant of Longest Run Subsequence that asks for a run subsequence R of S such that (1) $|R| = k$ and (2) R contains exactly r runs. In order to solve the general problem where we only ask for a solution of length at least k, we need to apply the algorithm for each k, with $r \leqslant k \leqslant |S|$.

Now, we describe the circuit on which our algorithm is based on. The set of variables is:

$$\{x_a : a \in \Sigma\}$$

Essentially, x_a represents the fact that we take an a-run (not necessarily maximal) in a substring of S.

Define a circuit $\mathcal{C}$ as follows. It has a root P and a set of intermediate vertices $P_{i,l,h}$, with $1 \leqslant i \leqslant |S|$, $1 \leqslant l \leqslant r$ and $1 \leqslant h \leqslant k$. The multilinear monomials of $P_{i,l,h}$ informally encode a run subsequence of $S[1, i]$ having length h and consisting of l runs. $P_{i,l,h}$ is recursively defined as follows:

$$P_{i,l,h} = \begin{cases} P_{i-1,l,h} + \sum_{j:1\leqslant j\leqslant i-1} P_{j,l-1,h-z}x_a & \text{if } i \geqslant 1, l \geqslant 1, h \geqslant 1, \\ & 1 \leqslant z \leqslant i-j-1, S[i] = a,\ a \in \Sigma, \\ & \text{and } S[j+1,i] \text{ contains an } a\text{-run} \\ & \text{of length } z \\ 1 & \text{if } i \geqslant 0, l = h = 0, \\ 0 & \text{if } i = 0, l > 0 \text{ or } h > 0. \end{cases} \quad (1)$$

Then, define $P = P_{|S|,r,k}$.

Next, we show that we can consider the circuit $\mathcal{C}$ to compute a run subsequence of S.

▶ **Lemma 7.** *There exists a run subsequence of S of length h consisting of l runs over symbols $a_1, \dots a_l$ if and only if there exists a multilinear monomial in $\mathcal{C}$ consisting of l monomials $x_{a_1}, \dots, x_{a_l}$.*

Proof. We will prove that there is a run subsequence of S of length k consisting of l runs over symbols $a_1, \dots a_l$ if and only if there exists a multilinear monomial in $\mathcal{C}$ of degree l, consisting of l distinct variables $x_{a_1}, \dots, x_{a_l}$. In order to prove this result, we prove by induction on i , $1 \leqslant i \leqslant |S|$, that there exists a run subsequence R of $S[1...i]$, such that $|R| = h$ and R contains l runs, an a_z-run for each $a_z \in \Sigma$, $1 \leqslant z \leqslant l$, if and only if there exists a multilinear monomial $x_{a_1} \dots x_{a_l}$ in $P_{i,l,h}$.

We start with the case $i = 1$. Assume that there is a run subsequence consisting of a single run of length 1 (say an a_1-run). It follows that $S[1] = a_1$ and, by Equation 1, $P_{1,1,1} = P_{0,0,0} \cdot x_{a_1} = x_{a_1}$. Conversely, if $P_{1,1,1} = P_{0,0,0} \cdot x_{a_1} = x_{a_1}$, then by construction $S[1] = a_1$, which is a run of length 1.

Assume that the lemma holds for $j < i$, we prove that it holds for i.

($\Rightarrow$) Assume that there exists a run subsequence R of $S[1,i]$ that consists of l runs and that has length h. Let the l runs in R be over symbols $a_1, \dots, a_l$ and assume that the rightmost run in R is an a_l-run. If $S[i]$ does not belong to the a_l-run in R, then R is a run subsequence in $S[1,i-1]$ and by induction hypothesis $P_{i-1,l,h}$ contains a multilinear monomial of degree l over variables $x_{a_1} \dots x_{a_l}$. If $S[i]$ belongs to the a_l-run in R, then consider the a_l-run in R and assume that it belongs to substring $S[j+1,i]$ of S, with $1 \leqslant j+1 \leqslant i$, and that it has length z. Consider the run subsequence R' of S obtained from R by removing the a_l-run. Then, R' is a run subsequence of $S[1,j]$ that does not contain a_l (hence it contains $l-1$ runs) and has length $h-z$. By induction hypothesis, $P_{j,l-1,h-z}$ contains a multilinear monomial of length $l-1$ over variables $x_{a_1} \dots x_{a_{l-1}}$. Hence by the first case of Equation 1, it follows that $P_{i,l,h}$ contains a multilinear monomial of length l over variables $x_{a_1} \dots x_{a_l}$.

($\Leftarrow$) Assume that $P_{i,l,h}$ contains a multilinear monomial of length l over variables $x_{a_1} \dots x_{a_l}$, we will prove that there is a run subsequence of S of length k consisting of l runs. By Equation 1, it follows that (1) $P_{i-1,l,h}$ contains a multilinear monomial of length l over variables $x_{a_1} \dots x_{a_l}$ or (2) $P_{j,l-1,h-z}$, for some $1 \leqslant j \leqslant i-1$, contains a multilinear monomial of length $l-1$ that does not contain one of $x_{a_1} \dots x_{a_l}$ (without loss of generality x_{a_l}) and $S[j+1,i]$ contains an a_l-run of length z.

In case (1), by induction hypothesis there exists a run subsequence in $S[1,i-1]$ (hence also in $S[1,i]$) of length h consisting of l runs over symbols $a_1, \dots, a_l$.

In case (2), by induction hypothesis there exists a run subsequence R' of $S[1,j]$ of length $h-z$ consisting of $l-1$ runs over symbols $a_1, \dots, a_{l-1}$. Now, by concatenating R' with the a_l-run of length z in $S[j+1,i]$, we obtain a run subsequence of R of $S[1,i]$ consisting of l runs and having length h. ◀

▶ **Theorem 8.** Longest Run Subsequence *can be solved by a randomized algorithm in* $O(2^r r|S|^3)$ *time and polynomial space.*

Proof. The correctness of the randomized algorithm follows by Lemma 7.

We compute P in polynomial time and we decide if $P_{|S|,r,k}$ contains a multilinear monomial of degree r in $O(2^r r|S|^2)$ time and polynomial space. The result follows from Lemma 7, Theorem 3, and from the observation that $|\mathcal{C}| = |S| \cdot l \cdot r$, with $l \leqslant |S|$. Finally, we have to iterate the algorithm for each k, with $r \leqslant k \leqslant |S|$, thus the overall time complexity is $O(2^r r|S|^3)$. ◀

4 Hardness of Kernelization

As discussed in Section 2, Longest Run Subsequence has a trivial polynomial kernel for parameter k and its FPT status implies an (exponential) kernel for parameters $|\Sigma|$ and r. In the following, we will prove that it is unlikely that Longest Run Subsequence admits a polynomial kernel for parameter $|\Sigma|$ and parameter r.

▶ **Theorem 9.** Longest Run Subsequence *does not admit a polynomial kernel for parameter* $|\Sigma|$*, unless NP* $\subseteq$ *coNP/poly.*

Proof. We will define an OR-cross-composition (see Definition 1) from the Longest Run Subsequence problem itself, whose unparameterized version is NP-Complete [10].

Consider t instances $(S_1, \Sigma_1, k_1), (S_2, \Sigma_2, k_2), \ldots, (S_t, \Sigma_t, k_t)$ of Longest Run Subsequence, where, for each i with $1 \leqslant i \leqslant t$, S_i is the input string built over the alphabet Σ_i, and $k_i \in \mathbb{N}$ is the length of the solution, respectively. We will define an equivalence relation $\mathcal{R}$ such that strings that are not encoding valid instances are equivalent, and two valid instances $(S_i, \Sigma_i, k_i), (S_j, \Sigma_j, k_j)$ are equivalent if and only if $|S_i| = |S_j|$, $|\Sigma_i| = |\Sigma_j|$, and $k_i = k_j$. We now assume that $|S_i| = n$, $|\Sigma_i| = m$ and $k_i = k$ for all $1 \leqslant i \leqslant t$.

We will build an instance of Longest Run Subsequence (S', k', Σ') where S' is a string built over the alphabet Σ' and k' an integer such that there is a solution of size at least k' for S' iff there is an i, $1 \leqslant i \leqslant t$ such that there is a solution of size at least k in S_i.

We first show how to redefine the input strings S_1, S_2, ..., S_t, such that they are all over the same alphabet. Notice that this will not be an issue, since we will construct a string S' such that a solution of Longest Run Subsequence is not spanning over two different input strings. For all instances (S_i, Σ_i, k_i), $1 \leqslant i \leqslant t$, we consider any ordering of the symbols in Σ_i and we define a string $\sigma(S_i)$ starting from S_i, by replacing the j-th, $1 \leqslant j \leqslant m$, symbol of Σ_i by j, that is its position in the ordering of Σ_i. That way, it is clear that all strings $\sigma(S_i)$, $1 \leqslant i \leqslant t$, are built over the same alphabet $\{1, 2, \ldots, m\}$.

Now, the instance (S', k', Σ') of Longest Run Subsequence is build as follows. First, Σ' is defined as follows:

$$\Sigma' = \{1, 2, \ldots, m\} \cup \{\#, \$\}$$

where # and \$ are two symbols not in Σ.

The string S' is defined as follows:

$$S' = \$^{2n}\sigma(S_1)\#^{2n}\$^{2n}\sigma(S_2)\#^{2n}, ..., \$^{2n}\sigma(S_t)\#^{2n}$$

where $\$^{2n}$ ($\#^{2n}$, respectively) is a string consisting of the repetition $2n$ times of the symbol \$ (#, respectively).

Finally,

$$k' = k + (t+1)(2n).$$

Since we are applying OR-cross-composition for parameter $|\Sigma|$, we need to show that property (ii) of Definition 1 holds. By construction, we see that $|\Sigma'| = m + 2$, which is independent of n and t and it is bounded by the size of the largest input instance.

We now show that S' contains a run subsequence of size at least k' if and only if there exists at least one string S_i, $1 \leqslant i \leqslant t$, that contains a run subsequence of size at least k.

(⇐) First, assume that some S_i, with $1 \leqslant i \leqslant t$, contains a run subsequence R_i of length at least k. Then, define the following run subsequence R' of S', obtained by concatenating these substrings of S':

- The concatenation of the leftmost i-th substrings $\$^{2n}$ of S',
- The substring $\sigma(R_i)$ of S_i,
- The concatenation of the rightmost $(t-i+1)$-th substrings $\#^{2n}$ of S'.

It follows that S' contains a run subsequence of length at least $i\times(2n)+k+(t-i+1)\times(2n) = k'$.

(⇒) Conversely, assume now that S' contains a run subsequence R' of length at least k'. First, we prove that R' contains exactly one \$-run and one #-run. Indeed, if it is not the case, we can add the leftmost (the rightmost, respectively) substring $\$^{2n}$ ($\#^{2n}$, respectively) as a run of R'.

Consider a run r in R', which is either the \$-run or the #-run of R'. Assume that R' contains a substring

$$rR'(S_i)R'(S_j)$$

or a substring

$$R'(S_i)R'(S_j)\ r$$

with $1 \leqslant i < j \leqslant t$, where $R'(S_i)$ ($R'(S_j)$, respectively) is a substring of $\sigma(S_i)$ (of $\sigma(S_j)$, respectively). We consider without loss of generality the case that $rR'(S_i)R'(S_j)$ is a substring of R'. Then, we can modify R', increasing its length, as follows: we remove $R'(S_i)$ and extend the run r with a string $\$^{2n}$ or a string $\#^{2n}$ (depending on the fact that r is a \$-run or a #-run, respectively) that is between $\sigma(S_i)$ and $\sigma(S_j)$. The size of R' is increased, since $|R'(S_i)| \leqslant n$.

Now, assume that R' contains a substring

$$r = \#^{2n}\ R'(S_i)\ \$^{2n}$$

where $R'(S_i)$ is a substring of $\sigma(S_i)$. We can replace r with the substring $\#^{2n}\#^{2n}\$^{2n}$, where $\#^{2n}$ is the substring between $\sigma(S_i)$ and $\sigma(S_{i+1})$ in S'. Again, the size of R' is increased, since $|R'(S_i)| \leqslant n$.

By iterating these modifications on R', we obtain that R' is one of the following string:

1. A prefix $\$^{j(2n)}$ concatenated with a substring $R'(S_j)$ of $\sigma(S_j)$, for some $1 \leqslant j \leqslant t$, concatenated with a suffix $\#^{2n(t-j+1)}$
2. A substring $R'(S_1)$ of $\sigma(S_1)$ concatenated with a string of $\#^{2nj}$ concatenated with a string of $\$^{2n(t-j)}$ concatenated with a substring $R'(S_t)$ of $\sigma(S_t)$.

Notice that in this second case, it holds that $|R'| = (t \times 2n) + |R'(S_1)| + |R'(S_t)| < k'$, since $|R'(S_1)| + |R'(S_t)| \leqslant 2n$, and we can assume that $k \geqslant 1$. Hence R' must be a string described at point 1. It follows that

$$|R'| = 2n(t+1) + |R'(S_j)|$$

Since $|R'| = 2n(t+1) + |R'(S_j)|$, then $R'(S_j)$ has length at least k.

We have described an OR-cross-composition of Longest Run Subsequence to itself. By Theorem 2, it follows that Longest Run Subsequence does not admit a polynomial kernel for parameters $|\Sigma|$, unless NP $\subseteq$ coNP/poly. ◀

We can complement the FPT algorithm of Section 3, with a hardness of kernelization for the same parameter.

▶ **Corollary 10.** Longest Run Subsequence *does not admit a polynomial kernel for parameters r, unless NP $\subseteq$ coNP/poly.*

Proof. The result follows from Theorem 9 and from the fact that $r \leqslant |\Sigma|$. ◀

5 APX-hardness for Bounded Number of Occurrences

In this section, we show that Longest Run Subsequence is hard even when the number of occurrences of a symbol in the input string is bounded by two. We denote this restriction of the problem by 2-Longest Run Subsequence. Notice that if the number of occurrences of a symbol is bounded by one, then the problem is trivial, as a solution of Longest Run Subsequence can have only runs of length one.

We prove the result by giving a reduction from the Maximum Independent Set problem on Cubic Graphs (MISC), which is known to be APX-hard [1]. We recall the definition of MISC:

Maximum Independent Set problem on Cubic Graphs (MISC)
- **Input**: A cubic graph $G = (V, E)$.
- **Output**: Does there exist an independent set in G of size at least q?

Given a cubic graph $G = (V, E)$, with $V = \{v_1, \dots v_n\}$ and $|E| = m$, we construct a corresponding instance S of 2-Longest Run Subsequence (see Fig. 2 for an example of our construction). First, we define the alphabet Σ:

$$\Sigma = \{w_i : 1 \leqslant i \leqslant n\} \cup \{x^i_{i,j}, x^j_{i,j}, e^1_{i,j}, e^2_{i,j} : \{v_i, v_j\} \in E, i < j\} \cup \{\sharp_{i,z} : 1 \leqslant i \leqslant m+n, 1 \leqslant z \leqslant 3\}$$

This alphabet is of size $n + 4m + 3(m+n) = 4n + 7m$.

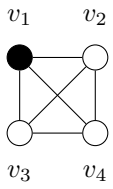
Now, we define a set of substrings of the instance S of 2-Longest Run Subsequence that we are constructing.

- For each $v_i \in V$, $1 \leqslant i \leqslant n$, such that v_i is adjacent to v_j, v_h, v_z, $1 \leqslant j < h < z \leqslant n$, we define a substring $S(v_i)$:

  $$S(v_i) = w_i x^i_{i,j} x^i_{i,h} x^i_{i,z} w_i$$

  Notice that in the definition of $S(v_i)$ given above, we have assumed without loss of generality that $1 \leqslant i < j < h < z \leqslant n$. If, for example, $1 \leqslant j < i < h < z \leqslant n$, the symbol associated with $\{v_i, v_j\}$ is then $x^i_{j,i}$ and $S(v_i)$ is defined as follows:

  $$S(v_i) = w_i x^i_{j,i} x^i_{i,h} x^i_{i,z} w_i$$



$$
\begin{aligned}
S = {} & w_1 x^1_{1,2} x^1_{1,3} x^1_{1,4} w_1 \sharp_{1,1} \sharp_{1,2} \sharp_{1,3} \\
& w_2 x^2_{1,2} x^2_{2,3} x^2_{2,4} w_2 \sharp_{2,1} \sharp_{2,2} \sharp_{2,3} \\
& w_3 x^3_{1,3} x^3_{2,3} x^3_{3,4} w_3 \sharp_{3,1} \sharp_{3,2} \sharp_{3,3} \\
& w_4 x^4_{1,4} x^4_{2,4} x^4_{3,4} w_4 \sharp_{4,1} \sharp_{4,2} \sharp_{4,3} \\
& e^1_{1,2} x^1_{1,2} e^2_{1,2} e^1_{1,2} x^2_{1,2} e^2_{1,2} \sharp_{5,1} \sharp_{5,2} \sharp_{5,3} \\
& e^1_{1,3} x^1_{1,3} e^2_{1,3} e^1_{1,3} x^3_{1,3} e^2_{1,3} \sharp_{6,1} \sharp_{6,2} \sharp_{6,3} \\
& e^1_{1,4} x^1_{1,4} e^2_{1,4} e^1_{1,4} x^4_{1,4} e^2_{1,4} \sharp_{7,1} \sharp_{7,2} \sharp_{7,3} \\
& e^1_{2,3} x^2_{2,3} e^2_{2,3} e^1_{2,3} x^3_{2,3} e^2_{2,3} \sharp_{8,1} \sharp_{8,2} \sharp_{8,3} \\
& e^1_{2,4} x^2_{2,4} e^2_{2,4} e^1_{2,4} x^4_{2,4} e^2_{2,4} \sharp_{9,1} \sharp_{9,2} \sharp_{9,3} \\
& e^1_{3,4} x^3_{3,4} e^2_{3,4} e^1_{3,4} x^4_{3,4} e^2_{3,4} \sharp_{10,1} \sharp_{10,2} \sharp_{10,3}
\end{aligned}
$$

Figure 2 A sample cubic graph (with 4 nodes and 6 edges) and the associated sequence. Black vertex of the graph corresponds to an independent set (of size 1 here), red symbols in the sequence correspond to the subsequence (of size $5 \cdot 1 + 4 \cdot 3 + 3 \cdot 6 + 3 \cdot 10 = 65$).

- For each edge $\{v_i, v_j\} \in E$, with $1 \leqslant i < j \leqslant n$, we define a substring $S(e_{ij})$:

  $$S(e_{ij}) = e^1_{i,j} x^i_{i,j} e^2_{i,j} e^1_{i,j} x^j_{i,j} e^2_{i,j}$$

- We define separation substrings $S_{Sep,i}$, with $1 \leqslant i \leqslant m + n$:

  $$S_{Sep,i} = \sharp_{i,1} \sharp_{i,2} \sharp_{i,3}$$

Now, given the lexical ordering[1] of the edges of G, the input string S is defined as follows (we assume that $\{v_1, v_z\}$ is the first edge and $\{v_p, v_t\}$ is the last edge in the lexicographic ordering of E):

$$S = S(v_1) S_{Sep,1} S(v_2) S_{Sep,2} \dots S(v_n) S_{Sep,n} S(e_{1,z}) S_{Sep,n+1} \dots S(e_{p,t}) S_{Sep,n+m}$$

Now, we prove some properties on the string S.

▶ **Lemma 11.** *Let $G = (V, E)$ be an instance of* MISC *and let S be the corresponding built instance of* 2-Longest Run Subsequence. *Then S contains at most two occurrences for each symbol of Σ.*

Proof. Notice that each symbol w_i, $1 \leqslant i \leqslant n$, appears only in substring $S(v_i)$ of S. Symbols $e^1_{i,j}$, $e^2_{i,j}$, with $\{v_i, v_j\} \in E$ and $1 \leqslant i < j \leqslant n$, appear only in substring $S(e_{i,j})$ of S. Each symbol $\sharp_{i,z}$, with $1 \leqslant i \leqslant m+n$ and $1 \leqslant z \leqslant 3$, appears only in substring $S_{Sep,i}$ of S. Finally, each symbol $x^i_{i,j}$, with $\{v_i, v_j\} \in E$, appears once in exactly two substrings of S, namely $S(v_i)$ and $S(e_{i,j})$. ◀

Now, we prove a property of solutions of 2-Longest Run Subsequence relative to separation substrings.

▶ **Lemma 12.** *Let $G = (V, E)$ be an instance of* MISC *and let S be the corresponding instance of* 2-Longest Run Subsequence. *Given a run subsequence R of S, if R does not contain some separation substring $S_{Sep,i}$, with $1 \leqslant i \leqslant m + n$, then there exists a run subsequence R' of S that contains $S_{Sep,i}$ and such that $|R'| > |R|$.*

[1] $\{v_i, v_j\} < \{v_h, v_z\}$ (assuming $i < j$ and $h < z$) if and only if $i < h$ or $i = h$ and $j < z$.

Proof. Notice that, since R does not contain substring $S_{Sep,i}$, with $1 \leqslant i \leqslant m+n$, it must contain a run r that connects two symbols that are on the left and on the right of $S_{Sep,i}$ in S, otherwise $S_{Sep,i}$ can be added to R increasing its length. Since each symbol in S, hence also in R, has at most two occurrences (see Lemma 11), then $|r| = 2$. Then, starting from R, we can compute in polynomial time a run subsequence R' by removing run r and by adding substring $S_{Sep,i}$. Notice that, after the removal of r, we can add $S_{Sep,i}$ since it contains three symbols each one having a single occurrence in S. Since $|S_{Sep,i}| = 3$, it follows that $|S_{Sep,i}| > r$ and $|R'| > |R|$. ◀

Given a cubic graph $G = (V, E)$ and the corresponding instance S of 2-Longest Run Subsequence, a run subsequence R of 2-Longest Run Subsequence on instance S is called *canonical* if:

- for each $S_{Sep,i}$, $1 \leqslant i \leqslant m+n$, R contains $S_{Sep,i}$ (a substring denoted by $R_{Sep,i}$)
- for each $S(v_i)$, with $v_i \in V$, R contains a substring $R(v_i)$ such that either $R(v_i) = w_i w_i$ or it is a substring of length 4 ($w_i x^i_{i,j} x^i_{i,h} x^i_{i,z}$ or $x^i_{i,j} x^i_{i,h} x^i_{i,z} w_i$); moreover if $\{v_i, v_j\} \in E$, then at least one of $R(v_i)$ or $R(v_j)$ has length 4
- for each $S(e_{i,j})$, with $\{v_i, v_j\} \in E$, R contains a substring $R(e_{i,j})$ such that $R(e_{i,j})$ is either of length 4 ($e^1_{i,j} x^i_{i,j} e^2_{i,j} e^2_{i,j}$ or $e^1_{i,j} e^1_{i,j} x^j_{i,j} e^2_{i,j}$), if one of $R(v_i)$, $R(v_j)$ has length 2, or of length 3 ($e^1_{i,j} e^1_{i,j} e^2_{i,j}$ or $e^1_{i,j} e^2_{i,j} e^2_{i,j}$).

▶ **Lemma 13.** *Let $G = (V, E)$ be an instance of* MISC *and let S be the corresponding instance of* 2-Longest Run Subsequence. *Given a run subsequence R of S, we can compute in polynomial time a canonical run subsequence of S of length at least $|R|$.*

Proof. Consider a run subsequence R of S. First, notice that by Lemma 12 we assume that R contains each symbol $\sharp_{i,p}$, with $1 \leqslant i \leqslant n+m$ and $1 \leqslant p \leqslant 3$. We start by proving some bounds on the run subsequence of $S(v_i)$ and $S(e_{i,j})$.

Consider a substring $R(v_i)$ of $S(v_i)$, $1 \leqslant i \leqslant n$. Each run subsequence of $S(v_i)$ can have length at most 4, since $|S(v_i)| = 5$ and if run $w_i w_i$ belongs to $R(v_i)$, then $R(v_i) = w_i w_i$. It follows that if $|R(v_i)| > 2$, then it cannot contain the two occurrences of symbol w_i. Notice that the two possible run subsequences of length 4 of $S(v_i)$ are $x^i_{i,j} x^i_{i,h} x^i_{i,z} w_i$ and $w_i x^i_{i,j} x^i_{i,h} x^i_{i,z}$.

Consider a run subsequence $R(e_{i,j})$ of $S(e_{ij}) = e^1_{i,j} x^i_{i,j} e^2_{i,j} e^1_{i,j} x^j_{i,j} e^2_{i,j}$. First, we prove that a run subsequence of $S(e_{ij})$ has length at most 4 and in this case it must contain at least one of $x^i_{i,j}$, $x^j_{i,j}$. By its interleaved construction, at most one of runs $e^1_{i,j} e^1_{i,j}$, $e^2_{i,j} e^2_{i,j}$ can belong to $R(e_{i,j})$. Moreover if $e^1_{i,j} e^1_{i,j}$ ($e^2_{i,j} e^2_{i,j}$, respectively) belongs to $R(e_{i,j})$, then $|R(e_{i,j})| \leqslant 4$, since the longest run in $S(e_{i,j})$ is then $e^1_{i,j} e^1_{i,j} x^j_{i,j} e^2_{i,j}$ ($e^1_{i,j} x^i_{i,j} e^2_{i,j} e^2_{i,j}$, respectively). If none of runs $e^1_{i,j} e^1_{i,j}$, $e^2_{i,j} e^2_{i,j}$ belongs to $R(e_{i,j})$, then $|R(e_{i,j})| \leqslant 4$, since $|S(e_{i,j})| = 6$; in this case both $x^i_{i,j}$ and $x^j_{i,j}$ must be in $R(e_{i,j})$ to have $|R(e_{i,j})| = 4$.

Now, we compute a canonical run subsequence R' of S of length at least $|R|$. Consider $R(v_i)$, $1 \leqslant i \leqslant n$, and $R(e_{i,j})$, with $\{v_i, v_j\} \in E$.

If $|R(v_i)| = 4$, then define $R'(v_i) = w_i x^i_{i,j} x^i_{i,h} x^i_{i,z}$ (or equivalently define $R'(v_i) = x^i_{i,j} x^i_{i,h} x^i_{i,z} w_i$).

If $|R(v_i)| = 3$, then by construction of $S(v_i)$ at least two of $x^i_{i,j}$, $x^i_{i,h}$, $x^i_{i,z}$ belong to $R(v_i)$. Then, at most one of $R(e_{i,j})$, $R(e_{i,h})$, $R(e_{i,z})$ can contain a symbol in $\{x^i_{i,j}, x^i_{i,h}, x^i_{i,z}\}$, assume w.lo.g. that $x^i_{i,j}$ belongs to $R(e_{i,j})$. We define $R'(v_i) = w_i x^i_{i,j} x^i_{i,h} x^i_{i,z}$ (or equivalently $R'(v_i) = x^i_{i,j} x^i_{i,h} x^i_{i,z} w_i$) and $R'(e_{i,j}) = e^1_{i,j} e^1_{i,j} e^2_{i,j}$ (or equivalently $R'(e_{i,j}) = e^1_{i,j} e^2_{i,j} e^2_{i,j}$).

Since $|R(e_{i,j})| \leqslant 4$, we have that

$$|R'(e_{i,j})| \geqslant |R(e_{i,j})| - 1$$

and

$$|R'(v_i)| = |R(v_i)| + 1.$$

It follows that the size of R' is not decreased with respect to the length of R.

If $|R(v_i)| = 2$, then define $R'(v_i) = w_i w_i$.

By construction of R', either $|R'(v_i)| = 4$ and

$$R'(v_i) = w_i x^i_{i,j} x^i_{i,h} x^i_{i,z} \text{ or } R'(v_i) = x^i_{i,j} x^i_{i,h} x^i_{i,z} w_i$$

or $|R'(v_i)| = 2$ and

$$R'(v_i) = w_i w_i.$$

Again the size of R' is not decreased with respect to the size of R.

In order to compute a canonical run subsequence, we consider an edge $\{v_i, v_j\} \in E$ and the run subsequences $R'(v_i)$ and $R'(v_j)$ of $S(v_i)$, $S(v_j)$, respectively. Consider the case that $R'(v_i) = w_i w_i$ and $R'(v_j) = w_j w_j$. Then by construction $|R'(e_{i,j})| = 4$ and assume with loss of generality that $R'(e_{i,j}) = e^1_{i,j} e^1_{i,j} x^j_{i,j} e^2_{i,j}$. Now, we can modify R' so that

$$R'(v_i) = w_i x^i_{i,j} x^i_{i,h} x^i_{i,z}$$

by eventually removing $x^i_{i,h}$, $x^i_{i,z}$ from $R'(e_{i,h})$ and $R'(e_{i,z})$. In this way, we decrease by at most one the length of each of $R'(e_{i,h})$, $R'(e_{i,z})$ and we increase of two the length of $R'(v_i)$. It follows that the length of R' is not decreased by this modification. By iterating this modification, we obtain that for each edge $\{v_i, v_j\} \in E$ at most one of $R'(v_i)$, $R'(v_j)$ has length two.

The run subsequence R' we have built is then a canonical run subsequence of S such that $|R'| \geqslant |R|$. ◀

Now, we are ready to prove the main results of the reduction.

▶ **Lemma 14.** *Let $G = (V, E)$ be an instance of* MISC *and let S be the corresponding instance of* 2-Longest Run Subsequence*. Given an independent set I of size at least q in G, we can compute in polynomial time a run subsequence of S of length at least $5q + 4(n - q) + 3m + 3(n + m)$.*

Proof. We construct a subsequence run R of S as follows:

- For each $v_i \in I$, define for the substring $S(v_i)$ the run subsequence $R(v_i) = w_i w_i$;
- For each $v_i \in V \setminus I$, define for the substring $S(v_i)$ the run subsequence $R(v_i) = w_i x^i_{i,j} x^i_{i,h} x^i_{i,z}$;
- For each $\{v_i, v_j\} \in E$, if $v_i \in I$ (or $v_j \in I$, respectively) define for the substring $S(e_{i,j})$ the run subsequence $R(e_{i,j}) = e^1_{i,j} x^i_{i,j} e^2_{i,j} e^2_{i,j}$ ($R(e_{i,j}) = e^1_{i,j} e^1_{i,j} x^j_{i,j} e^2_{i,j}$, respectively); if both $v_i, v_j \in V \setminus I$, define for the substring $S(e_{i,j})$ the run subsequence $R(e_{i,j}) = e^1_{i,j} e^1_{i,j} e^2_{i,j}$

Moreover, R contains each separation substring of S, denoted by $R_{Sep,i}$, $1 \leqslant i \leqslant n + m$.

First, we prove that R is a run subsequence, that is R contains a single run for each symbol in Σ. This property holds by construction for each symbol in Σ having only occurrences in $R(v_i)$, with $v_i \in V$, $R(e_{i,j})$, with $\{v_i, v_j\} \in E$, and $R_{Sep,i}$, $1 \leqslant i \leqslant n + m$. What is left to

prove is that $x^i_{i,j}$ appears in at most one of $R(v_i)$, with $v_i \in V$, $R(e_{i,j})$, with $\{v_i, v_j\} \in E$. Indeed, by construction, $R(e_{i,j})$ contains $x^i_{i,j}$ only if $R(v_i) = w_i w_i$. It follows that R is a run subsequence of S.

Consider the length of R. For each $v_i \in V \setminus I$, R contains a run subsequence of $S(v_i)$ of length 4. For each $v_i \in I$, R contains a run subsequence of length 2. For each $\{v_i, v_j\} \in E$, with $v_i, v_j \in V \setminus I$, R contains a run subsequence of $S(e_{i,j})$ of length 3; for each $\{v_i, v_j\} \in E$, with $v_i \in I$ or $v_j \in I$, R contains a run subsequence of $S(e_{i,j})$ of length 4. Finally, each separation substring $R_{Sep,i}$, $1 \leqslant i \leqslant n+m$, in R has length 3. Hence the total length of R is at least $5q + 4(n-q) + 3m + 3(n+m)$ (by accounting, for each $R(v_i)$ of length 2, the increasing of the length of the three run subsequences $R(e_{i,j})$, $R(e_{i,h})$, $R(e_{i,z})$ from 3 to 4 to $R(v_i)$). ◀

▶ **Lemma 15.** *Let $G = (V, E)$ be an instance of* MISC *and let S be the corresponding instance of* 2-Longest Run Subsequence*. Given a run subsequence of S of length at least $5q + 4(n-q) + 3m + 3(n+m)$, we compute in polynomial time an independent of G of size at least q.*

Proof. Consider a run subsequence R of S of length at least $5q + 4(n-q) + 3m + 3(n+m)$. By Lemma 13, we assume that R is a canonical run subsequence of S. It follows that we can define an independent V' of size at least q in G as follows:

$$V' = \{v_i : |R(v_i)| = 2\}$$

By the definition of canonical run subsequence, it follows that V' is an independent set, since if $|R(v_i)| = |R(v_j)| = 2$, with $1 \leqslant i, j \leqslant n$, then $\{v_i, v_j\} \notin E$. Furthermore, by the definition of canonical run subsequence, since $|R| \geqslant 5q + 4(n-q) + 3m + 3(n+m)$, there are at least q run subsequences $R(v_i)$, with $1 \leqslant i \leqslant n$, of length two such that $|R(e_{i,j})| = |R(e_{i,h})| = |R(e_{i,z})| = 4$, with $\{v_i, v_j\}, \{v_i, v_h\}, \{v_i, v_z\} \in E$. It follows that $|V'| \geqslant q$. ◀

Now, we can prove the main result of this section.

▶ **Theorem 16.** 2-Longest Run Subsequence *is APX-hard.*

Proof. We have shown a reduction from MISC to 2-Longest Run Subsequence. By Lemma 11, the instance of 2-Longest Run Subsequence we have built consists of a string with at most two occurrences for each symbol. We will now show that this reduction is an L-reduction from MISC to 2-Longest Run Subsequence (see Definition 4).

Consider an instance I of MISC and a corresponding instance I' of 2-Longest Run Subsequence. Then, given any optimal solution $opt(I')$ of 2-Longest Run Subsequence on instance I', by Lemma 15 it holds that

$$opt(I') \leqslant 5 \cdot opt(I) + 4(n - opt(I)) + 3m + 3(n+m) = opt(I) + 7n + 6m$$

In a cubic graph $G = (V, E)$, $|E| = \frac{3}{2}|V|$, hence $m = \frac{3}{2}n$. Furthermore, we can assume that an independent set has size at least $\frac{n}{4}$. Indeed, such an independent set V' can be greedily computed as follows: pick a vertex v in the graph, add it to V' and delete $N[v]$ from G. At each step, we add one vertex in V' and we delete at most 4 vertices from G.

Since $m = \frac{3}{2}n$ and $n \leqslant 4 \cdot opt(I)$, we thus have that

$$opt(I') \leqslant opt(I) + 7n + 6m = opt(I) + 16 \cdot n \leqslant opt(I) + 64 \cdot opt(I)$$

and then $\alpha = 65$ in Definition 4.

Conversely, consider a solution S' of length $5q + 4(n-q) + 3m + 3(n+m)$ of 2-Longest Run Subsequence on instance I'. First, notice that by Lemma 13, we can assume that S' and $opt(I')$ are both canonical. By Lemma 14, if $opt(I) = p$, then $opt(I') \geqslant 5p + 4(n-p) + 3m + 3(n+m)$. By Lemma 15, starting from S', we can compute in polynomial time a solution V' of MISC on instance I, with $|V'| \geqslant q$. It follows that

$$|opt(I) - |V'|| \leqslant |opt(I) - q| = |p - q| =$$

$$|5p + 4(n-p) + 3m + 3(n+m) - (5q + 4(n-q) + 3m + 3(n+m))| \leqslant$$

$$|opt(I') - (5q + 4(n-q) + 3m + 3(n+m))|$$

Then $\beta = 1$ in Definition 4. Thus we indeed have designed an L-reduction, therefore, the APX-hardness of 2-Longest Run Subsequence follows from the APX-hardness of MISC [1] and from Theorem 5. ◀

6 Conclusion

In this paper, we deepen the understanding of the complexity of the recently introduced problem Longest Run Subsequence. We show that the problem remains hard (even from the approximation point of view) also in the very restricted setting where each symbol occurs at most twice. We also complete the parameterized complexity landscape. From the more practical point of view, it is however unclear how our FPT algorithm could compete with implementations done in [10].

An interesting future direction is to further investigate the approximation complexity of the Longest Run Subsequence problem beyond APX-hardness. Note that a trivial $\min(|\Sigma|, occ)$-approximation algorithm (occ is the maximum number of occurrences of a symbol in the input S) can be designed by taking the solution having maximum length between: (1) a solution having one occurrence for each symbol in Σ and (2) a solution consisting of the a-run of maximum length, among each $a \in \Sigma$. This leads to a $\sqrt{|S|}$-approximation algorithm. Indeed, if the a-run of maximum length is greater than $\sqrt{|S|}$, then solution (2) has length at least $\sqrt{|S|}$, thus leading to the desired approximation factor. If this is not the case, then each symbol in Σ has less then $\sqrt{|S|}$ occurrences, thus a solution of Longest Run Subsequence on instance S has at length smaller than $|\Sigma|\sqrt{|S|}$. It follows that (1) is a solution with the desired approximation factor. We let for future work closing the gap between the APX-hardness and the $\sqrt{|S|}$-approximation factor of Longest Run Subsequence.

References

1 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.

2 Michael Alonge, Sebastian Soyk, Srividya Ramakrishnan, Xingang Wang, Sara Goodwin, Fritz J. Sedlazeck, Zachary B Lippman, and Michael C. Schatz. Fast and accurate reference-guided scaffolding of draft genomes. *Genome Biology*, 20:244, 2019. doi:10.1101/519637.

3 Alberto Apostolico, Gad M. Landau, and Steven Skiena. Matching for run-length encoded strings. *J. Complex.*, 15(1):4–16, 1999. doi:10.1006/jcom.1998.0493.

4 Hans L. Bodlaender, Bart M. P. Jansen, and Stefan Kratsch. Kernelization Lower Bounds by Cross-Composition. *SIAM J. Discrete Math.*, 28(1):277–305, 2014.

5 Lauren Coombe, Vladimir Nikolic, Justin Chu, Inanc Birol, and René Warren. ntJoin: Fast and lightweight assembly-guided scaffolding using minimizer graphs. *Bioinformatics*, 36:3885–3887, 2020. doi:10.1093/bioinformatics/btaa253.

6 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

7 Manish Goel, Hequan Sun, Wen-Biao Jiao, and Korbinian Schneeberger. SyRI: finding genomic rearrangements and local sequence differences from whole-genome assemblies. *Genome Biology*, 20:277, 2019. doi:10.1186/s13059-019-1911-0.

8 Ioannis Koutis. Faster Algebraic Algorithms for Path and Packing Problems. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings*, volume 5125 of *LNCS*, pages 575–586. Springer, 2008. doi:10.1007/978-3-540-70575-8_47.

9 Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, Approximation, and Complexity Classes. *J. Comput. Syst. Sci.*, 43(3):425–440, 1991. doi:10.1016/0022-0000(91)90023-X.

10 Sven Schrinner, Manish Goel, Michael Wulfert, Philipp Spohr, Korbinian Schneeberger, and Gunnar W. Klau. The longest run subsequence problem. In Carl Kingsford and Nadia Pisanti, editors, *20th International Workshop on Algorithms in Bioinformatics, WABI 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 172 of *LIPIcs*, pages 6:1–6:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.WABI.2020.6.

11 Ryan Williams. Finding paths of length k in $O^*(2^k)$ time. *Inf. Process. Lett.*, 109(6):315–318, 2009. doi:10.1016/j.ipl.2008.11.004.

Data Structures for Categorical Path Counting Queries

Meng He
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Serikzhan Kazi
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Abstract

Consider an ordinal tree T on n nodes, each of which is assigned a *category* from an alphabet $[\sigma] = \{1, 2, \ldots, \sigma\}$. We preprocess the tree T in order to support *categorical path counting queries*, which ask for the number of distinct categories occurring on the path in T between two query nodes x and y. For this problem, we propose a linear-space data structure with query time $\mathcal{O}(\sqrt{n} \lg \frac{\lg \sigma}{\lg w})$, where $w = \Omega(\lg n)$ is the word size in the word-RAM. As shown in our proof, from the assumption that matrix multiplication cannot be solved in time faster than cubic (with only combinatorial methods), our result is optimal, save for polylogarithmic speed-ups. For a trade-off parameter $1 \leq t \leq n$, we propose an $\mathcal{O}(n + \frac{n^2}{t^2})$-word, $\mathcal{O}(t \lg \frac{\lg \sigma}{\lg w})$ query time data structure. We also consider c-approximate categorical path counting queries, which must return an approximation to the number of distinct categories occurring on the query path, by counting each such category at least once and at most c times. We describe a linear-space data structure that supports 2-approximate categorical path counting queries in $\mathcal{O}(\lg n / \lg \lg n)$ time.

Next, we generalize the categorical path counting queries to weighted trees. Here, a query specifies two nodes x, y and an orthogonal range Q. The answer to thus formed *categorical path range counting* query is the number of distinct categories occurring on the path from x to y, if only the nodes with weights falling inside Q are considered. We propose an $\mathcal{O}(n \lg \lg n + (n/t)^4)$-word data structure with $\mathcal{O}(t \lg \lg n)$ query time, or an $\mathcal{O}(n + (n/t)^4)$-word data structure with $\mathcal{O}(t \lg^{\epsilon} n)$ query time. For an appropriate choice of the trade-off parameter t, this implies a linear-space data structure with $\mathcal{O}(n^{3/4} \lg^{\epsilon} n)$ query time. We then extend the approach to the trees weighted with vectors from $[n]^d$, where d is a constant integer greater than or equal to 2. We present a data structure with $\mathcal{O}(n \lg^{d-1+\epsilon} n + (n/t)^{2d+2})$ words of space and $\mathcal{O}(t \frac{\lg^{d-1} n}{(\lg \lg n)^{d-2}})$ query time. For an $\mathcal{O}(n \cdot \operatorname{polylog} n)$-space solution, one thus has $\mathcal{O}(n^{\frac{2d+1}{2d+2}} \cdot \operatorname{polylog} n)$ query time.

The inherent difficulty revealed by the lower bound we proved motivated us to consider data structures based on *sketching*. In unweighted trees, we propose a sketching data structure to solve the approximate categorical path counting problem which asks for a $(1 \pm \epsilon)$-approximation (i.e. within $1 \pm \epsilon$ of the true answer) of the number of distinct categories on the given path, with probability $1 - \delta$, where $0 < \epsilon, \delta < 1$ are constants. The data structure occupies $\mathcal{O}(n + \frac{n}{t} \lg n)$ words of space, for the query time of $\mathcal{O}(t \lg n)$. For trees weighted with d-dimensional weight vectors ($d \geq 1$), we propose a data structure with $\mathcal{O}((n + \frac{n}{t} \lg n) \lg^d n)$ words of space and $\mathcal{O}(t \lg^{d+1} n)$ query time.

All these problems generalize the corresponding categorical range counting problems in Euclidean space $\mathbb{R}^{d+1}$, for respective d, by replacing one of the dimensions with a tree topology.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases data structures, weighted trees, path queries, categorical queries, coloured queries, categorical path counting, categorical path range counting

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.15

1 Introduction

In orthogonal range searching, one preprocesses a given finite set $S \subset \mathbb{R}^d$ into a data structure so that the points inside an axis-aligned query (hyper-)rectangle can be efficiently searched. For example, orthogonal range counting asks for the number of points falling inside the

query rectangle, whereas the orthogonal range reporting problem asks to enumerate all such points. We refer the reader to [28, 7, 1] and references therein for the state-of-the-art in the discipline.

In some applications, of the actual interest may be the number of distinct *types*, or *categories*, of points that fall within the query rectangle. Apart from uses in business intelligence (enshrined in SQL keywords `DISTINCT` and `GROUP BY`), these distinct values find uses in SQL query optimization [9], too. Categorical (also known as *coloured*) range searching is thus an area of active research in computer science [22, 2, 29, 30, 31, 21, 33, 19, 8, 6].

A few aspects render the categorical variants of range searching harder than their "plain" counterparts. First, there can be far fewer categories than points. Second, such problems are not easily decomposable – for two disjoint regions S_1 and S_2, knowing just the number of distinct categories in each of them is insufficient to infer the count for the union $S_1 \cup S_2$.

For all the progress in categorical *reporting* queries (where one enumerates the distinct categories in the query region) [8, 6], with results almost matching the state of the art in regular 2D reporting [7], efficient categorical *counting* remains elusive, with the currently best results of $\mathcal{O}(n^2 \lg^2 n)$ words and $\mathcal{O}(\lg^2 n)$ query time [22, 29], or $\mathcal{O}(n \lg^6 n)$ words and $\mathcal{O}(\sqrt{n} \lg^7 n)$ query time [29], versus the optimal linear-space and $\mathcal{O}(\frac{\lg n}{\lg \lg n})$-time data structure for 2D orthogonal range counting [28]. In the exact opposite to the "plain" case, the categorical version of range counting is deemed to be harder than its reporting counterpart [29], when $d \geq 2$.

Meanwhile, given the versatility of trees as a data organization tool, information retrieval from tree-structured hierarchies is set to gain in importance. Hence researchers considered the generalizations of orthogonal range searching, where one of the dimensions is replaced by a tree topology, whereas the remaining coordinates of the points become the *weights* of the nodes [23]. Such a weight can be either a scalar, which corresponds to generalizing a Euclidean 2D point-set to trees, or a *vector*, when extending from $\mathbb{R}^d$, $d \geq 3$. Therefore, a tree weighted with d-dimensional *weight vectors* generalizes a point-set from $\mathbb{R}^{d+1}$. (Note that when $d = 0$, an unweighted tree thus generalizes a 1D set.)

The generalization from point-sets to trees gives rise to *path queries*, which ask a question about the nodes on the query path, whose weights fall inside the query rectangle; for example, a path counting query asks only for the number of such nodes, whereas the reporting variant asks to enumerate them [34, 26]. Research on path queries has spawned a wide range of metrics, such as range quantiles [26], minimum/maximum [5], mode/minority [12], and (α-)majority/minority [17].

Analogously to the Euclidean scenario, the qualitative side of the relation between node-entities is best captured in categorical variants of path queries. For example, let us annotate a phylogenetic tree for a set of genomes by marking each divergence with a *type* of mutation. The number of distinct mutation event types between two given species then could serve as a proxy for evolutionary "distance" between them. A *categorical path counting query*, which asks for the number of distinct categories on a query path, provides an adequate model in this case.

Generalizing the 1D categorical reporting problem, Durocher et al. [12] solved the *top-k colour reporting problem* on unweighted trees. We believe that in trees, neither the counting problem in the categorical setting, nor the scenario of weighted nodes has been studied before. In this paper, we formalize these problems and propose solutions to them.

We consider an ordinal tree[1] T on n nodes, such that each node z of T is associated with a category $\mathsf{c}(z) \in [\sigma]$. [2] Specified at query time is a query path $P_{x,y}$ between two nodes x, y in T. We are to preprocess T into a data structure to compute in an efficient manner the number $n_{real} = |\{\mathsf{c}(z) \mid z \in P_{x,y}\}|$. This is the **categorical path counting problem** studied in this paper.

Next, for $d \geq 1$, we consider a tree T in which each node, along with a category, is associated with a certain weight vector $\mathbf{w}(z) \in [n]^d$. In addition to a query path $P_{x,y}$, at query time specified also is an axis-aligned (hyper-)rectangle Q from $[n]^d$. We are to preprocess T into a data structure to compute in an efficient manner the number $n_{real} = |\{\mathsf{c}(z) \mid z \in P_{x,y} \wedge \mathbf{w}(z) \in Q\}|$. This is the **categorical path range counting problem** studied in the present paper.

For both problems, a *c-approximate* (for $c > 1$) answer is a number n_{appr} such that $n_{real} \leq n_{appr} \leq c \cdot n_{real}$. A $(1 \pm \epsilon)$-approximate (for $0 < \epsilon < 1$) answer is a number n_{appr} such that $\frac{|n_{appr} - n_{real}|}{n_{real}} \leq \epsilon$.

1.1 Previous Work

For points on a line, Gagie and Kärkkäinen [18] have proposed an $\mathcal{O}(n)$-word solution to the 1D categorical counting problem, with query time $\mathcal{O}(\lg^{1+\epsilon} n)$, [3] for any $\epsilon > 0$. Nekrich [33] proposed another $\mathcal{O}(n)$-space solution with query time $\mathcal{O}(\frac{\lg \sigma}{\lg \lg n})$, where σ is the number of categories.

Grossi and Vind [21] solve the 2D categorical range counting problem in linear space and $\wp(n)$ time, and higher-dimensional variants in almost-linear space and $\wp(n)$ time. The core idea is to divide the universe of categories into chunks of size $\lg n$, and use bitwise-`OR` when querying the restriction of the input set to each such chunk. The best result with polylogarithmic time in the 2D categorical counting problem remains at $\mathcal{O}(n^2 \lg^2 n)$ words and $\mathcal{O}(\lg^2 n)$ query time [22, 29], with [29] also proposing an $\mathcal{O}(X \lg^7 n)$ query-time data structure with $\mathcal{O}((\frac{n}{X})^2 \lg^6 n + n \lg^4 n)$ storage space, for a trade-off parameter $1 \leq X \leq n$; for $X = \sqrt{n}$, the space is thus $\mathcal{O}(n \lg^6 n)$ and the query time is $\mathcal{O}(\sqrt{n} \lg^7 n)$. Whereas Gupta et al. [22] use persistence, Kaplan et al. [29] proceed by a disjoint decomposition of the region covering an individual category, with the subsequent reduction to rectangle-stabbing. In higher dimensions ($d > 2$), [29] proposed an $\mathcal{O}(n^d \lg^{2d-2} n)$-word data structure with $\mathcal{O}(\lg^{2d-2} n)$ query time. They also show that an algorithm for categorical range counting in $\mathbb{R}^2$ that answers m queries over the set of $\mathcal{O}(n)$ points in $\wp(\min\{n, m\}^{\omega/2})$ time would yield an algorithm for obtaining the matrix product $MM^\intercal$ in $\wp(k^{\omega/2})$ time, for any $k \times k$ matrix M over $\{0, 1\}$, where ω is the best current exponent for Boolean matrix multiplication. Further, Kaplan et al. [29] proposed an $\mathcal{O}((\frac{n}{X})^{2d} + n \lg^{d-1} n)$-word data structure with $\mathcal{O}(X \lg^{d-1} n)$ query time, for a trade-off parameter $1 \leq X \leq n$. This implies an $\widetilde{\mathcal{O}}(n)$-space[4] data structure with $\widetilde{\mathcal{O}}(n^{\frac{2d-1}{2d}})$ query time.

Nekrich [33] proposed an $\mathcal{O}(n(\lg \lg n)^2)$-word data structure for $(4 + \epsilon)$-approximate 2D categorical counting in $\mathcal{O}((\lg \lg n)^2)$ time; this translates to a linear-space data structure for an $n \times n$ grid, that returns in $\mathcal{O}(1)$ time a $(1 + \epsilon)$-approximation for the number of points in a 3-sided 2D query range, for a constant $0 < \epsilon < 1$. El-Zein et al. [13] solved the approximate

[1] i.e. a tree in which the children of a node are ordered
[2] We set $[n] \triangleq \{1, 2, \ldots, n\}$ for any $n \in \mathbb{N}$.
[3] We use $\lg n \triangleq \log_2 n$, and explicitly specify the base otherwise.
[4] Notation $\widetilde{\mathcal{O}}$ leaves out polylogarithmic factors.

categorical range counting problem in 1D in succinct $\mathcal{O}(n)$ bits of space and $\mathcal{O}(1)$ time. The core technique is to sample the prefixes of an array with exponentially increasing number of distinct categories covered, and "sandwich" the query point between two sampled values using transdichotomous data structures [14].

Lai et al. [30] used sketching data structures [10] to solve the approximate categorical range counting problem in a probabilistic setting. In d dimensions, they proposed an $\mathcal{O}(dn \lg^{d-1} n)$-words-of-space data structure, to support queries in $\mathcal{O}(d \lg^{d+1} n)$ time, with probability $1 - \delta$, where $0 < \delta < 1$ is a given constant. Sketches approximate the number of distinct categories occurring in a collection; being small and additive, in the solution of Lai et al. they serve as summary structures.

To the best of our knowledge, the only categorical range searching problem considered so far for tree topologies is the *top-k colour reporting* problem. Therein, the categories have priorities, and the k highest-priority categories occurring on the given path are to be reported. Durocher et al. [11] introduce and solve this problem in (optimal) $\mathcal{O}(n)$ space and $\mathcal{O}(1 + k)$ time. They use heavy-path decomposition and chaining [32] to reduce the problem to 2D reporting in a narrow grid.

1.2 Our Contribution

For the categorical path counting problem, we propose a linear-space data structure with query time $\mathcal{O}(\sqrt{n} \lg \frac{\lg \sigma}{\lg w})$, where w is the word size on the word-RAM model. We show, by a reduction from Boolean matrix multiplication, that the query time is optimal within polylogarithmic factors, with current knowledge and when only combinatorial methods are allowed. This conditional lower bound is surprising, because the 1D counterpart in the Euclidean case admits a linear-space solution with a sub-logarithmic query time, and a similar conditional lower bound can only be proven in 2D. In other words, having a tree structure in the presence of categories is about as hard as having a second dimension, making the query time go up from polylogarithmic to polynomial, when the space usage is linear. This however is not the case in the previous work on path queries [12, 26, 23]. Specifically, for a trade-off parameter $1 \leq t \leq n$, we propose an $\mathcal{O}(n + \frac{n^2}{t^2})$-word, $\mathcal{O}(t \lg \frac{\lg \sigma}{\lg w})$ query time data structure (which corresponds to a linear-space data structure with $\mathcal{O}(\sqrt{n} \lg \frac{\lg \sigma}{\lg w})$ query time). We also describe a linear-space data structure that supports 2-approximate categorical path counting queries in $\mathcal{O}(\frac{\lg n}{\lg \lg n})$ time. These problems have not been considered in trees before.

We also generalize the categorical path counting queries to weighted trees. For $d = 1$, we propose an $\mathcal{O}(n \lg \lg n + (n/t)^4)$-word data structure with $\mathcal{O}(t \lg \lg n)$ query time, or an $\mathcal{O}(n + (n/t)^4)$-word data structure with $\mathcal{O}(t \lg^{\epsilon} n)$ query time. This implies a linear-space data structure with $\mathcal{O}(n^{3/4} \lg^{\epsilon} n)$ query time. The corresponding $\mathcal{O}(n \lg^6 n)$-word solution to categorical range counting in $\mathbb{R}^2$ by [29] achieves $\mathcal{O}(\sqrt{n} \lg^7 n)$ query time. Compared to the best result in the Euclidean counterpart, we thus sacrifice an $\widetilde{\mathcal{O}}(\sqrt[4]{n})$-factor in query time, to accommodate the tree structure.

We further extend the approach to the trees weighted with multidimensional vectors from $[n]^d$, where d is a constant integer greater than or equal to 2. We describe an $\mathcal{O}(n \lg^{d-1+\epsilon} n + (n/t)^{2d+2})$-word data structure with $\mathcal{O}(t \frac{\lg^{d-1} n}{(\lg \lg n)^{d-2}})$ query time. For an $\widetilde{\mathcal{O}}(n)$-space solution, this yields $\widetilde{\mathcal{O}}(n^{\frac{2d+1}{2d+2}})$ query time. When $d \geq 2$, this result matches the best corresponding result in $\mathbb{R}^{d+1}$ by Kaplan et al. [29], within polylogarithmic factors.

Our sketching data structure for unweighted trees solves the approximate categorical path counting problem, which asks for a $(1 \pm \epsilon)$-approximation for the number of distinct categories on the given path, with probability $1 - \delta$. The data structure occupies $\mathcal{O}(n + \frac{n}{t} \lg n)$

words of space, for the query time of $\mathcal{O}(t \lg n)$. For trees weighted with d-dimensional weight vectors ($d \geq 1$), we propose an $\mathcal{O}((n + \frac{n}{t} \lg n) \lg^d n)$-word data structure with $\mathcal{O}(t \lg^{d+1} n)$ query time. Here, $0 < \epsilon, \delta < 1$ are arbitrarily small constants.

2 Preliminaries

In this section we introduce the notation and give background on the concepts used in the paper.

2.1 Concepts and Notation

We denote by $|T|$ the size (i.e. the number of nodes) of the tree T, whose set of nodes is denoted as $V(T)$. For $x, y \in V(T)$, the path between x and y is denoted as $P_{x,y}$. For brevity, if no confusion ensues, we write $x \in T$ to denote $x \in V(T)$. We write $P_{x,y} \subseteq T$ to indicate that a path belongs to a tree. We denote the root of T by $\perp$; thus $P_{x,\perp}$ is the root-to-x path. In all our input trees, each $x \in T$ has a certain *category* $\mathtt{c}(x) \in [\sigma]$ associated with it. In addition, each $x \in T$ can be associated with a *weight* $\mathbf{w}(x) \in [n]$. In general, $\mathbf{w}(x)$ can be a *weight vector* drawn from $[n]^d$, for $d \geq 1$, and the i^{th} component of the weight vector is the i^{th} *weight*. In line with the current trends in orthogonal range searching, we assume the weights to be in the rank space [16]. For brevity, we shall also use *Iverson notation* [20]: For a Boolean predicate P, the symbol $[\![P]\!] \in \{0, 1\}$ equals 1 *iff* $P = \mathtt{true}$. A sequence of objects $I_1, I_2, \ldots, I_k$ is denoted as $\{I_j\}_{j=1}^{k}$. Finally, unless otherwise indicated, w denotes the word size in the word-RAM machine; one typically has $w = \Omega(\lg n)$.

2.2 Compact Representation of Ordinal Labeled Trees

Fast navigation in compactly-represented ordinal labeled trees and *tree extractions* are central to our solutions. In this section we review the pertinent results.

▶ **Lemma 1** (He et al. [24]). *Let T be an ordinal tree on n nodes. Then, T can be represented in $2n + o(n)$ bits of space, to support the following operations in $\mathcal{O}(1)$ time, for any $x, y \in T$: (a) $\mathtt{depth}(x)$ the number of ancestors of x; and (b) $\mathtt{level\_anc}(x, i)$ the i^{th} nearest ancestor of x ($\mathtt{level\_anc}(x, 1)$ being x itself); and (c) $\mathtt{LCA}(x, y)$ the lowest common ancestor of x and y.*

When the tree T is labeled over $[\sigma]$, with $\mathtt{label}(z)$ denoting the label assigned to $z \in T$, the common operators can be sub-scripted. Indeed, let a node (resp. ancestor) labeled α be referred to as an *α-node* (resp. *α-ancestor*). The following result is our main tool in navigating labeled trees:

▶ **Lemma 2** (He et al. [25]). *Let T be an ordinal tree on n nodes, each of which is assigned a label over $[\sigma]$, $\sigma \leq n$. Then, under the word-RAM with word size $w = \Omega(\lg n)$, T can be represented using $\mathcal{O}(n)$ words of space to support the following operations in $\mathcal{O}(\lg \frac{\lg \sigma}{\lg w})$ time, for any $x, y \in T$ and any $\alpha \in [\sigma]$: (a) $\mathtt{depth}_\alpha(x)$ the number of α-nodes on $P_{x,\perp}$; and (b) $\mathtt{level\_anc}_\alpha(x, i)$ the i^{th} nearest α-ancestor of x ($\mathtt{level\_anc}(x, 1) = x$ if x is an α-node); and (c) $\mathtt{pre\_rank}_\alpha(x)$ the number of α-nodes preceding x in preorder; and (d) $\mathtt{pre\_select}_\alpha(j)$ the j^{th} α-node in preorder.*

Labeled versions of the common operators serve to restrict the queries to the given labels only. For example, the number $\mathtt{depth}_\alpha(x) + \mathtt{depth}_\alpha(y) - 2 \cdot \mathtt{depth}_\alpha(z) + [\![\mathtt{label}(z) = \alpha]\!]$, where $z = \mathtt{LCA}(x, y)$, equals the number of α-labeled nodes on the given path $P_{x,y}$.

2.3 Tree Extraction

Tree extraction [26] selects a subset of nodes while preserving the relative preorder ranks, as well as the hierarchical relations among the nodes. Precisely, given a subset $X \subseteq V$ of nodes (X is called the *extracted nodes*), the *extracted tree* T_X is constructed from T via the following *edit operations*. Fix an arbitrary node $y \notin X$, and let $p \in T$ be the parent of y. Let y be the i^{th} child of p, in preorder. Let us erase, from T, the node y together with its incident edges. This frees the i^{th} slot in the list of children of p, as well as the k children $y_1, y_2, \ldots, y_k$ of the node y. Then, y_1 becomes the i^{th} child of p, y_2 becomes its $(i+1)^{\text{st}}$ child, and so on, until y_k becomes p's $(i+k-1)^{\text{st}}$ child. The node that was the $(i+1)^{\text{st}}$ child of p prior to deletion becomes the $(i+k)^{\text{th}}$ child of p, i.e. all the initial children occurring after the i^{th} are shifted to k positions to the right. After erasing all the nodes $y \notin X$ in the described way, the resulting forest F_X is either a tree (in which case we do nothing), or a forest, in which case we create a dummy root r (with preorder rank and depth set to 0) that becomes the parent of all the roots of the trees in F_X, again preserving the relative preorder ranks of the roots.

2.4 Semigroup Path Sum Query Problem

Trees with nodes associated with semigroup elements give rise to *semigroup path sum* problems:

▶ **Definition 3.** *Let us be given a semigroup $(G, \oplus)$ with the sum operator denoted as $\oplus$, and the set of the semigroup's elements denoted as G. Furthermore, let T be an ordinal tree on n nodes, each node x of which is assigned a d-dimensional weight vector $\boldsymbol{w}(x)$, as well as a semigroup element $g(x)$. Then, in a* d-dimensional semigroup path sum problem*, one is given a query path $P_{x,y} \subseteq T$, a query range Q in d-dimensional space, and is asked to evaluate $\bigoplus_{z \in P_{x,y} \wedge \boldsymbol{w}(z) \in Q} g(z)$.*

The framework of [23] can be used to extend a solution to the multidimensional semigroup path sum query problem in the sense of Definition 3 from $(d-1)$ to d dimensions (here, the *size* of a problem refers to the corresponding $|T|$):

▶ **Lemma 4** (Lemma 5 in [23])**.** *Let d be a positive integer constant. Let $G^{(d-1)}$ be an $\mathtt{s}(n)$-word data structure for a $(d-1)$-dimensional semigroup path sum problem of size n. Then, there is an $\mathcal{O}(\mathtt{s}(n) \lg n + n)$-word data structure $G^{(d)}$ for a d-dimensional semigroup path sum problem of size n, whose components include $\mathcal{O}(\lg n)$ structures of type $G^{(d-1)}$, each of which is constructed over a tree on $n+1$ nodes. Furthermore, $G^{(d)}$ can answer a d-dimensional semigroup path sum query by performing $\mathcal{O}(\lg n)$ $(d-1)$-dimensional queries using these components and returning the semigroup sum of the answers. Determining which queries to perform on structures of type $G^{(d-1)}$ requires $\mathcal{O}(1)$ time per query.*

3 Categorical Path Counting

In this section, we consider the categorical path counting problem in the exact and approximate formulations. First, in Section 3.1 we prove a conditional lower bound on the categorical path counting problem in unweighted trees. Then, Section 3.2 offers some background on the techniques used to solve the categorical path counting problem. Further, we design a data structure that matches the lower bound within polylogarithmic factors when using only combinatorial approaches (Section 3.3). We conclude by designing a 2-approximate solution with a much faster query time (Section 3.4).

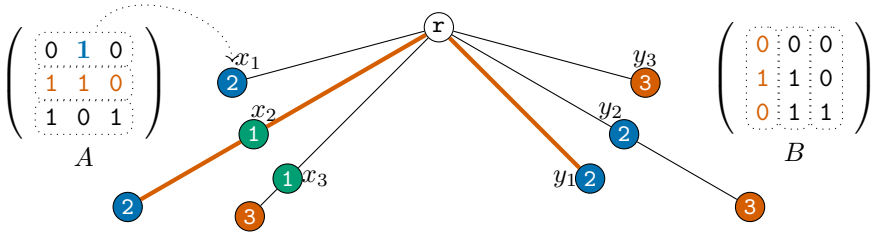


Figure 1 Two matrices A and B each of size $\sqrt{n} \times \sqrt{n}$ with $n = 9$ give rise to a tree over $\sqrt{n}+1$ categories. The dummy root is the node marked r, and the numbers inside circles, as well as the distinct colours, denote the category of the corresponding node. The path shown in thick coloured line corresponds to a path queried when computing the cell $(2, 1)$ of the product $A \times B$. This entry corresponds to the product of the second row and the first column, respectively of the matrices A and B (which are also coloured).

3.1 Hardness of Categorical Path Counting

In this section we show a reduction from the Boolean matrix multiplication problem to the categorical path counting problem over unweighted trees. Namely, we prove

▶ **Theorem 5.** *Let $p(n)$ (for $n \in \mathbb{N}$) be the preprocessing time of a categorical path counting data structure and $q(n)$ its query time, over an ordinal tree T on n nodes, each of which is assigned a category over a finite alphabet. Then Boolean matrix multiplication on two $\sqrt{n} \times \sqrt{n}$ matrices can be solved in $\mathcal{O}(p(n) + nq(n) + n)$ time.*

Proof. Let A and B be two $\sqrt{n} \times \sqrt{n}$ Boolean matrices, and we are to compute the product $C = A \times B$. Let $a_{i,j}, b_{i,j}$ and $c_{i,j}$ be the elements in row i and column j of the matrices A, B and C, respectively. For the i^{th} row of A we construct the set $A_i = \{j \mid a_{i,j} = 1\}$, and for the j^{th} column of B we construct the set $B_j = \{i \mid b_{i,j} = 1\}$. We notice that $c_{i,j} = [\![A_i \cap B_j \neq \emptyset]\!]$.

As $|A_i \cap B_j| = |A_i| + |B_j| - |A_i \cup B_j|$, it is sufficient to focus on computing $|A_i \cup B_j|$, which in turn motivates the following construction of a tree T of size $\mathcal{O}(n)$:

1. We create a dummy root r with dummy category $\sqrt{n}+1$;
2. The root r has $2\sqrt{n}$ children $x_1, x_2, \ldots, x_{\sqrt{n}}$ and $y_1, y_2, \ldots, y_{\sqrt{n}}$;
3. The subtree rooted at each x_i, $1 \leq i \leq \sqrt{n}$, is a single path of length $m_i = |A_i|$, consisting of nodes $x_{i,1}, x_{i,2}, \ldots, x_{i,m_i}$, listed in preorder, i.e. with $x_i = x_{i,1}$ and x_{i,m_i} being the leaf;
4. The subtree rooted at each y_j, $1 \leq j \leq \sqrt{n}$, is a single path of length $n_j = |B_j|$, consisting of nodes $y_{j,1}, y_{j,2}, \ldots, y_{j,n_j}$, listed in preorder, i.e. with $y_j = y_{i,1}$ and y_{j,n_j} being the leaf;
5. $\forall\, 1 \leq i \leq \sqrt{n}$ and $\forall\, 1 \leq j \leq m_i$, the node $x_{i,j}$ is assigned a category – the rank-j entry of A_i;
6. $\forall\, 1 \leq j \leq \sqrt{n}$ and $\forall\, 1 \leq i \leq n_j$, the node $y_{j,i}$ is assigned a category – the rank-i entry of B_j.

Thus T is a tree of size $\mathcal{O}(n)$, in which each node is assigned a category from $[\sqrt{n}+1]$. (See Figure 1 for an example of the tree constructed for two matrices A and B.) Now, it is clear that computing $|A_i \cup B_j|$ is nothing but a categorical path query with query parameters x_{i,m_i} and y_{j,n_j} (subtracting 1 from the result, to correct for the root r). Processing $\sqrt{n} \times \sqrt{n}$ queries each in time $q(n)$, the claimed time bound follows. ◀

The best algebraic methods of multiplying two $t \times t$ Boolean matrices are known to have complexity $\mathcal{O}(t^\omega)$ with $\omega < 2.3727$ (Williams [35]). Two $\sqrt{n} \times \sqrt{n}$ matrices can therefore be multiplied in $\mathcal{O}(n^{\omega/2})$ time. This means that, with current knowledge, one can not have preprocessing time $p(n)$ better than $\mathcal{O}(n^{\omega/2})$ and query time $q(n)$ better than $\mathcal{O}(n^{\omega/2-1})$ simultaneously, i.e. it must be either that $p(n)$ is $\Omega(n^{1.18635})$ or $q(n)$ is $\Omega(n^{0.18635})$.

The best known combinatorial algorithm for multiplying two $n \times n$ Boolean matrices is only polylogarithmically better than cubic [3, 4, 36]. Theorem 5 therefore implies that preprocessing time $p(n)$ can not be better than $n^{3/2}$ and query time $q(n)$ can not be better than $\sqrt{n}$ at the same time, by purely combinatorial methods with current knowledge, save for polylogarithmic speed-ups.

3.2 Uniform Partitioning of the Tree

Next, we review a tree mark-up technique that we use in our solutions in Sections 3.3 and 4.2.

▶ **Lemma 6** (Lemma 9 in [12]). *Given an ordinal tree T on n nodes and an integer $1 \leq t \leq n$ which is called the* blocking factor*, a subset $V' \subseteq V(T)$ of the nodes, called the* marked *nodes, can be selected in $\mathcal{O}(n)$ time so that: (i) $|V'| = \mathcal{O}(n/t)$; (ii) for any $x, y \in V'$ it follows that $\texttt{LCA}(x, y) \in V'$; and (iii) a path containing unmarked nodes only consists of less than t nodes and the edges between them.*

As Durocher et al. [12] only described which nodes should be marked without showing how to mark them in $\mathcal{O}(n)$ time, we propose a linear-time algorithm to mark these nodes, which is presented in the full version of the paper.

Path decomposition using the marked nodes (or, generally, nodes with certain labels) is encapsulated in a `decompose`-operator of Definition 7. Lemma 8 implements `decompose`, as a simple corollary to Lemma 2. In Definition 7 and Lemma 8, T is an ordinal tree on n nodes, each of which is assigned a label over an alphabet $[\sigma]$, where $\sigma \leq n$.

▶ **Definition 7.** *For any pair of nodes x, y of T, for any $\alpha \in [\sigma]$, consider the closest α-nodes $x', y' \in P_{x,y}$, to respectively x and y. Then, the operation $\texttt{decompose}(x, y, \alpha)$ returns the pair of nodes x' and y', or a special symbol **`undefined`** when no such x' and y' exist.*

▶ **Lemma 8.** *The tree T represented via Lemma 2 supports $\texttt{decompose}(x, y, \alpha)$ in $\mathcal{O}(\lg \frac{\lg \sigma}{\lg w})$ time.*

Proof. Let us preprocess the input tree T using Lemma 2. One then has the following cases and the corresponding courses of action:

Case 1 If $\texttt{LCA}(x, y) = y$, we have $x' = \texttt{level\_anc}_\alpha(x, 1)$. Node y' is set to be y itself if y is an α-node; otherwise, $y' = \texttt{level\_anc}_\alpha(x, a)$, where $a = \texttt{depth}_\alpha(x) - \texttt{depth}_\alpha(y)$. The result is *undefined* if y is not an α-node and $a = 0$. The case when x is the ancestor of y is symmetrical.

Case 2 If x and y are not ancestors of each other, we set $z = \texttt{LCA}(x, y)$. There are four subcases, depending on the values $a = \texttt{depth}_\alpha(x) - \texttt{depth}_\alpha(z)$ and $b = \texttt{depth}_\alpha(y) - \texttt{depth}_\alpha(z)$.

- $a > 0$, $b > 0$: One has $x' = \texttt{level\_anc}_\alpha(x, 1), y' = \texttt{level\_anc}_\alpha(y, 1)$;
- $a > 0$, $b = 0$: This case is reduced to **Case 1** by setting $y \coloneqq z$;
- $a = 0$, $b > 0$: This case is reduced to **Case 1** by setting $x \coloneqq z$;
- $a = 0$, $b = 0$: The result is *undefined* if z is not an α-node, and $x' = y' = z$, otherwise;

With $\mathcal{O}(1)$ amount of $\mathcal{O}(\lg \frac{\lg \sigma}{\lg w})$-time operations, the claimed running time follows. ◀

From the properties of tree extraction and Lemmas 2 and 8 it follows that

▶ **Proposition 9.** *In the conditions of Lemma 8, let $P_{x,y} \subseteq T$ be an arbitrary path and $\alpha \in [\sigma]$ an arbitrary label. Let T_α be a tree extraction from T of the node-set $X = \{z \in V(T) \mid \texttt{label}(z) = \alpha\}$. Let x' and y' be the nodes returned by $\texttt{decompose}(x, y, \alpha)$. Then, all the nodes in $P_{x,y} \cap X$ form a contiguous path π in T_α, with end-points $x_\alpha = 1 + \texttt{pre\_rank}_\alpha(x')$ and $y_\alpha = 1 + \texttt{pre\_rank}_\alpha(y')$. Furthermore, $\texttt{decompose}(x, y, \alpha)$ returns **`undefined`** iff $P_{x,y} \cap X = \emptyset$.*

3.3 Categorical Path Counting in Unweighted Trees

In this section, we solve the exact the categorical path counting problem. We do so by precomputing certain information, with additional work at query time. Hence the storage space and the explicit query-time work are balanced by a trade-off parameter.

Namely, the tree T is subject to the following preprocessing:

Nodes marking For the parameter $t \leq n$ to be chosen later, we mark $\mathcal{O}(n/t)$ nodes in T using Lemma 6. Let K be a copy of T labeled over $\{0,1\}$ in such a way that a node $z \in V(K)$ has label 1 *iff* its copy in T is marked. We preprocess K via Lemma 2 thereby enabling the use of Lemma 8;

Path emptiness Let G be a copy of T labeled over $[\sigma]$ in such a way that the node $z \in V(G)$ has label α *iff* its copy in T has category α. We preprocess G via Lemma 2 thereby enabling the use of Lemma 8;

Tabulation We store a table M such that, for the x^{th} and y^{th} (in preorder) marked node of T, one has $M[x,y] \triangleq |\{\mathtt{c}(z) \mid z \in P_{x',y'}\}|$ (i.e. $M[x,y]$ is the number of distinct categories occurring on the path – the *span* – $P_{x',y'} \subseteq T$). Here, x' and y' are found via Proposition 9 as the corresponding nodes to respectively x and y.

The data structures built in Section 3.3 result in the following

▶ **Theorem 10.** *Let T be an ordinal tree on n nodes, each of which is assigned a category over an alphabet $[\sigma]$, where $\sigma \leq n$. Then, T can be preprocessed into a data structure of size $\mathcal{O}(n + \frac{n^2}{t^2})$, for a given $1 \leq t \leq n$, so that a categorical path counting query is answered in $\mathcal{O}(t \lg \frac{\lg \sigma}{\lg w})$ time. The preprocessing time is $\mathcal{O}(\frac{n^2}{t} \lg \frac{\lg \sigma}{\lg w})$. In particular, setting $t = \sqrt{n}$ yields a linear-space data structure with $\mathcal{O}(\sqrt{n} \lg \frac{\lg \sigma}{\lg w})$ query time, and $\mathcal{O}(n^{3/2} \lg \frac{\lg \sigma}{\lg w})$ preprocessing time.*

Proof. We preprocess the input tree T as described in Section 3.3. The structures K, G and M contribute respectively $\mathcal{O}(n), \mathcal{O}(n)$ and $\mathcal{O}(n^2/t^2)$ words, and hence the claimed space.

We thus turn to answering queries and analyzing the query time. As answering the query when $|P_{x,y}| \leq t$ is subsumed in our analysis, we let $|P_{x,y}| > t$.

A call to $\mathtt{decompose}(x, y, 1)$ on K returns two nodes x' and y' such that $|P_{x,x'}|, |P_{y,y'}| \leq t$, and x', y' are marked.

Let x_M and y_M respectively be the relative preorder ranks of x' and y' among the marked nodes of T; one computes x_M and y_M using Proposition 9. We use x_M and y_M to address the table M.

The answer to our query is contained in the following sets of nodes: **Group 0 :** $P_{x',y'}$ (the *span*); **Group 1 :** $P_{x,x'} \setminus \{x'\}$; and **Group 2 :** $P_{y,y'} \setminus \{y'\}$. We note that Groups 1-2 are each of size at most t.

The strategy is to process each group sequentially, so that a category contributes to the answer as long as it appears neither in the groups one has so far traversed, nor in the portion of the path preceding the current node, in the current group.

Namely, the processing of Group 0 reduces to initializing the result counter `res` with $M[x_M, y_M]$. Next, one traverses Group 1 in the direction towards x. Let q be the current node, and p be the node immediately preceding q, on the current path $P_{x,x'}$ consistent with the direction of traversal. We check whether $\mathtt{c}(q)$ occurs in $P_{p,y'}$ using the data structure G and Proposition 9; if not, we increment `res`. Finally, we traverse Group 2 in the direction towards y. Let q be the current node, and p be the node immediately preceding q, on the current path $P_{y,y'}$ and in the direction of traversal. We check whether $\mathtt{c}(q)$ occurs in $P_{x,p}$ using the data structure G and Proposition 9; if it does not, we increment `res`.

We call the operations in Lemmas 2 and 8 $\mathcal{O}(t)$ times; the claimed query time bound follows.

To analyze the preprocessing time, consider the list $\mathcal{L}$ of all the pairs (u, v), $u < v$ of marked nodes, ordered non-decreasingly by $|P_{u,v}|$. The list $\mathcal{L}$ has length $\mathcal{O}(\frac{n^2}{t^2})$ and can be ordered in $\mathcal{O}(\frac{n^2}{t^2})$ time using e.g. counting sort (because the values $|P_{u,v}|$ are non-negative integers that are at most n). We compute the entries of the table M traversing $\mathcal{L}$ from left to right. For the given pair $(u, v) \in \mathcal{L}$, it is either (i) $|P_{u,v}| \leq t$, or (ii) the operation $\texttt{decompose}(u'', v'')$, called on the nodes $u'' \in P_{u,v}$ and $v'' \in P_{u,v}$ respectively closest to u and v, returns two marked nodes u' and v' with the properties claimed in Definition 7. If (i) holds, one explicitly traverses the path $P_{u,v}$ in time $\mathcal{O}(t \lg \frac{\lg \sigma}{\lg w})$, as previously described. In case (ii), one has $|P_{u',v'}| < |P_{u,v}|$ and the answer for the span $P_{u',v'}$ is available; hence, one again uses the algorithm described in the beginning of this proof. ◀

Under the assumption that matrix multiplication cannot be solved faster than cubic time [4, 36], the bounds given in Theorem 10 are optimal, save for polylogarithmic speed-ups.

3.4 2-Approximate Categorical Path Counting

We provide a 2-approximation for the number of distinct categories on $P_{x,y}$ by decomposing the path $P_{x,y}$ as $P_{x,z}$ followed by $P_{y,z}$, with $z = \texttt{LCA}(x, y)$, and computing the answers in $P_{x,z}$ and $P_{y,z}$ separately. It turns out that in contrast to general paths, a query path in which one end is an ancestor of the other lends itself to an efficient categorical counting.

We apply the *chaining* approach [32], by assigning weights to the nodes of T as follows. If for $q \in T$ one has $\texttt{c}(q) = \gamma$, then we identify q's lowest proper γ-ancestor p and set $\mathbf{w}(q) = \texttt{depth}(p)$. We set $\mathbf{w}(q) = -1$, if there is no such p. It can be seen that counting the number of distinct categories on $P_{p,q}$ is equivalent to counting the number of nodes on $P_{p,q}$ with weights in the range $(-\infty, \texttt{depth}(p))$. We use the result of He et al. [26] to encode, in a structure C, the weighted tree and support *path counting queries* (for a path $P_{x,y}$ and a range Q, a path counting query returns the number $|\{z \in P_{x,y} \mid \mathbf{w}(z) \in Q\}|$):

▶ **Lemma 11** (He et al. [26]). *Let T be an ordinal tree on n nodes, each having a weight drawn from $[m]$. Under the word-RAM model, T can be encoded in $\mathcal{O}(n)$ words to support path counting queries in $\mathcal{O}(\frac{\lg m}{\lg \lg n} + 1)$ time.*

For the data structures built in Section 3.4, one thus has

▶ **Theorem 12.** *An ordinal tree T on n nodes, each of which assigned a category, can be preprocessed into an $\mathcal{O}(n)$-word data structure to solve the 2-approximate categorical path counting problem in $\mathcal{O}(\frac{\lg n}{\lg \lg n})$ time. When one query node is an ancestor another, the answer is exact.*

Proof. The input tree T is preprocessed as described in Section 3.4.

The dominant-size data structure C is linear in size (Lemma 11), hence the claimed space.

We focus on answering the query on the path $P_{x,z}$, where $z = \texttt{LCA}(x, y)$, for the query nodes x and y. Given the query $P_{x,z}$, we execute a path counting query in C with arguments $P_{x,z}$ and $(-\infty, \texttt{depth}(z))$. After the verbatim procedure for $P_{y,z}$, we return the sum of (the answers to) the two queries as the sought 2-approximation. We note that when y is an ancestor of x, the answer is exact.

The total running time is dominated by at most two path counting queries; the claimed query time bound follows. ◀

4 Categorical Path Range Counting

In this section, we solve the categorical path counting problem in the case of weighted trees, including those weighted with multidimensional weight vectors. We assume that the number, d, of dimensions is a constant.

In solving the categorical path range counting problem, we still apply the marking technique of Lemma 6. The core idea remains, but we guard against over-counting using somewhat more complex data structures. Namely, in Section 4.1 we extend the repertoire of useful tree operations (Section 2.2) by a *path range emptiness* query, which, in the case of unweighted trees (Section 3.3), was simulated using labeled ancestors and labeled depths (Lemma 2).

4.1 Path Range Emptiness Queries

First, let us formally introduce path range emptiness queries:

▶ **Definition 13.** *For a constant $d \in \mathbb{N}$, let T be an ordinal tree on n nodes, each node z of which is assigned a weight vector $\boldsymbol{w}(z) \in [n]^d$. For any two nodes $x, y \in T$ and any axis-aligned hyper-rectangle Q from $[n]^d$, a* path range emptiness *query is a path query that returns* `false` *if the set $\{z \in T \mid z \in P_{x,y} \wedge \boldsymbol{w}(z) \in Q\}$ is empty, and* `true`*, otherwise.*

It follows from the solutions to the path reporting problem of [5] [5] and Lemma 4 that

▶ **Lemma 14** ([5, 23])**.** *Let T be an ordinal tree on n nodes, each of which is assigned a weight vector from $[n]^d$. Then, T can be preprocessed into a data structure so that a path emptiness query is answered in*

$d = 1$: either (a) $\mathcal{O}(\lg \lg n)$; or (b) in $\mathcal{O}(\lg^{\epsilon} n)$ time. The data structures occupy respectively (a) $\mathcal{O}(n \lg \lg n)$; and (b) $\mathcal{O}(n)$ words of space.

$d \geq 2$: $\mathcal{O}(\frac{\lg^{d-1} n}{(\lg \lg n)^{d-2}})$ time, for an $\mathcal{O}(n \lg^{d-1+\epsilon} n)$-word data structure.

Lemma 14 presents different trade-offs to be used in our solutions for different values of d. For brevity, we shall refer to the query time as $\tau_d(n)$ and to the space cost as $\mathtt{s}_d(n)$.

4.2 Categorical Path Range Counting in d Dimensions

As in Section 3, here, too, we trade off explicit traversals for the storage for precomputed information. There are a few notable differences to accommodate weights. Precisely, the tree T is preprocessed as follows:

Nodes marking. We mark the nodes of T using Lemma 6, with blocking factor t;

Weights partitioning. Along each of the d dimensions, we partition the space $[n]^d$ into $\lceil n/t \rceil$ slabs, using axis-aligned hyper-planes, in such a way that each slab contains exactly t (except, possibly, for the last slab, which may contain less than t) nodes of the tree T (this is always possible, as the weights are in rank space). Precisely, we maintain a list λ_i of slabs per weight component: $\lambda_{i,j} \triangleq \{z \in T \mid (j-1)t < w_i(z) \leq \min\{jt, n\}\}$, for $1 \leq i \leq d$ and $1 \leq j \leq \lceil n/t \rceil$. Somewhat abusing notation, we use "slab $\lambda_{i,j}$" to denote both the orthogonal range and the corresponding set of nodes defined above;

[5] The original works state the results for path reporting, but these results imply the results on path emptiness as stated in Lemma 14.

Path emptiness. For each category $\gamma \in [\sigma]$, we build the tree extraction T_γ of all the nodes with category γ. The nodes of T_γ inherit the weights of the original nodes in T. Each T_γ, in turn, is associated with the following data structures:
- The path emptiness data structure C_γ of Lemma 14;
- y-fast tries [15] $\{Y_{\gamma,i}\}_{i=1}^{d}$ such that $Y_{\gamma,j}$ maps the j^{th} weights of T_γ into rank space $[|T_\gamma|]$;

Mapping structures. Maintained using Lemma 2 are also trees K and G with the topology of T:
- K is labeled over $\{0,1\}$ such that a node $z \in K$ is labeled with 1 *iff* its copy in T is marked;
- G is labeled over $[\sigma]$ such that a node $z \in G$ is given a label γ *iff* its copy in T has category γ;

Tabulation. For each of the $\Theta((n/t)^{2d+2})$ *spans* we store, in a table M, the number of distinct categories occurring in the span. Precisely, let the indices $i_1, i_2, \ldots, i_d, j_1, j_2, \ldots, j_d$ be such that $\forall k 1 \leq i_k \leq j_k \leq \lceil n/t \rceil$ and two nodes x' and y' be marked. Then, the *span* corresponding to these indices is the set $\{z \in P_{x',y'} \mid z \in \cap_{k=1}^{d}(\cup_{l=i_k}^{j_k} \lambda_{k,l})\}$ (i.e. the set of nodes on the path $P_{x',y'}$ such that their weights fall into the relevant rectangle in $[n]^d$). To save space, the nodes x' and y' are referred to by their relative preorder ranks x_M and y_M among the marked nodes. Now, M is a table whose entry $M[x_M, y_M, i_1, j_1, i_2, j_2, \ldots, i_d, j_d]$ stores the number of distinct categories in the span corresponding to the given indices.

▶ **Lemma 15.** *The data structures built in Section 4.2 occupy $\mathcal{O}(\mathtt{s}_d(n) + (n/t)^{2d+2})$ words of space.*

Due to space considerations, we consign the proof of Lemma 15 to the full version of the paper.

We next describe how to resolve queries and analyze the query time:

▶ **Lemma 16.** *The data structures built in Section 4.2 answer a categorical path range counting query in $\mathcal{O}(t \cdot \tau_d(n))$ time.*

Proof. Let $P_{x,y}$ and $Q = \prod_{k=1}^{d}[a_k, b_k]$ be the query arguments. If $|P_{x,y}| \leq t$, we explicitly traverse the path $P_{x,y}$ and count the number of unique categories encountered; the exact procedure is subsumed in the discussion that follows. We therefore assume $|P_{x,y}| > t$, and split the path $P_{x,y}$ into $P_{x,x'}, P_{x',y'}$, and $P_{y,y'}$, as described in the proof of Theorem 10. One has that $|P_{x,x'}|, |P_{y,y'}| \leq t$.

The grid of the marked nodes and the slabs induce a decomposition of the query region into the span and the "rim" – the parts of the query region abutting the span. Of these, only the rim is meant to be explicitly traversed. The details follow.

First, we initialize the indices $i_1, i_2, \ldots, i_d$ and $j_1, j_2, \ldots, j_d$ as $i_k \coloneqq \lceil a_k/t \rceil$ and $j_k \coloneqq \lceil b_k/t \rceil$, for all $1 \leq k \leq d$. That is, i_k^{th} range contains a_k, and j_k^{th} range contains b_k. Furthermore, let x' and y' be respectively the x_M^{th} and y_M^{th} marked node, in preorder; one computes x_M and y_M using Proposition 9. Now the tuple $(x_M, y_M, i_1{+}1, j_1{-}1, i_2{+}1, j_2{-}1, \ldots, i_d{+}1, j_d{-}1)$ determines a span, for which the answer – the number of distinct categories occurring therein – is already precomputed. We initialize the counter variable `res` holding the answer to the query with the table entry $M[x_M, y_M, i_1 + 1, j_1 - 1, i_2 + 1, j_2 - 1, \ldots, i_d + 1, j_d - 1]$. With this span, we also associate the pair of query arguments $P^{(span)} = P_{x',y'}$ and $Q^{(span)} = \prod_{k=1}^{d}[i_k t + 1, \min\{(j_k - 1)t, n\}]$.

Next, our goal is to traverse the rim systematically, scanning for categories, while being careful not to double-count nor miss them. A description of such a traversal follows.

We split the query path $P_{x,y}$ into the "prefix" $P_{x,x'} \setminus \{x'\}$, the middle $P_{x',y'}$, and the "suffix" $P_{y,y'} \setminus \{y'\}$, using marked nodes x' and y' found earlier as in the proof of Theorem 10. Consider all the nodes $z \in P_{x',y'}$ whose weight vector $\mathbf{w}(z)$ pushes z outside of the span. The loci of such vectors in $[n]^d$ clearly can be covered with $\mathcal{O}(d) = \mathcal{O}(1)$ disjoint axis-aligned rectangles – henceforth *canonical rectangles* – in such a way that each canonical rectangle $r \in \mathcal{D}$ lies entirely within some $\lambda_{i,j}$. For each dimension k, there are at most two canonical rectangles within slabs λ_{k,i_k} and λ_{k,j_k}. We assume the availability of such a cover $\mathcal{D}$. From each canonical rectangle $r \in \mathcal{D}$, a *canonical set* $s(r) \triangleq \{z \in P_{x',y'} \mid \mathbf{w}(z) \in r\}$ is constructed. As each r lies inside a slab, one has $|s(r)| \leq t$.

We enumerate the nodes in the rim in (say) the following order: the nodes of the prefix, the nodes of the suffix, and the nodes in each canonical set. Within each set, the nodes are conceptually ordered in the direction from x to y (the traversal order is ascertained via Lemma 1). We walk through these sets, while referring as *previously seen* to the union of the span and the processed sets.

Let z, with $\gamma = \mathtt{c}(z)$, be the current node in our traversal of the rim. The category γ contributes towards `res` *iff* each query from the following list E of path range emptiness queries comes back as `false`. All queries in E are launched on C_γ, the query parameters being restrictions of previously seen sets to the tree T_γ. Namely, we (i) adjust the weights to the rank space of T_γ using the y-fast tries $\{Y_{\gamma,k}\}_{k=1}^{d}$; and (ii) map the associated path to T_γ using Proposition 9 (the path component of a canonical set is $P_{x',y'}$, whereas for the span we use the previously defined $P^{(span)}$ and $Q^{(span)}$). Finally, we also check the part of the current set (be it the prefix, the suffix, or a canonical set) that precedes z in our conventional x-to-y ordering. If the current set is the prefix, we launch a path range emptiness query for the path $P_{x,z} \setminus \{z\}$ and the range Q on C_γ. (For the suffix and canonical sets, this last step is analogous.)

Mapping the query path takes $\mathcal{O}(\lg \frac{\lg \sigma}{\lg w})$ time (Lemma 8); the mapping of the weight-ranges using the y-fast tries is an additive $\mathcal{O}(\lg \lg n)$ time. We note that both time bounds do not exceed $\tau_d(n)$ (the query time stated in Lemma 14). The rim consisting of $\mathcal{O}(d) = \mathcal{O}(1)$ sets of $\mathcal{O}(t)$ nodes, with $\mathcal{O}(1)$ time per fetching an entry, the claim for the query time follows. ◀

As the procedure of zooming into T_γs can be of independent interest, we formalize it in the full version of the paper.

Combining Lemmas 15 and 16 one has

▶ **Theorem 17.** *Let $d \in \mathbb{N}$ be a constant. Let T be an ordinal tree on n nodes, each node $z \in T$ of which is assigned a category $\mathtt{c}(z) \in [\sigma]$, as well as a d-dimensional weight vector $\boldsymbol{w}(z)$, in rank space. Let, furthermore, $1 \leq t \leq n$ be a parameter set prior to construction. Then, for the categorical path range counting problem there exists a data structure such that it uses*

$d = 1$: *either*

- *$\mathcal{O}(n \lg \lg n + (n/t)^4)$ words of space for the query time of $\mathcal{O}(t \lg \lg n)$; or*
- *$\mathcal{O}(n + (n/t)^4)$ words of space for the query time of $\mathcal{O}(t \lg^\epsilon n)$;*

In particular, a linear-space data structure has the query time $\mathcal{O}(n^{3/4} \lg^\epsilon n)$;

$d \geq 2$: *$\mathcal{O}(n \lg^{d-1+\epsilon} n + (n/t)^{2d+2})$ words of space for the query time of $\mathcal{O}(t \frac{\lg^{d-1} n}{(\lg \lg n)^{d-2}})$. In particular, a linear-space data structure has the query time $\widetilde{\mathcal{O}}(n^{\frac{2d+1}{2d+2}})$.*

5 Sketching Data Structures for Approximate Categorical Path Range Counting

In this section we consider a probabilistic approach to the approximate categorical path range counting problem. Section 5.1 reviews *sketches* [10, 30] that we use to approximate the number of distinct categories. Then, in Section 5.2, we solve the $(1 \pm \epsilon)$-approximate categorical counting problem proper, with probability $1 - \delta$, for arbitrarily small constants $0 < \epsilon, \delta < 1$.

5.1 Sketches

For an arbitrary vector $\mathbf{a} = (a_1, a_2, \ldots, a_\sigma) \in \mathbb{R}^\sigma$, Cormode et al. [10] introduce *Hamming norm* $|\mathbf{a}|_H$ of $\mathbf{a}$, defined as $|\mathbf{a}|_H \triangleq \sum_{i=1}^{\sigma} |a_i|^0$, with $|0|^0 \triangleq 0$. It is clear that $|\mathbf{a}|_H = |\{a_i \,|\, a_i \neq 0\}|$, i.e. the Hamming norm equals the number of non-zero components in $\mathbf{a}$. For our purposes, this original vector $\mathbf{a}$ is the *frequency array* $(a_1, a_2, \ldots, a_\sigma)$, with a_i standing for the number of the occurrences of the category i. While referring the reader to [10] and references therein for discussion in depth, we state the main result we build on:

▶ **Lemma 18** ([10, 30]). *Let $0 < \epsilon, \delta < 1$ be constants. Given a vector $\mathbf{a}$, there exists a sketch, $h(\mathbf{a})$, that requires $m = \mathcal{O}(\frac{1}{\epsilon^2} \cdot \lg \frac{1}{\delta})$ words and allows approximation of $|\mathbf{a}|_H$ within a factor of $1 \pm \epsilon$ of the true answer with probability $1 - \delta$. Updating the sketch and computing $|\mathbf{a}|_H$ both take $\mathcal{O}(m)$ time. Furthermore, if $\mathbf{a}$ and $\mathbf{b}$ are two vectors, then $h(\mathbf{a} \pm \mathbf{b}) = h(\mathbf{a}) \pm h(\mathbf{b})$.*

A clarification is in order regarding the *update* operation referred to in Lemma 18. The scenario of Cormode et al. [10] is that of observing a stream while maintaining an approximation to the number of the distinct values seen so far. Update refers to updating the approximation upon observing the next value in the stream. The gist of our solution is in treating certain paths $P_{x,\perp}$ each as a stream of its own and maintaining *several* sketch-summaries thereof. Our adaptation comprises (i) using the same transformation matrix [10, 30] throughout the computations; and (ii) building the sketches using $\delta' = \frac{\delta}{n^{2d+2}}$. Ensured by (i) is the "compatibility" of any two arbitrarily chosen summaries – sketches are obtained by a linear transformation [10] of (in our case) the frequency array, with linearity implying additivity. With (ii), the value of m in Lemma 18 works out to be $m = \mathcal{O}(\frac{1}{\epsilon^2} \lg \frac{1}{\delta / n^{2d+2}}) = \mathcal{O}(\lg n)$.

5.2 $(1 \pm \epsilon)$-Approximate Categorical Path Range Counting

We first solve the $(1 \pm \epsilon)$-approximate categorical path counting problem for unweighted trees; then we use Lemma 4 to extend the data structures to trees weighted with d-dimensional weight vectors.

First, we apply Lemma 19 (whose proof easily follows from the Pigeonhole Principle) with parameter t to mark $\mathcal{O}(n/t)$ nodes in the tree:

▶ **Lemma 19** ([27]). *Let $1 \leq t \leq n$ be an integer parameter. There exists a level l' no deeper than t such that, when one* marks *the nodes on every t^{th} level of the tree T, starting from l', then there are $\mathcal{O}(n/t)$ marked nodes in total.*

Next, at each marked node $z \in T$, one stores the sketch $h(z)$ as a summary of the categories occurring on the path $P_{z,\perp}$. Indeed, let $\mathbf{a}(z)$ be the (conceptual) frequency vector for the categories on the path $P_{z,\perp}$. Then we associate with z a length-m vector $h(z)$ – the sketch of $\mathbf{a}(z)$. One thus obtains

▶ **Lemma 20.** *The data structures built in Section 5.2 occupy* $\mathcal{O}(n + (n/t)\lg n)$ *words and answer a* $(1\pm\epsilon)$*-approximate categorical path counting query in* $\mathcal{O}(t \lg n)$ *time, with probability* $1 - \delta$.

Proof. There are $\mathcal{O}(n/t)$ marked nodes, each storing $m = \mathcal{O}(\lg n)$ words, hence the claimed space.

By the additivity stated in Lemma 18, the answer to a query with arbitrary query nodes x and y is simply $h(x) + h(y) - 2 \cdot h(\texttt{LCA}(x, y))$, corrected for $\mathsf{c}(\texttt{LCA}(x, y))$ using Lemma 18. Therefore, it is sufficient to show how to compute $h(x)$ for an arbitrary node x.

The path $P_{x,\perp}$ can be represented as $P_{x,x'} \cup P_{x',\perp}$, where x' is the closest marked ancestor of x. If there is no such x', then, by construction, the depth of x is no greater than t; this case is solved by an explicit traversal, as shown below. We therefore assume the existence of such x'. The case $x = x'$ is trivial, as we use $h(x')$ directly. If $x \neq x'$, then by construction $|P_{x,x'}| \leq t$. We initialize a zero-vector s of length m and the *current node* to x. We then start ascending the path $P_{x,x'}$ in the direction of x' until the current node equals x'. (Informally, the path $P_{x,x'}$ in the direction towards x' is our "stream", and the "next value" is the category of the next node encountered on this path.) For the category of the node currently being observed, the current sketch s is updated using Lemma 18. This increment thus being an $\mathcal{O}(m) = \mathcal{O}(\lg n)$-time operation, the traversal's time cost is $\mathcal{O}(tm) = \mathcal{O}(t \lg n)$. At the node x', we return the sum of s and of $h(x')$ (which is precomputed), as the sketch for $P_{x,\perp}$. ◀

When the marked nodes in Lemma 20 are assigned the sketches, they are assigned semigroup elements in the sense of Definition 3, the regular component-wise addition in vectors being the corresponding semigroup sum operator. Unmarked nodes are assigned conceptual zero-vectors in view of formal compliance with Definition 3; as the sketches stored at unmarked nodes are never consulted, this has no effect on our algorithm.

The combination of Lemma 20 and Lemma 4 thus yields the following

▶ **Theorem 21.** *Let* $0 < \epsilon, \delta < 1$ *be arbitrarily small constants, and* $d \geq 1$ *be an integer constant. Let, furthermore,* T *be an ordinal tree on* n *nodes, each node* z *of which is assigned a weight* $\boldsymbol{w}(z) \in [n]^d$*, as well as a category* $\mathsf{c}(z) \in [\sigma]$*. Then, there exists a data structure of* $\mathcal{O}((n + \frac{n}{t} \lg n) \lg^d n)$ *words that solves a* $(1 \pm \epsilon)$*-approximate categorical path range counting query in* $\mathcal{O}(t \lg^{d+1} n)$ *time, with success probability no less than* $1 - \delta$.

Proof. We iteratively apply Lemma 4 to Lemma 20. Since we start with G^0 (supplied by Lemma 20) and apply Lemma 4 exactly d times, the space cost of $\mathcal{O}((n + \frac{n}{t} \lg n) \lg^d n)$ words and the query time of $\mathcal{O}(t \lg^{d+1} n)$ follows.

Our data structure fails *iff* at least one of the $\Theta(n^{2d+2})$ possible queries fails. The total probability of failure therefore is at most the sum of the failure probabilities of each of these $\Theta(n^{2d+2})$ queries. When building the data structure, we thus use a stronger guarantee of $\delta' = \frac{\delta}{n^{2d+2}}$, which also means that the length m of the vectors storing sketches is $\mathcal{O}(\lg \frac{1}{\delta'}) = \mathcal{O}(\lg n)$. ◀

References

1 Peyman Afshani and Konstantinos Tsakalidis. Optimal Deterministic Shallow Cuttings for 3-d Dominance Ranges. *Algorithmica*, 80(11):3192–3206, 2018.

2 Pankaj K. Agarwal, Sathish Govindarajan, and S. Muthukrishnan. Range Searching in Categorical Data: Colored Range Searching on Grid. In *ESA*, pages 17–28, 2002.

3 Nikhil Bansal and Ryan Williams. Regularity Lemmas and Combinatorial Algorithms. *Theory Comput.*, 8(1):69–94, 2012.

4 Timothy M. Chan. Speeding up the Four Russians Algorithm by About One More Logarithmic Factor. In *SODA*, pages 212–217, 2015.

5 Timothy M. Chan, Meng He, J. Ian Munro, and Gelin Zhou. Succinct Indices for Path Minimum, with Applications. *Algorithmica*, 78(2):453–491, 2017.

6 Timothy M. Chan, Qizheng He, and Yakov Nekrich. Further Results on Colored Range Searching. In *SoCG*, volume 164, pages 28:1–28:15, 2020.

7 Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *SoCG*, pages 1–10, 2011.

8 Timothy M. Chan and Yakov Nekrich. Better Data Structures for Colored Orthogonal Range Reporting. In *SODA*, pages 627–636, 2020.

9 Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Towards Estimation Error Guarantees for Distinct Values. In *PODS*, pages 268–279, 2000.

10 Graham Cormode, Mayur Datar, Piotr Indyk, and S. Muthukrishnan. Comparing Data Streams Using Hamming Norms (How to Zero In). In *VLDB*, pages 335–345, 2002.

11 Stephane Durocher, Rahul Shah, Matthew Skala, and Sharma V. Thankachan. Top-k color queries on tree paths. In *SPIRE*, pages 109–115, 2013.

12 Stephane Durocher, Rahul Shah, Matthew Skala, and Sharma V. Thankachan. Linear-Space Data Structures for Range Frequency Queries on Arrays and trees. *Algorithmica*, 74(1), 2016.

13 Hicham El-Zein, J. Ian Munro, and Yakov Nekrich. Succinct Color Searching in One Dimension. In *ISAAC*, pages 30:1–30:11, 2017.

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.

15 Michael L. Fredman and Dan E. Willard. Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

16 Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *STOC*, pages 135–143, 1984.

17 Travis Gagie, Meng He, and Gonzalo Navarro. Tree Path Majority Data Structures. In *ISAAC*, volume 123, pages 68:1–68:12, 2018.

18 Travis Gagie and Juha Kärkkäinen. Counting Colours in Compressed Strings. In *CPM*, pages 197–207, 2011.

19 Arnab Ganguly, J. Ian Munro, Yakov Nekrich, Rahul Shah, and Sharma V. Thankachan. Categorical Range Reporting with Frequencies. In *ICDT*, pages 9:1–9:19, 2019.

20 Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley, 1994.

21 Roberto Grossi and Søren Vind. Colored Range Searching in Linear Space. In *SWAT*, volume 8503, pages 229–240, 2014.

22 Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Further results on generalized intersection searching problems: Counting, reporting, and dynamization. *J. Algorithms*, 19(2):282–317, 1995.

23 Meng He and Serikzhan Kazi. Path and Ancestor Queries over Trees with Multidimensional Weight Vectors. In *ISAAC*, volume 149, pages 45:1–45:17, 2019.

24 Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct ordinal trees based on tree covering. *ACM Trans. Algorithms*, 8(4):42:1–42:32, 2012.

25 Meng He, J. Ian Munro, and Gelin Zhou. A Framework for Succinct Labeled Ordinal Trees over Large Alphabets. *Algorithmica*, 70(4):696–717, 2014.

26 Meng He, J. Ian Munro, and Gelin Zhou. Data Structures for Path Queries. *ACM Trans. Algorithms*, 12(4):53:1–53:32, 2016.

27 David A. Hutchinson, Anil Maheshwari, and Norbert Zeh. An external memory data structure for shortest path queries. *Discret. Appl. Math.*, 126(1):55–82, 2003.

28 Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting. In *ISAAC*, pages 558–568, 2004.

29 Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. Efficient Colored Orthogonal Range Counting. *SIAM J. Comput.*, 38(3):982–1011, 2008.

30 Ying Kit Lai, Chung Keung Poon, and Benyun Shi. Approximate colored range and point enclosure queries. *J. Discrete Algorithms*, 6(3):420–432, 2008.

31 Kasper Green Larsen and Freek van Walderveen. Near-Optimal Range Reporting Structures for Categorical Data. In *SODA*, pages 265–276, 2013.

32 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.

33 Yakov Nekrich. Efficient range searching for categorical and plain data. *ACM Trans. Database Syst.*, 39(1):9:1–9:21, 2014.

34 Manish Patil, Rahul Shah, and Sharma V. Thankachan. Succinct representations of weighted trees supporting path queries. *J. Discrete Algorithms*, 17:103–108, 2012.

35 Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *STOC*, pages 887–898, 2012.

36 Huacheng Yu. An improved combinatorial algorithm for Boolean matrix multiplication. *Inf. Comput.*, 261:240–247, 2018.

Compressed Weighted de Bruijn Graphs

Giuseppe F. Italiano
Luiss University, Rome, Italy
Erable, INRIA Grenoble Rhône-Alpes, France

Nicola Prezza
DAIS, Ca' Foscari University of Venice, Italy

Blerina Sinaimeri
Luiss University, Rome, Italy
Erable, INRIA Grenoble Rhône-Alpes, France

Rossano Venturini
Dipartimento di Informatica, Università di Pisa, Pisa, Italy

Abstract

We propose a new compressed representation for weighted de Bruijn graphs, which is based on the idea of delta-encoding the variations of k-mer abundances on a spanning branching of the graph. Our new data structure is likely to be of practical value: to give an idea, when combined with the compressed *BOSS* de Bruijn graph representation, it encodes the weighted de Bruijn graph of a 16x-covered DNA read-set (60M distinct k-mers, $k = 28$) within 4.15 bits per distinct k-mer and can answer abundance queries in about 60 microseconds on a standard machine. In contrast, state of the art tools declare a space usage of at least 30 bits per distinct k-mer for the same task, which is confirmed by our experiments. As a by-product of our new data structure, we exhibit efficient compressed data structures for answering partial sums on edge-weighted trees, which might be of independent interest.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Data structures design and analysis

Keywords and phrases weighted de Bruijn graphs, k-mer annotation, compressed data structures, partial sums

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.16

Supplementary Material The code is written in C++ and is available at
Software (Source Code): `https://github.com/nicolaprezza/cw-dBg`

Funding *Giuseppe F. Italiano*: Partially supported by MUR, the Italian Ministry for University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data).
Rossano Venturini: Partially supported by MUR, the Italian Ministry for University and Research, under PRIN 2017 Project 2017K7XPAN *Algorithms, Data Structures and Combinatorics for Machine Learning* and Università di Pisa Project PRA_2020-2021_26.

1 Introduction

A DNA resequencing experiment consists of reconstructing the nucleotide sequence of large collections of short DNA fragments sampled from a reference genome [16, 18]. Depending on the genome's length and on the number of sampled fragments, each genome position is typically covered several times (up to a few hundred on average) by different fragments. One of the most common (lossy) representations of such a dataset is a de Bruijn graph of order k (k is usually chosen around 30) introduced in [31] and used by the majority of genome and transcriptome assemblers [2, 24, 20, 37]: this is a combinatorial structure storing all k-mers occurring in the DNA fragments, and connecting two k-mers with an edge when their length-$(k+1)$ concatenation also occurs in the dataset. While such a representation is already

useful to discover important features of the sequenced genome (for example, single-point mutations correspond to bubbles in De Bruijn graphs, see e.g., [35, 36]), a much more useful representation should also record the *abundance* of each k-mer, i.e., how often it appears in the dataset. This augmented structure takes the name *weighted de Bruijn graph* and has many applications in both genome and transcriptome analysis. For instance, in *de novo* genome projects it can be used for error correction [22, 37] or to infer genome characteristics as repeat structures or rate of heterozygosity [21]. In transcriptome projects the role of k-mer abundances is even more important as the abundance of a gene does not only reflect its copy-number in the genome, but also and mostly its expression level. Hence, k-mer abundances are crucial for quantifying the expression level of transcripts [26, 30] but also for distinguishing SNPs from sequencing errors, or between different types of alternative splicing events [17, 23].

Due to the massive size of sequencing data, the main challenge is to design data structures for weighted de Bruijn graphs, which can scale up to very large instances such as genomes, genome collections or metagenomics datasets (see for example [24]). In this paper, we tackle exactly this problem and present new space-efficient data structures for storing weighted de Bruijn graphs.

1.1 State of the art

As far as (unweighted) de Bruijn graphs are concerned, Bowe et al. in [3] presented a space-efficient data structure – *BOSS* in the following – based on the concept of *prefix sorting*. Their idea is to sort co-lexicographically the k-mers appearing in the dataset and to append their outgoing labels to a sequence. It turns out that this sequence is a generalization to de Bruijn graphs of the celebrated Burrows-Wheeler transform [4], the text permutation at the core of the FM-index [12]. In its most basic form, the BOSS representation requires just 4 bits per edge to be stored and supports fast navigation queries on the de Bruijn graph.

It would be highly desirable to achieve the same rate of space efficiency also for weighted de Bruijn graphs. A naive strategy to extend the BOSS representation to weighted de Bruijn graphs could be to associate to each node a counter explicitly storing k-mer abundances. The large variance of these counters (that is, difference between the largest and mean abundance), however, requires some compression strategy in order to save space with respect to a straightforward uncompressed solution.

Although many data structures already present in the literature could support k-mer quantification, their representations are not space efficient [24]. For instance an extension of Bloom filters, known as counting Bloom filters [19], allocates a fixed number of bits per k-mer to store its count. This is clearly not space efficient for datasets where most of the k-mers have low abundance. To deal with this, variable-length counters have been proposed. Among them the counting quotient filter has been introduced in [28] and based on it two tools have been proposed: deBGR [27] and Squeakr [29]. Both tools are *approximate* in the sense that the returned counters could be wrong with low probability, but can also store exact counts at the price of a higher space usage. On a RNA-seq Human experiment composed of 1.4 billion distinct k-mers ($k = 28$) with average abundance per k-mer equal to 27, deBGR [27] reports a space usage of 37 bit/k-mer, while Squeakr [29] uses 79.8 bit/k-mer. Another experiment on k-mer counting tools [29] reports that, on a dataset composed of 6.6 billion distinct k-mers ($k = 28$) with average abundance per k-mer equal to 14.9, the RAM usage of Squeakr [29] is of 77.8 bit/k-mer, the RAM usage of KMC2 [8] (signature-based) is of 139 bit/k-mer, and the RAM usage of Jellyfish2 [25] (hash-based) is of 80 bit/k-mer.

Finally, in [33] an approximate data structure called Set-Min sketch has been proposed. This method takes advantage of the power-law distribution of the k-mer counts to reduce the error rate of the returned results. Set-Min sketch is implemented in the tool fress which achieves better memory consumption but does not guarantee exact counts. For a dataset of 6.6 billion distinct k-mers ($k = 28$) with average abundance per k-mer equal to 14.9, the declared RAM usage of fress is 16.9 bit/k-mer with an error rate of 0.01.

1.2 Our contribution

In this paper we propose a new representation for weighted de Bruijn graphs, which is based on the idea of delta-encoding the abundance variations on a spanning branching of the graph. To show the usefulness of our representation we test it on different real datasets obtaining each time a significant space improvement with respect to the state of the art. As an example, when combined with the compressed BOSS de Bruijn graph representation [3], our new data structure stores the weighted de Bruijn graph of a 16x-covered DNA read-set (60M distinct k-mers, $k = 28$) within 4.15 bits per distinct k-mer: just 1.4 bits per k-mer on top of the BOSS representation[1] [3].

Our data structure does not make any assumption on the distribution of the k-mer counts, which makes it appropriate to any dataset for both DNA and RNA. As a by-product of our data structure, we exhibit efficient compressed data structures for answering partial sums on edge-weighted trees, which might be of independent interest.

2 Notation

Logarithms are to the base 2. To simplify notation, we take $\log 1 = 1$. Notation $[n]$ denotes the set $\{1, 2, \ldots, n\}$. Throughout the paper, we work with the DNA alphabet $\Sigma = \{A, C, G, T\}$. A k-mer is a string from Σ^k. If $w \in \Sigma^*$ is a string, then $w[i, j]$ denotes substring $w[i] \cdot w[i+1] \cdots w[j]$, where the symbol "$\cdot$" represents the concatenation of characters.

De Bruijn graphs. A de Bruijn graph is a directed graph $G = (\mathcal{V}, \mathcal{E})$ whose nodes are in bijection with the k-mers appearing in the dataset. To distinguish between k-mers and the corresponding nodes of G, we use the notation $\hat{s}$ to denote the node of G corresponding to k-mer $s \in \Sigma^k$.

Let $s, s' \in \Sigma^k$ be two k-mers. We say that s *is adjacent to* s' if $s[2, k] = s'[1, k-1]$ and the $(k+1)$-mer $s \cdot s'[k]$ occurs in the dataset. The directed edges of G are in bijection with all pairs of adjacent k-mers: $(\hat{s}, \hat{s}')$ belongs to $\mathcal{E}$ if and only if s is adjacent to s'. A *weighted* de Bruijn graph associates an integer positive weight $c(\hat{s})$ to each of its nodes. Weight $c(\hat{s})$ (equivalently, $c(s)$ when we refer to the corresponding k-mer) is called the *abundance* of s and corresponds to the number of times that k-mer s appears in the dataset.

3 Compressed Weighted de Bruijn Graphs

To compress the weights (abundances) of a de Bruijn graph, one could exploit the following observation: since consecutive genomic positions generate adjacent k-mers, weights of adjacent nodes in the de Bruijn graph are likely to be very similar. Let $s, s' \in \Sigma^k$ be such that s is

[1] Note that we divide the total space by the number of *distinct* k*-mers* rather than the number of de Bruijn graph edges as done in [3]. More details on this in Section 4.

adjacent to s'. Recall that with $c(s)$ we denote the k-mer's abundance and we extend this notation to the nodes of G as $c(\hat{s}) = c(s)$. Let $(\hat{s}, \hat{s}')$ be the corresponding edge of G, and let $w(\hat{s}, \hat{s}') = c(\hat{s}) - c(\hat{s}')$ be the difference between the weights of $\hat{s}$ and $\hat{s}'$. Note that $w(\hat{s}, \hat{s}')$ might be negative.

A first idea could be to store (i) the compressed integer $w(\hat{s}, \hat{s}')$ on each edge, and (ii) the explicit value $c(\hat{u})$ whenever $\hat{u}$ has in-degree equal to 0. In this case, we say that $\hat{u}$ is a *sampled node*. At this point, one could retrieve $c(\hat{s})$ also for any non-sampled node $\hat{s}$ by summing $c(\hat{u})$ to the weights of the edges on a path from $\hat{s}$ to $\hat{u}$, where $\hat{u}$ is a sampled node. This solution has two main drawbacks: it is not time-efficient (in the worst case we need to visit the whole graph G to compute one single weight) and it stores one integer per edge, whereas the original graph contained one integer per node (the abundance).

In fact, all we need is a *spanning branching*[2] $T = (V, E)$ of G. The idea is to store (i) the compressed weight $w(\hat{s}, \hat{s}')$ only for the edges of T and (ii) the weights $c(\hat{r}_1), \ldots, c(\hat{r}_t)$ for the t roots $\hat{r}_1, \ldots, \hat{r}_t$ of T. To simplify the description, in the following we will assume that $c(\hat{r}_i) = 0$ for $1 \leq i \leq t$. This information is sufficient to reconstruct the weight of each node of T, but still leaves us with a few issues:

(1) We would like to design a fast data structure to compute the *partial sum* of the values $w(\hat{s}, \hat{s}')$ on the edges of an arbitrary node-to-root path in T;
(2) We have to decide how to encode each $w(\hat{s}, \hat{s}')$ on the edges of T. This may affect the computation of a branching that minimizes such cost;
(3) We do not know to which k-mer (node of G) each node of T corresponds to.

Issues (1) and (2) can be solved with a compressed data structure for answering *partial sums on trees*. In Section 3.1 we first discuss our new compressed solution to partial sums on the special case of arrays, which may be of independent interest. Then, in Section 3.2 we generalize this solution to partial sums on trees. The solution will introduce a cost in bits for encoding each weight $w(\hat{s}, \hat{s}')$. The branching $T = (V, E)$ will then be chosen so as to minimize the sum of the costs on its edges. Issue (3) requires us to keep a mapping between the nodes of G and T. In Section 3.3 we show how to solve this problem with a simplified (and practical) variant of the structure of Section 3.2. Finally, in Section 4 we present experimental results on real DNA datasets, comparing the space usage of our data structure with the state of the art.

3.1 Partial Sums on Arrays

Given a (static) sequence $s[1, n]$ of **positive** integers such that $u = \sum_{i=1}^{n} s[i]$, we would like to design a data structure to support partial sum operations, i.e., to answer queries $\mathsf{sum}(i) = \sum_{j=1}^{i} s[j]$, for any $i \in [n]$. Note that there are succinct data structures that are able to support sum queries in constant time [10, 11, 32] within the information-theoretic lower bound of $\lceil \log \binom{u}{n} \rceil = n \log \frac{u}{n} + O(n)$ bits. These classic results are summarized in the following theorem.

▶ **Theorem 1** ([10, 11, 32]). *Given a (static) sequence $s[1, n]$ of positive integers such that $u = \sum_{i=1}^{n} s[i]$, there exists a data structure that can answer any* sum *query in $O(1)$ time using $n \log(u/n) + O(n)$ bits of space.*

[2] In this paper *spanning branching* stands for *spanning forest of arborescences*, that is, a collection of disjoint directed trees spanning the de Bruijn graph. We note that a spanning forest of *undirected* trees would work as well; however, as we discuss in Section 3.3 this introduces some technical complications.

This space bound is always at most the space required by writing down all partial sums explicitly, i.e., $n\lceil \log u \rceil$ bits. However, it is possible to get a better space bound in terms of a well-known data-aware measure called *gap* measure [15], where the gap measure of the sequence s is defined as $\mathsf{gap}(s) = \sum_{i=1}^{n} \lceil \log(s[i]+1) \rceil$ bits. Notice that in general it is not possible to represent each $s(i)$ with $\lceil \log(s(i)+1) \rceil$ bits, and thus data structures must incur a space overhead on top of the gap measure. Note that the gap measure is always at most the information-theoretic lower bound of $\lceil \log \binom{u}{n} \rceil$ bits: this is because the gap measure is maximised when $s[i] = u/n$ for every i. However, in practice the gap measure could be much smaller than the information-theoretic lower bound.

Gupta et al. [15] designed a data structure that is able to answer each sum query in $O(\log \log u)$ time and uses $\mathsf{gap}(s) + O(n \log \log(u/n) + n \log(u/n)/\log n)$ bits of space. Delpratt et al. [7] defined the delta measure $\Delta(s) = \sum_{i=1}^{n} |\delta(s[i])|$, where $|\delta(x)|$, for $x \geq 1$, is the size in bits of encoding the number x with Elias' δ coding. They present a data structure that answers each sum query in $O(\log \log u)$ time using $\Delta(s) + o(n)$ bits of space. Since for any x we have $\delta(x) = \log x + 2 \log \log x + O(1)$ bits, $\Delta(s) \leq \mathsf{gap}(s) + 2n \log \log(u/n) + O(n)$ bits [7]. This improves over Gupta et al. due to the lower order term in the space bound.

In the remainder of this section we prove the following theorem. Even if our result on de Bruijn graphs requires partial sums on trees instead of arrays, we present this result on arrays here mainly for two reasons. First, it will be used to solve partial sums on trees in Section 3.1. Second, we believe it can be of independent interest since it provides different query-time/space trade-offs for partial sums on arrays.

▶ **Theorem 2.** *Given a (static) sequence $s[1,n]$ of positive integers such that $u = \sum_{i=1}^{n} s[i]$, there exists a data structure that answer sum queries in $O(\frac{1}{\epsilon})$ time using $(1+\epsilon)\mathsf{gap}(s) + O(n \log \log(u/n) + u^{\frac{1}{c}})$ bits of space, for any $0 < \epsilon \leq 1$ and any constant $c \geq 1$.*

We first present a solution using standard ideas and tools. This solution will be then refined to obtain the claimed result. We start by concatenating in a vector D the Elias' δ encoding of each value of s. Elias' γ and δ codes [10] are two standard encodings for positive integers. Notice that binary encoding could also suffice however we believe that the proof is clearer using the Elias's one. Elias' $\gamma(x)$ of an integer $x > 0$ is obtained by writing $\lfloor \log x \rfloor$ in unary, followed by the value $x - 2^{\lfloor \log x \rfloor}$ written with $\lfloor \log x \rfloor$ bits. Elias' δ instead encodes x by writing $\lfloor \log x \rfloor + 1$ with Elias' γ, followed by the value $x - 2^{\lfloor \log x \rfloor}$ written with $\lfloor \log x \rfloor$ bits. We would like also to be able to compute the starting position in D of the encoding of any integers of s. This can be easily done by representing the sequence $L[1,n]$ with the data structure of Theorem 1. The entry $L[i]$ equals the length of the encoding of $s[i]$. This way, the starting position of the encoding of $s[i]$ in D is $\mathsf{sum}(i)$ in L. Note that D and L suffice to decompress any value of the original sequence s in constant time. The space required by these two vectors is bounded by $\mathsf{gap}(s) + O(n \log \log(u/n))$ bits. In order to support sum queries, we can partition s into blocks of size $b = \lceil \frac{1}{\epsilon} \rceil$ each. We store in an array $B = [1, n/b]$ the partial sums in s up to the beginning of each block. This way we can easily support any $\mathsf{sum}(i)$ in $O(\frac{1}{\epsilon})$ time. We first get from B the sum up to the block, say j, that contains the ith element of s. Then, we decompress the elements in the jth blocks one after the other up to the element $s[i]$. This clearly requires $O(1/\epsilon)$ time per operation. However, the space needed by B is $\epsilon n \log u$ bits. This gives an overall space usage of $\mathsf{gap}(s) + \epsilon n \log u + O(n \log \log(u/n) + n)$ bits, which is worse than what is claimed in Theorem 2, since the term $\epsilon n \log u$ may be potentially larger than the term $\epsilon \mathsf{gap}(s)$.

To improve the space bound, we use a different partitioning strategy. The goal is to partition s into variable size blocks such that the cost of encodings the values in each block (but the last one) is between $\frac{\log u}{\epsilon}$ bits and $\frac{3 \log u}{\epsilon}$ bits. This guarantees that there are at most $\frac{\epsilon \mathsf{gap}(s)}{\log u} + 1$ blocks and, thus, the cost of storing the vector B is at most $\epsilon \mathsf{gap}(s) + \log u$ bits.

We observe that a partition with the above characteristics is always possible. For example we can use the following greedy algorithm. We start with an empty block and we process the sequence from left to right. While processing an element we have two possibilities: we either include the element in the current block or we create a new block with this element. We take the former decision only if the overall cost of encoding the elements in the block is less than $\frac{\log u}{\epsilon}$ bits. As the code of an element is at most $\log u + O(\log \log u)$ bits, the above partitioning strategy gives blocks of size between $\frac{\log u}{\epsilon}$ bits and $\frac{3 \log u}{\epsilon}$ bits as claimed.

As now we have variable-size blocks, we need to store a binary vector V of size n bits to keep track of blocks boundaries: an entry $V[i]$ is 1 if and only if the ith element of s is the first element in a block. We use a data structure to support rank/select operations on V in constant time [32].

A query $\mathsf{sum}(i)$ is answered as follows. We first use rank/select operations on V to compute the block j of position i and its offset p within this block. We now need to decode the first p elements of the jth block and sum them to the value $B[j]$. Unfortunately, there may be $\Theta(\log u)$ elements encoded in a block and, thus, we cannot use a trivial decoding. Instead, we conceptually split each block into subblocks such that each subblock is either (i) formed of elements whose overall encoding sizes is no more than $\frac{1}{2} \log u$ bits, or (ii) a single element. This split into subblocks can be done by an easy variant of the greedy partitioning strategy above. Note that there are $O(\frac{1}{\epsilon})$ subblocks per block. We use a table of size $O(\sqrt{u} \log u)$ bits to precompute the sum of any prefix of any subblock given its encoding. This way, computing the sum of the first p elements in jth block costs $O(\frac{1}{\epsilon})$ time as required. Note that the splitting above can be changed to use $\frac{1}{c} \log u$ bits, for any constant $c \geq 1$ instead $\frac{1}{2} \log u$ bits. This way we need a table of size $O(2^{\frac{1}{c} \log u} \log u) = O(u^{\frac{1}{c}} \log u)$ bits and the query time remains $O(\frac{1}{\epsilon})$. By adjusting infinitesimally the constant c, the table takes $O(u^{\frac{1}{c}})$ bits of space.

We conclude by showing how to adapt any solution for the partial sums problem to the variant of the problem in which the sequence s has both positive and negative integers. Let us define $\mathsf{gap}^{\pm}(s) = \sum_{i=1}^{n} \lceil \log(|s[i]| + 1) \rceil$

This variant can be easily reduced to two instances of the original problem as follows. We first use a binary vector $S[1, n]$ that records the signs of the values in s, i.e., $S[i] = 1$ if and only if $s < 0$, $S[i] = 0$ otherwise. We use the data structure by Raman et al. [32] to support constant time rank/select operation on S using $\log \binom{n}{p} + o(n)$ bits of space, where p is the number of positive integers in s. Then, we create two sequences $s^+[1, p]$ and $s^-[1, n - p]$ that store positive and negative integers of s, respectively. Any partial sum query can now be answered with two partial sums queries on s^+ and s^-. Combining this reduction with Theorem 2 yields the following corollary.

▶ **Corollary 3.** *Given a (static) sequence $s[1, n]$ of positive and negative integers such that $u = \sum_{i=1}^{n} |s[i]|$, there exists a data structure that answer each* sum *operation in $O(\frac{1}{\epsilon})$ time using $(1 + \epsilon)\mathsf{gap}^{\pm}(s) + \log \binom{n}{p} + O(n \log \log(u/n) + u^{\frac{1}{c}})$ bits of space, for any $0 < \epsilon \leq 1$ and any constant $c \geq 1$, where p is the number of positive integers.*

3.2 Partial Sums on Trees

Let $T = (V, E)$ be a rooted weighted tree and let $w(i, j)$ denote the (possibly negative) integer weight of edge $(i, j) \in E$. We would like to build a space-efficient data structure that answers partial sum queries of the form $\mathsf{sum}(v) = \sum_{(i,j) \in \Pi(v)} w(i, j)$ on paths, i.e., that reports the sum of the weights on the path $\Pi(v)$ connecting node v to the root, where v is represented in pre-order (the solution for post-order is symmetrical and we do not discuss it).

Chan et al. [5] tackle this problem in a more general framework, where weights belong to a semigroup of size γ. Let $n = |V|$ be the number of nodes. In this model, they provide a data structure taking $n \log \gamma + o(n \log \gamma) + 2n$ bits and answering queries in $O(\alpha(n))$ (inverse Ackermann) time. Note that this space is succinct but does not achieve compression: each weight $w(i, j)$ is stored in $\log \gamma$ bits, independently of its magnitude. Typically, one would like a data structure using much less space: ideally, close to $\mathsf{gap}^{\pm}(T) = \sum_{(i,j)\in E} \lceil \log(|w(i, j)|+1) \rceil$ bits of space.

A folklore approach can reduce partial sums on trees to partial sums on arrays. The idea consists of linearizing the weights according to a Euler tour of the tree. Recall that tree edges are directed towards the root and note that the Euler tour visits each tree edge (i, j) twice: the first time in the direction (j, i) and the second time in the direction (i, j). If $(i, j) \in E$, we assign to the reverse edge (j, i) weight $w(j, i) = -w(i, j)$. We initialize an empty sequence W and, for each edge (i, j) visited along the tour (that is, in the direction $i \to j$), append $-w(i, j)$ at the end of W. Then, it holds that $\mathsf{sum}(v) = \sum_{j=1}^{t} W[j]$, where t is the index corresponding to the first time we see node v in the Euler tour (i.e., $W[t]$ contains the weight $w(\pi(v), v)$, where $\pi(v)$ is the parent of v). This equality holds because, in $W[1], \ldots, W[t]$, all values that correspond to edges not belonging to the path from v to the root appear two times with opposite signs and thus they cancel out.

We can use the data structure of Corollary 3 on the sequence W to have $O(\frac{1}{\epsilon})$ query time. The space is $(2 + \epsilon)\mathsf{gap}^{\pm}(s) + O(n \log \log(u/n) + u^{\frac{1}{c}})$ bits because every gap occurs twice in W.

Another folklore approach uses the Heavy-Light decomposition of the dynamic trees of Sleator and Tarjan [34]. The idea is to split the tree into heavy paths and use a data structure for prefix sums on each of these heavy paths. By virtues of the decomposition, any root-to-a-node path crosses $O(\log n)$ heavy paths and, thus, a query can be answered with $O(\log n)$ prefix sums on arrays. This solution has a better space bound as every gap is represented exactly once, but its query time is logarithmic.

In the remainder of this section we prove the following theorem, which improves over the two folklore solutions above.

▶ **Theorem 4.** *Given a (static) rooted tree $T = (V, E)$ with positive integer weights $w(i, j)$ associated with each edge $(i, j) \in E$ such that $u = \sum_{(i,j)\in E} w(i, j)$, there exists a data structure that answers* sum *queries in $O(\frac{1}{\epsilon})$ time using $(1+\epsilon)\mathsf{gap}(T)+O(n \log \log(u/n)+u^{\frac{1}{c}})$ bits of space, for any $0 < \epsilon \leq 1$ and any constant $c \geq 1$.*

The following corollary generalizes the solution to trees with positive and negative weights.

▶ **Corollary 5.** *Given a (static) rooted tree $T = (V, E)$ with positive and negative integer weights $w(i, j)$ associated with each edge $(i, j) \in E$ such that $u = \sum_{(i,j)\in E} |w(i, j)|$, there exists a data structure that solves queries* sum *in $O(\frac{1}{\epsilon})$ time using $(1 + \epsilon)\mathsf{gap}^{\pm}(T) + \log \binom{n}{p} + O(n \log \log(u/n) + u^{\frac{1}{c}})$ bits of space, for any $0 < \epsilon \leq 1$ and any constant $c \geq 1$, where p is the number of positive integers.*

These results are obtained by adapting to trees the solutions of Theorem 2 and Corollary 3, respectively.

The high level ideas behind our approach are as follows. We first partition the tree into subtrees. Subtrees are either disjoint or intersect only at their common root. Then, we store the sums on the paths from the tree's root to each subtree root. The sum $\mathsf{sum}(v)$ is computed by taking the sum up to the root of v's subtree and summing up the sum of the path within the subtree. The sum within each subtree is computed by using an approach similar to the one in the previous subsection.

The partitioning of the tree is crucial for the space and time efficiency of the solution. For this aim we use a weighted variant of the tree covering procedure described by Geary et al. [14, Sec. 2.1] to decompose T into subtrees. Given any parameter $M > 2$, the original tree covering procedure as described in [14, Sec. 2.1] is used to decompose T into $\Theta(n/M)$ sub-trees containing $O(M)$ nodes each. Two subtrees are either disjoint or intersect only at their common root. The covering is built by visiting the tree in post-order and by grouping nodes into components. The procedure lets the current component grow until its size falls in the range $[M, 3M-4]$. When a component reaches the required size, a new empty component is created. This greedy procedure guarantees that every component (except for the one containing the root) has between M and $3M-4$ nodes. See Geary et al. [14, Sec. 2.1] for more details.

In our solution we use a simple variant of this covering procedure. We fix M to be $\frac{\log u}{\epsilon}$ and we use the greedy procedure above with the only difference that every node i has an encoding cost which equals the size of encoding $w(\pi(i), i)$, where $\pi(i)$ is the parent of i. We build our components to have a cost bounded by M. This way, we can control the size of the encoding of each subtree, which is between $\frac{\log u}{\epsilon}$ bits and $\frac{3\log u}{\epsilon} + 2\log u$ bits. We now need to δ-encode the weights w of a subtree one after the other by following an ordering (specified later) which is suitable for solving queries efficiently. Our goal is, given a node v represented in pre-order, to compute the sum from tree's root to node v. This is solved by summing up: i) the sum from the tree's root to the root of the subtree containing v; ii) the sum of the weights on the path from the subtree's root to node v. Geary et al. [14, Sec. 4.3] show how to find the pre-order number of the root of the subtree containing v in constant time and $o(n)$ bits of space. This index can be used to access a vector containing the sum mentioned at point (i). Point (ii) is computed in $O(\frac{1}{\epsilon})$ time in a way similar to what we have done for arrays. We split the representation of a subtree into $O(\frac{1}{\epsilon})$ subblocks and use tables storing precomputed answers to all the possible subblocks representations. However, the solution here is more involved than the one we used for arrays because we need to compute the sum only of some of the elements of a subblock, i.e., those elements that belong to the query path, and exclude the other elements. This can be done by using a mask that marks the elements belonging to the query path. Given the representation of a subblock and a mask, a precomputed table returns the sum of the marked elements only. The main issue is now to compute the required mask. This can be done by splitting the subtree into subsubtrees whose encoding takes between $\frac{1}{12}\log u$ bits and $\frac{1}{4}\log u$ bits. This is done by using once again the covering procedure described above. We δ-encode the elements in these subsubtrees by visiting nodes in BFS order. We also need to store the topology of each subsubtree. To do that, we can use a balanced sequence representation of each subsubtree so that we need $2n + o(n)$ bits overall. Observe that each subsubtree has at most $\frac{1}{4}\log u$ nodes. Thus, its balanced parentheses representation always takes at most $\frac{1}{2}\log u$ bits and we can use a table of size $O(u^{\frac{1}{2}}\log^2 u)$ bits that, given a subsubtree topology ($\frac{1}{2}\log u$ bits) and a node represented as a pre-order position in the subsubtree topology ($O(\log\log u)$ bits), returns the required mask, i.e., a mask of $\frac{1}{4}\log u$ bits that marks only nodes on the path from the root to the node. Another table of size $O(u^{\frac{1}{2}}\log u)$ bits is used to, given any possible combination of the delta-encoded weights of a subsubtree (at most $\frac{1}{4}\log u$ bits) and any possible mask (at most $\frac{1}{4}\log u$ bits), return the sum of the marked elements only. Note that to perform the above operations we need, given a pre-order node v, to obtain (a) the packed balanced parentheses sequence representation of the subsubtree containing v and (b) the pre-order position of v within its subsubtree. Information (a) can be obtained as done above by using the procedure described by Geary et al. [14, Sec. 4.3]: we locate the pre-order number of the root of the subsubtree and use it as index in an array containing the packed balanced

sequence representations of the subsubtrees ($O(n)$ bits of space). In the same section, Geary et al. [14, Sec. 4.3] show also how to obtain information (b) in constant time and $o(n)$ bits of additional space. Finally note that, as done in the previous section, we can replace the size $\frac{1}{4}\log u$ of the subsubtrees by $\frac{1}{c}\log u$ for any constant $c \geq 1$ and obtain the claimed space bound.

3.3 Adding the dBg Topology

In this section we discuss how to combine a simplified (and practical) variant of the data structure of Corollary 5 with the BOSS de Bruijn graph representation of Bowe et al. [3]. Our final data structure represents a weighted de Bruijn graph in compressed space and supports computing the abundance of any given input k-mer. In Section 4 we present experimental results based on our data structure. Our implementation is available at `https://github.com/nicolaprezza/cw-dBg/`.

Let n and m be the number of nodes and edges of the input de Bruijn graph G, respectively. First of all, we compute a branching T of G minimizing measure $\mathtt{gap}^{\pm}(T)$. This can be achieved using Gabow et al.'s optimized version [13] of Edmonds' algorithm [9], running in $O(m + n\log n)$ time.

The second step is to observe that, by definition of branching, its topology is embedded in the topology of the de Bruijn graph. Consider the list $\hat{s}_1, \ldots, \hat{s}_n$ of the n nodes of the de Bruijn graph sorted by the co-lexicographic order of their corresponding k-mers. Note that this is precisely the order in which nodes are stored in the BOSS representation [3]. Let $e_i^1, \ldots, e_i^{t_i}$ be the t_i incoming edges of node $\hat{s}_i$, and consider the list $e_1, \ldots, e_m = \langle e_i^1, \ldots, e_i^{t_i}\rangle_{i=1,\ldots,n}$ of all the m edges in the graph, sorted by their target node. We keep one bitvector $\mathtt{IN\_B}[1, m]$ marking the edges, in the above order, that are included in the branching. It is clear that the topology of the de Bruijn graph, combined with bitvector `IN_B`, fully specifies the topology of the branching. Importantly, we observe that de Bruijn graphs are usually very sparse, i.e., $m \approx n$. This implies that `IN_B` is composed mostly of bits equal to 1, thus its zero-order entropy $\lceil \log \binom{m}{n} \rceil$ is likely to be very small. We thus store `IN_B` with the zero-order compressed representation of Raman et al. [32], supporting constant-time rank and select queries.

The third step is to build a simplified version of the structure of Corollary 5 over each arborescence of the branching. We do this as follows. Given a parameter ρ (the *sample rate*), $1 \leq \rho \leq n$, we use the tree covering procedure of Geary et al. [14, Sec. 2.1] to decompose each arborescence into $\Theta(n'/\rho)$ subtrees of $O(\rho)$ nodes each, where n' is the number of nodes of the arborescence. We furthermore explicitly store the abundances of the subtree's roots in a vector, sorting them by the order in which nodes appear in the BOSS data structure (that is, by the co-lexicographic order of their corresponding k-mers). The sampled abundances take $O((\frac{n}{\rho} + t)\log u)$ bits, where t is the number of arborescences. An additional zero-order compressed bitvector marks sampled nodes. Using the representation of Raman et al. [32], this bitvector takes $O((\frac{n}{\rho} + t)\log \rho) + o(n)$ bits and supports access, rank, and select operations in constant time.

We store the weight of each edge of T (that is, the difference between the abundances of its endpoint k-mers) using a version of Elias' gamma encoding supporting fast random access queries. First, we convert each (possibly negative) weight w into a positive integer $w' \geq 1$ using the formula

$$w' = \begin{cases} -2w, & \text{if } w < 0 \\ 2w + 1, & \text{otherwise} \end{cases}$$

In practice, using the above conversion, rather than storing explicitly the sign of w in a separate bitvector, has an important practical advantage, as the minimum branching algorithm compresses also the sign of the weights. Indeed, we observed experimentally that this choice compresses each sign to about 0.3 bits on average (rather than 1 bit per sign required when storing them in a plain bitvector). The random access gamma-compressed weights are implemented as follows. We concatenate the binary representation of each w' (devoid of its most significant bit) in an uncompressed bitvector X, and its length in unary in a bitvector Y (for example: $w' = 27$ would be encoded as 1011 in X and 10000 in Y). We compress Y using the representation of Raman et al. [32]. The i-th weight can be extracted from this representation in $O(1)$ time with one select operation on Y and one packed access operation on X. Since we compress Y, it is not hard to see that our representation takes at most $\mathsf{gap}^{\pm}(T) + O(n \log\log(u/n))$ bits of space. Finally we mention that, in order to achieve further compression, we used the compressed bitvector of Raman et al. [32] also to implement the components of the BOSS representation [3].

Let s be a k-mer. To retrieve its abundance, we do the following:

1. We use the de Bruijn graph representation to retrieve the node $\hat{s}$ (in the BOSS representation, a position in the Burrows-Wheeler transform of the graph) corresponding to s. This step takes $O(k)$ time via the backward search algorithm (see [3] for full details) since we assume constant-sized DNA alphabet $\Sigma = \{A, C, G, T\}$.
2. Starting from $\hat{s}$, we move upward towards the root of the tree containing $\hat{s}$ in the branching, stopping as soon as a sampled node $\hat{s}'$ is found (that is, a node whose abundance has been stored explicitly). Each move in the tree is implemented with a constant-time application of the FL function [3]. Along the walk, we sum the abundance of the sampled node $\hat{s}'$ to the weights of the edges on the path connecting $\hat{s}'$ with $\hat{s}$. Overall, this step takes $O(\rho)$ time.

Observe that Step 1 can be avoided if we are already given the representation of a node in the de Bruijn graph (for example, if we are navigating it). Now we can also explain why we use a branching instead of a minimum spanning undirected forest. With a branching, the parent of node $\hat{s}$ (in its corresponding arborescence) is always one of its incoming edges (precisely, the one marked in bitvetor `IN_B`). With an undirected spanning forest, instead, the parent of node $\hat{s}$ could be one of its outgoing edges; this would force us to use an additional bitvector, increasing the overall space and slowing down operations.

To sum up, our implementation offers the following trade-offs. Let G be a de Bruijn graph with m edges and n nodes, $w(\hat{i}, \hat{j}) = c(\hat{i}) - c(\hat{j})$ be the weight associated with each edge $(\hat{i}, \hat{j})$ of G, where $c(\hat{i})$ is the abundance of node $\hat{i}$, $T = (V, E)$ be a branching of G with t connected components minimizing measure $\mathsf{gap}^{\pm}(T) = \sum_{(\hat{i},\hat{j})\in E} \lceil \log(|w(\hat{i}, \hat{j})| + 1) \rceil$, $u = \sum_{(\hat{i},\hat{j})\in E} |w(\hat{i}, \hat{j})|$, and $1 \leq \rho \leq n$ be the user-defined sample rate. Our data structure uses $\mathsf{gap}^{\pm}(T) + O\left(n \log\log(u/n) + \left(\frac{n}{\rho} + t\right) \log u\right) + \log \binom{m}{n}$ bits on top of the compressed BOSS representation [3] and allows us to retrieve:

(1) the abundance of a given k-mer $s \in \Sigma^k$ in $O(k + \rho)$ time, and

(2) the abundance of a given node in the de Bruijn graph (represented as a position in BOSS) in $O(\rho)$ time.

Note that, in practice, ρ is usually chosen to be $\rho \gg k$ (in our experiments, $\rho = 64$ and $k = 28$) so the two query times are not expected to differ much in practical applications.

Our experiments (see next section) highlight that, in practice, the number t of arborescences is negligible compared to the size of the graph.

4 Experiments

In order to get a feeling of its practical value, we implemented a first preliminary version of our compressed representation of weighted de Bruijn graphs. In the following, we refer to this implementatio as cw-dBg; its code is available at `https://github.com/nicolaprezza/cw-dBg`.

Tools. We ran cw-dBg on four datasets and compared the results obtained against the state-of-the-art algorithms for this problem: Squeakr [29], deBGR [27], and the very recent tool fress [33] which appeared online in November 2020. Both Squeakr and deBGR are based on the so-called *counting quotient filter (CQF)* data structure [28]. Squeakr supports two modes: approximate and exact. When run in exact mode, the k-mers are inserted in the CQF using an invertible $2k$-bit hash function. In approximate mode, Squeakr uses a p-bit hash function, with $p \leq 2k$, and thus can have false positives with an error rate depending on the chosen p. deBGR is based on Squeakr and builds an approximate data structure with smaller error rate by increasing the space complexity of approximate Squeakr by just $18\% - 20\%$. Finally, fress [33] implements an approximate data structure called *Set-Min sketch*, inspired by the *Count-Min sketch* data structure [6], that takes advantage of the power-law distribution of the k-mer counts to reduce both the error rate of the returned results and the space usage (by an entropy-compression mechanism). Similarly to Count-Min sketch, Set-Min sketch uses several hash functions to map a given k-mer to sketches of its abundance; at query time, the abundance of the k-mer is computed by combining the retrieved sketches. As shown by the authors, Set-Min sketch is more space-efficient than minimal perfect hash functions and provides better error guarantees compared to equally-dimensioned Count-Min sketches.

Datasets. We selected datasets to cover applications that could be as widely different as possible. In particular, we tested the above tools against the following datasets:

- IAVs Inf. 1 ($1.2Gbp$) and IAVs Inf. 2 ($578Mbp$) are samples from the dataset presented in [1] of human lung cells infected by Influenza A viruses (IAVs) and correspond to a sample of 10 millions reads and a sample of $4,814,148$ reads from chromosome 2, respectively.
- E. coli ($2.3Gbp$) consists of $22,720,100$ Illumina reads of *E. Coli* K-12 strain MG1655 (available in the ENA repository under the following study: PRJEB2323, `https://www.ebi.ac.uk/ena/browser/view/ERR022075`).
- Human ($6.5Gbp$) consists of $63,917,134$ Illumina reads of human RNA (available in the ENA repository under the following study: PRJNA609878, `https://www.ebi.ac.uk/ena/browser/view/SRX7829390`).

Experimental settings. The experiments were carried out on a single core of an 8-core Intel i9-9900K server with 64 GB of RAM and running Linux 5.4.0, 64 bits. Our code is written in C++ and compiled with `gcc` 9.3.0.

As deBGR assumes k-mers of size 28 (which cannot be changed), for consistency we also used this value in all our experiments for all tools. Table 1 reports some characteristics of the datasets. Notice that, as it was previously mentioned, the number of arborescences is very small, when compared to the size of the graph, and thus negligible in practice. We used sample rate $\rho = 64$ for cw-dBg.

deBGR and Squeakr (the latter both in its approximate and exact version) were run using only 1 thread. When running deBGR, the user needs to specify two parameters related to the log of the number of slots in the counting quotient filter. The *CQF size* argument is estimated

Table 1 Characteristics of the de Bruijn graph resulted from each of the dataset for k-mers of size 28. The columns correspond to: number of bases and number of distinct k-mers in the dataset, number of edges and number of arborescences in the de Bruijn graph, average and max abundance of the k-mers and the total number of distinct abundances in the dataset (which influences fress' space).

Dataset	# bases	# k-mers	# edges	# Arb.	Avg ab.	Max ab.	# ab.
IAVs Inf. 1	1,200,000,000	57,629,309	65,880,010	259	16.14	607,028	13,452
IAVs Inf. 2	577,697,760	34,914,869	41,295,094	100	12.82	55,110	9,007
E. coli	2,317,450,200	87,414,957	281,845,871	1	19.49	8,520,260	2,657
Human	6,455,630,534	362,818,017	463,413,958	1226	13.04	11,054,076	21,472

using the lognumslots.sh script provided by the authors and corresponds approximately to the log of the number of k-mers. The *exact CQF size* argument is estimated as 20% less then the *CQF size* (personal communication with the authors of deBGR).

Finally, fress is an approximate tool and allows to specify a parameter related to the error rate: in our experiments, we ran it using an error rate of $\epsilon = 0.01$, as used in fress' paper [33]. Notice that the error rate estimates the expected number of collision which could possibly lead to output a wrong abundance. An error rate of 0.01 means that in N queries, the number of expected collisions in the hash table is ϵN.

Experimental results. In Table 2 we present the detailed space required by our representation for each of the datasets considered, while Table 3 reports the space usage of all the tools considered.

The bit/k-mer of each component in Tables 2 and 3 were computed using the number of distinct k-mers over the alphabet $\{A, C, G, T\}$ present in the original dataset. We note that this is not the same calculation performed by Bowe et al. [3] when estimating the size of their BOSS representation: in that case, the authors divide by the number of *edges* in the de Bruijn graph when also including dummy (that is, left-padded) k-mers, obtaining roughly 4 bits per edge. In fact, the two values (number of edges and number of k-mers) are close to each other only when the number of dummy k-mers is small (which is not always the case, as explained below). We believe that dividing the total bit-size by the number of distinct k-mers appearing in the dataset is more general, since it applies to any k-mer counting data structure like the ones shown in Table 3. Notice that for this reason in Table 2 the space reported for the BOSS representation of the E. coli dataset is larger than the typical 4 bit/edge reported by the authors of BOSS [3] as the number of dummy k-mers for this dataset is significantly large (approximately equal to the number of distinct k-mers).

The results in Table 2 and Table 3 show that our representation uses at most 4.75 bit/k-mer (and as little as 1.42 bit/k-mer) to encode the abundances. Even when adding the BOSS representation (required to map k-mers to their corresponding abundances), we obtain at most 12.64 bit/k-mer (and as little as 4.14 bit/k-mer) while the exact version of Squeakr uses 83.26 bit/k-mer in the best scenario. In almost all our experiments cw-dBg required substantially much less space, an by several orders of magnitude, than all the other tools. The only exception occured in the E. coli dataset, where fress performed slightly better than cw-dBg by using 10.95 bit/k-mer (versus 12.64 bit/k-mer of cw-dBg).

We stress out, however, that fress achieves this slight space saving at the cost of returning a wrong abundance 1% of the times in the worst case. Indeed, if its error rate is lowered from 0.01 to 0.005, fress uses 14.6 bit/k-mer on this dataset. The efficiency of fress on this dataset

Table 2 The space required for each dataset by our representation of the weighted de Bruijn graph. The columns correspond to: the size of the file containing the final representation, the space required by the BOSS representation of the de Bruijn graph; the space required by the compressed abundances divided in the space required by the delta-compressed weights (stored on the edges of the branching) and the one for the branching topology and the sampled abundances on the root of each subtree; the total space required by our representation and the average query time in microseconds for retrieving the abundance of a given k-mer (represented as a string). Space is measured in bits per distinct k-mer (see Table 1) and we used sample rate $\rho = 64$.

Dataset	Final size (MB)	BOSS (bit/k-mer)	Compressed ab. (bit/k-mer)			Total size (bit/k-mer)	Query time (μs/query)
			weights	branching	total		
IAVs Inf. 1	29	2.73	0.95	0.48	1.43	4.15	47.56
IAVs Inf. 2	20	2.87	1.19	0.53	1.72	4.59	44.24
E. coli	132	7.88	2.93	1.82	4.75	12.64	62.34
Human	223	3.14	1.40	0.61	2.01	5.14	67.42

Table 3 For each dataset we report the space required for the representation of the weighted de Bruijn graph by each of the considered tools.

	IAVs Inf. 1		IAVs Inf. 2		E. coli		Human	
	MB	bit/k-mer	MB	bit/k-mer	MB	bit/k-mer	MB	bit/k-mer
cw-dBg	29	4.15	20	4.59	132	12.64	223	5.14
Squeakr approx.	325	47.19	179	42.80	325	31.11	1126	24.05
Squeakr exact	965	140.40	499	119.75	965	92.57	3686	83.26
deBGR	912	132.46	376	89.55	912	87.32	5529	119.00
fress	733	106.56	135	32.35	115	10.95	733	16.90

is explained by the fact that the number of distinct k-mer abundances is much smaller than in the other three datasets (see Table 1). This is clearly the case where we expect fress to work well, as it fully takes advantage of the power-law distribution of the abundances by entropy-compressing their values.

Finally, we estimated the average query time of cw-dBg, by querying a set of $5,000$ randomly chosen 28-mers. The largest measured query time for cw-dBg on our machine was 67.42 μs/query on the Human dataset (see Table 2 for the query times on all datasets). As expected, hash-based data structures have better query time: on the same Human dataset, Squeakr answers queries in average 0.19 μs/query using 1 thread. However, as we discussed before, hash-based structures have a much higher space usage, in addition to being approximate. The exact variant of Squeakr uses orders of magnitude more space than cw-dBg.

5 Conclusions and open problems

We propose a new compressed representation for weighted de Bruijn graphs based on the idea of delta-encoding the variations of k-mer abundances on a spanning branching of the graph. As a by-product of independent interest, we exhibit efficient compressed data structures for answering partial sums on edge-weighted trees.

We show both theoretically and experimentally that our approach uses significantly less memory than the one used by the state-of-the-art exact representations. Our de Bruijn graph representation is general, in other words it is not restricted by the application (e.g., variation finding or RNA-seq), and can be used as part of any algorithm that represents NGS data

with de Bruijn graphs. Future extensions will include implementing a strategy similar to the one used by fress in order to take advantage of the small number of distinct abundance observed in practice. The basic idea is to use a table storing all the distinct abundances sorted and, for each node, encode its abundance's offset in the table with our data structure (that is, storing the deltas of these offsets rather than of the original abundances). We also plan to integrate our structure in a usable bioinformatics tool.

References

1 Usama Ashraf, Clara Benoit-Pilven, Vincent Navratil, Cécile Ligneau, Guillaume Fournier, Sandie Munier, Odile Sismeiro, Jean-Yves Coppée, Vincent Lacroix, and Nadia Naffakh. Influenza virus infection induces widespread alterations of host cell splicing. *NAR Genomics and Bioinformatics*, 2(4), November 2020.

2 Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son Pham, Andrey D. Prjibelski, Alexey V. Pyshkin, Alexander V. Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. Spades: A new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012. PMID: 22506599. doi:10.1089/cmb.2012.0021.

3 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn Graphs. In Ben Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics*, pages 225–235, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

4 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

5 Timothy M. Chan, Meng He, J. Ian Munro, and Gelin Zhou. Succinct indices for path minimum, with applications. *Algorithmica*, 78(2):453–491, 2017. doi:10.1007/s00453-016-0170-7.

6 Graham Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *Proceedings of the 2005 SIAM International Conference on Data Mining, SDM 2005, Newport Beach, CA, USA, April 21-23, 2005*, pages 44–55, 2005. doi:10.1137/1.9781611972757.5.

7 O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Compressed prefix sums. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 4362 of *Lecture Notes in Computer Science*, pages 235–247. Springer, 2007. doi:10.1007/978-3-540-69507-3_19.

8 Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, January 2015. doi:10.1093/bioinformatics/btv022.

9 Jack Edmonds. Optimum branchings. *Journal of Research of the national Bureau of Standards Section B*, 71(4):233–240, 1967.

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. memorandum 61. *Computer Structures Group, Project MAC, MIT, Cambridge, Mass., nd*, 1971.

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

13 Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

14 Richard F Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms (TALG)*, 2(4):510–534, 2006.

15 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.

16 David L. Hyten, Steven B. Cannon, Qijian Song, Nathan Weeks, Edward W. Fickus, Randy C. Shoemaker, James E. Specht, Andrew D. Farmer, Gregory D. May, and Perry B. Cregan. High-throughput snp discovery through deep resequencing of a reduced representation library to anchor and orient scaffolds in the soybean whole genome sequence. *BMC Genomics*, 11(1):38, 2010. doi:10.1186/1471-2164-11-38.

17 Katerina Kechris, Yee Hwa Yang, and Ru-Fang Yeh. Prediction of alternatively skipped exons and splicing enhancers from exon junction arrays. *BMC Genomics*, 9(1):551, 2008. doi:10.1186/1471-2164-9-551.

18 Ruiqiang Li, Yingrui Li, Xiaodong Fang, Huanming Yang, Jian Wang, Karsten Kristiansen, and Jun Wang. Snp detection for massively parallel whole-genome resequencing. *Genome Research*, 19(6):1124–1132, 2009. doi:10.1101/gr.088013.108.

19 Li Fan, Pei Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000. doi:10.1109/90.851975.

20 Leandro Lima, Blerina Sinaimeri, Gustavo Sacomoto, Hélène Lopez-Maestre, Camille Marchet, Vincent Miele, Marie-France Sagot, and Vincent Lacroix. Playing hide and seek with repeats in local and global de novo transcriptome assembly of short RNA-seq reads. *Algorithms Mol Biol*, 12, 2017.

21 Binghang Liu, Yujian Shi, Jianying Yuan, Xuesong Hu, Hao Zhang, Nan Li, Zhenyu Li, Yanxiang Chen, Desheng Mu, and Wei Fan. Estimation of genomic characteristics by analyzing k-mer frequency in de novo genome projects. *arXiv preprint*, 2013. arXiv:1308.2012.

22 Yongchao Liu, Jan Schröder, and Bertil Schmidt. Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data. *Bioinformatics*, 29(3):308–315, November 2012. doi:10.1093/bioinformatics/bts690.

23 Hélène Lopez-Maestre, Lilia Brinza, Camille Marchet, Janice Kielbassa, Sylvère Bastien, Mathilde Boutigny, David Monnin, Adil El Filali, Claudia Marcia Carareto, Cristina Vieira, Franck Picard, Natacha Kremer, Fabrice Vavre, Marie-France Sagot, and Vincent Lacroix. SNP calling from RNA-seq data without a reference genome: identification, quantification, differential analysis and impact on the protein sequence. *Nucleic Acids Research*, 44(19):e148–e148, 2016. doi:10.1093/nar/gkw655.

24 Camille Marchet, Christina Boucher, Simon J Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Research*, 31(1):1–12, 2021.

25 Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, January 2011. doi:10.1093/bioinformatics/btr011.

26 Ali Mortazavi, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. Mapping and quantifying mammalian transcriptomes by rna-seq. *Nature Methods*, 5(7):621–628, 2008. doi:10.1038/nmeth.1226.

27 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics*, 33(14):i133–i141, July 2017. doi:10.1093/bioinformatics/btx261.

28 Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 775–787, New York, NY, USA, 2017. Association for Computing Machinery.

29 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, October 2017. doi:10.1093/bioinformatics/btx636.

30 Rob Patro, Geet Duggal, Michael I. Love, Rafael A. Irizarry, and Carl Kingsford. Salmon provides fast and bias-aware quantification of transcript expression. *Nature methods*, 14(4):417–419, 2017. URL: https://pubmed.ncbi.nlm.nih.gov/28263959.

31 Pavel A. Pevzner. 1-tuple dna sequencing: Computer analysis. *Journal of Biomolecular Structure and Dynamics*, 7(1):63–73, 1989. PMID: 2684223. doi:10.1080/07391102.1989.10507752.

32 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

33 Yoshihiro Shibuya and Gregory Kucherov. Set-min sketch: a probabilistic map for power-law distributions with application to k-mer annotation. *SeqBIM 2020*, 2020. doi:10.1101/2020.11.14.382713.

34 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

35 Raluca Uricaru, Guillaume Rizk, Vincent Lacroix, Elsa Quillery, Olivier Plantard, Rayan Chikhi, Claire Lemaitre, and Pierre Peterlongo. Reference-free detection of isolated SNPs. *Nucleic Acids Research*, 43(2):e11–e11, November 2014. doi:10.1093/nar/gku1187.

36 Reda Younsi and Dan MacLean. Using $2k + 2$ bubble searches to find single nucleotide polymorphisms in k-mer graphs. *Bioinformatics*, 31(5):642–646, October 2014. doi:10.1093/bioinformatics/btu706.

37 Birney E. Zerbino DR. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, 18(5):821–9., 2008. PMID: 18349386. doi:doi:10.1101/gr.074492.107.

Optimal Construction of Hierarchical Overlap Graphs

Shahbaz Khan
University of Helsinki, Finland

Abstract

Genome assembly is a fundamental problem in Bioinformatics, where for a given set of overlapping substrings of a genome, the aim is to reconstruct the source genome. The classical approaches to solving this problem use assembly graphs, such as *de Bruijn graphs* or *overlap graphs*, which maintain partial information about such overlaps. For genome assembly algorithms, these graphs present a trade-off between overlap information stored and scalability. Thus, Hierarchical Overlap Graph (HOG) was proposed to overcome the limitations of both these approaches.

For a given set P of n strings, the first algorithm to compute HOG was given by Cazaux and Rivals [IPL20] requiring $O(||P|| + n^2)$ time using superlinear space, where $||P||$ is the cumulative sum of the lengths of strings in P. This was improved by Park et al. [SPIRE20] to $O(||P|| \log n)$ time and $O(||P||)$ space using segment trees, and further to $O(||P|| \frac{\log n}{\log \log n})$ for the word RAM model. Both these results described an open problem to compute HOG in optimal $O(||P||)$ time and space. In this paper, we achieve the desired optimal bounds by presenting a simple algorithm that does not use any complex data structures. At its core, our solution improves the classical result [IPL92] for a special case of the All Pairs Suffix Prefix (APSP) problem from $O(||P|| + n^2)$ time to optimal $O(||P||)$ time, which may be of independent interest.

2012 ACM Subject Classification Mathematics of computing → Trees; Theory of computation → Data compression; Theory of computation → Pattern matching

Keywords and phrases Hierarchical Overlap Graphs, String algorithms, Genome assembly

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.17

Related Version *Previous Version*: `https://arxiv.org/abs/2102.02873` [14]

Funding This work was funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFEBIO).

Acknowledgements I would like to thank Alexandru I. Tomescu for helpful discussions, and for critical review and insightful suggestions which helped me in refining the paper. I would also like to thank Veli Mäkinen for pointing out the similarity with the classical result for APSP problem.

1 Introduction

Genome assembly is one of the oldest and most fundamental problems in Bioinformatics [21]. Due to practical limitations, sequencing an entire genome as a single complete string is not possible, rather a collection of the *substrings* of the genome (called *reads*) are sequenced. The goal of a sequencing technology is to produce a collection of reads that cover the entire genome and have sufficient overlap amongst the reads. This allows the source genome to be reconstructed by ordering the reads using this overlap information. The genome assembly problem thus aims at computing the source genome given such a collection of overlapping reads. Most approaches of genome assembly capture this overlap information into an *assembly graph*, which can then be efficiently processed to assemble the genome. The prominent approaches use assembly graphs such as *de Bruijn graphs* [22] and *Overlap graphs* (also called string graphs [17]), which have been shown to be successfully used in various practical assemblers [28, 3, 18, 2, 23, 24].

The de Bruijn graphs are built over k length substrings (or k-mers) of the reads as nodes, and arcs denoting $k-1$ length overlaps among the k-mers. Their prominent advantage is that their size is linear in that of the input. However, their limitations include losing information about the relationship of k-mers with the reads, and in general not being able to represent overlaps of size other than $k-1$ among the reads (except [7, 5, 4]). On the other hand, Overlap graphs have each read as a node, and edges between every pair of nodes represent their corresponding maximum overlap. In practice, only the edges having certain threshold value of overlap are considered. Though they store more overlap information than de Bruijn graphs, they do not maintain whether two pairs of strings have the same overlap. Moreover, they are inherently quadratic in size in the worst case, and computing the edge weights (even optimally [13, 26, 16]) is difficult in practice for large data sets.

As a result, Hierarchical Overlap Graphs (HOG) were proposed in [9, 10] as an alternative to overcome such limitations of the two types of assembly graphs. The HOG has nodes for all the longest overlaps between every pair of strings, and edges connecting strings to their suffix and prefix, using linear space. Note that Overlap graphs have edges representing longest overlaps between strings requiring quadratic size, whereas HOG has additional nodes for longest overlaps between strings requiring linear size by exploiting pairs of strings having the same longest overlaps. Thus, it is a promising alternative to both de Bruijn graph and Overlap graph to better solve the problem of genome assembly. Also, since it maintains if two pairs of strings have the same overlap, it also has the potential to better solve the approximate *shortest superstring problem* [27] having applications in both genome assembly and data compression [25, 6]. Some applications of HOG have been studied in [9, 8].

Cazaux and Rivals [10] presented the first algorithm to build HOG efficiently. They showed how HOG can be computed for a set of n strings P in $O(||P|| + n^2)$ time, where $||P||$ represents the cumulative sum of lengths of strings in P. However, they required $O(||P|| + n \times \min(n, \max_{p \in P} |p|))$ space, which is superlinear in input size. Park et al. [20] improved it to $O(||P|| \log n)$ time requiring linear space using Segment trees [11], assuming a constant sized character set. For the word RAM model, they further improved it to $O(||P|| \frac{\log n}{\log \log n})$ time. For practical implementation, both these results build HOG using an intermediate Extended HOG (EHOG) which reduces the memory footprint of the algorithm. In both the results, the *bottleneck* is solving a special case of All Pairs Suffix Prefix (APSP) problem. Given a set P of n strings, the goal of the APSP problem is to compute the maximum overlaps between every pair of strings. This classical problem was optimally solved by Gusfield et al. [13] using $O(||P|| + n^2)$ time and $O(||P||)$ space, where the solution is reported for the n^2 pairs. However, for computing HOG we only require the set of *maximum overlaps*, and not their association with the corresponding pairs of strings, making the result suboptimal due to the extra $O(n^2)$ factor. Also, both these results [10, 20] mentioned as an open problem the construction of HOG using optimal $O(||P||)$ time and space. We answer this open question positively and solve the special case of APSP optimally as follows.

▶ **Theorem 1** (Optimal HOG). *For a set of strings P, the Hierarchical Overlap Graph can be computed using $O(||P||)$ time and space.*

Moreover, unlike [20] our algorithm does not use any complex data structures for its implementation. Also, we do not assume any limitations on the character set. Finally, like [10, 20] our algorithm can also use EHOG as an intermediate step for improving memory footprint in practice. Note that the size EHOG and HOG can even be identical for some instances, but their ratio can tend to infinity for some families of graphs [10]. Thus, despite the existence of optimal algorithm for computing EHOG, an optimal algorithm for computing HOG is significant from both theoretical and practical viewpoints.

Note. Another result [19] simultaneously achieve the same optimal bound by reducing the problem to computing *borders* [15]. However, our result is simpler and more self-contained.

Outline of the paper. We first describe notations and preliminaries that are used in our paper in Section 2. In Section 3, we briefly describe the previous approaches to compute HOG. Thereafter, Section 4 describes our core result in three stages for simplicity of understanding, each building over the previous, to give the optimal algorithm. Finally, we present the conclusions in Section 5.

2 Preliminaries

Given a finite set $P = \{p_1, ..., p_n\}$ of n non-empty strings over a finite set of characters, we denote the size of a string p_i by $|p_i|$ and the cumulative size of P by $||P|| = \sum_{i=1}^{n} |p_i|$ ($\geq n$ as strings are not empty). For a string p, any substring that starts from the first character of p is called a *prefix* of p, whereas any substring which ends at the last character of p is called a *suffix* of p. A prefix or suffix of p is called *proper* if it is not same as the whole p. For an ordered pair of string (p_1, p_2), a string is called their *overlap* if it is both a proper suffix of p_1 and a proper prefix of p_2, where $ov(p_1, p_2)$ denotes the *longest* such overlap. Also, for the set of strings P, $Ov(P)$ denotes the set of all $ov(p_i, p_j)$ for $1 \leq i, j \leq n$. An empty string is denoted by ϵ. We also use the notions of HOG, EHOG and the Aho-Corasick trie as follows.

▶ **Definition 2** (Hierarchical Overlap Graph [10]). *Given a set of strings $P = \{p_1, \cdots, p_n\}$, its Hierarchical Overlap Graph is a directed graph $\mathcal{H} = (V, E)$, where*

- *$V = P \cup Ov(P) \cup \{\epsilon\}$ and $E = E_1 \cup E_2$, having*
- *$E_1 = \{(x, y) : x$ is the longest proper prefix of y in $V\}$ as* tree edges*, and*
- *$E_2 = \{(x, y) : y$ is the longest proper suffix of x in $V\}$ as* suffix links*.*

The *extended HOG* of P (referred as $\mathcal{E}$) is also similarly defined [10], having additional nodes corresponding to every overlap (not just longest) between each pair of strings in P, with the same definition of edges. The construction of both these structures uses the Aho-Corasick Trie [1] which is computable in $O(||P||)$ time and space. The Aho-Corasick Trie of P (referred as $\mathcal{A}$) contains all prefixes of strings in P as nodes, with the same definition for edges. All these structures are essentially *trees* having the empty string ϵ as the root, and the strings of P as its *leaves*. A *tree edge* (x, y) is labelled with the substring of y not present in x. Hence, despite being a graph due to the presence of *suffix links* (also called *failure links*), we abuse the notions used for tree structures when applying to $\mathcal{A}, \mathcal{E}$ or $\mathcal{H}$ (ignoring suffix links). Also, while referring to a node v of $\mathcal{A}, \mathcal{E}$ or $\mathcal{H}$, we represent its corresponding string with v as well.

Consider Figure 1 for a comparison of $\mathcal{A}$, $\mathcal{E}$ and $\mathcal{H}$ for $P = \{aabaa, aadbd, dbdaa\}$. Since $\mathcal{A}$ contains all prefixes as nodes, the tree edges have labels of a single character. However, $\mathcal{E}$ contains all overlaps among strings of P, so it can potentially have fewer internal nodes ($\{a, aa, db, dbd\}$) than $\mathcal{A}$. Further, $\mathcal{H}$ contains only longest overlaps so it can potentially have even fewer internal nodes ($\{aa, dbd\}$).

Now, to compute $\mathcal{E}$ or $\mathcal{H}$ one must only remove some internal nodes from $\mathcal{A}$ and adjust the edge labels accordingly. This requires the computation of all overlaps among strings in P for $\mathcal{E}$, which is further restricted to only the longest overlaps for $\mathcal{H}$. For a string $p_i \in P$ (leaf of $\mathcal{A}$), all its prefixes are its ancestors in $\mathcal{A}$, whereas all its suffixes are on the path following the suffix links from it (referred as *suffix path*). Thus, every internal node is implicitly the prefix of its descendant leaves, and to be an overlap it must merely be a suffix of some string

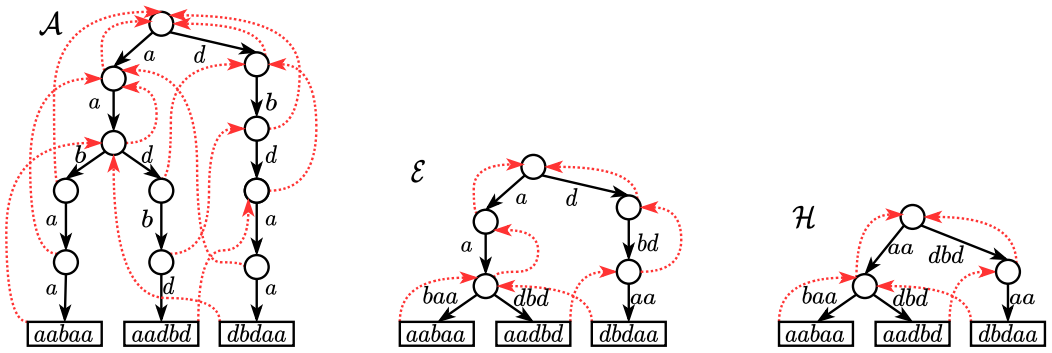


Figure 1 Given $P = \{aabaa, aadbd, dbdaa\}$, the figure shows from left to right the Aho-Corasick Trie ($\mathcal{A}$), Extended Hierarchical Overlap Graph ($\mathcal{E}$) and Hierarchical Overlap Graph ($\mathcal{H}$) of P.

in P [27]. Hence to compute internal nodes of $\mathcal{E}$ (or overlap) from $\mathcal{A}$ one simply traverses the suffix paths from all the leaves of $\mathcal{A}$, and remove the non-traversed internal nodes (see Figure 1). However, to compute $\mathcal{H}$ from $\mathcal{A}$ (or $\mathcal{E}$) we need to find only the longest overlaps, which is equivalent to solving a special case of the APSP problem, requiring only the set of all maximum overlaps. We use the following criterion (also used by [13]) to identify the internal nodes of $\mathcal{H}$.

▶ **Lemma 3** ([13]). *An internal node v in $\mathcal{A}$ (or $\mathcal{E}$) of P, is $ov(p_i, p_j)$ for two strings $p_i, p_j \in P$ iff v is an overlap of (p_i, p_j) and no descendant of v is an overlap of (p_i, p_j).*

Proof. The ancestor of a node v in $\mathcal{A}$ is its proper prefix and hence is shorter than v. Since two internal nodes of $\mathcal{A}$ which are both overlaps of (p_i, p_j), are prefixes of p_j and hence have an ancestor-descendant relationship, where the descendant is longer in length. Thus, the longest overlap $ov(p_i, p_j)$ cannot have a descendant which is an overlap of (p_i, p_j). ◀

Hence to compute $Ov(P)$ (or nodes of $\mathcal{H}$), we need to check each internal node v if it is the lowest overlap (in $\mathcal{A}$) for some pair (p_i, p_j). This implies that v is a suffix of some p_i, such that for some descendant leaf p_j, no suffix of p_i is on path from v to p_j (see Figure 1).

3 Previous results

Cazaux and Rivals [10] were the first to study $\mathcal{H}$, where they used $\mathcal{E}$ [8] as an intermediate step in the computation of $\mathcal{H}$. They showed that $\mathcal{E}$ can be constructed in $O(||P||)$ time and space from $\mathcal{A}$ [1], which itself is computable in $O(||P||)$ time and space. In order to compute $\mathcal{H}$, the main *bottleneck* is the computation of $Ov(P)$ (i.e. solving APSP), after which we simply remove the internal nodes not in $Ov(P)$ from $\mathcal{E}$ (or $\mathcal{A}$), in $O(||P||)$ time and space. They gave an algorithm to compute $Ov(P)$ in $O(||P|| + n^2)$ time using $O(||P|| + n \times \min(n, \max\{|p_i|\}))$ space. This procedure was recently improved by Park et al. [20] to require $O(||P|| \log n)$ time and $O(||P||)$ space using segment trees, assuming constant sized character set. For the word RAM model they further improve the time to $O(||P|| \frac{\log n}{\log \log n})$. The main ideas of the previous results can be summarized as follows.

Computing $Ov(P)$ in $O(||P|| + n^2)$ time [10]

The algorithm computes $Ov(P)$ by considering the internal nodes in a bottom-up manner, where a node is processed after its descendants. Firstly, for each internal node u, they compute the list $R_l(u)$ (called $\mathcal{L}_u$ in our algorithm) of all leaves having u as a suffix. Now,

while processing a node u, they check whether $u = ov(v, x)$, i.e., u is a suffix of some leaf v such that the path to at least one of u's descendant leaf (say x) does not have a suffix of v. To perform this task, they maintain a bit-vector for all leaves (suffix v), which is marked if no such descendant path exists from u for such leaves. For a leaf v, the bit is implicitly marked if all children of u have the bit for v marked. Otherwise, if $v \in R_l(u)$ it is marked adding u to $\mathcal{H}$, else left unmarked. The space requirement is dominated by that of this bit-vector, and it is computed only for the branching nodes, taking total $O(||P|| + n^2)$ time.

Computing $Ov(P)$ in $O(||P|| \log n)$ time [20]

The algorithm firstly orders the strings in P lexicographically in $O(||P||)$ time (requires constant sized character set). This allows them to define an interval of leaves which are the descendants of each internal node in $\mathcal{E}$. Now, for each leaf v (suffix) they start with an unmarked array corresponding to all leaves (prefix). Then starting from v they follow its suffix path and at each internal node u, check if some descendant leaf x (prefix) is unmarked. In such a case $u = ov(v, x)$ and hence u is added to $\mathcal{H}$. Before moving further in the next suffix path the interval corresponding to all the descendant leaves (prefix) of u is marked in the array. Since both query and update (mark) over an interval can be performed in $O(\log n)$ time using a segment tree, the total time taken is $O(||P|| \log n)$ using $O(||P||)$ space.

4 Our algorithm

Our main contribution is an alternative procedure to compute $Ov(P)$ in $O(||P||)$ time and space which results in an optimal algorithm for computing $\mathcal{H}$ for P in $O(||P||)$ time and space. Our overall approach is similar to that of the original algorithm [10] with the exception of a procedure to mark the internal nodes that belong to $\mathcal{H}$, i.e., Mark$\mathcal{H}$. The algorithm except for the procedure Mark$\mathcal{H}$ takes $O(||P||)$ time and space (also shown in [10]). We describe our algorithm for Mark$\mathcal{H}$ in three stages, first for a single prefix leaf requiring $O(||P||)$ time, and then for all prefix leaves requiring overall $O(||P|| + n^2)$ time, and finally improving it to overall $O(||P||)$ time, which is optimal. The algorithm can be applied to any of $\mathcal{A}$ or $\mathcal{E}$, both computable in $O(||P||)$ time and space.

Note: The second stage of our algorithm is equivalent to [13], and achieves the same bounds as [10] for computing $\mathcal{H}$, though using a simpler technique and linear space.

4.1 Outline of Approach

We first describe our overall approach in Algorithm 1. After computing $\mathcal{A}$, for each internal node v, we compute the list $\mathcal{L}_v$ of all the leaves having v as its suffix. As described earlier, this can be done by following the suffix path of each leaf x, adding x to $\mathcal{L}_y$ for every internal node y on the path. Using this information of suffix (in $\mathcal{L}_v$) and prefix (implicit in $\mathcal{A}$) we mark the nodes of $\mathcal{A}$ to be added in the HOG $\mathcal{H}$. We shall describe this procedure Mark$\mathcal{H}$ later on. Thereafter, in order to compute $\mathcal{H}$ we simply merge the unmarked internal nodes of $\mathcal{A}$ with its parents. This process is carried on using a DFS traversal of $\mathcal{A}$ (ignoring suffix links) where for each unmarked internal node v, we move all its edges to its parent, prepending their labels with the label of the parent edge of v.

As previously described, $\mathcal{A}$ can be computed in $O(||P||)$ time and space [1]. Computing $\mathcal{L}_v$ for all $v \in \mathcal{A}$ requires each leaf p_i to follow its suffix path in $O(|p_i|)$ time, and add p_i to at most $|p_i|$ different $\mathcal{L}_y$, requiring total $O(||P||)$ time for all $p_i \in P$. This also limits the

Algorithm 1 HIERARCHICAL OVERLAP GRAPHS.

$\mathcal{A} \leftarrow$ Aho-Corasik Trie of P `// Trie with suffix links`

foreach *internal node* v *of* $\mathcal{A}$ **do** $\mathcal{L}_v \leftarrow \emptyset$ `// List of leaves with suffix` v

foreach *leaf* x *of* $\mathcal{A}$ **do** `// Compute all` $\mathcal{L}_v$

  $y \leftarrow$ Suffix link of x in $\mathcal{A}$

  while $y \neq \epsilon$ **do** `//` ϵ `is the root of` $\mathcal{A}$

    Add x to $\mathcal{L}_y$

    $y \leftarrow$ Suffix link of y in $\mathcal{A}$

$in\mathcal{H} \leftarrow$Mark$\mathcal{H}(\mathcal{A}, \mathcal{L})$ `// Procedure to mark nodes of` $\mathcal{H}$ `in flags` $in\mathcal{H}$

foreach *node* $v \in \mathcal{A}$ *in DFS order* **do** `// Compute` $\mathcal{H}$

  if $in\mathcal{H}[v] = 0$ **then** Merge v with its parent

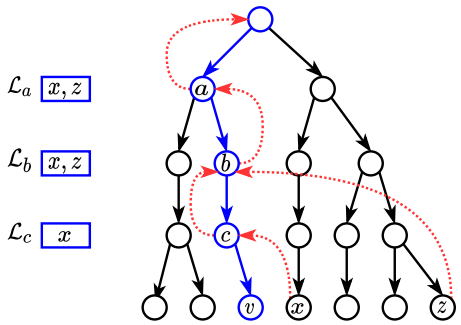


Figure 2 Overlaps of v with all leaves, where $c = ov(x, v)$ and $b = ov(z, v)$ are in $Ov(P)$.

size of $\mathcal{L}_v$ for all $v \in \mathcal{A}$ to $O(||P||)$. Since merge operation on a node v requires $O(deg(v))$ cost, computing $\mathcal{H}$ using $in\mathcal{H}$ requires total $O(|\mathcal{A}|) = \mathcal{O}(||\mathcal{P}||)$ time as well. Thus, we have the following theorem (also proved in [10]).

▶ **Theorem 4.** *For a set of strings P, the computation of Hierarchical Overlap Graph except for Mark$\mathcal{H}$ operation requires $O(||P||)$ time and space.*

4.2 Marking the nodes of $\mathcal{H}$

We shall describe our procedure to mark the nodes of $\mathcal{H}$ in three stages for simplicity of understanding. First, we shall describe how to mark all internal nodes representing all longest overlaps $ov(\cdot, v)$ from a single leaf v (prefix) in $\mathcal{A}$, using $O(||P||)$ time. Thereafter, we extend this to compute such overlaps from all leaves in $\mathcal{A}$ together using $O(||P|| + n^2)$ time (equivalent to [13]). Finally, we shall improve this to our final procedure requiring optimal $O(||P||)$ time. All the three procedures require $O(||P||)$ space.

Marking all nodes $ov(\cdot, v)$ for a leaf v

In order to compute all longest overlaps of a leaf v (see Figure 2), we need to consider all its prefixes (ancestors in $\mathcal{A}$) according to Lemma 3. Here the internal nodes a, b and c are prefixes of v and also suffixes of x, whereas z only has suffixes a and b. Thus, we have $\mathcal{L}_a = \mathcal{L}_b = \{x, z\}$ and $\mathcal{L}_c = \{x\}$. Thus, given that a, b and c are ancestors of v, a and b are

valid overlaps of (x, v) and (z, v), whereas c is only a valid overlap of (x, v). Using Lemma 3, for being the longest overlap of a pair of strings, no descendant should be an overlap of the same pair of strings. Hence, $c = ov(x, v)$ and $b = ov(z, v)$, but a is not the longest overlap for any pair of strings because of b and c. Processing $\mathcal{L}_u$ for all nodes on the ancestors of the leaf (prefix) requires $O(||P||)$ time. Thus, a simple way to mark all the longest overlaps of strings with prefix v in $O(||P||)$ time, is as follows:

Mark$\mathcal{H}$ for $ov(\cdot, v)$:
Traverse the ancestral path of v from the root to v, storing for each leaf x of $\mathcal{A}$ the last internal node y having x in $\mathcal{L}_y$. On reaching v, mark the stored internal nodes for each x.

Algorithm 2 MARK$\mathcal{H}(\mathcal{A}, \mathcal{L})$.

```
foreach internal node v of A do inH[v] ← 0      // Flag for membership in H
foreach leaf v of A do inH[v] ← 1               // Leaves implicitly in H
inH[ϵ] ← 1                                      // Root implicitly in H
foreach leaf v of A do S_v ← ∅                  // Stack of exposed suffix
foreach node v ∈ A in DFS order do              // Compute all inH[v]
    if internal node v first visited then
        foreach leaf x in L_v do Push v on S_x  // Expose v on stacks of L_v
    if internal node v last visited then
        foreach leaf x in L_v do Pop v from S_x // Remove v from stacks of L_v
    if leaf v visited then
        foreach leaf node x do
            if S_x ≠ ∅ then
                inH[Top of S_x] ← 1             // Mark ov(x, v)
Return inH
```

Marking all nodes in $Ov(P)$

We now describe how to perform this procedure for all leaves (prefix) together (see Algorithm 2) using stacks to keep track of the last encountered internal node for each leaf (suffix). The main reason behind using stacks is to avoid processing $\mathcal{L}_u$ multiple times (for different prefixes). For each internal node, we initialize the flag denoting membership in $\mathcal{H}$ to zero, whereas the root and leaves of $\mathcal{A}$ are implicitly in $\mathcal{H}$. For each leaf (suffix) we initialize an empty stack. Now, we traverse $\mathcal{A}$ in DFS order (ignoring suffix links). As in the case for single leaf (prefix), the stack S_x maintains the last internal node v containing a leaf x (suffix) in $\mathcal{L}_v$. This node v is added to the stack S_x of the leaf x (suffix) when v is first visited by the traversal, and removed from the stack S_x when it is last visited. This exposes the previously added internal nodes on the stack. Finally, on visiting a leaf v (prefix), each non-empty stack S_x of a leaf x (suffix) exposes the internal node last added on its top, which is the longest overlap $ov(x, v)$ by Lemma 3. We mark such internal nodes as being present in $\mathcal{H}$. The correctness follows from the same arguments used for the first approach.

In order to analyze the procedure we need to consider the processing of $\mathcal{L}_v$ and S_x for all $v, x \in \mathcal{A}$, in addition to traversing $\mathcal{A}$. Since the total size of all $\mathcal{L}_v$ is $O(||P||)$, processing it twice (on the first and last visit of v) requires $O(||P||)$ time. This also includes the time to push and pop nodes from the stacks, requiring $O(1)$ time while processing $\mathcal{L}_v$. However, on visiting the leaf (prefix) by the traversal, we need to evaluate all S_x and mark the top of non-empty stacks. Since we consider n leaves (prefix), each processing all stacks of n leaves (suffix), we require $O(n^2)$ time. For analyzing size, we need to consider only S_x in addition to $\mathcal{L}_v$. Since the nodes in all S_x are added once from some $\mathcal{L}_v$, the total size of all stacks S_x is bounded by the size of all lists $\mathcal{L}_v$, i.e. $O(||P||)$ (as proved earlier). Thus, this procedure requires $O(||P|| + n^2)$ time and $O(||P||)$ space to mark all nodes in $Ov(P)$.

Optimizing Mark$\mathcal{H}$

As described earlier, the only operation not bounded by $O(||P||)$ time is the marking of internal nodes, while processing the leaves (prefix) considering the stacks of all leaves (suffix). Note that this procedure is overkill as the same top of the stack can be marked again when processing different leaves (prefix), whereas total nodes entering and leaving stacks are proportional to total size of all $\mathcal{L}_u$, i.e., $O(||P||)$. Thus, we ensure that we do not have to process stacks of all leaves (suffix) on processing the leaves (prefix) of $\mathcal{A}$, and instead, we only process those stacks which were not processed earlier to mark the same top. Note that the same internal node may be marked again when exposed in different stacks, but we ensure that it is not marked again while processing the same stack.

Consider Algorithm 3 (showing modified code in red and additions in blue), we maintain a doubly linked-list $\mathcal{S}$ of non-empty stacks whose tops are not marked. Now, whenever a new node is added to a stack, it clearly has an unmarked top, so it is added to $\mathcal{S}$. And when a node is removed from a stack, the stack is added to $\mathcal{S}$ if the new top is not previously marked and stack in not already in $\mathcal{S}$. Similarly, if the stack is empty or has a previously marked top, it is removed from $\mathcal{S}$ if it was present in $\mathcal{S}$. Since $\mathcal{S}$ is a list, its members are additionally maintained using flags $in\mathcal{S}$ for each stack corresponding to leaves (suffix) of $\mathcal{A}$, so that the same stack is not added multiple times in $\mathcal{S}$. Also, each stack in $\mathcal{S}$ maintains a pointer to its location in $\mathcal{S}$, so that it can be efficiently removed if required. Now, on processing the leaves (prefix) of $\mathcal{A}$, we only process the stacks in $\mathcal{S}$, marking their tops and removing them from $\mathcal{S}$. Clearly, stacks are added to $\mathcal{S}$ only while processing $\mathcal{L}_v$, hence overall we can mark $O(|\mathcal{L}_v|)$ nodes for all v, requiring total $O(||P||)$ time. And the time taken in removing stacks from $\mathcal{S}$ is bounded by the total size of all S_x, which is also $O(||P||)$. Thus, we can perform Mark$\mathcal{H}$ using optimal $O(||P||)$ time and space, which results in our main result (using Theorem 4).

▶ **Theorem 1** (Optimal HOG). *For a set of strings P, the Hierarchical Overlap Graph can be computed using $O(||P||)$ time and space.*

Remark: The classical result for APSP [13] (equivalent to our second stage) was optimized [12] to get *output-sensitive* $O(||P|| + n')$ time (where n' is number of pairs with non-zero overlap) by maintaining a list of non-empty stacks (similar to our list $\mathcal{S}$ of stacks with non-marked heads). However, their approach does not suffice for computing $\mathcal{H}$ optimallty as in the worst case $n' = O(n^2) >> O(||P||)$.

Algorithm 3 MARK$\mathcal{H}(\mathcal{A}, \mathcal{L})$

```
foreach internal node v of A do inH[v] ← 0      // Flag for membership in H
foreach leaf v of A do inH[v] ← 1               // Leaves implicitly in H
inH[root] ← 1                                   // Root implicitly in H

foreach leaf v of A do S_v ← ∅                  // Stack of exposed suffix
S ← ∅                                           // List of stacks with unmarked tops
foreach leaf v of A do inS[v] ← 0               // Flag for membership of S_v in S

foreach node v ∈ A in DFS order do              // Compute all inH[v]
    if internal node v first visited then
        foreach leaf x in L_v do                // Expose v on stacks of L_v
            Push v on S_x
            if inS[x] = 0 then                  // Add S_x to S if not present
                inS[x] ← 1
                Add S_x to S
    if internal node v last visited then
        foreach leaf x in L_v do                // Remove v from stacks of L_v
            Pop v from S_x
            if S_x ≠ ∅ and inH[Top of S_x] = 0 then   // S_x eligible in S
                if inS[x] = 0 then              // S_x not present in S
                    inS[x] ← 1
                    Add S_x to S
            else                                // S_x either empty or with marked top
                if inS[x] = 1 then              // S_x present in S
                    inS[x] ← 0
                    Remove S_x from S
    if leaf v visited then
        foreach S_x ∈ S do
            inH[Top of S_x] ← 1                 // Mark ov(x, v)
            inS[x] ← 0
            Remove S_x from S                   // Remove S_x with marked top from S

Return inH
```

5 Conclusions

Genome assembly is one of the most prominent problems in Bioinformatics, and it traditionally relies on de Bruijn graphs or Overlap graphs, each having limitations of either loss of information or quadratic space requirements. Hierarchical Overlap Graphs provide a promising alternative that may result in better algorithms for genome assembly. The previous results on computing these graphs were not scalable (due to the quadratic time-bound) or required complicated data structures (segment trees). Moreover, computing HOG in optimal time and space was mentioned as an open problem in both the previous results [10, 20]. We present a simple algorithm that achieves the desired bounds, using only elementary data structures such as stacks and lists. At its core, we present an improved algorithm for a

special case of All Pairs Suffix Prefix problem. We hope our algorithm directly, or after further simplification, results in a greater adaptability of HOGs in developing better genome assembly algorithms.

References

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

2 Dmitry Antipov, Anton I. Korobeynikov, Jeffrey S. McLean, and Pavel A. Pevzner. hybridspades: an algorithm for hybrid assembly of short and long reads. *Bioinform.*, 32(7):1009–1015, 2016.

3 Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son K. Pham, Andrey D. Prjibelski, Alex Pyshkin, Alexander Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. Spades: A new genome assembly algorithm and its applications to single-cell sequencing. *J. Comput. Biol.*, 19(5):455–477, 2012.

4 Djamal Belazzougui and Fabio Cunial. Fully-functional bidirectional burrows-wheeler indexes and infinite-order de bruijn graphs. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 10:1–10:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

5 Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J. Puglisi. Bidirectional variable-order de bruijn graphs. *Int. J. Found. Comput. Sci.*, 29(8):1279–1295, 2018.

6 Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. *J. ACM*, 41(4):630–647, 1994.

7 Christina Boucher, Alexander Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-order de bruijn graphs. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, pages 383–392. IEEE, 2015.

8 Rodrigo Cánovas, Bastien Cazaux, and Eric Rivals. The compressed overlap index. *CoRR*, abs/1707.05613, 2017. `arXiv:1707.05613`.

9 Bastien Cazaux, Rodrigo Cánovas, and Eric Rivals. Shortest DNA cyclic cover in compressed space. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30 - April 1, 2016*, pages 536–545. IEEE, 2016.

10 Bastien Cazaux and Eric Rivals. Hierarchical overlap graph. *Inf. Process. Lett.*, 155, 2020.

11 Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.

12 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

13 Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Inf. Process. Lett.*, 41(4):181–185, 1992.

14 Shahbaz Khan. Optimal construction of hierarchical overlap graphs. *CoRR*, abs/2102.02873, 2021. `arXiv:2102.02873`.

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

16 Jihyuk Lim and Kunsoo Park. A fast algorithm for the all-pairs suffix-prefix problem. *Theor. Comput. Sci.*, 698:14–24, 2017.

17 Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(2):79–85, 2005. `doi:10.1093/bioinformatics/bti1114`.

18 Sergey Nurk, Dmitry Meleshko, Anton I. Korobeynikov, and Pavel A. Pevzner. metaspades: A new versatile de novo metagenomics assembler. In Mona Singh, editor, *Research in Computational Molecular Biology - 20th Annual Conference, RECOMB 2016, Santa Monica, CA, USA, April 17-21, 2016, Proceedings*, volume 9649 of *Lecture Notes in Computer Science*, page 258. Springer, 2016.

19 Sangsoo Park, Sung Gwan Park, Bastien Cazaux, Kunsoo Park, and Eric Rivals. A linear time algorithm for constructing hierarchical overlap graphs. *CoRR*, abs/2102.12824, 2021 *(accepted for publishing at CPM 2021)*. `arXiv:2102.12824`.

20 Sung Gwan Park, Bastien Cazaux, Kunsoo Park, and Eric Rivals. Efficient construction of hierarchical overlap graphs. In Christina Boucher and Sharma V. Thankachan, editors, *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 277–290. Springer, 2020.

21 Hannu Peltola, Hans Söderlund, Jorma Tarhio, and Esko Ukkonen. Algorithms for some string matching problems arising in molecular genetics. In *IFIP Congress*, pages 59–64, 1983.

22 P. A. Pevzner. l-Tuple DNA sequencing: computer analysis. *Journal of Biomolecular Structure & Dynamics*, 7(1):63–73, 1989.

23 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences of the United States of America*, 98(17):9748–9753, 2001.

24 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinform.*, 26(12):367–373, 2010.

25 Z. Sweedyk. A 2½-approximation algorithm for shortest superstring. *SIAM J. Comput.*, 29(3):954–986, 1999.

26 William H. A. Tustumi, Simon Gog, Guilherme P. Telles, and Felipe A. Louza. An improved algorithm for the all-pairs suffix-prefix problem. *J. Discrete Algorithms*, 37:34–43, 2016.

27 Esko Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(3):313–323, 1990.

28 Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821—829, May 2008. `doi:10.1101/gr.074492.107`.

A Compact Index for Cartesian Tree Matching

Sung-Hwan Kim
Pusan National University, South Korea

Hwan-Gue Cho
Pusan National University, South Korea

Abstract

Cartesian tree matching is a recently introduced string matching problem in which two strings match if their corresponding Cartesian trees are the same. It is considered appropriate to find patterns regarding their shapes especially in numerical time series data. While many related problems have been addressed, developing a compact index has received relatively less attention. In this paper, we present a $3n + o(n)$-bit index that can count the number of occurrences of a Cartesian tree pattern in $\mathcal{O}(m)$ time where n and m are the text and pattern length. To the best of our knowledge, this work is the first $\mathcal{O}(n)$-bit compact data structure for indexing for this problem.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases String Matching, Suffix Array, FM-index, Compact Index, Cartesian Tree Matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.18

Acknowledgements We would like to thank anonymous reviewers for their valuable and constructive comments.

1 Motivation

String matching is an important and fundamental problem that is widely applied in various applications of computer science and engineering. Many practical problems involve a large volume of sequential data and hence string matching problems in finding particular patterns therein. Owing to many reasons such as noise in data, and a variety forms of isomorphism, many variants of the string matching problem have been introduced from various perspectives and different application domains, e.g., parameterized string matching [3] for program source codes, δ-approximate [7] and order preserving matching [22] for numerical time series, structural pattern matching for biological sequences [29], as well as general approximate matching problems such as jumbled pattern matching with Parikh vectors [1] and pattern matching with mismatches [5] and gaps [25].

Developing an indexing method for these approximate string matching problems is important particularly when either the text or the patterns are provided in advance and the other is given online. We are interested in indexing the provided text before pattern queries are given. By indexing the text appropriately, pattern search can be performed efficiently during the query time compared with scanning the text repeatedly for every single pattern. However, indexing strings for approximate string matching problems is usually challenging because of their complex nature. Whereas many problems involve significant space and time complexities [5, 25, 1], some problems may involve efficient data structures [3, 14, 8, 13].

Cartesian tree matching is a variant of the string matching problem that was recently introduced by Park *et al.* [28]. In this problem, two strings over a totally ordered set $\mathbb{U}$ match if their corresponding Cartesian trees are identical. It is considered appropriate to find patterns regarding their shapes and suitable for time series data. Many interesting properties of Cartesian trees have attracted a lot of interest regarding this matching problem, and researches have been conducted in many perspectives such as multiple pattern matching [18],

dictionary matching [19], cover [21], longest common substring [10], and indeterminate matching [15] since the problem was introduced. Meanwhile, the development of a compact index for this problem has received relatively less attention. In the original paper [28], the authors show that the suffix tree for Cartesian tree matching can be built efficiently using the suffix tree construction algorithm for generalized matching problems [6]. Some researchers have addressed this problem as a special case of their indexing framework [2, 23]. However, all these methods are from the standpoint of generalized indexing methods, and the index size is $\mathcal{O}(n \lg n)$ in bits. Because a Cartesian tree of size n can be represented in $2n - o(n)$ bits, it is distant from the optimal space.

The main challenge of the compact data structure for Cartesian tree matching is how we achieve space compactness within $\mathcal{O}(n)$ bits while keeping the search time bounded within a time linear to the pattern length. As mentioned in [28], an the unary code can be used, which allows us to achieve $2n$ bits to represent the text string. However, one single character can be of length $\mathcal{O}(n)$ in its unary code in the worst case. This would cause $\mathcal{O}(n)$ time required to search for a pattern, thereby preventing the index from performing searches efficiently.

In this paper, we present a $3n + o(n)$-bit data structure[1] that supports a counting query in $\mathcal{O}(m)$ time where n and m are the text and pattern lengths, respectively. We introduce a novel concept of the *trimmed LF-mapping*, which allows us to develop a data structure supporting time-efficient queries for counting the number of occurrences within the space bound. To the best of our knowledge, this is the first $\mathcal{O}(n)$-bit index introduced for this problem.

The main theorem of this paper is as follows.

▶ **Theorem 1.** *There exists a $3n + o(n)$-bit data structure that can count the number of occurrences of a Cartesian tree pattern in $\mathcal{O}(m)$ time where n and m are the text and pattern length, respectively.*

The rest of the paper is organized as follows. In Section 2, we establish some notations used in the paper and we give a brief review on backgrounds including succinct bitvectors, which is a building block of the proposed data structure. We define the encoding scheme used to transform the suffixes of the text string in Section 3. In Section 4, we describe the underlying information used in the proposed data structure, and we give a conceptual description of how the searching procedure is performed. In Section 5, we propose a space-efficient representation of the structure described in the previous section. Then we present the searching algorithm on the proposed data structure in Section 6, and we conclude the paper in Section 7.

2 Preliminaries

Notation. By T, P, X and Y, we denote strings over a totally ordered set $\mathbb{U}$; especially, T and P are called text and pattern string. We assume that every element of these strings is distinct in each of them; if not, ties can be broken by position. We use 0-based index for strings and arrays; $T[0]$ indicates the first character. $|T|$ is the length of T. $T[i..j]$ is a substring $T[i] \circ T[i+1] \circ \cdots \circ T[j]$ of T where $\circ$ is the concatenation operator. We define $T[i..j]$ for $i > j$ as an empty string ϵ. For $T[0..i]$ and $T[i..|T|-1]$, we may use $T[..i]$ and $T[i..]$ for brevity. $\mathsf{lcp}(X, Y)$ is the length of the longest common prefix of X and Y. For an integer array A and a property $P(\cdot)$, we denote by $\langle A[i] \rangle_{i|P(A[i])}$, its subsequence obtained by concatenating $A[i]$'s such that $P(A[i])$ holds.

[1] Our data structure can be called an *encoding structure* in the sense that the text string T is not necessary during the pattern matching process.

$E(\cdot)$ is the encoding function described in Section 3.1 that we use to transform suffixes for indexing. We use $\oplus$ to represent a prepending operation as described in Section 3.1. SA is the suffix array and LF is the so-called LF-mapping that represent the correspondence between adjacent suffixes in terms of the lexicographical rank; these are defined in Section 3.2 L and F are integer arrays described in Section 4.1. $B_k^{(L)}$ and $B_k^{(F)}$ are bitvectors described in Section 5.1. On these bitvectors we use several navigating operations such as $\mathsf{down}_k(\cdot)$, $\mathsf{up}_k(\cdot)$, $\mathsf{map}_k(\cdot)$ as defined in Algorithms 1, with which we also define the trimmed LF-mapping $\mathsf{tLF}_k(\cdot)$ in Algorithm 2.

Cartesian Tree. Given a string X over $\mathbb{U}$, its Cartesian tree is a binary tree that is defined as follows. The element with the smallest value becomes the root. The left (and right) subtree is constructed recursively with the elements on the left (right) side. The Cartesian tree of a string of length n can be constructed in $\mathcal{O}(n)$ time using a stack-based algorithm. There are several representation of Cartesian trees regarding our work. These representation use the relation between the element corresponding to each iteration and the elements popped from the stack at the iteration during the construction of the Cartesian tree. Cartesian signature [9] uses the number of pop operations performed on the stack at corresponding iteration, and the parent-distance representation [28] uses the positional distance between the current element and the popped element.

Bitvectors. Bitvectors are basic building blocks of the proposed data structure in this paper. A data structure that supports the following queries in $\mathcal{O}(1)$ time for a length-n bit vector B can be stored in $n + o(n)$ bits [16]:

- Accessing $B[i]$.
- $B.\mathsf{rank}_x(i)$: the number of occurrences of x in $B[0..i-1]$.
- $B.\mathsf{select}_x(i) = j$ such that $B.\mathsf{rank}_x(j) = i - 1$ and $B[j] = x$.

For convenience, we define $B.\mathsf{select}_x(0) = -1$. We also define rank and select queries on integer arrays and strings. Although we do not use them in the final searching algorithm, it is useful for describing how the proposed method works.

3 Encoding and Sorting Suffixes

For many variants of the string matching problem, such as parameterized string matching [3, 14], structural pattern matching [13], and order-preserving matching [8], and Cartesian tree matching [28], it is conventional to use an encoding scheme that transforms strings in a certain form such that two strings match iff their corresponding encoded strings are exactly the same. After encoding the suffixes of the text provided, we can build a data structure for standard string matching such as a suffix tree and a suffix array on the encoded suffixes to enable an efficient pattern search.

In this section, we present the encoding scheme that transforms the suffixes of the given text string for Cartesian tree matching problem, and define the suffix array on them, which will be used in the rest of the paper.

3.1 Modified Parent-Distance Representation

The encoding scheme we use in this paper is similar to that in the original paper [28], which is called *parent-distance representation*. In this representation, each element has at most one parent. Let X be a string. For each $0 \leq i < |X|$, $X[i]$ does not have a parent if it is the smallest one among $X[0], \cdots, X[i]$. If there is an element $X[j]$ that is smaller than $X[i]$ for some $0 \leq j < i$, $X[i]$ points to the rightmost one among such elements as its parent.

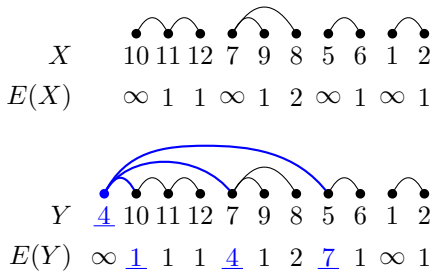


Figure 1 Illustration of changes in encoded strings when a character is prepended. For some k, the first k ∞'s are substituted with integers according to their positions after prepending a character. Arcs represent parents. In this case, we represent it as $E(Y) = 3 \oplus E(X)$.

The relation between an element and its parent is represented as the distance between their positions. For the elements that do not have a parent, 0 is used in the original paper. In this paper, we use ∞ instead of 0, which means the element having no parent will be represented as the greatest symbol in its encoded form. More formally, the encoding scheme is defined as follows:

▶ **Definition 2** (Encoding). *For a string X over $\mathbb{U}$, its encoded string $E(X)$ is defined as follows. For $0 \le i \le |X| - 1$,*

$$E(X)[i] = i - \max \pi_X(i) \tag{1}$$

where $\pi_X(i) = \{j \mid 0 \le j < i \text{ and } X[j] < X[i]\} \cup \{-\infty\}$. For convenience, we also define the encoded string of the empty string to be the empty string: i.e. $E(\epsilon) = \epsilon$.

As shown in [28], we can compute the (modified) parent-distance representation of a string in linear time.

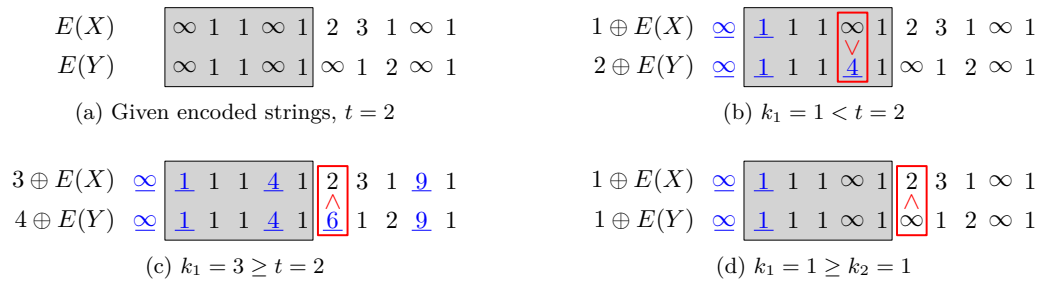
Our index will use a backward searching mechanism as FM-index families do. In order to devise a backward searching algorithm, we need to find out the relation between adjacent suffixes $T[i-1..]$ and $T[i..]$ in terms of their encoded strings. Note that $T[i-1..]$ is a string that can be obtained by prepending a character $T[i-1]$ at the beginning of $T[i..]$.

Let X and Y be strings over $\mathbb{U}$ such that $Y = x \circ X$ for some $x \in \mathbb{U}$; Y is a string that can be obtained by prepending a character x at the beginning of X. We want to observe their differences in terms of their encoded strings. This will be used to proceed from a suffix $T[i..]$ to its previous suffix $T[i-1..]$ during the searching process.

Figure 1 shows an example. When we prepend a character x to a string X, we can observe that, for some k, the first k ∞'s are substituted by some integers in its encoded form; and the integers with which ∞'s are substituted are determined by their positions. Then a single ∞ is prepended at the beginning, which completes $E(x \circ X) = E(Y)$. In other words, the operation of prepending a character can be characterized by an integer indicating the number of ∞'s to be substituted. In the rest of the paper, we represent the relation between two encoded strings such that $Y = x \circ X$ as $E(Y) = k \oplus E(X)$.

For an encoded string $E(X)$ and an integer $0 \le k \le E(X).\mathsf{rank}_\infty(|X|)$, $k \oplus E(X)$ is an encoded string defined as:

$$(k \oplus E(X))[i] = \begin{cases} \infty & \text{if } i = 0, \\ i & \text{if } E(X)[i-1] = \infty \text{ and } i - 1 \le E(X).\mathsf{select}_\infty(k), \\ E(X)[i-1] & \text{otherwise .} \end{cases} \tag{2}$$



(a) Given encoded strings, $t = 2$

(b) $k_1 = 1 < t = 2$

(c) $k_1 = 3 \geq t = 2$

(d) $k_1 = 1 \geq k_2 = 1$

Figure 2 Illustration of changes in lexicographical order when a character is prepended. $E(X)$ and $E(Y)$ are converted into $k_1 \oplus E(X)$ and $k_2 \oplus E(Y)$ due to a prepended character. t is the number of ∞'s within the longest common prefix (indicated with shaded boxes) of $E(X)$ and $E(Y)$. Underlined blue characters: changed elements after prepending a character; Red thick boxes: The position in which the order of two encoded strings is determined.

Given two encoded strings $E(X)$ and $E(Y)$, prepending characters at their beginning possibly changes their lexicographical order. Figure 2 decribes how the order can change by prepending different characters. From this observation, we establish an important lemma about the lexicographical order after prepending single characters at the beginning of these strings, which is frequently used throughout the paper.

▶ **Lemma 3.** *Let X and Y be strings over $\mathbb{U}$ such that $E(X) < E(Y)$, and let $t = E(X).\mathsf{rank}_\infty(l)$ where $l = \mathsf{lcp}(E(X), E(Y))$. For integers $0 \leq k_1 \leq E(X).\mathsf{rank}_\infty(|X|)$ and $0 \leq k_2 \leq E(Y).\mathsf{rank}_\infty(|Y|)$, $k_1 \oplus E(X) < k_2 \oplus E(Y)$ if and only if $k_1 \geq t$ or $k_1 \geq k_2$.*

Proof. ($\Rightarrow$) We prove by contrapositive. Let us assume that $k_1 < t$ and $k_1 < k_2$. Since $k_1 < t$, the (k_1+1)-th ∞ is within the longest common prefix of $E(X)$ and $E(Y)$. Thus $E(X).\mathsf{select}_\infty(k_1+1) = E(Y).\mathsf{select}_\infty(k_1+1)$. Let $i = E(X).\mathsf{select}_\infty(k_1+1)$. Then $(k_1 \oplus E(X))[i+1] = \infty > i+1 = (k_2 \oplus E(Y))[i+1]$. Since $(k \oplus E(X))[..i] = (k \oplus E(Y))[..i]$, this implies $k_1 \oplus E(X) > k_2 \oplus E(Y)$.

($\Leftarrow$) We have two cases: (i) $k_1 \geq t$, and (ii) $k_1 \geq k_2$.

- **Case 1 ($k_1 \geq t$):** If $k_2 < t$, the (k_2+1)-th ∞ of $E(X)$ and $E(Y)$ is within their longest common prefix. Let $i = E(X).\mathsf{select}_\infty(k_2+1)$. The (k_2+1)-th ∞ of $E(X)$ is to be substituted with $i+1$, while the (k_2+1)-th ∞ of $E(Y)$ remains the same. Thus $(k_1 \oplus E(X))[i+1] = i+1 < \infty = (k_2 \oplus E(Y))[i+1]$, which implies $k_1 \oplus E(X) < k_2 \oplus E(Y)$. If $k_2 \geq t$, all ∞'s within the longest common prefix are to be substituted according to their positions, and the lexicographical order of $k_1 \oplus E(X)$ and $k_2 \oplus E(Y)$ is determined by $(k_1 \oplus E(X))[l+1]$ and $(k_2 \oplus E(Y))[l+1]$ where $l = \mathsf{lcp}(E(X), E(Y))$. If $E(Y)[l] \neq \infty$, $(k_1 \oplus E(X))[l+1] = E(X)[l] < E(Y)[l] = (k_2 \oplus E(Y))[l+1]$. If $E(Y)[l] = \infty$, $(k_1 \oplus E(X))[l+1] = E(X)[l] \leq l < l+1 = (k_2 \oplus E(Y))[l+1]$. In both cases, we have $k_1 \oplus E(X) < k_2 \oplus E(Y)$.
- **Case 2 ($k_1 \geq k_2$):** Since we have proved the case of $k_1 \geq t$, we can assume that $t > k_1 \geq k_2$. Thus all ∞'s being substituted with integers are within their longest common prefix. Let $i = E(X).\mathsf{select}_\infty(k_2+1)$ and $l = \mathsf{lcp}(E(X), E(Y))$. If $k_1 > k_2$, $(k_1 \oplus E(X))[..i] = (k_2 \oplus E(Y))[..i]$, and $(k_1 \oplus E(X))[i+1] = i+1 < \infty = E(Y)[i] = (k_2 \oplus E(Y))[i+1]$. If $k_1 = k_2$, $(k_1 \oplus E(X))[..l] = (k_2 \oplus E(Y))[..l]$, and $(k_1 \oplus E(X))[..l+1] = E(X)[l] < E(Y)[l] = (k_2 \oplus E(Y))[..l+1]$. Hence $k_1 \oplus E(X) < k_2 \oplus E(Y)$. ◀

3.2 Sorting Suffixes

In this subsection, we define the suffix array of a given text string $T[0..n-1]$. Let $\mathcal{S}(T)$ be the set of encoded suffixes:

$$\mathcal{S}(T) = \{E(T[i..]) \mid 0 \leq i \leq n\} \tag{3}$$

Note that we include $T[n..] = \epsilon$, so the size of $\mathcal{S}(T)$ is $n+1$. This acts like the unique termination symbol at the end of the text, which is a standard assumption in many indexing methods. The suffix array SA is an array of length $n+1$ that stores the encoded suffixes in the lexicographically sorted order. Each encoded suffix is represented by its starting position on the text; i.e. if $E(T[j..])$ is the $(i+1)$-th smallest string among the encoded suffixes, we define $\mathsf{SA}[i] = j$: For $0 \leq i \leq n$,

$$\mathsf{SA}[i] = j \text{ iff } i = \left|\{X \in \mathcal{S}(T) \mid X < E(T[j..])\}\right| \tag{4}$$

Since the suffix array is a permutation of $\langle 0, \cdots, n\rangle$, we can also define its inverse: $\mathsf{SA}^{-1}[i] = j$ iff $\mathsf{SA}[j] = i$.

3.3 Suffix Range

Remember that P matches $T[j..j+|P|-1]$ if and only if $E(P)$ is a prefix of $E(T[j..])$. Because we have sorted the encoded suffixes, if $E(P)$ is a prefix of encoded suffixes $E(T[j_1..]), E(T[j_2..]), \cdots, E(T[j_k..])$, then these encoded suffixes are consecutive in their sorted order on the suffix array. Therefore, we can use two integers $0 \leq p_s \leq p_e \leq n$ such that $p_s \leq \mathsf{SA}^{-1}[j_l] \leq p_e$ for $1 \leq l \leq k$ to represent these encoded suffixes.

▶ **Definition 4** (Suffix range). *For an encoded pattern $E(P)$, an integer pair (p_s, p_e) is called the suffix range of $E(P)$ if $p_s \leq i \leq p_e \Leftrightarrow E(T[\mathsf{SA}[i]..])[0..|P|-1] = E(P)$ for all $0 \leq i \leq n$.*

4 Basic Idea on Updating Suffix Ranges

The proposed data structure performs a backward search that processes the pattern in the right-to-left direction. Assuming that the pattern $P[0..m-1]$ is not an empty string, it starts with the suffix range of $E(P[m-1..]) = \infty$. Then it repeatedly updates the suffix range by prepending $P[i]$ at the beginning of the currently searched pattern from $m-2$ to 0. At each iteration, we compute the suffix range of $E(P[i..m-1])$, hence we can obtain the desired suffix range of the entire pattern in the end.

Each iteration of this procedure can be described as follows. Let P be the currently searched pattern. When we prepend a character at the beginning of the currently searched pattern, this updated pattern can be written as $k \oplus E(P)$. Let (p_s, p_e) be the suffix range of an encoded pattern $E(P)$. What we want is to update the suffix range (p_s, p_e) into the suffix range (p'_s, p'_e) of $k \oplus E(P)$. It can be seen as two stages:

1. Identifying the target suffixes: from the set of indices $\mathcal{I} = \{i \mid p_s \leq i \leq p_e\}$, we identify the set of indices $\mathcal{I}' = \{i \mid p_s \leq i \leq p_e \text{ and } E(T[j-1..])[0..|P|] = k \oplus E(P) \text{ where } j = \mathsf{SA}[i]\} \subset \mathcal{I}$ whose corresponding suffixes are to be included in the updated suffix range after prepending their corresponding character $T[j-1]$.
2. Mapping $E(T[\mathsf{SA}[i]..])$ to its previous suffix $E(T[\mathsf{SA}[i]-1..])$ for each target suffix: we compute the set $\mathcal{I}'' = \{\mathsf{SA}^{-1}[\mathsf{SA}[i]-1] \mid i \in \mathcal{I}'\}$.

i	$\mathsf{SA}[i]$	$\mathsf{LF}[i]$	$F[i]$	$L[i]$	$E(T[\mathsf{SA}[i]..])$
0	15	1	-1	0	ϵ
1	14	8	0	0	∞
2	4	9	2	0	∞ 1 1 2 3 5 1 ∞ 1 2 3
3	0	0	1	-1	∞ 1 1 2 ∞ 1 1 2 3 5 1 ∞ 1 2 3
4	11	10	3	0	∞ 1 2 3
5	5	2	3	2	∞ 1 2 3 ∞ 1 ∞ 1 2 3
6	1	3	2	1	∞ 1 2 ∞ 1 1 2 3 5 1 ∞ 1 2 3
7	9	11	1	0	∞ 1 ∞ 1 2 3
8	13	12	0	0	∞ ∞
9	3	13	0	0	∞ ∞ 1 1 2 3 5 1 ∞ 1 2 3
10	10	7	0	1	∞ ∞ 1 2 3
11	8	14	0	0	∞ ∞ 1 ∞ 1 2 3
12	12	4	0	3	∞ ∞ ∞
13	2	6	0	2	∞ ∞ ∞ 1 1 2 3 5 1 ∞ 1 2 3
14	7	15	0	0	∞ ∞ ∞ 1 ∞ 1 2 3
15	6	5	0	3	∞ ∞ ∞ ∞ 1 ∞ 1 2 3

Figure 3 Underlying information of the proposed index for sequence T = 4 6 9 8 2 10 15 14 12 3 13 1 11 7 5. Examples of suffix ranges are indicated by boxes in column $E(T[\mathsf{SA}[i]..])$. Shaded, red (with thick borders), and blue (with thin borders) boxes indicate the suffix ranges of P =4 2, P' =3 4 2, and P'' =1 4 2, respectively. Boxes in columns $F[i]$ and $L[i]$ indicate the entries used to obtain the suffix ranges of P' and P'' from that of P.

Computing the new indices in the stage 2 ($\mathsf{SA}^{-1}[\mathsf{SA}[i]-1]$) is the so-called *LF-mapping*, which is the core operation of many compact string matching indexes that use the backward searching mechanism. In this section, we define the LF-mapping along with its representation with two integer arrays. Then we also present how to identify the target suffixes using these arrays. Note that this representation using two integer arrays in this section is a conceptual representation; the space-efficient representation that is actually used in the proposed data structure is described in Section 5.

4.1 LF-mapping with Two Integer Arrays

In the context of backward searching methods such as FM-index, LF-mapping is a function indicating the correspondence between two adjacent suffixes $E(T[j..])$ and $E(T[j-1..])$ in terms of indices on the suffix array.

For an integer $0 \leq i \leq n$, let $T[j..]$ be a suffix such that $\mathsf{SA}[i] = j$. Let $j' = j + n \mod (n+1)$. The correspondence between $T[j..]$ and $T[j'..]$ in the sorted suffixes are stored in the array LF. More specifically, $\mathsf{LF}[i]$ indicates the lexicographical rank of $T[j'..]$.

$$\mathsf{LF}[i] = \mathsf{SA}^{-1}[\mathsf{SA}[i] + n \mod (n+1)] \tag{5}$$

We define two integer arrays L and F of length $n+1$, which contain the essential underlying information of the proposed data structure. Let us consider a particular suffix $T[\mathsf{SA}[i]..]$. When we prepend a character $T[\mathsf{SA}[i]-1]$ at the beginning of $T[\mathsf{SA}[i]..]$, the corresponding suffix is $E(T[\mathsf{SA}[i]-1..]) = E(T[\mathsf{SA}[i]-1] \circ T[\mathsf{SA}[i]..])$. We can uniquely determine an integer k such that $E(T[\mathsf{SA}[i]-1..]) = k \oplus E(T[\mathsf{SA}[i]..])$. This k is the value related to the Cartesian tree signature [9], which is also mentioned as an alternative representation for the matching problem in [28]. It can be computed during the construction of the Cartesian tree or its parent-distance representation. We define the array L to store such k's for each of the suffixes.

$$L[i] = \begin{cases} -1 & \text{if } \mathsf{SA}[i] = 0, \\ K(\mathsf{SA}[i]-1) & \text{otherwise.} \end{cases}$$

where $K(j)$ is the integer k such that $E(T[j..]) = k \oplus E(T[j+1..])$. (6)

We can see that for the suffixes having the same $L[i]$ values, their LF-mapping is order-preserving.

▶ **Lemma 5.** *For $0 \le i < j \le n$ such that $L[i] = L[j]$, $\mathsf{LF}[i] < \mathsf{LF}[j]$.*

Proof. Note that $\mathsf{LF}[i] < \mathsf{LF}[j]$ iff $L[i] \oplus E(T[\mathsf{SA}[i]..]) < L[j] \oplus E(T[\mathsf{SA}[j]..])$. Applying Lemma 3 with $X = T[\mathsf{SA}[i]..]$, $Y = T[\mathsf{SA}[j]..]$, $k_1 = L[i]$, and $k_2 = L[j]$, we obtain $L[i] \oplus E(T[\mathsf{SA}[i]..]) < L[j] \oplus E(T[\mathsf{SA}[j]..])$ because $i < j \Leftrightarrow E(T[\mathsf{SA}[i]..]) < E(T[\mathsf{SA}[j]..])$ and $L[i] = L[j]$. ◀

Using this order-preserving property, we can represent the correspondence between two adjacent suffixes $E(T[\mathsf{SA}[\mathsf{LF}[i]]..])$ and $E(T[\mathsf{SA}[i]..])$ using their associated k-values described above. As the L array represents k-values for each suffix $E(T[\mathsf{SA}[i]..])$, we write these k-values for their associated suffixes $E(T[\mathsf{SA}[\mathsf{LF}[i]]..])$ to make another array F as follows:

$$F[\mathsf{LF}[i]] = L[i] \tag{7}$$

Using the arrays L and F, we can conceptually compute $\mathsf{LF}[i]$ as follows. Let $x = L[i]$. We can compute the number $c = L.\mathsf{rank}_{L[i]}(i+1)$ of occurrences $L[i]$ in $L[0..i]$. We find $0 \le j \le n$ such that the number of occurrences of x in $F[0..j]$ is c and $F[j] = x$: i.e. $j = F.\mathsf{select}_{L[i]}(c)$ is the position of the c-th occurrence of x on F. Then we have $j = \mathsf{LF}[i]$.

4.2 Identifying Target Suffixes

To devise a backward searching algorithm, we need to compute the suffix range (p'_s, p'_e) of $k \oplus E(P)$ from the suffix range (p_s, p_e) of $E(P)$. As we have mentioned at the beginning of this section, this update procedure consists of two stages, which is identifying the target suffixes according to k followed by applying LF-mapping for each of the identified target suffixes. In this subsection, we present how to identify the target suffixes using L array during the suffix range update.

We have two cases: (i) there are ∞'s remaining in $(k \oplus E(P))[1..]$, and (ii) all ∞'s in $E(P)$ are to be substituted into integers so there is no ∞ in $(k \oplus E(P))[1..]$. In the first case, we know that the same number of ∞'s are to be substituted in the target suffix after applying the LF-mapping. On the other hand, in the second case, the number of ∞'s that are to be substituted in the target suffix is not fixed, but k is the lower bound of the number of substituted ∞'s.

▶ **Lemma 6.** *For an encoded string $E(P)$ and an integer k such that $0 \le k \le E(P).\mathsf{rank}_\infty(|P|)$, let (p_s, p_e) and (p'_s, p'_e) be the suffix ranges of $E(P)$ and $k \oplus E(P)$, respectively. For $p_s \le i \le p_e$, $p'_s \le \mathsf{LF}[i] \le p'_e$ if and only if*

$$\begin{cases} L[i] = k & \text{if } k < E(P).\mathsf{rank}_\infty(|P|), \\ L[i] \ge k & \text{if } k = E(P).\mathsf{rank}_\infty(|P|). \end{cases} \tag{8}$$

Proof. (⇐) We can rewrite the right-hand of the if-and-only-if statement as: **(i)** $L[i] = k$ or **(ii)**$L[i] > k$ and $k = E(P).\mathsf{rank}_\infty(|P|)$. (Case 1: $L[i] = k$) Let Q be a string such that $E(Q) = E(P) \circ (\infty)^n$. Then the fact that $p_s \leq i \leq p_e$ is equivalent to $E(P) \leq E(T[\mathsf{SA}[i]..]) < E(Q)$. Also, $k \oplus E(P) \leq k \oplus E(T[\mathsf{SA}[i]..]) < k \oplus E(Q)$. Since $L[i] = k$, $k \oplus E(P) \leq L[i] \oplus E(T[\mathsf{SA}[i]..]) = E(T[\mathsf{SA}[\mathsf{LF}[i]]..]) < k \oplus E(Q)$ by Lemma 3. (Case 2: $L[i] > k$ and $k = E(P).\mathsf{rank}_\infty(|P|)$) Note that $E(T[\mathsf{SA}[i]..]).\mathsf{select}_\infty(k') \geq |P|$ for all $k' > k$. Hence, $(L[i] \oplus E(T[\mathsf{SA}[i]..]))[0..|P|] = k \oplus (E(T[\mathsf{SA}[i]..])[0..|P| - 1]) = k \oplus E(P)$.

(⇒) We prove by contrapositive. The negation of the right-hand part of the if-and-only-if statement can be rewritten as follows: **(i)** $L[i] < k$ or **(ii)**$L[i] > k$ and $k < E(P).\mathsf{rank}_\infty(|P|)$. Note that because i is within the suffix range of $E(P)$, $E(T[\mathsf{SA}[i]..]).\mathsf{rank}_\infty(|T[\mathsf{SA}[i]..]|) \geq E(P).\mathsf{rank}_\infty(|P|) \geq k$; moreover, $E(T[\mathsf{SA}[i]..]).\mathsf{select}_\infty(j) = E(P).\mathsf{select}_\infty(j)$ for $1 \leq j \leq E(P).\mathsf{rank}_\infty(|P|)$. (Case 1: $L[i] < k$): Let $j = E(T[\mathsf{SA}[i]..]).\mathsf{select}_\infty(L[i] + 1) + 1 = E(P).\mathsf{select}_\infty(L[i] + 1) + 1$. Then we have $(L[i] \oplus E(T[\mathsf{SA}[i]..])[j] = \infty \neq j = (k \oplus E(P))[j]$. (Case 2: $L[i] > k$ and $k < E(P).\mathsf{rank}_\infty(|P|)$) Let $j = E(T[\mathsf{SA}[i]..]).\mathsf{select}_\infty(k + 1) + 1$. $(L[i] \oplus E(T[\mathsf{SA}[i]..])[j] = j \neq \infty = E(P)[j - 1] = (k \oplus E(P))[j]$. ◀

5 $3n + o(n)$-bit Representation

In this section, we present how to represent two arrays L and F in a space-efficient way. After we present a $6n + o(n)$-bit representation, we show how to reduce the space occupancy into $3n + o(n)$ bits by representing $3n + 2$ bits among them within $\mathcal{O}(\lg n) = o(n)$ bits.

5.1 Representation of L and F with Unary Coding

In this subsection, we present how to efficiently represent the two arrays L and F. First, we define subsequences of L and F for $k \geq -1$:

$$L_k = \langle L[i] \rangle_{i|L[i] \geq k} \tag{9}$$

Similarly,

$$F_k = \langle F[i] \rangle_{i|F[i] \geq k} \tag{10}$$

We also define bitvectors corresponding to L_k and F_k:

$$B_k^{(L)}[i] = 0 \text{ iff } L_k[i] = k, \text{ otherwise } 1. \tag{11}$$

and

$$B_k^{(F)}[i] = 0 \text{ iff } F_k[i] = k, \text{ otherwise } 1. \tag{12}$$

Note that the number of bits at level $k + 1$ is the number of 1-bits at level k. The i-th 1-bit at level k is associated with i-th bit at level $k + 1$. Using this correspondence between bits on consecutive levels, we can build tree-like structures on L and F. This is similar to the (pointer-less) wavelet tree [17, 27] with a non-standard shape. Wavelet tree with different shapes have been used to compress the space occupancy and speed up the query time [12], in which Huffman prefix tree is typically used where each element can be seen to be represented as its Huffman code. For our data structure, it can be seen as representing each element x of L and F as the unary code of $x + 1$: i.e. a bit string $1^{x+1}0$.

Note the sum of $L[i]$ over $0 \leq i \leq n$ such that $L[i] \geq 0$ is less than n and there is the unique i such that $L[i] = -1$. Thus the sum of $L[i] + 1$ over all $0 \leq i \leq n$ is less than $2n$. The total number of 0's is $n + 1$. The number of bits for representing L and F is at most

$(2n-1)+(n+1)=3n$ each; thus the total number of bits over the entire data structure is at most $6n$. We build the rank and select dictionaries on these bitvectors, which would occupy $6n+o(n)$ bits in total.

▶ Remark. One might also find that it is similar to Direct Addressable Code (DAC,[4]) with block size $b=1$. Each chunk $C_{i,j}$ of DAC consists of two parts: a single flag bit ($B_i[j]$), and a b-bit block of codes ($A_{i,j}$). In this case, every block $A_{i,j}$ has a single bit, B_i indicates whether this block is the highest one. Each element of L (or F) can be represented across as many levels as its value. The bitvector B_i of DAC is actually the same as the bits that comprise our data structure. Perhaps we may use this observation to extend our data structure into other string matching problems where L and F values can be represented with a variable number of integers.

We define the operations that are used to navigate across bitvectors in Algorithm 1. By $\mathsf{down}_k(i)$ we can move from a position on $B_{-1}^{(L)}$ to its corresponding position on $B_k^{(L)}$. Similarly, $\mathsf{up}_k(i)$ computes the corresponding position on $B_{-1}^{(F)}$ to a position on $B_k^{(F)}$. We can use $\mathsf{map}_k(i)$ to jump from $B_k^{(L)}$ to $B_k^{(F)}$; 0's (resp. 1's) on $B_k^{(L)}$ correspond to 0's (resp. 1's) on $B_k^{(F)}$ in order.

Algorithm 1 Navigating operations on $B_k^{(L)}$'s and $B_k^{(F)}$'s.

```
1  function down_k(i):
2      for l = −1 To k − 1 do
3          i ← B_l^(L).rank_1(i)
4      end
5      return i

6  function up_k(i):
7      for l = k To 0 do
8          i ← B_{l−1}^(F).select_1(i + 1)
9      end
10     return i

11 function map_k(i):
12     x ← B_k^(L)[i]
13     return B_k^(F).select_x(B_k^(L).rank_x(i) + 1)
```

5.2 Trimmed LF-mapping

The unary representation of L and F can reduce the required space into $\mathcal{O}(n)$ in bits. However, accessing a single element $L[i]$ takes $\Theta(L[i])$ time and $L[i] \leq n$. As a result, computing the LF-mapping may take $\Theta(n)$ time in the worst case, which would prevent us from achieving $\mathcal{O}(m)$ query time. In this section, we introduce the concept of *trimmed LF-mapping*, which enables us to update the suffix range efficiently. Although an individual computation of the trimmed LF-mapping of a particular suffix may produce an incorrect mapping, it effectively works for the simultaneous mapping of target suffixes during the suffix range updates. The trimmed-mapping function $\mathsf{tLF}(i)$ is defined as follows.

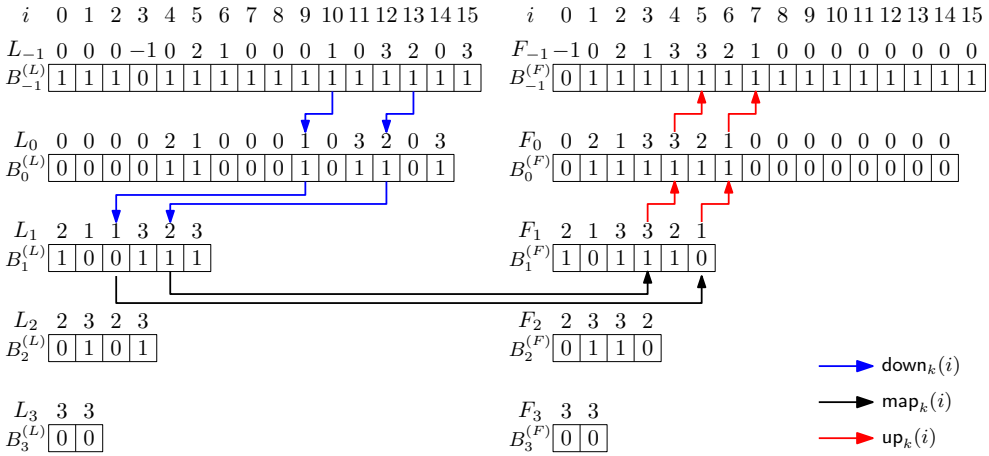


Figure 4 Computing $\mathsf{tLF}_k(i)$, which consists of $\mathsf{down}_k(\cdot)$, $\mathsf{map}_k(\cdot)$, and $\mathsf{up}_k(\cdot)$. If we arrive at a 0-bit after executing $\mathsf{down}_k(i)$, $\mathsf{tLF}_k(i) = \mathsf{LF}[i]$. If not, $\mathsf{tLF}_k(\cdot)$ may be jumbled; but it does not actually matter for updating suffix ranges. In this example, $\mathsf{tLF}_1(10)$ and $\mathsf{tLF}_1(13)$ are depicted.

Algorithm 2 Computing Trimmed LF-mapping.

```
1 function tLF_k(i):
2     i ← down_k(i)
3     i ← map_k(i)
4     i ← up_k(i)
5     return i
```

An example of $\mathsf{tLF}_k(\cdot)$ is illustrated in Figure 4, which computes $\mathsf{tLF}_1(10)$ and $\mathsf{tLF}_1(13)$. Let us explain the computation of $\mathsf{tLF}_1(10)$, which is described as the outer path in Figure 4. After performing $\mathsf{down}_1(10)$, we arrive at position 2 at level 1. By $\mathsf{map}_1(2)$ we move to its corresponding position on the F-side tree, which is position 5 at level $B_1^{(F)}$. Then we obtain $\mathsf{tLF}_1(10) = 7$ by performing $\mathsf{up}_1(5)$. Similarly, we can compute $\mathsf{tLF}_1(13) = 5$ by performing $\mathsf{down}_k(\cdot)$, $\mathsf{map}_k(\cdot)$, $\mathsf{up}_k(\cdot)$ in order, each step of which we arrive at positions 4, 3 and 5 respectively.

Notice the different behavior of $\mathsf{map}_k(i)$ depending on the value $B_k^{(L)}(i)$, in the perspective of the correspondence that $\mathsf{map}_k(\cdot)$ makes between B_k and L_k. Suppose we arrive at position i at level k. If $B_k^{(L)}[i] = 0$, we have $L_k[i] = F_k[j]$ where $j = \mathsf{map}_k(i)$. We also observe that the mapping between L_k and F_k using 0-bits is order-preserving. From this observation, we can obtain the position of the l-th occurrence of x on F by computing $\mathsf{tLF}_k(i)$ where i is the position of the l-th occurrence of x on L if we can reach a 0-bit at the deepest level during the computation. $\mathsf{tLF}_k(\cdot)$ can be used to compute LF directly. If we compute $\mathsf{tLF}_{L[i]}(i)$, we reach a 0 after calling $\mathsf{down}_{L[i]}(i)$, which means we reach the end of the unary code of a particular element. Conceptually, $\mathsf{tLF}_{L[i]}(i)$ is equivalent to perform a rank query on L for $x = L[i]$, followed by performing select query for the corresponding occurrence of x on F. Thus we can successfully compute $\mathsf{LF}[i]$.

▶ **Lemma 7.** *For $0 \le i \le n$, we can compute $\mathsf{LF}[i]$ in $\Theta(L[i])$ time; more specifically, $\mathsf{tLF}_{L[i]}(i) = \mathsf{LF}[i]$*

Proof. Let $i' = \mathsf{down}_{L[i]}(i)$ and $i'' = \mathsf{map}_{L[i]}(i')$. Clearly, $L_{L[i]}[i'] = F_{L[i]}[i''] = L[i]$ and $L_{L[i]}.\mathsf{rank}_{L[i]}(i') = F_{L[i]}.\mathsf{rank}_{L[i]}(i'')$ because they are order-preserving. Let $i^* = \mathsf{up}_{L[i]}(i'')$. Then we have $L[i] = F[i^*]$ and $L.\mathsf{rank}_{L[i]}(i) = F.\mathsf{rank}_{L[i]}(i^*)$. Therefore $\mathsf{LF}[i] = i^* = \mathsf{tLF}_{L[i]}(i)$ by Lemma 5. $\mathsf{down}_{L[i]}(i)$ and $\mathsf{up}_{L[i]}(i'')$ take $\Theta(L[i])$ time each, and $\mathsf{map}_{L[i]}(i')$ takes $\mathcal{O}(1)$ time. Therefore, computing $\mathsf{tLF}_{L[i]}(i)$ takes $\Theta(L[i])$ time. ◀

On the other hand, when $B_k^{(L)}[i] = 1$, $L_k[i]$ is not necessarily the same as $F_k[j]$ where $j = \mathsf{map}_k(i)$. However, we can see that the l-th occurrence of an element on L that is not less than k is associated with the l-th occurrence of an element on F that is not less than k by this mapping at level k. Although $\mathsf{map}_k(i)$ may not give the exact value of $\mathsf{LF}[i]$, we can effectively use this to update a suffix range, which will be described in Section 6.

5.3 Representing $B_k^{(L)}$ and $B_k^{(F)}$ Compactly

The total number of bits to represent all of $B_k^{(L)}$'s and $B_k^{(F)}$'s in their raw form is at most $6n$ bits. We can reduce the required number of bits by representing certain levels of bitvectors compactly. Note that bit vectors at level -1 consist of $n+1$ bits, and there is only one 0-bit in each of $B_{-1}^{(L)}$ and $B_{-1}^{(F)}$. Thus we can represent them using $\mathcal{O}(\lg n)$ bits by representing the position of the unique 0-bit using a single integer. We can also make an observation that $B_0^{(F)}$ is a form of 01^p0^{n-p-1} where p is the number of occurrences of 0's on F. Therefore $B_0^{(F)}$ can be represented by a pair of integers that represent the interval in which 1-bits are located. Using this representation, the bitvectors can be represented in up to $3n + o(n)$ bits in total.

▶ **Lemma 8.** *$B_{-1}^{(L)}$, $B_{-1}^{(F)}$ and $B_0^{(F)}$ can be stored in $\mathcal{O}(\lg n)$ bits while supporting* rank *and* select *queries in $\mathcal{O}(1)$ time.*

Proof. It is trivial for $B_{-1}^{(L)}$ and $B_{-1}^{(F)}$ because the number of 1-bits is 1 so a single integer occupying $\mathcal{O}(\lg n)$ bits can represent these bit vectors. For $B_0^{(F)}$, we claim that $B_0^{(F)}$ has a form of 01^p0^{n-p-1} for some p. Note that $E(T[\mathsf{SA}[0]..]) = E(T[|T|..]) = \epsilon$ is the smallest encoded suffix, and $E(T[|T|-1..]) = \infty$ is the smallest among non-empty encoded suffixes because, for $1 \le i \le n$, $E(T[\mathsf{SA}[i]..])$ have a common prefix ∞. It is clear that $L[0] = 0 = F[\mathsf{LF}[0]] = F[1]$. Since $F[0] = -1$, $F[1]$ is the first occurrence of 0, thus we have $B_0^{(F)}[0] = 0$. Because the number of ∞'s in the longest common prefix of any two non-empty encoded suffixes is at least 1, $L[i] \oplus E(T[\mathsf{SA}[i]..]) < L[j] \oplus E(T[\mathsf{SA}[j]..])$ if $L[i] > 0$ and $L[j] = 0$ for any $0 < i, j \le n$ by Lemma 3. Therefore, we have $\mathsf{LF}[i] < \mathsf{LF}[j]$ for such i and j. By the definition of F, $F[\mathsf{LF}[i]] \ge 1$ and $F[\mathsf{LF}[j]] = 0$. ◀

6 Searching Algorithm with Trimmed LF-mapping

In this section, we devise a searching algorithm to compute the suffix range of a pattern P in $\mathcal{O}(|P|)$ time using the data structure described in the earlier section. Algorithm 3 shows the procedure to compute the suffix range of a given pattern. Starting with $E(P[|P|-1..]) = \infty$, it prepends $P[m-2], \cdots, P[0]$ at each iteration. Examples of updating a suffix range within an individual iteration are depicted in Figure 5.

6.1 Identifying the Target Suffixes

We need to identify the target suffixes in order to update the suffix range correctly. As we have discussed in Section 4.2, given a currently searched encoded pattern $E(P)$ and an integer k, we need to identify the suffixes $T[\mathsf{SA}[i]..]$'s such that $L[i] = k$ if $k < E(P).\mathsf{rank}_\infty(|P|)$,

Algorithm 3 Compute the suffix range of a pattern.

```
1  function SuffixRange(P[0..m − 1]):
2      Compute E(P)
3      Set K[i] ← 0 for 0 ≤ i ≤ m − 2
4      for i=0 To m − 1 do
5          if E(P)[i] ≠ ∞ then
6              K[i − E(P)[i]] ← K[i − E(P)[i]] + 1
7      end
8      p_s ← 1  // the start position of the current suffix range.
9      p_e ← n  // the end position of the current suffix range.
10     t ← 1  // the number of ∞'s in the currently searched pattern.
11     for i=2 To m do
12         k ← K[m − i]  // the number of ∞'s to be substituted.
13         p_s ← down_k(p_s)
14         p_e ← down_k(p_e + 1) − 1
15         if k < t then
               // Equivalent to map_k(·) for the left- and rightmost 0's.
16             p_s ← B_k^(F).select_0(B_k^(L).rank_0(p_s) + 1)
17             p_e ← B_k^(F).select_0(B_k^(L).rank_0(p_e + 1))
18         p_s ← up_k(p_s)
19         p_e ← up_k(p_e)
20         t ← t − k + 1
21     end
22     return (p_s, p_e)
```

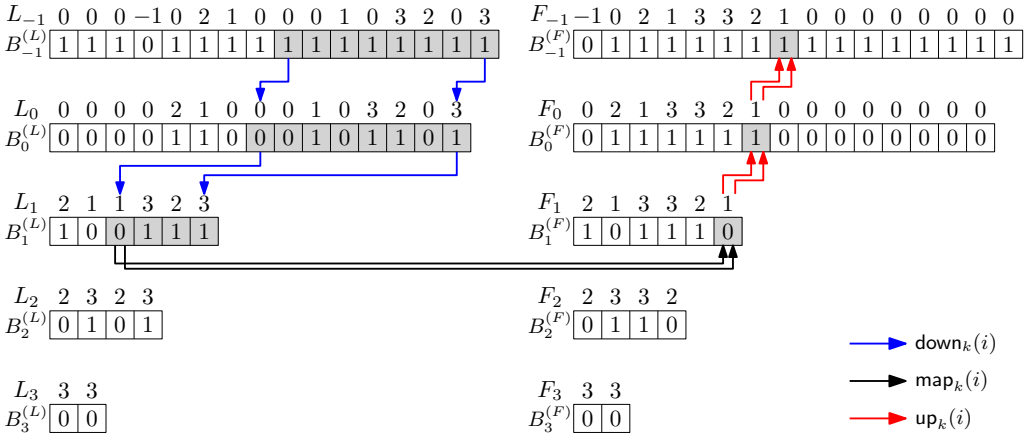
$L[i] \geq k$ if $k = E(P).\mathsf{rank}_\infty(|P|)$. This task can be done by computing the interval $[i_s, i_e]$ at level k that corresponds to the current suffix range $[p_s, p_e]$ (i.e. the interval on level -1) where $i_s = \mathsf{down}_k(p_s)$ and $i_e = \mathsf{down}_k(p_e + 1) - 1$. This is because L_k is a subsequence of L, whose elements are equal to or greater than k.

If $k < E(P).\mathsf{rank}_\infty(|P|)$, the target suffixes correspond to the elements that are equal to k, which can be determined by positions $\{i_s \leq i \leq i_e \mid B_k^{(L)}[i] = 0\}$. For example, in Figure 5-(a), the corresponding interval at level $k = 1$ of the suffix range $[8, 15]$ is $[2, 5]$. We can have $\{2\}$ as the set of positions whose value of the bitvector $B^{(L)}$ is 0. This position 2 at level 1 corresponds to the position 10 at level -1 that represent the entire elements of L. It means that $E(T[\mathsf{SA}[10]..])$ is the only target suffix, and we need to compute its LF-mapping to obtain the updated suffix range. We have $\mathsf{LF}[10] = 7$, and the updated suffix range is $[7, 7]$, which exactly matches the one obtained by the proposed algorithm.

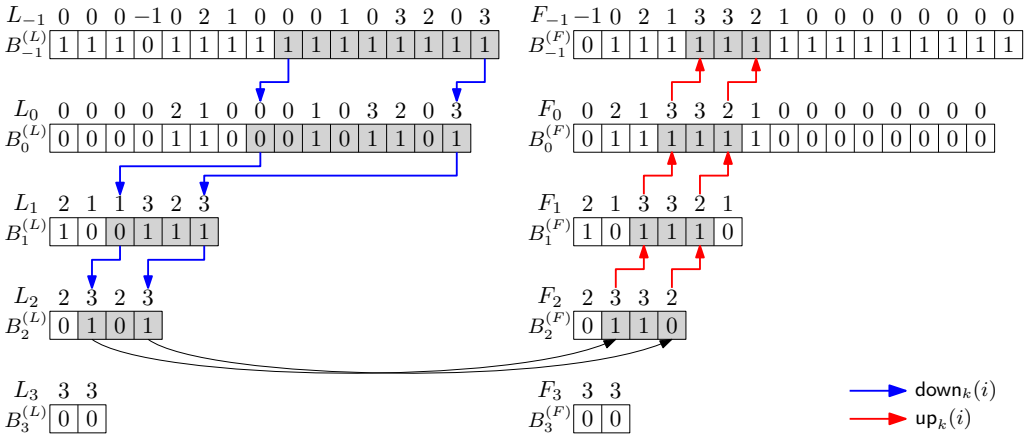
If $k = E(P).\mathsf{rank}_\infty(|P|)$, all the elements within the interval $i_s \leq i \leq i_e$ correspond to the target suffixes. For example, in Figure 5-(b), we have $[i_s, i_e] = [1, 3]$. The positions at level -1 that corresponds to positions 1,2, and 3 at level $k = 2$, are 12, 13, and 15, respectively. Since $\mathsf{LF}[12] = 4$, $\mathsf{LF}[13] = 6$, and $\mathsf{LF}[15] = 5$, the updated suffix range is $[4, 6]$.

6.2 Computing the LF-mapping of the Target Suffixes

We have shown that the target suffixes can be correctly identified after calling $\mathsf{down}_k(\cdot)$. Thus the correctness of the algorithm relies on the correctness of Lines 15–17, which compute the positions on $B_k^{(F)}$ that correspond to the positions on $B_k^{(L)}$. If this mapping can identify the target suffixes in terms of an interval on F_k, then computing $\mathsf{up}_k(\cdot)$ will give the desired suffix range that is correctly updated with respect to the given $E(P)$ and k.



(a) Case 1: $k = 1 < E(P).\mathsf{rank}_\infty(|P|) = 2$



(b) Case 2: $k = E(P).\mathsf{rank}_\infty(|P|) = 2$

Figure 5 An iteration of Algorithm 3 to update a suffix range. Given a suffix range $[8, 15]$ for the currently searched pattern $E(P) = \infty\infty$, it computes the suffix range for $k \oplus E(P)$.

It is obvious for the case $k < E(P).\mathsf{rank}_\infty(|P|)$. Because 0-bits (and their corresponding $L[i]$-values) in both bitvectors $B^{(L)}_k$ (and L_k) and $B^{(F)}_k$ (and F_k) are associated in order, it is sufficient to find the positions on $B^{(F)}_k$ that correspond to the first and last occurrence of 0-bit within the interval on $B^{(L)}_k$. We can determine the (relative) positions of the first and last 0-bits within the interval on $B^{(L)}_k$ using $B^{(L)}_k.\mathsf{rank}(\cdot)$ queries, and perform the mapping into their corresponding positions using $B^{(F)}_k.\mathsf{select}(\cdot)$ queries. This can be done by performing $\mathsf{map}_k(\cdot)$ at the leftmost and rightmost 0's in the interval.

It is not trivial if we are in the case $k = E(P).\mathsf{rank}_\infty(|P|)$. We observe that $\mathsf{up}_k(i) < \mathsf{up}_k(j)$ for $0 \le i < j < |B^{(F)}_k|$ because $\mathsf{up}_k(\cdot)$ is order-preserving. Suppose we can compute the number of suffixes $E(T[\mathsf{SA}[i]..])$'s such that $L[i] \ge k$ and $\mathsf{LF}[i] < p'_s$ where (p'_s, p'_e) is the suffix range of $k \oplus E(P)$, say a to denote this number. Let b be the number of the target suffixes. Then the suffixes that belong to the updated suffix range (p'_s, p'_e) must correspond to the interval $[a, a + b - 1]$ on $B^{(F)}_k$. We claim the following:

▶ **Lemma 9.** *Let (p_s, p_e) be the suffix range of an encoded pattern $E(P)$, and let i and j be integers such that $L[i], L[j] \ge E(P).\mathsf{rank}_\infty(|P|)$. If $i < p_s \le j$ or $i \le p_e < j$, then $\mathsf{LF}[i] < \mathsf{LF}[j]$.*

Proof. Let $l = \mathsf{lcp}(E(T[\mathsf{SA}[i]..]), E(T[\mathsf{SA}[j]..]))$ and $t = E(T[\mathsf{SA}[i]..]).\mathsf{rank}_\infty(l)$. Clearly, $t \leq E(P).\mathsf{rank}_\infty(|P|)$ because one of $E(T[\mathsf{SA}[i]..])$ and $E(T[\mathsf{SA}[j]..])$ has $E(P)$ as its prefix while the other does not. We also have $E(T[\mathsf{SA}[i]..]) < E(T[\mathsf{SA}[j]..])$ since $i < j$. By Lemma 3, $L[i] \oplus E(T[\mathsf{SA}[i]..]) < L[j] \oplus E(T[\mathsf{SA}[j]..])$ ◀

Note that it does not matter if $\mathsf{tLF}_k(\cdot)$ is not order-preserving within $p_s \leq i, j \leq p_e$. Among the suffixes $E(T[\mathsf{SA}[i]..])$ such that $L[i] \geq k$, the suffixes that are smaller than $k \oplus E(P)$ will be mapped into those that are smaller than $k \oplus E(P)$ after applying the LF-mapping function, and the larger suffixes remain larger. Let $\mathcal{I}$ be the interval at level k that correspond to a suffix range (p_s, p_e). Then computing the set $\{\mathsf{map}_k(i) \mid i \in \mathcal{I}\}$ is identical to $\mathcal{I}$ itself, which proves the correctness of the mapped interval on $B_k^{(F)}$ in this case.

6.3 Time Complexity

The search time of Algorithm 3 depends on how many times $\mathsf{down}_k(\cdot)$ and $\mathsf{up}_k(\cdot)$ are called over the iterations. Note that each of $\mathsf{down}_k(\cdot)$ and $\mathsf{up}_k(\cdot)$ takes $\mathcal{O}(k)$ time. We have $\sum_i k_i < |P|$ because k_i is the number of ∞'s to be substituted at iteration i. Once an ∞ is substituted with an integer, it remains the same until the end of the search process. The total number of substitution is bounded by the pattern length $|P|$, thus the search time is also bounded by $\mathcal{O}(|P|)$ time. This completes the proof of the main theorem.

7 Conclusion and Open Problems

In this paper, we have proposed a $3n+o(n)$-bit index for the Cartesian tree matching problem. We achieved this space bound by representing the correspondence between adjacent encoded suffixes using unary code. To bound the search time within linear time, we introduced the concept of a *trimmed LF-mapping function*. The trimmed LF-mapping function has special properties which enable us to compute the updated suffix ranges efficiently.

We also have open problems that should be addressed in the future work as follows:

- **Can we achieve $2n + o(n)$ bits?** We need to achieve $2n + o(n)$ bits to make it *succinct* regarding the entropy bound of Cartesian trees. We do not think that $B_k^{(L)}$ can be further reduced, but we think $B_k^{(F)}$ can be represented in a space-efficient way. Nevertheless, it seems quite challenging to represent $B_k^{(F)}$ within $o(n)$ bits to achieve succinctness.
- **Can we efficiently locate the occurrences?** The standard method that uses a sampled suffix array may take a long time especially when a pattern is long and it is also very frequent, because the time complexity has an $\mathcal{O}(|P| \cdot occ)$ term, which would result in $\mathcal{O}(n^2)$ time for locating occurrences. Although it might be a rare case in practice, the algorithm should be improved in order to bound the worst case complexity.
- **Can we apply the trimmed LF-mapping to other indexing problems?** We may apply the trimmed LF-mapping when L-values or pattern lengths are bounded by $\mathcal{O}(1)$. We can also extend it into some other matching problems where the prepending operation to a suffix cannot be represented in a single value.

References

1 Amihood Amir, Timothy M. Chan, Moshe Lewenstein, and Noa Lewenstein. On hardness of jumbled indexing. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 114–125, 2014. doi:10.1007/978-3-662-43948-7_10.

2 Amihood Amir and Eitan Kondratovsky. Sufficient conditions for efficient indexing under different matchings. In *Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 6:1–12, 2019. doi:10.4230/LIPIcs.CPM.2019.6.

3 Brenda S. Baker. Parameterized pattern matching: Algorithm and applications. *Journal of Computer and System Sciences*, 52:28–42, 1996. doi:10.1006/jcss.1996.0003.

4 Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management*, 49(1):392–404, 2013. doi:10.1016/j.ipm.2012.08.003.

5 Vincent Cohen-Addad, Laurent Feuilloley, and Tatiana Starikovskaya. Lower bounds for text indexing with mismatches and differences. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1146–1164, 2019. doi:10.1137/1.9781611975482.70.

6 Richard Cole and Ramesh Hariharan. Faster suffix tree construction with missing suffix links. *SIAM Journal on Computing*, 33(1):26–42, 2003. doi:10.1137/S0097539701424465.

7 Maxime Crochemore, Costas S. Iliopoulos, Thierry Lecroq, and Y. J. Pinzon. Approximate string matching in musical sequences. In *Proceedings of the Prague Stringology Conference*, pages 1–11, 2001.

8 Maxime Crochemore, Costas S. Iliopulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Order-preserving indexing. *Theoretical Computer Science*, 638:122–135, 2016. doi:10.1016/j.tcs.2015.06.050.

9 Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68:610–625, 2014. doi:10.1007/s00453-012-9683-x.

10 Simone Faro, Thierry Lecroq, and Kunsoo Park. Fast practical computation of the longest common cartesian substring of two strings. In *Proceedings of the Prague Stringology Conference*, pages 48–60, 2020.

11 Paolo Ferragina and Giovanni Manzini. Compression boosting in optimal linear time using the burrows-wheeler transform. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 655–663, 2004. URL: https://dl.acm.org/doi/10.5555/982792.982892.

12 Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffery Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006. doi:10.1145/1198513.1198521.

13 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Structural pattern matching - succinctly. In *Proceedings of the 28th International Symposium on Algorithms and Computation (ISAAC)*, pages 35:1–13, 2017. doi:10.4230/LIPIcs.ISAAC.2017.35.

14 Arnab Ganguly, Rahul Shah, and SharmaV. Thankachan. pbwt: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 397–407, 2017. doi:10.1137/1.9781611974782.25.

15 Paweł Gawrychowski, Samah Ghazawi, and Gad M. Landau. On indeterminate string matching. In *Proceedings of the 31th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 14:1–14, 2020. doi:10.4230/LIPIcs.CPM.2020.14.

16 Alexander Golynski, Roberto Grossi, Ankur Gupta, Rajeev Raman, and Satti Srinivasa Rao. On the size of succinct indices. In *Proceedings of the 15th European Symposium on Algorithms (ESA)*, pages 371–382, 2007. doi:10.1007/978-3-540-75520-3_34.

17 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003. URL: https://dl.acm.org/doi/10.5555/644108.644250.

18 Geonmo Gu, Siwoo Song, Simone Faro, Thierry Lecroq, and Kunsoo Park. Fast multiple pattern cartesian tree matching. In *Proceedings of the 14th International Workshop on Algorithms and Computations (WALCOM)*, pages 107–119, 2020. doi:10.1007/978-3-030-39881-1_10.

19 Diptarama Hendrian. Generalized dictionary matching under substring consistent equivalence relations. In *Proceedings of the 14th International Workshop on Algorithms and Computations (WALCOM)*, pages 120–132, 2020. doi:10.1007/978-3-030-39881-1_11.

20 Juha Kärkkäinen and Simon J. Puglisi. Fixed block compression boosting in fm-indexes. In *Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 174–184, 2011. doi:10.1007/s00453-018-0475-9.

21 Natsumi Kikuchi, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. Computing covers under substring consistent equivalence relations. In *Proceedings of the 27th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 131–146, 2020. doi:10.1007/978-3-030-59212-7_10.

22 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014. doi:10.1016/j.tcs.2013.10.006.

23 Sung-Hwan Kim and Hwan-Gue Cho. Indexing isodirectional pointer sequences. In *Proceedings of the 31st International Symposium on Algorithms and Computation (ISAAC 2020)*, pages 35:1–35:15, 2020. doi:10.4230/LIPIcs.ISAAC.2020.35.

24 Sung-Hwan Kim and Hwan-Gue Cho. Simpler fm-index for parameterized string matching. *Information Processing Letters*, page 106026, 2020. (Online available). doi:10.1016/j.ipl.2020.106026.

25 Moshe Lewenstein. Indexing with gaps. In *Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 135–143, 2011. doi:10.1007/978-3-642-24583-1_14.

26 Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 229–241, 2007. doi:10.1007/978-3-540-75530-2_21.

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

28 Sung Gwan Park, Amihood Amir, Gad M. Landau, and Kunsoo Park. Cartesian tree matching and indexing. In *Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 16:1–14, 2019. doi:10.4230/LIPIcs.CPM.2019.16.

29 Tetsuo Shibuya. Generalization of a suffix tree for RNA structural pattern matching. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 393–406, 2000. doi:10.1007/s00453-003-1067-9.

A Further Discussion

In this section, we discuss several considerations that can be addressed in the context of developing the compact index of Cartesian tree matching and its extension, including the problems mentioned in Section 7.

A.1 Reducing the Required Number of Bits Further

Although we achieved $3n + o(n)$ bits by representing $B_{-1}^{(L)}$, $B_{-1}^{(F)}$ and $B_0^{(F)}$ in $\mathcal{O}(\lg n)$ bits, we did not use any techniques to reduce the space occupancy for the other bitvectors. A straightforward method to reduce the space occupancy further is to use compressed bit vectors. We may also apply compression boosting techniques such as level-wise [11] and block-wise compression [26, 20] to bitvectors.

We can also notice that there must be some redundancies in bitvectors, because $B_k^{(F)}$ is not an arbitrary permutation of $B_F^{(L)}$. Although we used the fact that $B_0^{(F)}$ has a certain form, perhaps the other bitvectors $B_k^{(F)}$'s can be represented more efficiently rather than independently of $B_k^{(L)}$'s. We can make several observations regarding bitvectors and operations related to them, some of which are as follows.

▶ **Proposition 10.** *For $k \geq -1$ and $0 \leq i, j < |B_k^{(L)}|$, the following facts hold:*

1. *If $B_k^{(L)}[i] = 0$, then $\mathsf{map}_k(i) \geq i$.*
2. *If $B_k^{(L)}[i] = 1$, then $\mathsf{map}_k(i) \leq i$.*
3. *If $i < j$ and $B_k^{(L)}[i] = 1$ and $B_k^{(L)}[j] = 0$, then $\mathsf{map}_k(i) < \mathsf{map}_k(j)$.*

Proof. (1) For $0 \leq j < |B_k^{(L)}|$, let $f(j) = F_k.\mathsf{select}_x(L_k.\mathsf{rank}_x(j) + 1)$ where $x = L_k[j]$. Suppose there exists $0 \leq i < |B_k^{(L)}|$ such that $B_k^{(L)}[i] = 0$ and $i' = \mathsf{map}_k(i) < i$. Because the number of occurrences of k in $L_k[0..i]$ and $F_k[0..i']$ is the same, there must exist $0 \leq j < i$ such that $L[j] \neq k$ and $f(j) > i$. When we trace up to F_{-1}, which is order-preserving, this implies $\mathsf{LF}[i] < \mathsf{LF}[j]$. However, since we have $j < i$ and $L[j] > k = L[i]$, $E(T[\mathsf{SA}[\mathsf{LF}[j]]..]) = L[j] \oplus E(T[\mathsf{SA}[j]..]) < L[i] \oplus E(T[\mathsf{SA}[i]..]) = E(T[\mathsf{SA}[\mathsf{LF}[i]]..])$. This implies $\mathsf{LF}[j] < \mathsf{LF}[i]$. Contradiction.

(2) Note that $B_k^{(F)}.\mathsf{rank}_0(i+1)+B_k^{(F)}.\mathsf{rank}_1(i+1) = B_k^{(L)}.\mathsf{rank}_0(i+1)+B_k^{(L)}.\mathsf{rank}_1(i+1) = i$. Suppose there exists $\mathsf{map}_k(i) > i$ for i such that $B_k^{(L)}[i] = 1$. Then $B_k^{(F)}.\mathsf{rank}_1(i+1) < B_k^{(L)}.\mathsf{rank}_1(i+1)$. Since $\mathsf{map}_k(j) \geq j$ for $0 \leq j < |B_k^{(L)}|$ such that $B_k^{(L)}[j] = 0$, we also have $B_k^{(F)}.\mathsf{rank}_0(i+1) \leq B_k^{(L)}.\mathsf{rank}_0(i+1)$. Therefore $B_k^{(F)}.\mathsf{rank}_0(i+1) + B_k^{(F)}.\mathsf{rank}_1(i+1) < B_k^{(L)}.\mathsf{rank}_0(i+1) + B_k^{(L)}.\mathsf{rank}_1(i+1)$. Contradiction.

(3) Immediate from $\mathsf{map}_k(i) \leq i < j \leq \mathsf{map}_k(j)$. ◀

We believe that we can reduce the number of bits required to implement $B_k^{(F)}$'s if we can find more properties on the relation between $B_k^{(L)}$ and $B_k^{(F)}$. However, it is not trivial if we can break it down into $o(n)$ bits. One possibility to achieve $o(n)$ bits is the use of sparse bitvector if we can make $B_k^{(F)}$'s significantly imbalanced in terms of the number of 0- and 1-bits.

A.2 Locating Occurrences

To locate the occurrences, we can sample entries of the suffix array $\mathsf{SA}[i]$ as we conventionally do for other problems. More specifically, we build an array $\hat{\mathsf{SA}}$ storing $\mathsf{SA}[i]$ if it is divisible by δ.

$$\hat{\mathsf{SA}} = \langle \mathsf{SA}[i] \rangle_{i|\mathsf{SA}[i]=0 \mod \delta} \tag{13}$$

Using a bitvector M of length $n+1$, we mark the sampled entry; i.e., $M[i] = 1$ iff $\mathsf{SA}[i] = 0 \mod \delta$. Algorithm 4 traces from a suffix backward until it meets the sampled entry of the suffix array.

Algorithm 4 Locating Occurrences.

```
1  function Locate(i, t = E(P).rank_∞(|P|)):
2      R ← {}
3      for j = 0 To δ − 1 do
4          if i = 0 then break
5          if M[i] = 1 then R ← R ∪ {ŜA[M.rank_1(i)] + j}
6          k ← min{t, L[i]}
7          i ← tLF_k(i)
8          t ← t − k + 1
9      end
10     return R
```

Unlike the conventional technique of the sampled suffix array, tracing from a single suffix does not necessarily report one entry in the sampled suffix array because $\mathsf{tLF}_k(i)$ does not guarantee to be the same as $\mathsf{LF}[i]$; during the iterations of the algorithm, they may be jumbled and it possibly jumps into another suffix other than its previous one. Nevertheless, reported positions are correct if we collect the reported positions over all suffixes in the suffix range. Note that at the end of each iteration j of the outer loop, $E(T[\mathsf{SA}[i]..])$ has the form of $k_1 \oplus (\cdots (k_j \oplus E(P)) \cdots)$, thus even if it jumps into another suffix, it is still correct one.

The time for locating all the occurrence is $\mathcal{O}((|P|+\delta) \cdot occ)$. The time taken for computing k in Line 6 can be done in $\mathcal{O}(\min\{t, L[i]\})$ time, because we can repeatedly move into the next level using $B_k^{(L)}.\mathsf{rank}_1(\cdot)$ until either we arrive at level t or we reach a 0-bit at level $L[i]$. A single computation of $\mathsf{tLF}_k(i)$ may take $\mathcal{O}(L[i])$ time, but it is bounded by $\mathcal{O}(|P|+\delta)$ because the sum of all k's over all iterations is the number of ∞'s that are substituted. There are at most $|P|$ ∞'s remaining in the searched part, and there are possibly $\delta - 1$ more ∞'s to be substituted during the iterations.

A.3 Extensions

We may apply the trimmed LF-mapping when L-values or pattern lengths are bounded by $\mathcal{O}(1)$. For example, the indexing methods for the parameterized string matching [14, 24] have a $\log \sigma$ factor in their space complexity, which comes from the fact that $L[i]$ is bounded by σ. If they are bounded by some constant c, we may adopt this technique with unary coded L-values to achieve $\mathcal{O}(n)$ bits and $\mathcal{O}(m)$ search time.

We can extend the concept of the trimmed LF-mapping introduced in this paper for other matching problems where the prepending operation to a suffix cannot be represented in a single value. The resulting structure would look like the combination of wavelet trees [17, 27] and DAC [4] with a larger size of blocks. For example, in [23], the number of pointers starting at a particular position is constrained to be at most 1. The main reason of this constraint was that it may cause an unbounded time complexity if there are multiple pointers having the same starting position. We can use the mechanism of the trimmed LF-mapping to relax this constraint.

A.4 Construction Time

In [28], it is shown that the suffix tree for the Cartesian tree matching problem can be built in randomized $\mathcal{O}(n)$ time or deterministic $\mathcal{O}(n \lg n)$ time based on the suffix tree construction algorithm based on the character oracles [6]. Once the suffix tree is constructed, we can compute the suffix array in $\mathcal{O}(n)$ time using the tree traversal. The array SA^{-1}, LF, L, and F can also be computed in order, each of which takes $\mathcal{O}(n)$ time; note that we can precompute $K(\cdot)$ required to compute L in $\mathcal{O}(n)$ time using Lines 2–7 of Algorithm 3. The remaining task is to build the corresponding bitvectors, which does not exceed this time bound.

String Sanitization Under Edit Distance: Improved and Generalized

Takuya Mieno
Kyushu University, Fukuoka, Japan
Japan Society for the Promotion of Science, Tokyo, Japan

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Leen Stougie
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Michelle Sweering
CWI, Amsterdam, The Netherlands

Abstract

Let W be a string of length n over an alphabet Σ, k be a positive integer, and $\mathcal{S}$ be a set of length-k substrings of W. The **ETFS** problem (**E**dit distance, **T**otal order, **F**requency, **S**anitization) asks us to construct a string X_{ED} such that: (i) no string of $\mathcal{S}$ occurs in X_{ED}; (ii) the order of all other length-k substrings over Σ (and thus the frequency) is the same in W and in X_{ED}; and (iii) X_{ED} has minimal edit distance to W. When W represents an individual's data and $\mathcal{S}$ represents a set of confidential patterns, the **ETFS** problem asks for transforming W to preserve its privacy and its utility [Bernardini et al., ECML PKDD 2019].

ETFS can be solved in $\mathcal{O}(n^2k)$ time [Bernardini et al., CPM 2020]. The same paper shows that **ETFS** cannot be solved in $\mathcal{O}(n^{2-\delta})$ time, for any $\delta > 0$, unless the Strong Exponential Time Hypothesis (SETH) is false. Our main results can be summarized as follows:

- An $\mathcal{O}(n^2 \log^2 k)$-time algorithm to solve **ETFS**.
- An $\mathcal{O}(n^2 \log^2 n)$-time algorithm to solve **AETFS** (**A**rbitrary lengths, **E**dit distance, **T**otal order, **F**requency, **S**anitization), a generalization of **ETFS** in which the elements of $\mathcal{S}$ can have arbitrary lengths.

Our algorithms are thus optimal up to subpolynomial factors, unless SETH fails.

In order to arrive at these results, we develop new techniques for computing a variant of the standard dynamic programming (DP) table for edit distance. In particular, we simulate the DP table computation using a directed acyclic graph in which every node is assigned to a smaller DP table. We then focus on redundancy in these DP tables and exploit a tabulation technique according to dyadic intervals to obtain an optimal alignment in $\tilde{\mathcal{O}}(n^2)$ total time[1]. Beyond string sanitization, our techniques may inspire solutions to other problems related to regular expressions or context-free grammars.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, data sanitization, edit distance, dynamic programming

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.19

Funding *Takuya Mieno*: JSPS KAKENHI Grant Number JP20J11983
Leen Stougie: Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003
Michelle Sweering: Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003

[1] The notation $\tilde{\mathcal{O}}(f)$ denotes $\mathcal{O}(f \cdot \mathrm{polylog}(f))$.

Acknowledgements We wish to thank Grigorios Loukides (King's College London) for useful discussions about improving the presentation of this manuscript.

1 Introduction

Let us start with an example to ensure that the reader is familiar with the basic motivation behind the computational problem investigated here. Consider a sequence W of items representing a user's on-line purchasing history. Further consider a fragment (or a subsequence) of W denoting that the user has first purchased unscented lotions and zinc/magnesium supplements and then unscented soaps and cotton balls in extra-large bags. By having access to W and to the respective domain knowledge, one can infer that the user is probably pregnant and close to the delivery date.

Data sanitization, also known as *knowledge hiding*, is a privacy-preserving data mining process aiming to prevent the mining of confidential knowledge from published datasets; it has been an active area of research for the past 25 years [15, 29, 31, 19, 32, 20, 1, 2, 18, 21, 27, 11, 7, 8, 10]. Informally, it is the process of disguising (hiding) confidential information in a given dataset. This process typically incurs some data utility loss that should be minimized. Thus, naturally, privacy constraints and utility objective functions lead to the formulation of combinatorial optimization problems. From a fundamental perspective, it is thus relevant to be able to establish some formal guarantees.

A string W is a sequence of letters over some alphabet Σ. Individuals' data, in domains ranging from web analytics to transportation and bioinformatics, are typically represented by strings. For example, when Σ is a set of items, W can represent a user's purchasing history [5]; when Σ is a set of locations, W can represent a user's location profile [33]; and when Σ is the DNA alphabet, W can represent a patient's genome sequence [25]. Such strings commonly fuel up a gamut of applications; in particular, frequent pattern mining applications [4]. For example, frequent pattern mining from location history data facilitates route planning [14]; frequent pattern mining from market-basket data facilitates business decision making [5]; frequent pattern mining from genome sequences facilitates clinical diagnostics [25]. To support these applications in a privacy-preserving manner, individual sequences are often being disseminated after they have been sanitized.

Towards this end, Bernardini et al. have recently formalized the following string sanitization problem under edit distance [8]. Let W be a string of length n over an alphabet Σ, k be a positive integer, and $\mathcal{S}$ be a set of length-k substrings of W. Set $\mathcal{S}$ is conceptually seen as an *antidictionary*: a set of *sensitive patterns* modelling private or confidential information. The **ETFS** problem (**E**dit distance, **T**otal order, **F**requency, **S**anitization) asks us to construct a string X_{ED} such that: (i) no string of $\mathcal{S}$ occurs in X_{ED}; (ii) the order of all other length-k substrings over Σ is the same in W and in X_{ED}; and (iii) X_{ED} has minimal edit distance to W. In order to obtain a feasible solution string, we may need to extend Σ to $\Sigma_{\#} = \Sigma \cup \{\#\}$, which includes a special letter $\# \notin \Sigma$.

▶ **Example 1.** Let $W = \texttt{ecabaaaaabbbadf}$ over alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}\}$ be the input string. Further let $k = 3$ and the set of sensitive patterns be $\mathcal{S} = \{\texttt{aba}, \texttt{baa}, \texttt{aaa}, \texttt{aab}, \texttt{bba}\}$. Consider the following three feasible (sanitized) strings: $X_{\mathrm{TR}} = \texttt{eca\#cab\#abb\#bbb\#bad\#adf}$, $X_{\mathrm{MIN}} = \texttt{ecabbb\#badf}$ and $X_{\mathrm{ED}} = \texttt{ecab\#aa\#abbb\#badf}$. All three strings contain *no sensitive pattern* and preserve the *total order* and thus the *frequency* of all non-sensitive length-3 patterns of W: X_{TR} is the trivial solution of interleaving the non-sensitive length-3 patterns of W with $\#$; X_{MIN} is the *shortest* possible such string; and X_{ED} is a string *closest* to W in terms of edit distance.

A simple $\mathcal{O}(n^2k|\Sigma|)$-time solution [8] to **ETFS** can be obtained via employing approximate regular expression matching. Consider the regular expression R that encodes all feasible solution strings. The size of R is $\mathcal{O}(nk|\Sigma|)$. By aligning W and R using the standard quadratic-time algorithm [28], we obtain an optimal solution X_{ED} in $\mathcal{O}(n^2k|\Sigma|)$ time for **ETFS**. Bernardini et al. showed that this can be improved to $\mathcal{O}(n^2k)$ time [9]. Let us informally describe their algorithm. (A formal description of their algorithm follows in Section 2.) We use a dynamic programming (DP) table similar to the standard edit distance algorithm. We write the letters of the input string W on the top of the first row. Since we do not know the exact form of the output string X_{ED}, we write the non-sensitive length-k patterns to the left of the first column interleaved by special # letters. We then proceed to fill this table using recursive formulae. The formulae are more involved than the edit distance ones to account for the possibility to *merge* consecutive non-sensitive patterns (e.g., `eca` and `cab` are merged to `ecab` in Example 1) and to expand the #'s into longer gadgets that may contain up to $k-1$ letters from Σ (e.g., `#aa#` in Example 1). Once the DP table is filled, we construct an X_{ED} by tracing back an optimal alignment.

Bernardini et al. also showed, via a reduction from the weighted edit distance problem [12], that **ETFS** cannot be solved in $\mathcal{O}(n^{2-\delta})$ time, for any $\delta > 0$, unless the strong exponential time hypothesis (SETH) [22, 23] is false. We were thus also motivated to match this lower bound.

Our Results and Techniques

Our first main result is the following.

▶ **Theorem 2.** *The **ETFS** problem can be solved in $\mathcal{O}(n^2 \log^2 k)$ time.*

Let us also stress that the algorithm underlying Theorem 2 works under edit distance with arbitrary weights at no extra cost.

We also consider a generalized version of **ETFS**, which we denote by **AETFS** (**A**rbitrary lengths, **E**dit distance, **T**otal order, **F**requency, **S**anitization). The only difference in **AETFS** with respect to **ETFS** is that $\mathcal{S}$ can contain elements (sensitive patterns) of arbitrary lengths. This generalization is evidently more useful as it drops the restriction of fixed-length sensitive patterns; it also turns out to be algorithmically much more challenging. In both **ETFS** and **AETFS**, we make the standard assumption that substrings of W are represented as intervals over $[0, n-1]$, and thus each element in $\mathcal{S}$ has an $\mathcal{O}(1)$-sized representation. We further assume that $\mathcal{S}$ satisfies the properties of *closure* and *minimality* (formally defined in Section 2), which in turn ensure that $\mathcal{S}$ has an $\mathcal{O}(n)$-sized representation.

▶ **Example 3.** Consider the same input string $W =$ `ecabaaaaabbbadf` as in Example 1. Further let $k = 3$ and the set of sensitive patterns be $\mathcal{S} = \{$`aba`, `aa`, `abbba`$\}$. Then, string $Y_{\mathrm{ED}} =$ `ecab#abb#bbbadf` is a feasible string and is a closest to W in terms of edit distance. Notice that, we cannot merge all of the three consecutive non-sensitive patterns `abb`, `bbb`, and `bba` into one since it will result in an occurrence of the sensitive pattern `abbba`; we thus rather create `abb#bbba`.

Our second main result is the following.

▶ **Theorem 4.** *The **AETFS** problem can be solved in $\mathcal{O}(n^2 \log^2 n)$ time.*

Our algorithms are thus optimal up to subpolynomial factors, unless SETH fails. Let us describe the main ideas behind the new techniques we develop. As in Example 3, a sensitive pattern of length greater than k might be generated by merging multiple non-sensitive

patterns. In **AETFS**, we have to consider avoiding such *invalid* merge operations. If we enumerate all *valid* combinations of merging non-sensitive patterns, and run the DP for **ETFS** for all the cases, then we can obtain an optimal solution to **AETFS**. Our main idea for reducing the time complexity is to carefully maintain a directed acyclic graph (DAG) for representing all such valid combinations. We first construct the DAG, and then plug a small DP table into each node of the DAG. This technique gives us an $\mathcal{O}(n^3)$-time solution to **AETFS**. To achieve $\tilde{\mathcal{O}}(n^2)$ time, we focus on redundancy in the DP tables. When the size of the DP tables is large, there must be multiple sub-tables corresponding to the same pair of strings. Before propagating, we precompute lookup table structures of size $\mathcal{O}(n^2 \log^2 n)$ according to dyadic intervals on $[0, n-1]$. To this end, we modify the data structure proposed in [13]. Then, we decompose the DP tables into sub-tables according to these dyadic intervals. We compute only boundaries of such sub-tables using the precomputed lookup table structures, and thus, we obtain an optimal alignment for **AETFS** in $\mathcal{O}(n^2 \log^2 n)$ total time. By applying the same technique to **ETFS**, we obtain an $\mathcal{O}(n^2 \log^2 k)$-time solution, which improves the state of the art by a factor of $k/\log^2 k$ [9].

In a nutshell, our main technical contribution is that we manage to align a string of length n and a specific regular expression of size $\Omega(nk|\Sigma|)$ in $\tilde{\mathcal{O}}(n^2)$ time. We can also view the solution spaces of **ETFS** and **AETFS** as context-free languages. The main idea of our **AETFS** algorithm is to first preprocess a set N of non-terminals, such that we can later use them in $\mathcal{O}(n)$ time each. We then write the context-free language as a new language, which is accepted by a Deterministic Acyclic Finite State Automaton (DASFA), taking the set N as its terminals. In this paper, we develop several techniques to reduce the size of the DAFSA (cf. DAG) to $\tilde{\mathcal{O}}(n)$ and efficiently precompute the set N (cf. lookup tables) in $\tilde{\mathcal{O}}(n^2)$ time. Thus, beyond string sanitization, our techniques may inspire solutions to other problems related to regular expressions or context-free grammars.

Paper Organization

Section 2 introduces the basic definitions and notation used throughout, and also provides a summary of the currently fastest algorithm for solving **ETFS** [9]. In Section 3, we describe our lookup table structures. In Section 4, we present the $\mathcal{O}(n^3)$-time algorithm for solving **AETFS**. This algorithm is refined to an $\tilde{\mathcal{O}}(n^2)$-time algorithm, which is described in Section 5. Along the way, in Section 5, we also describe an $\tilde{\mathcal{O}}(n^2)$-time algorithm for solving **ETFS**.

2 Preliminaries

Strings

An *alphabet* Σ is a finite set of elements called *letters*. Let $S = S[0]S[1]\cdots S[n-1]$ be a *string* of length $|S| = n$ over an alphabet Σ of size $\sigma = |\Sigma|$. Let $\Gamma = \{\ominus, \oplus, \otimes\}$ be a set of special letters with $\Gamma \cap \Sigma = \emptyset$. By Σ^* we denote the set of all strings over Σ, and by Σ^k the set of all length-k strings over Σ. For two indices $0 \le i \le j \le n-1$, $S[i\,..\,j] = S[i]\cdots S[j]$ is the *substring* of S that starts at position i and ends at position j of S. By ε we denote the *empty string* of length 0. A *prefix* of S is a substring of the form $S[0\,..\,j]$, and a *suffix* of S is a substring of the form $S[i\,..\,n-1]$. Given two strings U and V we say that U has a *suffix-prefix overlap* of length $\ell > 0$ with V if and only if the length-ℓ suffix of U is equal to the length-ℓ prefix of V, i.e., $U[|U|-\ell\,..\,|U|-1] = V[0\,..\,\ell-1]$.

We fix a string W of length n over an alphabet Σ. We assume that $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$. If this is not the case, we use perfect hashing [17] to hash $W[i]$, for all $i \in [1, n]$, and obtain another string over $\Sigma = \{1, \ldots, n\}$ in $\mathcal{O}(n)$ time with high probability or in $\mathcal{O}(n \log n)$ time

deterministically via sorting. We consider the obtained string to be W. We also fix an integer $0 < k < n$. Unless specified otherwise, we refer to a length-k string or a *pattern* interchangeably. An occurrence of a pattern is uniquely defined by its starting position. Let $\mathcal{S}_k$ be the set representing the sensitive patterns as starting positions over $\{0, \ldots, n-k\}$ with the following closure property: for every $i \in \mathcal{S}_k$, any j for which $W[j\,..\,j+k-1] = W[i\,..\,i+k-1]$, must also belong to $\mathcal{S}_k$. That is, if an occurrence of a pattern is in $\mathcal{S}_k$, then all its occurrences are in $\mathcal{S}_k$. A substring $W[i\,..\,i+k-1]$ of W is called *sensitive* if and only if $i \in \mathcal{S}_k$; $\mathcal{S}_k$ is thus the complete set of occurrences of sensitive patterns. The difference set $\mathcal{I} = \{0, \ldots, n-k\} \setminus \mathcal{S}_k$ is the set of occurrences of length-k *non-sensitive* patterns.

For any substring U, we denote by $\mathcal{I}_U$ the set of occurrences in U of non-sensitive length-k strings over Σ. (We have that $\mathcal{I}_W = \mathcal{I}$.) We call an occurrence i the *t-predecessor* of another occurrence j in $\mathcal{I}_U$ if and only if i is the largest element in $\mathcal{I}_U$ that is less than j. This relation induces a *strict total order* on the occurrences in $\mathcal{I}_U$. We call a subset $\mathcal{J}$ of $\mathcal{I}_U$ a *t-chain* if for all elements in $\mathcal{J}$ except the minimum one, their t-predecessor is also in $\mathcal{J}$. For two strings U and V, chains $\mathcal{J}_U$ and $\mathcal{J}_V$ are *equivalent*, denoted by $\mathcal{J}_U \equiv \mathcal{J}_V$, if and only if $|\mathcal{J}_U| = |\mathcal{J}_V|$ and $U[u\,..\,u+k-1] = V[v\,..\,v+k-1]$, where u is the j-th smallest element of $\mathcal{J}_U$ and v is the j-th smallest of $\mathcal{J}_V$, for all $j \leq |\mathcal{J}_U|$.

Given two strings U and V the *edit distance* $d_{\mathrm{E}}(U, V)$ is defined as the minimum number of elementary edit operations (letter insertion, deletion, or substitution) that transform one string into the other. Each edit operation can also be associated with a cost: a fixed positive value. Given two strings U and V the *weighted edit distance* $d_{\mathrm{WE}}(U, V)$ is defined as the minimal total cost of a sequence of edit operations to transform one string into the other. We assume throughout that the three edit operations all have unit weight. However, as mentioned in Section 1, our algorithm for the **ETFS** problem (formally defined below) also works for arbitrary weights at no extra cost. The standard algorithm to compute the edit distance between two strings U and V [26] works by creating a $(|U|+1) \times (|V|+1)$ DP table D with $D[i][j] = d_{\mathrm{E}}(U[0\,..\,i-1], V[0\,..\,j-1])$. The sought edit distance is thus $d_{\mathrm{E}}(U, V) = D[|U|][|V|]$. Since we compute each table entry from the entries to the left, top and top-left in $\mathcal{O}(1)$ time, the algorithm runs in $\mathcal{O}(|U| \cdot |V|)$ time. Moreover, we can find an optimal (minimum cost) alignment by tracing back through the table.

The ETFS Problem

We formally define **ETFS**, one of the problems considered in this paper.

▶ **Problem 5 (ETFS).** *Given a string W of length n, an integer $k > 1$, and a set $\mathcal{S}_k$ (and thus set $\mathcal{I}$), construct a string X_{ED} which is at minimal (weighted) edit distance from W and satisfies:*
C1 *X_{ED} does not contain any sensitive pattern.*
P1 *$\mathcal{I}_W \equiv \mathcal{I}_{X_{\mathrm{ED}}}$, i.e., the t-chains $\mathcal{I}_W$ and $\mathcal{I}_{X_{\mathrm{ED}}}$ are equivalent.*

The AETFS Problem

The length of sensitive patterns in the **ETFS** setting is fixed. In what follows, we define a generalization of the **ETFS** problem which allows for arbitrary length sensitive patterns. Let $\mathcal{S}$ be a set of intervals with the two following properties (closure property and minimality property): (i) For every $[i, j] \in \mathcal{S}$, any $[i', j']$ for which $W[i'\,..\,j'] = W[i\,..\,j]$, must also belong to $\mathcal{S}$; and (ii) any proper sub-interval of $[i, j] \in \mathcal{S}$ is not in $\mathcal{S}$. It is easy to see that $|\mathcal{S}| \leq n$

from its minimality. Now, we redefine notions of sensitive and non-sensitive patterns as follows: A sensitive pattern is an *arbitrary length* substring $W[i\,..\,j]$ of W for each $[i,j] \in \mathcal{S}$. For a fixed k, a non-sensitive pattern is a length-k substring of W containing no sensitive pattern as a substring.

▶ **Problem 6 (AETFS).** *Given a string W of length n, an integer $k > 1$, and a set $\mathcal{S}$ (and thus set $\mathcal{I}$), construct a string Y_{ED} which is at minimal edit distance from W and satisfies:*
C1 *Y_{ED} does not contain any sensitive pattern.*
P1 *$\mathcal{I}_W \equiv \mathcal{I}_{Y_{\mathrm{ED}}}$, i.e., the t-chains $\mathcal{I}_W$ and $\mathcal{I}_{Y_{\mathrm{ED}}}$ are equivalent.*

The ETFS-DP Algorithm

For independent reading we describe here ETFS-DP, the algorithm from [9] that solves the **ETFS** problem in $\mathcal{O}(n^2k)$ time. The output string X_{ED} is a string that contains all non-sensitive patterns in the same order as in W. For each pair of consecutive non-sensitive patterns, their occurrences in X_{ED} are either (i) overlapping by $k-1$ letters (e.g., `eca` and `cab` in Example 1) or (ii) delimited by a string over $\Sigma_{\#}$ which contains no length-k string over Σ (e.g., `#aa#` in Example 1). We call such strings *gadgets*. For case (ii), we use the following regular expressions:

$$\Sigma^{<k} = (a_1|a_2|\ldots|a_{|\Sigma|}|\varepsilon)^{k-1},$$

where $\Sigma = \{a_1, a_2, \ldots, a_{|\Sigma|}\}$. Also, the special letters $\ominus, \oplus, \otimes \in \Gamma$ correspond to regular expressions $(\Sigma^{<k}\#)^*$, $\#(\Sigma^{<k}\#)^*$, and $(\#\Sigma^{<k})^*$, respectively. Let $N_0, N_1, \ldots, N_{|\mathcal{I}|-1}$ be the sequence of non-sensitive patterns sorted in the order in which they occur in W. In what follows, we fix string $T = \ominus N_0 \oplus N_1 \oplus \cdots \oplus N_{|\mathcal{I}|-1}\otimes$ of length $(k+1)|\mathcal{I}|+1$. String T corresponds to the regular expression R that represents the set of all feasible solutions (feasible strings) in which all non-sensitive patterns in the string are delimited by stings over $\Sigma_{\#}$. Moreover, we need to consider feasible strings in which a non-sensitive pattern overlaps the next one. Let M be a binary array of length $|\mathcal{I}|$ such that for each $0 \le i \le |\mathcal{I}|-1$, $M[i] = 1$ if $i > 0$ and N_{i-1} has a suffix-prefix overlap of length $k-1$ with N_i, and $M[i] = 0$ otherwise. Namely, $M[i] = 1$ implies that N_{i-1} and N_i can be merged for $0 < i \le |\mathcal{I}|-1$.

Let E be a table of size $((k+1)|\mathcal{I}|+1) \times (n+1)$. The rows of E correspond to string T defined above and the columns to string W. Note that the leftmost column corresponds to the empty string ε as in the standard edit distance DP table. Each cell $E[i][j]$ contains the edit distance between the regular expression corresponding to $T[0\,..\,i]$ and $W[0\,..\,j-1]$. We classify the rows of E into three categories: *gadget rows*; *possibly mergeable rows*; and *ordinary rows*. We call every row corresponding to a special letter in Γ a gadget row. Namely, rows with index $i \equiv 0 \bmod (k+1)$ are gadget rows. Also, we call every row corresponding to the last letter of a non-sensitive pattern a possibly mergeable row. Namely, rows with index $i \equiv -1 \bmod (k+1)$ are possibly mergeable rows. All the other rows are called ordinary rows. The recursive formula of ordinary rows is the same as in the standard edit distance solution:

$$E[i][j] = \min \begin{cases} E[i-1][j]+1, & \text{(insert)} \\ E[i][j-1]+1, & \text{(delete)} \\ E[i-1][j-1] + I[T[i] \neq W[j-1]], & \text{(match or substitute)}, \end{cases}$$

where I is an indicator function: $I[T[i] \neq W[j-1]] = 1$ if $T[i] \neq W[j-1]$, and 0 otherwise. Next, consider a possibly mergeable row $E[i][\cdot]$ which is the last row of the non-sensitive pattern N_h. If $M[h] = 0$, then the recursive formula is the same as that of ordinary rows.

Otherwise ($M[h] = 1$), N_{h-1} and N_h can be merged. Merging them means that the values in the previous mergeable row $E[i-k-1][\cdot]$ will be propagated to $E[i][\cdot]$ directly without considering the k rows below. Thus, the recursive formula is:

$$E[i][j] = \min \begin{cases} E[i-1][j]+1, & \text{(insert)} \\ E[i][j-1]+1, & \text{(delete)} \\ E[i-1][j-1]+I[T[i] \neq W[j-1]], & \text{(match or substitute)} \\ E[i-k-1][j]+1, & \text{if } M[h]=1 \text{ (insert and merge)} \\ E[i-k-1][j-1]+I[T[i] \neq W[j-1]], & \text{if } M[h]=1 \text{ (match or sub. and merge).} \end{cases}$$

Next, consider a gadget row $E[i][\cdot]$ which corresponds to a special letter in Γ. Because of the form of regular expressions corresponding to special letters, a # can either be inserted or substituted directly after a non-sensitive pattern, or be preceded by another # no more than k positions earlier. This results in the following recursive formula:

$$E[i][j] = \min \begin{cases} E[i-1][j]+1, & \text{(insert)} \\ E[i-1][j-1]+1, & \text{(substitute)} \\ E[i][j-1]+1, \ldots, E[i][\max\{0, j-k\}]+1, & \text{(delete or extend gadget).} \end{cases}$$

For completeness, we write down the recursive formula for initializing the leftmost column:

$$E[i][0] = \begin{cases} E[i-k-1][0]+1, & \text{if } i \equiv -1 \bmod (k+1) \wedge M[h]=1 \text{ (merge)} \\ E[i-1][0]+1, & \text{otherwise (no merge).} \end{cases}$$

Unlike in the standard setting [26], the edit distance between W and any string matching the regular expression R is not necessarily found in its bottom-right entry $E[|\mathcal{I}|(k+1)][|W|]$. Instead, it is found among the rightmost k entries of the last row (in case X_{ED} ends with a string in $\otimes$), and the rightmost entry of the second-last row (when X_{ED} ends with the last letter of the last non-sensitive pattern). After computing the edit distance value, we construct an X_{ED}. To do so, when computing each entry $E[i][j]$, we memorize a backward-pointer to an entry from which the minimum value for $E[i][j]$ was obtained. We then construct X_{ED} from right to left with respect to the sequence of edit operations corresponding to an optimal alignment obtained by the backward-pointers.

3 Compact Lookup Table Structure for Squared Blocks

In this section we consider the standard edit distance table, and propose a data structure which can answer some queries on a $b \times b$ sub-table of the DP table, which we call a *block*, corresponding to two strings of the same length. Our data structure is similar to the one proposed in [13], tailored, however, to our needs. We next provide some further definitions about blocks. Let B be a $b \times b$ block to be processed. The top (resp. bottom) row of B is called the input (resp. output) row of B. Similarly, the leftmost (resp. rightmost) column of B is called the input (resp. output) column of B. A cell in the input (resp. output) row or column is called an input (resp. output) cell.

In the following, we propose a lookup table for $b \times b$ blocks that computes all output cell values of a block in $\mathcal{O}(b)$ time for any given block and input cell values of the block. We modify the following known result to enhance it with a trace-back functionality.

▶ **Theorem 7** (Theorem 1 in [13]). *Given two strings both of length b corresponding to a $b \times b$ block, we can construct a data structure of size $\mathcal{O}(b^2)$ in $\mathcal{O}(b^2 \log b)$ time such that given any values for the input row and column of the block, the data structure can compute the output row and column of the block in $\mathcal{O}(b)$ time.*

In [13], the authors did not refer to tracing back, i.e., it is not clear how to obtain an optimal alignment using Theorem 7. We prove that we can trace back a shortest path to an output cell in a $b \times b$ block in $\mathcal{O}(b)$ time using $\mathcal{O}(b^2 \log b)$ additional space. This yields an optimal alignment. We next briefly describe the data structure of Theorem 7 and explain how we modify it.

3.1 Constructing a Data Structure for a Pair of Strings

For constructing the data structure of Theorem 7, Brubach and Ghurye [13] utilize the result of [30]. Instead, we use the following result by Klein [24], which is more general.

▶ **Theorem 8** ([24]). *Given an N-node planar graph with non-negative edge labels, we can construct a data structure of size $\mathcal{O}(N \log N)$ in $\mathcal{O}(N \log N)$ time such that given a node s in the graph and another node t on the boundary of the infinite face, the data structure can compute the maximum (or minimum) distance from s to t in $\mathcal{O}(\log N)$ time. Also, if the graph has constant degree, then we can compute the shortest s-t path in time linear in the length of the path.*

The lookup table structure for a block is constructed as follows (see [13] for details). Let B be a $b \times b$ block to be preprocessed. First, we regard B as a grid-graph of size $b \times b$. Namely, each node corresponds to a cell in the block, and each edge corresponds to an edit operation. Also, each edge is labeled by the weight of its corresponding edit operation. Then, we construct the data structure of Theorem 8 for the grid-graph. We denote this data structure by $\mathcal{D}_B$. Next, for each input cell u and each output cell v, we compute the weight of the shortest path from u to v, and store them to table M_B of size $(2b-1) \times (2b-1)$. Each row (resp. column) of M_B corresponds to each output (resp. input) cell of B. A table is called *monotone* if each row's minimum value occurs in a column which is equal to or greater than the column of the previous row's minimum. It is *totally monotone* if the same property is true for every sub-table defined by an arbitrary subset of the rows and columns of the given table. It is known that we can construct M_B so that it is totally monotone [16]. We thus construct $\mathcal{D}_B$ and M_B in $\mathcal{O}(b^2 \log b)$ time and space.

By Theorem 7, the size of the final data structure (that depends on the size of M_B) is $\mathcal{O}(b^2)$. However, $\mathcal{O}(b^2 \log b)$ working space is used for constructing $\mathcal{D}_B$. In our algorithm, we also use table M_B and keep the temporary data structure $\mathcal{D}_B$ to support tracing back operations efficiently.

3.2 Answering Queries and Tracing Back

Given a query input row and column, we can compute the output row and column in $\mathcal{O}(b)$ time using the SMAWK algorithm [3] for finding the minimum value in each row of an implicitly-defined totally monotone table, since M_B is totally monotone [13]. Note that, for each output cell v of B, we can also obtain an input cell s_v which is the starting cell of a shortest path ending at v from the result of SMAWK algorithm. Thus, by using data structure $\mathcal{D}_B$, we can obtain a shortest s_v-v path in time linear in the length of the path. To summarize, we obtain the following lemma.

▶ **Lemma 9.** *Given two strings both of length b corresponding to a $b \times b$ block, we can construct a data structure of size $\mathcal{O}(b^2 \log b)$ in $\mathcal{O}(b^2 \log b)$ time such that given any values for the input row and column of the block, the data structure can compute the output row and column of the block in $\mathcal{O}(b)$ time. Furthermore, given an output cell v and any other cell u in the block, we can compute a shortest u-v path in time linear in the length of the path.*

This data structure works under edit distance with arbitrary weights at no extra cost.

4 Sensitive Patterns of Arbitrary Lengths

In this section we propose a data structure with which we can solve the **AETFS** problem in time $\mathcal{O}(n^3)$. First, let us consider whether ETFS-DP can be applied directly to the **AETFS** problem. The **AETFS** problem is a generalization of the **ETFS** problem, and there are some differences between them: if there exists a *long sensitive pattern* of length longer than k, then we cannot apply the same logic for the possibly mergeable rows to the **AETFS** problem. This is because merging multiple non-sensitive patterns of length k may create a long sensitive pattern, while this sensitive pattern must be hidden. In contrast, if there exists a *short sensitive pattern* of length less than k, then we cannot apply the same logic for the gadget rows to the **AETFS** problem, since this may introduce a short sensitive pattern in a gadget. Thus **AETFS** is much more challenging.

Let $L = \mathcal{O}(n^2)$ denote the total length of long sensitive patterns. As a first step towards our main result, we prove the following lemma.

▶ **Lemma 10.** *The **AETFS** problem can be solved in $\mathcal{O}(k|\mathcal{I}|n + Ln)$ time.*

Note that Lemma 10 yields $\mathcal{O}(n^2k)$ time for **ETFS** because in this case $L = 0$. Lemma 10 thus generalizes Theorem 2 in [9]. In what follows, we propose a new data structure for solving the **AETFS** problem and prove Lemma 10. The main idea is to use multiple DP tables and link them under specific rules. Interestingly, our data structure is shaped as a DAG consisting of DP tables.

4.1 Long Sensitive Patterns

If there is a long sensitive pattern, we need to consider the case where multiple non-sensitive patterns are contained in a single sensitive pattern. (Recall that all non-sensitive patterns have fixed length k.) In this case, we cannot apply the ETFS-DP algorithm from [9] directly.

Let us consider the situation in which we have just finished computing a possibly mergeable row. We may be able to choose the next move from two candidates: either go down to the next (gadget) row or jump to the next possibly mergeable row if possible. We consider a *decision tree* $\mathcal{T}$ that represents all combinations of such choices at all possibly mergeable rows (inspect Figure 1). We regard $\mathcal{T}$ as a *tree of tables*, i.e., each node of $\mathcal{T}$ represents a small DP table. Let $E[0\,..\,(k+1)|\mathcal{I}|][0\,..\,n]$ be the DP table of the ETFS-DP algorithm described in Section 2. There are three types of nodes in $\mathcal{T}$: *root*, *#-node*, and *m-node*. The root represents sub-table $E[0\,..\,k][0\,..\,n]$. For each depth b with $1 \leq b \leq |\mathcal{I}| - 1$, the #-node at depth b represents sub-table $E[b(k+1)\,..\,(b+1)(k+1) - 1][0\,..\,n]$, and each m-node at depth b represents a copy of possibly mergeable row $E[(b+1)(k+1) - 1][0\,..\,n]$. Each edge (u, v) of $\mathcal{T}$ means that the bottom row values of u will be propagated to the top row of v. If there are multiple incoming edges $(u_1, v), (u_2, v), \ldots, (u_p, v)$ of a single node v, then we virtually consider a row $r[0\,..\,n]$ as the previous possibly mergeable row of v such that $r[j]$ is the minimum value between all j-th values in the last rows of $u_1, u_2, \ldots, u_p$ for each $0 \leq j \leq n$. We call a path that consists of only m-nodes an *m-path*.

We can solve the **AETFS** problem if we can simulate all valid combinations of merge operations represented by $\mathcal{T}$. However, we do not have to check all combinations explicitly. This is due to the fact that we can *prune* branches and *merge* nodes in the decision tree $\mathcal{T}$ as follows[2].

[2] Once the merge operation is applied to the decision tree, it is no longer a tree. However, we continue calling it the decision "tree" for convenience.

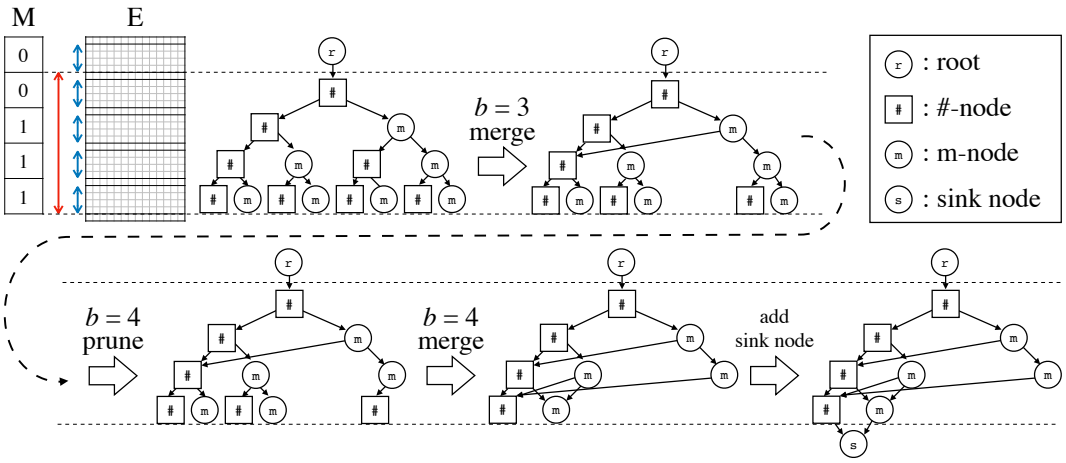


Figure 1 An example for pruning and merging a decision tree. The red arrow represents a long sensitive pattern, and blue arrows represent non-sensitive patterns. The tree on the top-left represents the decision tree after pruning or merging operations for depths $b = 0, 1, 2$. For $b = 3$, we merge two #-nodes by Rule 3. For $b = 4$, we prune an m-node by Rule 2 since merging four non-sensitive patterns following the node results creating a sensitive pattern. Furthermore, we merge three #-nodes by Rule 3, and merge two m-nodes by Rule 4. Finally, we add the sink node at the bottom.

For incremental $b = 1, 2, \ldots, |\mathcal{I}| - 1$, we edit $\mathcal{T}$ according to the following four rules:

Rule 1. If $M[b] = 0$, then prune all edges to m-nodes at depth b.

Rule 2. If there is a path $(v_1, v_2, \ldots, v_p)$ such that the depth of v_p is b, then $(v_2, \ldots, v_p)$ is an m-path, and v_1 and v_p respectively correspond to the length-k prefix and the length-k suffix of the same sensitive pattern. Hence prune the edge (v_{p-1}, v_p). In other words, we prune the edge (v_{p-1}, v_p) if merging the path strings results in creating a long sensitive pattern.

Rule 3. If there are multiple #-nodes at depth b, then merge all of them into a single #-node.

Rule 4. If there are multiple m-nodes $\{v_1, v_2, \ldots\}$ at depth b such that each u_i does not correspond to the length-k prefix of any sensitive pattern, where u_i is the parent of the starting node of the longest m-path ending at v_i, then merge such m-nodes into a single m-node.

Finally, we add the *sink node* under the decision tree such that the sink node corresponds to the bottom row $E[|\mathcal{I}|(k+1)][0\,..\,n]$, and each node at depth $|\mathcal{I}| - 1$ has only one outgoing edge to the sink node. We also rename the root to the *source node*.

After executing all pruning and merging operations, the decision tree becomes a DAG whose all source-to-sink paths represent all valid choices (inspect Figure 3 in Appendix A).

We call such DAG the *decision DAG*, and we denote it by $\mathcal{G}$. Although the size of $\mathcal{T}$ can be exponentially large, we can directly construct $\mathcal{G}$ in a top-down fashion from an instance of **AETFS** in $\mathcal{O}(|\mathcal{G}|)$ time.

Correctness

We show that no valid path is eliminated and all invalid paths are eliminated while constructing $\mathcal{G}$ from $\mathcal{T}$. Clearly, crossing #-nodes creates no invalid path. In what follows, we mainly focus on m-nodes that can create invalid paths.

It is easy to see that a path $(v_1, v_2, \ldots, v_p)$ is invalid if and only if (i) there is a sub-path $(v_s, \ldots, v_t)$ where v_s and v_t respectively correspond to the length-k prefix and suffix of the same sensitive pattern, and $v_{s+1}, \cdots, v_t$ are all m-nodes or (ii) there is an edge (v_{i-1}, v_i) such that $M[d_i] = 0$ where d_i is the depth of v_i. By Rule 1, we delete all invalid paths which satisfy condition (ii), and do not delete any valid path. By Rule 2, we delete an invalid path which satisfies condition (i), and do not delete any valid path. By Rule 3, we merge #-nodes, however, it does not matter since this operation does not cause deleting or creating any path. By Rule 4, we may merge m-nodes, however, the m-nodes to be merged are carefully chosen to not interfere with Rule 2. Thus, this also does not cause deleting or creating any path. Therefore, $\mathcal{G}$ is constructed correctly.

The Size of the DAG

We next analyze the size of $\mathcal{G}$. Clearly, the total number of #-nodes is equal to $|\mathcal{I}| - 1$. Also, the source node and the sink node are unique. The number of m-nodes, each of which is a child of some #-node, is equal to the number of #-nodes, i.e., $|\mathcal{I}| - 1$. The number of the rest of m-nodes is at most L. Also, each node has at most two outgoing edges.

Each #-node and the source node represent a sub-table of size $(k + 1) \times (n + 1)$. Each m-node represents a possibly mergeable row, and the sink node represents the last gadget row. Therefore, the total size of $\mathcal{G}$ is $\mathcal{O}(|\mathcal{I}|kn + |\mathcal{I}|n + Ln) = \mathcal{O}(k|\mathcal{I}|n + Ln)$.

Time Complexity

The decision DAG $\mathcal{G}$ is computed in $\mathcal{O}(k|\mathcal{I}| + L)$ time without creating the original decision tree $\mathcal{T}$ by applying the above four rules for incremental $b = 1, 2, \ldots, |\mathcal{I}| - 1$. Also, we can compute each cell in $\mathcal{G}$ in amortized constant time [9]. Thus, the total time is $\mathcal{O}(k|\mathcal{I}|n + Ln)$.

4.2 Short Sensitive Patterns

Running the ETFS-DP algorithm may introduce short sensitive patterns in its gadgets. We explain how to modify the recursive formulae of the gadget row to account for short sensitive patterns. We first prove that w.l.o.g. all gadgets are either a single # or can be optimally aligned such that:

1. All #'s in gadgets are substituted by letters in W;
2. All letters in gadgets are matched with letters in W; and
3. No further letters are inserted between letters of the same gadget.

If some extra inserted letters of W are aligned with a gadget, we can add some extra #'s to change them into substitutions without increasing the cost, changing the number of non-sensitive patterns or increasing the number of sensitive patterns. Similarly, if some letters of the gadget are not matched with the same letters in W, these gadget letters can be replaced by #. Finally, if some #'s in the gadget are not aligned with any letter in W, we can either remove them or move them to the place of an adjacent gadget letter while deleting that letter. Inspect the following example.

▶ **Example 11.** Let the following optimal alignment from Example 1 with cost 4. Gadgets are in red.

```
e c a b - a a a a a b b - b a d f
e c a b # a a # a b b b # b a d f
```

We transform it to another optimal alignment of the same cost that respects the above conditions:

```
e c a b a a a a a b b - b a d f
e c a b # a # a b b b # b a d f
```

Let us first consider the leftmost gadget: (1) #'s are substituted by letters in W; (2) all letters are matched with letters in W; and (3) no further letters are inserted between the gadget's letters. Note that the rightmost gadget is a single # and so the modified alignment satisfies all conditions above.

A single # cannot introduce any sensitive pattern, so just as in the ETFS-DP algorithm we can get a cost of $E[i-1][j]+1$ corresponding to the case that a single # is inserted after $W[j-1]$ or a cost of $E[i-1][j-1]+1$ corresponding to the case that a single # is aligned with $W[j-1]$. For longer gadgets the possibilities are a bit more restricted than in the ETFS-DP algorithm. Assuming the gadget to have the structure described above, it follows that the previous # cannot be aligned before $W[F[j-1]]$, where $F[j]$ is defined to be the largest integer such that $W[F[j]\,..\,j-1]$ contains a sensitive or non-sensitive pattern (if it exists). More formally:

$$F[j] = \max(\{i < j \mid W[i\,..\,j-1] \text{ contains a sensitive pattern}\} \cup \{j-k\} \cup \{0\}).$$

F can be computed in $\mathcal{O}(kn)$ time. We denote the point-wise minimum of the copies of the preceding merge row with r; in the case of **ETFS** this is just the previous merge row. This gives us the following formula for the gadget rows. For all $0 \le i \le (k+1)|\mathcal{I}|$ with $i \equiv 0 \mod k+1$,

$$E[i][j] \quad = \quad \min \begin{cases} r[j]+1 \\ r[j-1]+1 \\ E[i][j-1]+1, E[i][j-2]+1, \ldots, E[i][F[j-1]+1]+1. \end{cases}$$

(Notice that a string position and its corresponding table index differ by one.)

To conclude, we also need to consider the range in which the edit distance value lies. Since the last row corresponds to $\otimes$, the value stored in $E[|\mathcal{I}|(k+1)][j]$, for all $0 \le j \le n$, is the cost of an optimal alignment between $W[0\,..\,j+e_j-1]$ and a string in the regular expression whose length-(e_j+1) suffix is $\#W[j\,..\,j+e_j-1]$, where $e_j = \min(\max\{e \mid W[j\,..\,j+e-1] \text{ does not contain any sensitive or non-sensitive pattern}\} \cup \{n-j\})$. The edit distance between W and any string matching the regular expression is found among the rightmost $n-F[n]$ entries of the last row or the rightmost entry of the second-last row. Thus, we obtain:

$$d_{\mathrm{E}}(Y_{\mathrm{ED}}, W) = \quad \min \begin{cases} E[|\mathcal{I}|(k+1)-1][n], \\ E[|\mathcal{I}|(k+1)][n], E[|\mathcal{I}|(k+1)][n-1], \ldots, E[|\mathcal{I}|(k+1)][F[n]+1]. \end{cases}$$

For each $E[i][j]$ and $r[j]$ we store a pointer to an entry which led to this minimum value. We can then trace back as in ETFS-DP, taking the minimizing entry of the above equation as a starting point, and obtain Y_{ED} in an additional $\mathcal{O}(kn)$ time. Therefore the total time complexity of **AETFS** is $\mathcal{O}(k|\mathcal{I}|n + Ln)$ and we arrive at Lemma 10.

5 $\tilde{\mathcal{O}}(n^2)$-Time Algorithms using Dyadic Intervals

In this section we improve ETFS-DP and the algorithm of Lemma 10. We first show an algorithm to compute *gadget rows* in amortized constant time per cell in the rows. Secondly, we focus on the redundancy in the computation of *ordinary rows*, and propose an algorithm to

compute them by using the lookup table structure of Section 3 according to dyadic intervals. These two improvements yield an $\tilde{\mathcal{O}}(n^2)$-time algorithm for **ETFS**. Finally, we employ a similar lookup table technique to contract m-paths in the decision DAG, which yields an $\tilde{\mathcal{O}}(n^2)$-time algorithm for **AETFS** as well.

5.1 Speeding Up Gadget Rows Computation

First, we show how to speedup the gadget rows computation. For each #-node u in the decision DAG $\mathcal{G}$, we denote by d_u the in-degree of u. Let $G_u[0\,..\,n]$ be the gadget row in u. For each $0 \leq i \leq d_u - 1$, let $M_u^i[0\,..\,n]$ be a possibly mergeable row of a node which has an edge pointing to u. The recursive formula for $G_u[i]$ is as follows: $G_u[0] = \min\{M_u^0[0]+1, \ldots, M_u^{d-1}[0]+1\}$, and

$$G_u[j] = \min \begin{cases} M_u^0[j-1]+1, \ldots, M_u^{d-1}[j-1]+1, \\ M_u^0[j]+1, \ldots, M_u^{d-1}[j]+1, \\ G_u[j-1]+1, \ldots, G_u[F[j-1]+1]+1, \end{cases}$$

for $1 \leq j \leq n$. We assume that $M_u^0, \ldots, M_u^{d-1}$ and F are given. It costs $\mathcal{O}(n(k+d_u))$ time to compute G_u naïvely. The next lemma states that we can actually compute G_u in $\mathcal{O}(nd_u)$ time.

▶ **Lemma 12.** *Given $M_u^0, \ldots, M_u^{d-1}$ and F, we can compute every $G_u[i]$ in $\mathcal{O}(d_u)$ time for incremental $i = 0, \ldots, n$.*

Proof. Let us fix an arbitrary #-node u and omit subscripts related to u. Let r_j be the index such that $G[r_j]$ is the rightmost minimum value in the range $G[F[j-1]+1\,..\,j]$, and let $m_j = G[r_j]$ be that minimum value. Then, it can be seen that $G[p] = m_j + 1$, for any $r_j < p \leq j$, since $G[p] > G[r_j]$ and $G[p] \leq G[r_j]+1$ by the recursive formula. Clearly, $r_0 = 0$. We assume that r_{j-1} is known before computing $G[j]$. If $r_{j-1} < F[j-1]+1$, then $G[j-1] = m_{j-1}+1$ is the minimum in $G[F[j-1]+1\,..\,j-1]$, and $r_j = \arg\min\{G[j-1], G[j]\}$. Otherwise, $F[r_{j-1}] = m_j$ is the minimum in $G[F[j-1]+1\,..\,j-1]$, and $r_j = \arg\min\{G[r_{j-1}], G[j]\}$. Note that F is a non-decreasing array, i.e., $F[j] \geq F[j-1]$. Thus, we can compute $G[j]$ and r_j in $\mathcal{O}(d)$ time. ◀

By Lemma 12, we can compute all gadget rows in a total of $\mathcal{O}(n\sum_{u\in\mathcal{G}} d_u) = \mathcal{O}(n|\mathcal{I}| + nL)$ time.

5.2 ETFS in $\mathcal{O}(n^2 \log^2 k)$ Time

In this section we describe an algorithm which solves **ETFS** in $\mathcal{O}(n^2 \log^2 k)$ time. The key to losing the factor k is the fact that the string T on the left is highly repetitive and only consists of substrings of the length-n string W (interleaved by some letters in Γ). Therefore we can compute the DP table efficiently using only few precomputed sub-tables as in the Four Russians method [6].

First, we partition W into substrings of length 2^i (or shorter if $2^i \nmid |W|$) for each $i \in \{0, 1, 2, \ldots, \lfloor \log k \rfloor\}$. This gives a set $\mathcal{A}$ of at most $2n$ different strings. Moreover, note that each length-k pattern in W can be written as the concatenation of at most $2\lfloor \log k \rfloor + 2$ such strings.

For every pair of strings in $(w_1, w_2) \in \mathcal{A}^2$ with $|w_1| = |w_2|$, we precompute the lookup table for the strings w_1 and w_2 according to Lemma 9. Now we can compute the non-merge case of each possibly mergeable row using at most $2 \cdot \lceil n/2^i \rceil$ precomputed lookup tables of size $2^i \times 2^i$ (or smaller) for each $i \in \{0, 1, 2, \ldots, \lfloor \log k \rfloor\}$.

Time Complexity

Precomputing a lookup table for two strings of length up to 2^i takes $\mathcal{O}(2^{2i}i)$ time. In total this gives a precomputation time of

$$\mathcal{O}\left(\sum_{i=0}^{\lfloor \log k \rfloor} \frac{n}{2^i} \cdot \frac{n}{2^i} \cdot 2^{2i} i\right) = \mathcal{O}(n^2 \log^2 k).$$

Each possibly mergeable row can now be computed in $\mathcal{O}(n \log k)$ time from the previous merge and gadget row, since each non-sensitive length-k pattern can be partitioned into at most $2\lfloor \log k \rfloor + 2$ precomputed strings. Gadget rows can be computed in $\mathcal{O}(n)$ time each from the preceding possibly mergeable rows using the technique described by Lemma 12.

For the traceback, note that $|X_{\mathrm{ED}}| = \mathcal{O}(kn)$, i.e., the length of an optimal alignment path over E is $\mathcal{O}(kn)$. We do not know how the path behaves inside each block. However we can compute the sub-path inside a block in time linear in the path's length by using Lemma 9. The gadget rows can be traced back in a further $\mathcal{O}(n)$ time. Thus, we can trace back in a total time of $\mathcal{O}(kn)$. Therefore the total time complexity is $\mathcal{O}(n^2 \log^2 k)$. We arrive at the following result.

▶ **Theorem 2.** *The **ETFS** problem can be solved in $\mathcal{O}(n^2 \log^2 k)$ time.*

5.3 AETFS in $\mathcal{O}(n^2 \log^2 n)$ Time

In this section we describe how to further reduce the decision DAG $\mathcal{G}$ from Section 4 by precomputing parallel m-paths. First, we give some observations for m-paths. An m-path is said to be *maximal* if the m-path cannot be extended either forward or backward. Any two maximal m-paths do not share any nodes, since every m-node in $\mathcal{G}$ has at most one outgoing edge to m-nodes and at most one incoming edge from m-nodes. Also, the number of maximal m-paths is at most $|\mathcal{I}|$ since the parent of the first m-node of each maximal m-path is a different #-node or the source node. In what follows, suppose $\mathcal{G}$ contains a total of p maximal m-paths of length $\ell_1, \ldots, \ell_p$ with $\sum_{i=1}^{p} \ell_i \leq n + \ell n$, where ℓ is the length of the longest sensitive pattern. Recall that an m-node represents a possibly mergeable row of size $1 \times (n+1)$, and thus, we will identify an m-path of length x with a DP table of size $x \times (n+1)$.

Let us now describe our DAG reduction. An example of the DAG reduction is demonstrated in Figure 2. In the prepocessing phase, we first construct a lookup table structure for all possible m-paths corresponding to dyadic intervals of lengths at most ℓ over the range $[1, |\mathcal{I}| - 1]$ of the depths of m-paths, in a similar way as in Section 5.2. Next, for each maximal m-path in $\mathcal{G}$, we decompose it into shorter m-paths according to dyadic intervals. We then contract each such m-path into a single node named *j-node* consisting of a single row, which *jumps* from the beginning of the m-path to the end and represent consecutive merges. Also, we have to take into account edges leaving the m-path. Note that paths that leave the m-path early always leave to a #-node, so we do not have to worry about introducing any sensitive patterns. We therefore create a new *copy* of the m-path preceded by an additional node named *c-node* consisting of a single row, which takes the point-wise minimum of the parent nodes of the m-paths.

After finishing the DAG reduction, we fill the DP tables from top to bottom: all j-nodes are computed by using the lookup table; all new m-paths are computed in the original fashion, including all outgoing edges; and all the other nodes are computed as in Section 5.2. Also, we can trace back and find the solution to **AETFS** by storing appropriate backward-pointers and the data structures of Lemma 9 just as in Section 5.2.

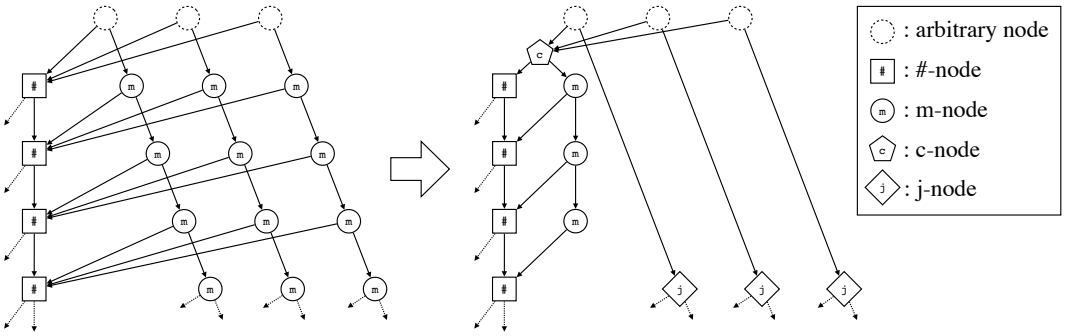


Figure 2 An example of contracting a part of the decision DAG. Before contracting, there are three parallel m-paths each of length $2^2 = 4$. We contract each m-path to a j-node corresponding to the case in which four consecutive non-sensitive patterns are merged. Also, we create an m-path of length 3 preceded by a c-node corresponding to the case in which at least one gadget is inserted.

Time Complexity

Constructing the lookup tables takes $\mathcal{O}(n^2 \log^2 \ell)$ time since there are at most $\lfloor \log \ell \rfloor$ different path lengths, and for each $i \in \{0, 1, \ldots, \lfloor \log \ell \rfloor\}$, we preprocess at most $(n/2^i)^2$ blocks of size $2^i \times 2^i$ each in $\mathcal{O}(2^{2i} i)$ time. We can also easily contract $\mathcal{G}$ in $\mathcal{O}(n^2)$ time by traversing the DAG. Note that the number of nodes in the original DAG is $\mathcal{O}(n^2)$ (Section 4.1). We partition each path of length ℓ_i into at most $2(\log \ell + \ell_i/\ell)$ precomputed paths: at most $\ell_i / 2^{\lfloor \log \ell \rfloor}$ paths of length $2^{\lfloor \log \ell \rfloor}$ and at most 2 of each shorter length. Therefore the j-nodes can be computed in $\mathcal{O}(n \cdot \sum_{i=1}^{p} 2(\log \ell + \ell_i/\ell)) = \mathcal{O}(n^2 \log \ell)$ time. The c-nodes and the following m-nodes can be computed in $\mathcal{O}(n^2 \log \ell)$ time, because there is at most one c- or m-node per depth and per precomputed path length. The #-nodes can each be computed in $\mathcal{O}(n \log^2 k)$ time using the method described in Section 5.2. Finally, tracing back takes only $\mathcal{O}(kn)$ time by using the backward-pointers and the data structures of Lemma 9.

Summarizing this section, we have shown that the **AETFS** problem can be solved in time $\mathcal{O}(n^2 \log^2 k + \min\{n^2 \log^2 \ell, Ln\})$. We arrive at the following result.

▶ **Theorem 4.** *The **AETFS** problem can be solved in $\mathcal{O}(n^2 \log^2 n)$ time.*

We defer investigating the generalization of this result for arbitrary weights to the full version of this paper.

References

1 Osman Abul, Francesco Bonchi, and Fosca Giannotti. Hiding sequential and spatiotemporal patterns. *IEEE Trans. Knowl. Data Eng.*, 22(12):1709–1723, 2010. doi:10.1109/TKDE.2009.213.

2 Osman Abul and Harun Gökçe. Knowledge hiding from tree and graph databases. *Data Knowl. Eng.*, 72:148–171, 2012. doi:10.1016/j.datak.2011.10.002.

3 Alok Aggarwal, Maria M Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1-4):195–208, 1987.

4 Charu C. Aggarwal. Applications of frequent pattern mining. In Charu C. Aggarwal and Jiawei Han, editors, *Frequent Pattern Mining*, pages 443–467. Springer, 2014. doi:10.1007/978-3-319-07821-2_18.

5 Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 3–14. IEEE Computer Society, 1995. doi:10.1109/ICDE.1995.380415.

6 Vladimir L. Arlazarov, Yefim A. Dinitz, MA Kronrod, and Igor A. Faradzhev. On economical construction of the transitive closure of an oriented graph. *Doklady Akademii Nauk*, 194(3):487–488, 1970.

7 Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. String sanitization: A combinatorial approach. In Ulf Brefeld, Élisa Fromont, Andreas Hotho, Arno J. Knobbe, Marloes H. Maathuis, and Céline Robardet, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I*, volume 11906 of *Lecture Notes in Computer Science*, pages 627–644. Springer, 2019. doi:10.1007/978-3-030-46150-8_37.

8 Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giovanna Rosone, and Michelle Sweering. Combinatorial algorithms for string sanitization. *ACM Trans. Knowl. Discov. Data*, 15(1), 2020. doi:10.1145/3418683.

9 Giulia Bernardini, Huiping Chen, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Leen Stougie, and Michelle Sweering. String Sanitization Under Edit Distance. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2020.7.

10 Giulia Bernardini, Alessio Conte, Garance Gourdel, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giulia Punzi, Leen Stougie, and Michelle Sweering. Hide and mine in strings: Hardness and algorithms. In Claudia Plant, Haixun Wang, Alfredo Cuzzocrea, Carlo Zaniolo, and Xindong Wu, editors, *20th IEEE International Conference on Data Mining, ICDM 2020, Sorrento, Italy, November 17-20, 2020*, pages 924–929. IEEE, 2020. doi:10.1109/ICDM50108.2020.00103.

11 Luca Bonomi, Liyue Fan, and Hongxia Jin. An information-theoretic approach to individual sequential data sanitization. In Paul N. Bennett, Vanja Josifovski, Jennifer Neville, and Filip Radlinski, editors, *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, San Francisco, CA, USA, February 22-25, 2016*, pages 337–346. ACM, 2016. doi:10.1145/2835776.2835828.

12 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 79–97. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.15.

13 Brian Brubach and Jay Ghurye. A succinct four Russians speedup for edit distance computation and one-against-many banded alignment. In *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

14 Meng Chen, Xiaohui Yu, and Yang Liu. Mining moving patterns for predicting next location. *Inf. Syst.*, 54:156–168, 2015. doi:10.1016/j.is.2015.07.001.

15 Chris Clifton and Don Marks. Security and privacy implications of data mining. In *In ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages 15–19, 1996.

16 Maxime Crochemore, Gad M Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM journal on computing*, 32(6):1654–1673, 2003.

17 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

18 Aris Gkoulalas-Divanis and Grigorios Loukides. Revisiting sequential pattern hiding to enhance utility. In Chid Apté, Joydeep Ghosh, and Padhraic Smyth, editors, *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pages 1316–1324. ACM, 2011. doi:10.1145/2020408.2020605.

19 Aris Gkoulalas-Divanis and Vassilios S. Verykios. An integer programming approach for frequent itemset hiding. In Philip S. Yu, Vassilis J. Tsotras, Edward A. Fox, and Bing Liu, editors, *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management, Arlington, Virginia, USA, November 6-11, 2006*, pages 748–757. ACM, 2006. doi:10.1145/1183614.1183721.

20 Aris Gkoulalas-Divanis and Vassilios S. Verykios. Exact knowledge hiding through database extension. *IEEE Trans. Knowl. Data Eng.*, 21(5):699–713, 2009. doi:10.1109/TKDE.2008.199.

21 Robert Gwadera, Aris Gkoulalas-Divanis, and Grigorios Loukides. Permutation-based sequential pattern hiding. In Hui Xiong, George Karypis, Bhavani M. Thuraisingham, Diane J. Cook, and Xindong Wu, editors, *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*, pages 241–250. IEEE Computer Society, 2013. doi:10.1109/ICDM.2013.57.

22 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.

23 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.

24 Philip N Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 146–155. Society for Industrial and Applied Mathematics, 2005.

25 Daniel C. Koboldt, Karyn M. Steinberg, David E. Larson, Richard K. Wilson, and Elaine R. Mardis. The next-generation sequencing revolution and its impact on genomics. *Cell*, 155(1):27–38, 2013.

26 Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, 1966.

27 Grigorios Loukides and Robert Gwadera. Optimal event sequence sanitization. In Suresh Venkatasubramanian and Jieping Ye, editors, *Proceedings of the 2015 SIAM International Conference on Data Mining, Vancouver, BC, Canada, April 30 - May 2, 2015*, pages 775–783. SIAM, 2015. doi:10.1137/1.9781611974010.87.

28 Eugene W. Myers and Webb Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

29 Stanley R. M. Oliveira and Osmar R. Zaïane. Protecting sensitive knowledge by data sanitization. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003), 19-22 December 2003, Melbourne, Florida, USA*, pages 613–616. IEEE Computer Society, 2003. doi:10.1109/ICDM.2003.1250990.

30 Jeanette P Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.

31 Vassilios S. Verykios, Ahmed K. Elmagarmid, Elisa Bertino, Yücel Saygin, and Elena Dasseni. Association rule hiding. *IEEE Trans. Knowl. Data Eng.*, 16(4):434–447, 2004. doi:10.1109/TKDE.2004.1269668.

32 Yi-Hung Wu, Chia-Ming Chiang, and Arbee L. P. Chen. Hiding sensitive association rules with limited side effects. *IEEE Trans. Knowl. Data Eng.*, 19(1):29–42, 2007. doi:10.1109/TKDE.2007.250583.

33 Josh Jia-Ching Ying, Wang-Chien Lee, Tz-Chiao Weng, and Vincent S. Tseng. Semantic trajectory mining for location prediction. In Isabel F. Cruz, Divyakant Agrawal, Christian S. Jensen, Eyal Ofek, and Egemen Tanin, editors, *19th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2011, November 1-4, 2011, Chicago, IL, USA, Proceedings*, pages 34–43. ACM, 2011. doi:10.1145/2093973.2093980.

A Omitted Figure

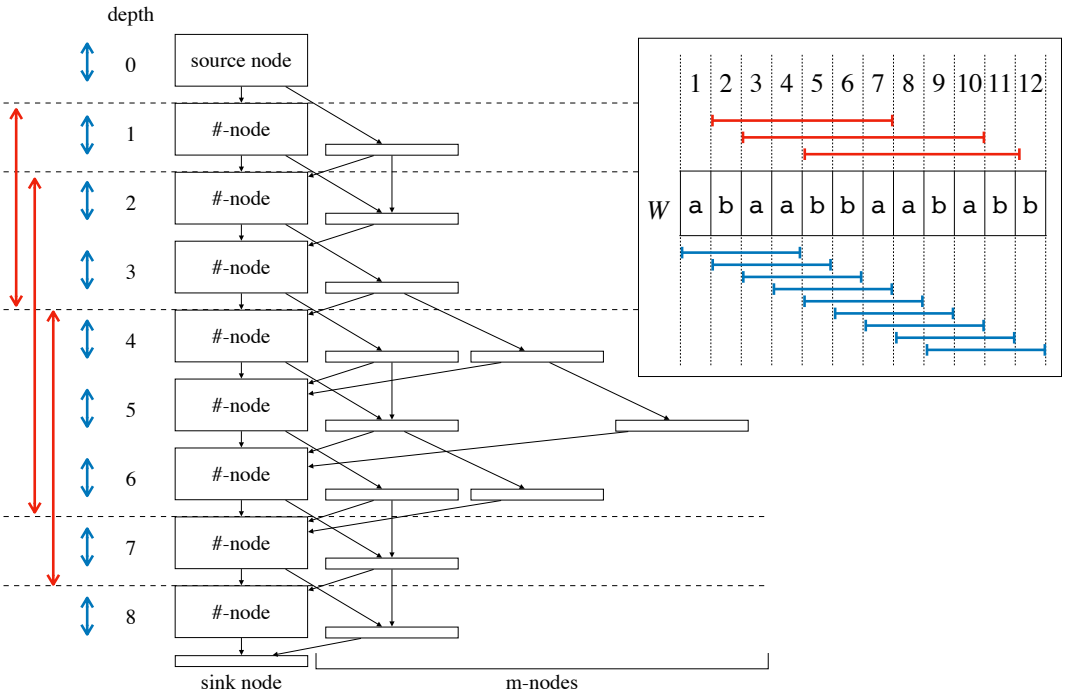


Figure 3 The decision DAG for $W = \texttt{abaabbaababb}$, $k = 4$, and $\mathcal{S} = \{[2, 7], [3, 10], [5, 11]\}$.

An Invertible Transform for Efficient String Matching in Labeled Digraphs

Abhinav Nellore[1]
Oregon Health & Science University, Portland, Oregon 97239, USA

Austin Nguyen
Oregon Health & Science University, Portland, Oregon 97239, USA

Reid F. Thompson
Oregon Health & Science University, Portland, Oregon 97239, USA
VA Portland Healthcare System, Portland, Oregon 97239, USA

Abstract

Let $G = (V, E)$ be a digraph where each vertex is unlabeled, each edge is labeled by a character in some alphabet Ω, and any two edges with both the same head and the same tail have different labels. The powerset construction gives a transform of G into a weakly connected digraph $G' = (V', E')$ that enables solving the decision problem of whether there is a walk in G matching an arbitrarily long query string q in time linear in $|q|$ and independent of $|E|$ and $|V|$. We show G is uniquely determined by G' when for every $v_\ell \in V$, there is some distinct string s_ℓ on Ω such that v_ℓ is the origin of a closed walk in G matching s_ℓ, and no other walk in G matches s_ℓ unless it starts and ends at v_ℓ. We then exploit this invertibility condition to strategically alter any G so its transform G' enables retrieval of all t terminal vertices of walks in the unaltered G matching q in $O(|q| + t \log |V|)$ time. We conclude by proposing two defining properties of a class of transforms that includes the Burrows-Wheeler transform and the transform presented here.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Mathematics of computing → Combinatorics on words

Keywords and phrases pattern matching, string matching, Burrows-Wheeler transform, labeled graphs

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.20

Related Version *arXiv preprint*: `https://arxiv.org/abs/1905.03424`

Acknowledgements We thank Rachel Ward, Ben Langmead, Chris Wilks, and the anonymous reviewers for the 32nd Annual Symposium on Combinatorial Pattern Matching, all of whose feedback improved this paper considerably.

1 Introduction

Consider the decision problem of whether there is a walk in a finite edge-labeled digraph matching a query string of labels. Intuitively, the offline version of this problem is straightforwardly solved in time linear in the size of the string and independent of the size and order of the graph using an index that sorts the walks matching every possible query string so they can essentially be performed as if they were one walk. Many approaches [37, 30, 35, 36, 18, 32, 2, 15] further rely on the last-to-first (LF) mapping property exhibited by a class of invertible transforms [9, 28] that includes the Burrows-Wheeler transform (BWT) [3]. Wheeler graphs [15] provide a unifying formalism for these LF mapping-based strategies. A Wheeler graph admits a particular total order of its (unlabeled) vertices[2] such

[1] corresponding author
[2] in contradistinction to the order of a graph, which is the number of its vertices

that the terminal vertices of all walks matching any query string comprise a single interval. LF mapping enables performing all walks in the Wheeler graph matching a given query string as if they were one walk from interval to interval of the totally ordered vertices.

But not every labeled digraph is a Wheeler graph. Indexing a non-Wheeler graph using the framework outlined above involves constructing an equivalent Wheeler graph of which subsets of vertices each represent a distinct vertex of the non-Wheeler graph. A query string is matched by a walk to a vertex of this original graph if and only if it is also matched by a walk to an associated vertex of the equivalent Wheeler graph. After confirming the presence of at least one match in the equivalent Wheeler graph, it is often important to perform a locate query, which retrieves the vertices of the original graph at which matching walks terminate. This requires additionally storing the vertex associations between the original graph and its equivalent Wheeler graph [37, 35, 18].

There are compressed indexes of Wheeler graphs supporting efficient locate queries [15, 14, 31], but an equivalent Wheeler graph can be large and unwieldy regardless, especially when the non-Wheeler graph it represents has cycles. In practice, such cycles are typically indexed so only query strings up to some maximum size are supported [35, 18], limiting the size and order of the equivalent Wheeler graph, but it is worth seeking alternative strategies that more readily and more elegantly accommodate matching query strings of arbitrary size and locating matches. The central issue is that LF mapping is a navigational instrument restricted to a line of vertices, mapping intervals into intervals. In this sense, a non-Wheeler graph is linearized by an equivalent Wheeler graph. Linearization may be cumbersome when the topology of the non-Wheeler graph deviates substantially from the topology of a Wheeler graph. Moreover, the terminal vertices of walks in a non-Wheeler graph matching a given query string are in general redundantly represented by the corresponding matching interval of totally ordered vertices of an equivalent Wheeler graph – that is, vertices on this interval may represent subsets of vertices of the non-Wheeler graph that are not disjoint. A locate query then returns a multiset of vertices of the non-Wheeler graph that must be deduplicated [37, 35, 18]. The performance of such a query depends on the size of the multiset, which can significantly exceed the number of unique vertices it comprises. A full-featured navigational instrument would go beyond LF mapping and be able to map arbitrary subsets of vertices into arbitrary subsets of vertices, eliminating this performance bottleneck while also accommodating any finite labeled digraph.

The powerset construction [33, 20] is just such an instrument, providing a transform of one labeled digraph into another labeled digraph of which each vertex represents a subset of vertices of the original graph. Moreover, this transformed graph can be used to solve the string matching decision problem with the same performance as an LF mapping-based framework. We show (1) under a particular condition, the original graph is uniquely determined by just the transformed graph, and (2) this invertibility condition can be exploited in a framework for efficiently locating query matches in *any* finite labeled digraph, in analogy to how invertibility of the BWT for strings enables the FM-index [10, 11], a widely used [24, 27, 23, 26] compressed suffix array.

Here is a brief summary of our framework: start with any finite digraph whose edges are labeled on some alphabet, and add edges to form a cycle that includes all vertices and matches a generalized de Bruijn sequence [12] on a different alphabet. As explained in Section 4, this operation is analogous to adding a sentinel ("the dollar sign") to a string before obtaining its BWT. Think of the digraph as a nondeterministic finite automaton (NFA) whose every vertex is both an initial state and a final state and whose labeled edges encode the transition function. Use the powerset construction to obtain a deterministic finite

automaton (DFA) from this NFA. Now think of the DFA as a digraph whose labeled edges encode the transition function and whose states are unlabeled vertices except for the empty state, which is excluded from the graph along with all states unreachable from the initial state. Refer to that initial state as the root. We call this transformed graph a nength of the original graph. There is at most one edge with a given label directed from any vertex of the nength. A walk in the original graph matches a query string if and only if a walk starting at the root of its nength also matches that string. When such a walk is present in the nength, traversals of the nength's edges labeled on the alphabet of the generalized de Bruijn sequence and starting at the terminal vertex of this walk rapidly and nonredundantly retrieve the terminal vertices of all walks in the original graph matching the query string.

Our main message is this: existing software [37, 35, 18] for offline string matching in a labeled digraph linearizes that graph and subsequently indexes the result using BWT-based approaches familiar from text indexing. These approaches exploit invertibility of the BWT or a related transform via LF mapping to downsample vertex indices stored to support locate queries. However, when linearization is awkward and gives a massive equivalent Wheeler graph, downsampling is severely constrained and of diminished utility. In this event, it is possible to use a nength instead, which obviates the need for graph linearization and still exploits invertibility – achieved via a mechanism different from LF mapping – to reduce the index size while keeping locate queries efficient. Nength has the additional advantage that it naturally accommodates matching query strings of arbitrary size in any labeled digraph, whether or not it has cycles.

Table 1 A glossary of terms used in this paper.

term	definition
head of a digraph's edge	vertex at which the edge is directed
tail of a digraph's edge	vertex from which the edge is directed
order of a graph	number of vertices of the graph; the word "order" is also used in this paper to refer to ordering objects, and the appropriate denotation should be clear from context
size of a graph	number of edges of the graph
walk in a digraph	sequence alternating between vertices and edges of the digraph such that each edge in the sequence is directed from the vertex immediately before it and at the vertex immediately after it
closed walk in a digraph	walk in the digraph that starts and ends at the same vertex
path in a digraph	walk in the digraph that repeats neither vertices nor edges
necklace	circular string of characters; if, e.g., 101 is said to be a necklace, then the set of its two-character substrings is $\{10, 01, 11\}$, and 011 and 110 refer to the same necklace
transform	function, irrespective of its domain and codomain

Our presentation is organized as follows. Section 2 introduces the powerset construction using graph theoretic language and proves a general invertibility condition. Section 3 gives an algorithm for locating query matches with a nength, which relies on this invertibility condition via the generalized de Bruijn sequence construction sketched above, and describes a basic data structure for storing a nength. Section 4 elaborates on the analogy between nength and the BWT and proposes two defining properties of a class of transforms that includes both. A glossary of terms required to understand this paper is provided in Table 1. Other terms introduced here or invoked in more specific contexts than may be typical in the literature are italicized and defined on first use.

2 Transform

Let $G = (V, E)$ be a finite digraph where each vertex is unlabeled, each edge bears exactly one label on some alphabet Ω, and any two edges with both the same head and the same tail have different labels. Let $G' = (V', E')$ be another digraph specified by G according to the following conditions:

1. each vertex $v'_i \in V'$ is unlabeled but associated with a distinct bit vector $\mathbf{b}'_i$ of size $|V|$ called the *state* of v'_i whose ℓth bit is $b'_{i\ell}$ and whose bits are never all zero,
2. exactly one edge labeled $\omega_k \in \Omega$ extends from v'_i to v'_j for $v'_i, v'_j \in V'$ if and only if $\{v_m \in V : b'_{jm} = 1\}$ is the set of heads of ω_k-labeled edges of G whose tails are among $\{v_m \in V : b'_{im} = 1\}$, and
3. $|V'|$ is as large as possible such that all vertices in V' are reachable from a vertex designated as the root whose state has only nonzero bits.

Condition 1 above implies G' is finite because G is finite, and there are $2^{|V|} - 1$ possible nonzero states. Condition 2 implies a vertex of G' is the tail of no more than one edge labeled by a given character in Ω. Condition 3 implies G' is weakly connected.

A vertex v'_i of G' represents a set of vertices of G, and the state $\mathbf{b}'_i$ records these vertices. Note G' can be thought of as the DFA obtained via the powerset construction [33, 20] from, in general, an NFA. In the NFA, Ω is the set of input symbols, each state is both an initial state and a final state, each state corresponds to a distinct vertex of G, and the transition function is prescribed by the edges of G. In the DFA, the initial state corresponds to the root of G'. Further, every vertex of G' corresponds to a distinct state of the DFA. States of the DFA unreachable from the initial state and transitions to the DFA's empty state are not represented in G'. See [4, 34] for recent innovations in parallelization of the powerset construction.

G' facilitates following all walks in G matching some query string q on Ω as if they were one walk: stand at the root of G', start walking the sequence of edges whose labels match q, and either (1) it is not possible to reach step number $p \leq |q|$ because no walk in G matches the size-p prefix of q, or (2) the nonzero bits of the state of the vertex reached at step number p correspond to the terminal vertices of all walks in G matching the size-p prefix of q. Since it essentially sorts all walks in G and is obtained from the powerset construction, call G' the *powerset sort* of G. G' permits solving the decision problem of whether there exist one or more walks in G matching a query string q in time linear in $|q|$, independent of G's size $|E|$ and order $|V|$. If the states of vertices of the powerset sort G' are stored beforehand, a positive determination is accompanied by the terminal vertices of matches in G. However, storing these states together with G' is costly.

Call a vertex $v'_i \in V'$ for which $b'_{i\ell}$ is the only nonzero bit of $\mathbf{b}'_i$ the *singleton* v'^*_ℓ of G'; that is, as an alternative notation, use an asterisk to denote singletons, and index them according to how corresponding vertices in G are indexed. Now suppose for every $v_\ell \in V$, there is some distinct string s_ℓ on Ω such that v_ℓ is origin of some closed walk matching s_ℓ, and no other walk in G matches s_ℓ unless it starts and ends at v_ℓ. Call s_ℓ an *identifying string* of v_ℓ; call a closed walk matching s_ℓ an *identifying walk* of v_ℓ. It is clear that if the state of each vertex of G' is specified, G is uniquely determined by G': (1) for every vertex v_ℓ of G, there is a walk matching an identifying string of v_ℓ from G''s root to the singleton v'^*_ℓ; (2) for every $\omega_k \in \Omega$, if there are any ω_k-labeled edges of G whose tail is v_ℓ, their heads are specified by the state of the head of the ω_k-labeled edge extending from v'^*_ℓ; and (3) this implies the head and tail of every edge of G are known from G' and the $\{\mathbf{b}'_w\}$. But a stronger statement can be made: G' itself encodes the $\{\mathbf{b}'_w\}$ (up to permutation equivalence) via identifying walks, and it is an invertible transform of G without requiring that the $\{\mathbf{b}'_w\}$ are recorded. We prove the following.

▶ **Theorem 1.** *G is uniquely determined by its powerset sort G' when every vertex of G has an identifying walk.*

Proof. G' is given, but the state of each of its vertices is not. Write $\{v \rightarrow\}$ to refer to the set of strings matching walks starting at some vertex v of some graph. Note by construction, $\{v'^{*}_{\ell} \rightarrow\} = \{v_\ell \rightarrow\}$ for $v'^{*}_{\ell} \in V'$ and $v_\ell \in V$; that is, the strings matching walks starting at a singleton of G' capture precisely the set of possible matches to walks starting at its corresponding vertex in G. More generally,

$$\{v'_i \rightarrow\} = \bigcup_{w \in Y} \{v'^{*}_{w} \rightarrow\} \qquad \text{for } Y = \{m : b'_{im} = 1\}; \tag{1}$$

that is, the strings matching walks starting at a given vertex v'_i of G' capture precisely the set of possible matches to walks starting at any vertex v_ℓ of G for which $b'_{i\ell} = 1$. But by definition, for any singletons $v'^{*}_{\ell}, v'^{*}_{p} \in V'$ with $\ell \neq p$, $\{v'^{*}_{\ell} \rightarrow\}$ contains an identifying string that is not in $\{v'^{*}_{p} \rightarrow\}$. Together with (1), this says for any vertices $v_\ell \in V$ and $v'_i, v'_j \in V'$, $\{v'_i \rightarrow\} \subseteq \{v'_j \rightarrow\}$ if and only if $b'_{i\ell} = 1 \implies b'_{j\ell} = 1$. It follows that a given vertex $v'_i \in V'$ is a singleton if and only if for any vertex $v'_j \in V'$ with $j \neq i$, $\{v'_j \rightarrow\} \not\subseteq \{v'_i \rightarrow\}$. Further, $b'_{j\ell} = 1$ if and only if $\{v'^{*}_{\ell} \rightarrow\} \subseteq \{v'_j \rightarrow\}$ for $v'^{*}_{\ell}, v'_j \in V'$. This implies the states of all vertices of G' can be determined up to permutation equivalence, and thus G is uniquely determined by its powerset sort G'. ◀

Let $\mathbf{A}_k$ be the adjacency matrix of G specific to $\omega_k \in \Omega$; that is, its (ℓ, p)th entry is 1 when an ω_k-labeled edge extends from v_ℓ to v_p for $v_\ell, v_p \in V$ and is 0 otherwise. Observe that G' represents a system of matrix equations where multiplication is Boolean and a given edge labeled $\omega_k \in \Omega$ extending from $v'_i \in V'$ to $v'_j \in V'$ corresponds to the equation $\mathbf{A}_k \mathbf{b}'_i = \mathbf{b}'_j$. Perhaps surprisingly, Theorem 1 says this system has a unique solution up to permutation equivalence when it is constructed from a graph G for which every vertex has an identifying walk, despite how none of the adjacency matrices or states is known in advance.

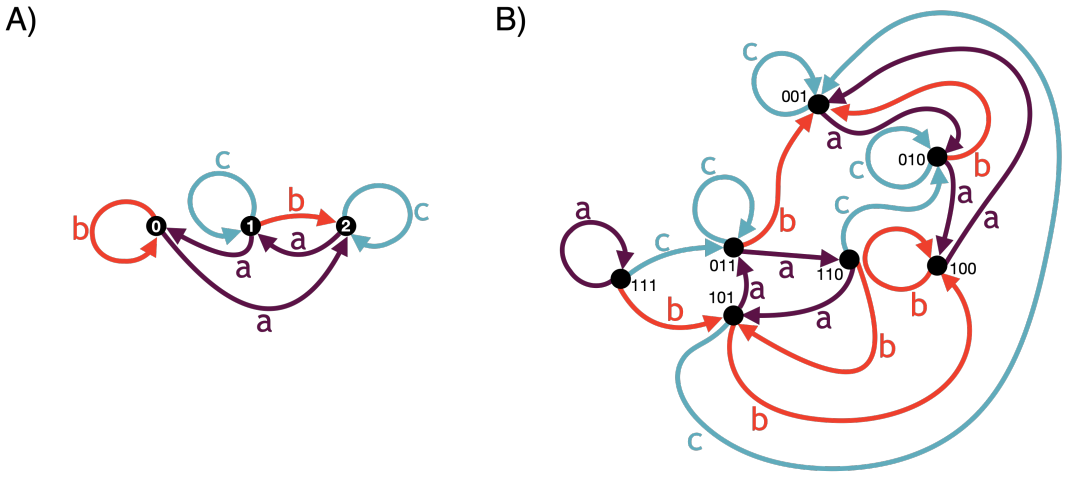


Figure 1 A) is an example digraph on the alphabet $\{a, b, c\}$ where each vertex has an identifying walk, and B) is its powerset sort. Edges with different labels have different colors. States are written next to associated vertices of B), and state bits are ordered correspondingly to vertex indices of A). A complete set of identifying strings for A) is $\{bb, cba, abc\}$. It is easily seen all walks in B) matching a given identifying string end at the same vertex, which is always a singleton.

For a given identifying string s_ℓ of $v_\ell \in V$, any walk matching s_ℓ in G' terminates at v'^*_ℓ, and the origin of that walk has a state whose ℓth bit is nonzero because an identifying walk is closed. Call any set of identifying strings in which there is at least one identifying string per vertex of G a *complete set of identifying strings*. An example digraph is Figure 1A, and its powerset sort is Figure 1B. A complete set of identifying strings for Figure 1A is $\{bb, cba, abc\}$, and in Figure 1B, every walk matching any one of these identifying strings ends at the same singleton. The next section relies on Theorem 1 to develop a framework for efficient location of matches to a query string in any finite labeled digraph.

3 Location

A de Bruijn sequence $B(r, n)$ of order n on a size-r alphabet is a necklace of size r^n such that every possible size-n string on the alphabet occurs exactly once as a substring. $B(r, n)$ is optimally short in the sense that a necklace of size r^n has exactly as many substrings of size n as there are possible strings of size n on a size-r alphabet. References [13, 12] introduce generalized de Bruijn sequences, a natural generalization of de Bruijn sequences to necklaces of arbitrary size. Let x be some necklace, and let $\gamma_z(x)$ be the size of the set of size-z substrings of x. A generalized de Bruijn sequence $B_G(r)$ on a size-r alphabet for $r \leq |B_G(r)|$ is a necklace for which

$$\gamma_z(B_G(r)) = \min(r^z, |B_G(r)|)\,.$$

When $\gamma_z(B_G(r)) = |B_G(r)| = r^z$, $B_G(r)$ is a de Bruijn sequence of order z. Note $\lceil \log_r |B_G(r)| \rceil$ is the smallest value of z such that $\gamma_z(B_G(r)) = |B_G(r)|$, and thus every size-$\lceil \log_r |B_G(r)| \rceil$ substring of $B_G(r)$ occurs exactly once as a substring of $B_G(r)$. References [13, 12] give a proof that there exists at least one generalized de Bruijn sequence $B_G(r)$ for any combination of $r \geq 2$ and $|B_G(r)| \geq 1$ and provide several examples of generalized de Bruijn sequences. Also refer to [25], an antecedent with most of the elements of this proof.

Let $\widetilde{\Omega}$ be an alphabet of size at least 2 such that $\widetilde{\Omega} \cap \Omega = \emptyset$. Call $\widetilde{\Omega}$ the *sentinel alphabet*. Perform the following steps to alter any G with at least two vertices[3] to form a new graph $\widetilde{G} = (V, \widetilde{E})$:

1. Obtain some generalized de Bruijn sequence c of size $|V|$ on $\widetilde{\Omega}$.

2. Add edges to G to form a cycle $G_C = (V, E_C)$ that includes every vertex and matches c.

$\widetilde{G}$ is a labeled digraph on the alphabet $\Omega_U := \widetilde{\Omega} \cup \Omega$. Call the cycle subgraph G_C of $\widetilde{G}$ the *identifying cycle* of $\widetilde{G}$. Because c is a generalized de Bruijn sequence, every vertex $v_\ell \in V$ is the origin of a walk in G_C matching some size-$\lceil \log_{|\widetilde{\Omega}|} |V| \rceil$ string such that no other walk in $\widetilde{G}$ matches that string. This walk is part of a closed walk in the identifying cycle, and thus from Theorem 1, $\widetilde{G}$'s powerset sort $\widetilde{G}' = (\widetilde{V}', \widetilde{E}')$ is invertible. Refer to the powerset sort of any digraph augmented with an identifying cycle as a *nength* of that digraph. $\widetilde{G}'$ is a nength of G. Note that G', the powerset sort of the original graph G, is a subgraph of the nength $\widetilde{G}'$: walks in $\widetilde{G}$ matching some query string q on Ω end at vertices mirroring those in G at which walks matching q end.

Call a walk starting at any vertex $\widetilde{v}'_i \in \widetilde{V}'$ that is not a singleton a *locating walk* of $\widetilde{v}'_i$ if it traverses only edges labeled on the sentinel alphabet $\widetilde{\Omega}$, ends at a singleton, and otherwise visits no singletons. Because it does not traverse any edges labeled on Ω, a locating walk

[3] For G with one vertex, G' is invariably the same graph as G, making for a trivial case that need not be considered.

represents only walks in $\widetilde{G}$ confined to the identifying cycle; because it ends at a singleton, a locating walk represents exactly one walk in $\widetilde{G}$; because G_C matches a generalized de Bruijn sequence, the size of a locating walk does not exceed $\lceil \log_{|\widetilde{\Omega}|} |V| \rceil$; because if any locating walk repeated a vertex, it would be possible to construct an arbitrarily long locating walk, a locating walk is always a path. The identifying cycle imposes a cyclic order on vertices of $\widetilde{G}$. Assign indices to vertices of $\widetilde{G}$ such that state bits respect this order. Then a locating walk of $\widetilde{v}'_i$ can be used to determine a nonzero bit of the state $\widetilde{\mathbf{b}}'_i$ of $\widetilde{v}'_i$, where a modular subtraction of the size of the locating walk from the index of its terminal singleton gives the index of the nonzero bit. This modular subtraction corresponds to a backwards walk in G_C to recover the vertex starting the walk represented by the locating walk in $\widetilde{G}'$. The full state is recovered by following all locating walks of $\widetilde{v}'_i$. We prove the following.

▶ **Proposition 2.** *Every $\widetilde{v}'_i \in \widetilde{V}'$ that is not a singleton has exactly as many locating walks in $\widetilde{G}'$ as there are nonzero bits of the state $\widetilde{\mathbf{b}}'_i$ of $\widetilde{v}'_i$, with each nonzero bit determined by a different locating walk.*

Proof. Suppose there were more locating walks than nonzero bits of $\widetilde{\mathbf{b}}'_i$. Then by the pigeonhole principle, there would be at least two distinct locating walks to singletons for which appropriate modular subtractions of steps from indices determined the same bit, some $\widetilde{b}'_{i\ell}$. But since both these walks represent walks in the identifying cycle G_C starting at $v_\ell \in V$, (1) if they had the same number of the steps, they would necessarily correspond to the same walk in G_C, a contradiction, and (2) if they had different numbers of steps, the longer walk would reach a singleton before its end, a contradiction. ◀

So $\widetilde{G}'$ offers a straightforward way to obtain the vertices of G matching any size-p prefix of a query string q on Ω: stand at the root of $\widetilde{G}'$, start walking the sequence of edges whose labels match q, and if step p is reached at some vertex, follow locating walks and perform appropriate modular subtractions to obtain the state of that vertex, whose nonzero bits correspond to the terminal vertices of walks in G matching the size-p prefix of q.

It is not necessary to store all of $\widetilde{G}'$ to enable these locate queries. Call a vertex of $\widetilde{G}'$ a *spanner* if it is not a singleton and either is reachable from the root by following only edges labeled on Ω or is the root itself. Call a vertex of $\widetilde{G}'$ a *locator* if it is not a singleton and is not reachable from the root by following only edges labeled on Ω, but is reachable on some locating walk of a spanner. Store only singletons, spanners, locators, edges labeled on Ω whose tails are singletons, edges labeled on Ω_U whose tails are spanners, and edges labeled on $\widetilde{\Omega}$ whose tails are locators. Any other components of $\widetilde{G}'$ are not visited or traversed during string matching or on locating walks. Further, note it is enough to know only that an edge is labeled on the sentinel alphabet $\widetilde{\Omega}$ rather than Ω to follow locating walks; the particular label of an edge on $\widetilde{\Omega}$ need not be recorded.

Let $\mathbf{M}$ be a matrix with $|\Omega_U|$ columns where (1) each row corresponds to a different vertex of $\widetilde{G}'$, (2) each column corresponds to a different character in Ω_U, (3) an entry is the null pointer if and only if there is no edge of $\widetilde{G}'$ whose tail corresponds to the entry's row and whose label corresponds to the entry's column, and (4) an entry is a pointer to some row if and only if that row corresponds to the head of an edge of $\widetilde{G}'$ whose tail corresponds to the entry's row and whose label corresponds to the entry's column. $\mathbf{M}$ can be used to perform a walk in $\widetilde{G}'$ by following pointers from row to row. Arrange the row order of $\mathbf{M}$ so its first $|V|$ rows correspond to singletons, and ensure these vertices are in an order prescribed by G_C. This implicitly stores their indices – that is, when a walk in $\widetilde{G}'$ using $\mathbf{M}$ ends at some (0-indexed) row $\ell < |V|$, that row corresponds to a singleton whose index according to G_C is ℓ. Arrange that the root of $\widetilde{G}'$ corresponds to the row of $\mathbf{M}$ right after the first $|V|$ rows so

pattern matching always starts there. Also arrange that all rows corresponding to spanners precede all rows corresponding to locators, and all columns corresponding to characters in $\widetilde{\Omega}$ precede all columns corresponding to characters in Ω. Further, since the particular labels of edges labeled on $\widetilde{\Omega}$ are inconsequential, reorder the first $\widetilde{\Omega}$ entries of each row of $\mathbf{M}$ so all nonnull pointers precede all null pointers.

Algorithm 1 A depth-first approach to determining the state of a vertex of some nength $\widetilde{G}'$. A given entry of $\mathbf{M}$ that is a nonnull pointer is taken to be the index of the row of $\mathbf{M}$ pointed. The first row of $\mathbf{M}$ corresponds to the index 0. Bit indices of bit vectors respect congruence modulo $|V|$.

Input: $\mathbf{M}$, index i of row of $\mathbf{M}$ corresponding to vertex $\widetilde{v}'_i$ of $\widetilde{G}'$ whose state is desired, $|V|$, $|\widetilde{\Omega}|$
Output: size-$|V|$ state $\widetilde{\mathbf{b}}'_i$ of $\widetilde{v}'_i$
Initialization : state $\widetilde{\mathbf{b}}'_i \leftarrow \mathbf{0}$, stack $S \leftarrow \{\}$
if $i < |V|$ **then**
  $\widetilde{b}'_{ii} \leftarrow 1$
  return $\widetilde{\mathbf{b}}'_i$
end if
for $k := 0$ **to** $|\widetilde{\Omega}| - 1$ **do**
  if M_{ik} is null **then**
    break
  end if
  push the tuple $(M_{ik}, 1)$ onto S
end for
while S is not empty **do**
  pop some (m, p) off S
  if $m < |V|$ **then**
    $\widetilde{b}'_{i(m-p)} \leftarrow 1$
  else
    for $k := 0$ **to** $|\widetilde{\Omega}| - 1$ **do**
      if M_{mk} is null **then**
        break
      end if
      push the tuple $(M_{mk}, p + 1)$ onto S
    end for
  end if
end while
return $\widetilde{\mathbf{b}}'_i$

The entries of $\mathbf{M}$ necessary for pattern matching can now be stored as an array $\mathbf{m}$ in row-major order, where there is: (1) a single block of rows corresponding to a singletons, with each row taking $|\Omega|$ elements; (2) a single block of rows corresponding to spanners, with each row taking $|\Omega_U|$ elements; and (3) a single block of rows corresponding to locators, with each row taking $|\widetilde{\Omega}|$ elements. Straightforward pointer arithmetic then gives the location in memory of any entry of $\mathbf{M}$ required for string matching or state determination. Our procedure for state determination is formalized in Algorithm 1. The logic for retrieving specific entries of $\mathbf{M}$ using $\mathbf{m}$ is excluded there.

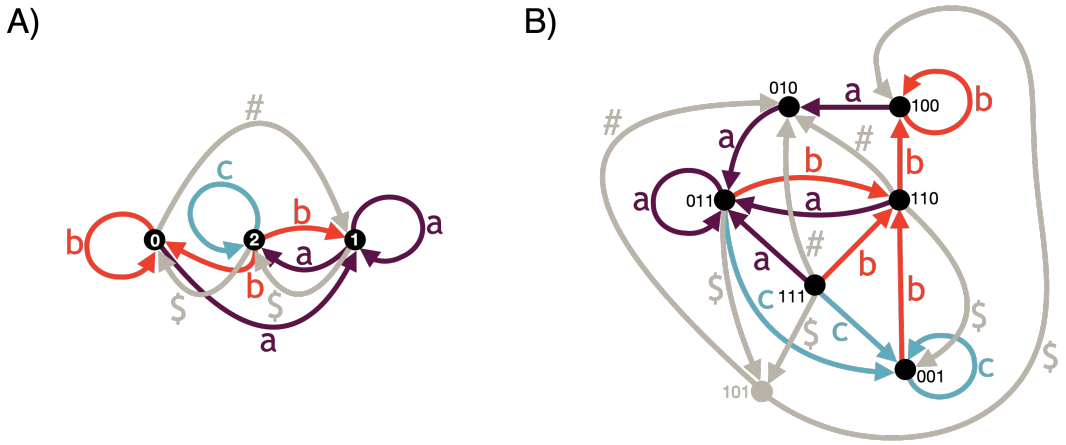


Figure 2 A) is a digraph with an identifying cycle on the sentinel alphabet $\{\$, \#\}$ matching the generalized de Bruijn sequence \$\$#. B) is the corresponding nength, excluding edges labeled on $\widetilde{\Omega}$ whose tails are singletons and edges labeled on Ω whose tails are the sole locator (i.e., the vertex with state 101). States are written next to associated vertices of B), and state bits are ordered correspondingly to vertex indices of A). Note the identifying cycle imposes an order on the vertices respected by their indices. Edges comprising the identifying cycle in A) and the components they contribute to B) are in gray. Otherwise, edges are assigned colors according to their labels.

Because of how a subgraph of $\widetilde{G}'$ is G', $\widetilde{G}'$ has $|V'| - |V|$ spanners. The number of singletons of $\widetilde{G}'$ is the number $|V|$ of vertices of $\widetilde{G}$. Suppose $\widetilde{G}'$ has $|\widetilde{V}'_L|$ locators. Then $\mathbf{m}$ takes up

$$\left(|\Omega_U|\left(|V'| - |V|\right) + |\Omega||V| + |\widetilde{\Omega}||\widetilde{V}'_L|\right)\lceil \log_2(|V'| + |\widetilde{V}'_L| + 1)\rceil$$

bits.

Figure 2A is a graph with an identifying cycle on the sentinel alphabet $\{\$, \#\}$. Without the identifying cycle, vertex 0 has no identifying walk. Figure 2B is the corresponding nength, excluding edges labeled on $\widetilde{\Omega}$ whose tails are singletons and edges labeled on Ω whose tails are the sole locator (i.e., the vertex with state 101).

Assume $|\widetilde{\Omega}| < |V|$ and that entries of $\mathbf{M}$ can be accessed in constant time. In the worst case, our algorithm for state determination runs in $O(t \log_{|\widetilde{\Omega}|} |V|)$ time, where t is the number of nonzero bits of the state, since a locating walk can take up to $\lceil \log_{|\widetilde{\Omega}|} |V| \rceil$ steps. Identifying the t terminal vertices of walks in G matching a query string q by following pointers in $\mathbf{M}$ thus takes $O(|q| + t \log_{|\widetilde{\Omega}|} |V|)$ time. The size of the sentinel alphabet $\widetilde{\Omega}$ can be as small as 2, with larger alphabet sizes improving the performance of state determination while increasing the number of columns of $\mathbf{M}$.

By contrast, naively storing all states of the powerset sort G' of G in a $|V'| \times |V|$ binary matrix for their immediate retrieval gives $O(|q|)$ performance. But storing $|V'||V|$ state bits may be forbidding for large G, and the storage overhead of locating walks in the array $\mathbf{m}$ may be comparatively small for small $|\widetilde{\Omega}|$; for example, a single locator may be the head of many edges labeled on $\widetilde{\Omega}$, achieving compression by simultaneously representing configurations of state bits that are the same across the edges' tails.

It is not absolutely necessary to ensure G_C matches a generalized de Bruijn sequence to obtain the $O(t \log_{|\widetilde{\Omega}|} |V|)$ performance guarantee for state determination. References [13, 12] provide an alternative characterization of a generalized de Bruijn sequence $B_G(r)$ as a necklace on a size-r alphabet that satisfies

1. $\gamma_d(B_G(r)) = r^d$
2. $\gamma_{d+1}(B_G(r)) = |B_G(r)|$

with $d := \lfloor \log_r |B_G(r)| \rfloor$. A necklace satisfying condition 2 above but not necessarily condition 1 also gives the performance guarantee. For some intuition about the difference, consider the following example borrowed from [13, 12]: on the binary alphabet $\{0, 1\}$, 00001011101 is a generalized de Bruijn sequence, but 10011110000 satisfies only condition 2. While every size-4 substring is distinct in each sequence, the former has all eight possible size-3 substrings, while the latter has only seven and is missing 101. A necklace satisfying condition 2 but not necessarily condition 1 is called an m-ary closed sequence in [25], and construction algorithms were developed decades ago [19, 8]. However, it is desirable that G_C matches a necklace that also satisfies condition 1. Maximizing complexity in this way can in general reduce the average number of steps of a locating walk: fewer instances of particular short kmers in the necklace make for fewer steps along the identifying cycle to distinguish vertices. So it is worth exploring how to construct generalized de Bruijn sequences efficiently.

Our basic data structure for storing $\mathbf{M}$ can be refined to reduce its size. A degree of freedom we do not explore thoroughly here is that rows corresponding to locators and spanners can be reordered so $\mathbf{M}$ has more structured sparsity. For example, ordering rows of $\mathbf{M}$ to cluster them according to which of their columns contains null pointers permits eliminating null pointers from the array $\mathbf{m}$ if the indices of nonnull columns on intervals of rows are recorded in an auxiliary data structure. Rows can also be ordered lexicographically by treating them like they are strings on an ordered alphabet of pointers; intervals of rows with the same prefix can then be compressed. It may also be possible to reduce the total number of rows of $\mathbf{M}$ by arranging that the identifying cycle largely follows existing paths in G; when G is suitably sparse, this could give rise to $\widetilde{G}'$ with a preponderance of pairs of edges sharing the same head and tail, thereby economizing the number of its vertices. The designs of identifying cycles and of compressed representations of $\mathbf{M}$ are thus potentially fruitful areas for further research.

4 Discussion

Consider the case where G is a cycle graph whose every vertex has an identifying walk, and sort the $|E|$ identifying strings in lexicographic order, writing the result as an $|E| \times |E|$ matrix $\mathbf{B}$. The ith instance of a given character $\omega_k \in \Omega$ in the first (F) column of $\mathbf{B}$ corresponds to the same edge as the ith instance of ω_k in the last (L) column of $\mathbf{B}$. This LF mapping property means that G is implicitly encoded in $\mathbf{B}$'s last column, which is the BWT of G. To recover G from its BWT, note first the F column is exactly the characters of the L column in sorted order – that is, F is composed of successive blocks of characters from Ω, with one block per character. Write the F and L columns next to each other. Now:

1. start at some arbitrary row,
2. apply LF mapping to the character in the F column to move to the row whose character in the L column corresponds to the same edge, and
3. repeat step 2 until the starting row of step 1 is reached.

The sequence of characters in the F column encountered on following these instructions completely recapitulates the cycle comprising G. If a given query string q on Ω matches several walks in G, all these walks correspond to a single interval of rows of $\mathbf{B}$. LF mapping can be applied to obtain this interval in time linear in $|q|$, independent of $|E|$ and $|V|$, with an appropriately designed rank data structure over the BWT [10].

LF mapping is generally encountered as a byproduct of an ordering procedure. A recent generalization [5] of the BWT that applies to an arbitrary labeled digraph G obtains a partial co-lexicographic order of its vertices. In this framework, walks in G matching a query string always terminate at some convex subset of the ordered vertices. In the worst case, search performance using a proposed extension to the FM-index based on this transform is $O(|q||E|\log|V|)$. The result is consistent with a recently obtained conditional lower bound for string matching in labeled digraphs: unless the strong exponential time hypothesis is false, no index constructed in time polynomial in $|E|$ can deliver a search performance of $O(|q|^{\delta}|E|^{\beta})$ with either $\delta < 1$ or $\beta < 1$ [6, 7].

A nength can have up to $2^{|V|}-1$ vertices, so the asymptotic scalings of both its construction time and index size include an exponential factor whose argument is $|V|$. However, we expect the situation is not so grim for many classes of graphs. Indeed, [5] establishes an upper bound of $2^p(|V|-p+1)-1$ on the number of states of the DFA obtained by applying the powerset construction to an NFA with $|V|$ states, where p is the width of a partial co-lexicographic order of the NFA's states. Reference [5] further notes the parameter p serves as a complexity measure for graphs, where Wheeler graphs have $p = 1$.

Various analogs to the BWT respecting some of its features while discarding others are possible. In the analog described in [5], invertibility is achieved in the general case by explicitly recording a strategic abbreviation of submatrices of the graph's adjacency matrix that exploits the partial co-lexicographic order of vertices. As the complexity measure p increases, this representation collapses to exactly the adjacency matrix. So the representation is guaranteed to be invertible because at worst, it is a literal encoding of the original graph. In the analog to the FM-index built on this representation, compression is achieved at the expense of search performance, both of which degrade as p increases.

Our perspective is that the powerset construction itself provides a BWT-like transform. A nength sorts possible matches without ordering them. While it no longer has a semblance of what makes the BWT navigable – LF mapping – what makes the BWT invertible is preserved: every vertex has an identifying walk. (Note, for example, if multiple closed walks in a cycle graph matched the same string, the BWT matrix $\mathbf{B}$ would be a sequence of blocks of identical rows. LF mapping would then obtain multiple cycles rather than a single cycle, and invertibility would be lost.) Just as the BWT is an invertible transform of a cycle graph into a string with a beginning and an end, a result of ordering, a nength is an invertible transform of a labeled graph into a different graph for which each vertex is the tail of at most one edge labeled by a given character, a result of sorting.

For both the BWT and nength, properties linked to invertibility can be exploited to rapidly locate matched patterns. An arbitrary finite string on Ω can be extended by an extra sentinel character that is not in Ω. The ends of this string can then be joined to form an aperiodic necklace. The BWT of this necklace is invertible because each character has a distinct distance from the sentinel. The identifying cycle labeled on the sentinel alphabet performs the same function for a nength as a sentinel does for a BWT; there is not necessarily a natural distance between two given vertices of an arbitrary graph, but adding an identifying cycle vests the graph with a distance function on its vertices. Ensuring this cycle matches a generalized de Bruijn sequence gives a performance guarantee for state determination via nength navigation.

The paper introducing Wheeler graphs [15] articulates two main features of the original BWT: (1) it is invertible, and (2) it "helps" compression. The paper also notes some variants of the BWT in the literature do not have these features. Since there are indeed so many such variants, it is worth considering how to define properties of a potentially

broad class of transforms that includes both the BWT and nength. We believe references to compression should be avoided. A labeled digraph can be thought of as a potentially compressed representation of many strings, apart from any transform. Moreover, the BWT does not itself do any compressing, and that it tends to help in approaches to lossless compression of text is, of course, an artifact of the distribution of data encountered in practical settings; there exists some distribution of data for which it would typically "hurt" compression. Rather, we believe at its core, the BWT is a tool for maximally efficient string matching. We also believe invertibility alone is not one of its defining properties. How the BWT achieves invertibility matters.

Given these considerations, we propose defining a search transform as follows. Let X be a configuration of unlabeled objects together with directed relationships, where each relationship connects a subset of objects and has a set of labels, potentially on multiple alphabets. A *search transform* is any transform of X into a different configuration X' of objects and relationships such that

1. X' enables an index that answers whether a structured query pattern of relationships is present in X in time independent of the numbers of objects and relationships X contains, and
2. X is uniquely determined by X' precisely because for every object in X, there is some nonempty query pattern matched only at that object.

Above, we draw a distinction between a search transform and an index enabled by a search transform. A total order of the vertices of a Wheeler graph together with auxiliary data supporting LF mapping-based navigation is a search transform because an FM-index built on it solves the string matching decision problem in time independent of the size and order of the graph [15]. This is despite how using the more compact r-index [16, 17, 29, 21, 22, 1] in place of the FM-index solves the string matching decision problem in time polylogarithmic in the graph's size [14]. Both nength and the BWT are also search transforms. However, the compound transform that linearizes a non-Wheeler graph and subsequently orders the vertices of an equivalent Wheeler graph is excluded from our definition because it is invertible in part via the map between the non-Wheeler graph and the equivalent Wheeler graph. Note our definition leaves room for possible transforms that facilitate matching of patterns more involved than strings. We leave exploration of these possibilities for future work.

References

1 Hideo Bannai, Travis Gagie, and I Tomohiro. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.

2 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In *International workshop on algorithms in bioinformatics*, pages 225–235. Springer, 2012.

3 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical report, Systems Research Center, 1994.

4 Hyewon Choi and Bernd Burgstaller. Non-blocking parallel subset construction on shared-memory multicore architectures. In *Proceedings of the Eleventh Australasian Symposium on Parallel and Distributed Computing-Volume 140*, pages 13–20, 2013.

5 Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2585–2599. SIAM, 2021.

6 Massimo Equi, Veli Mäkinen, and Alexandru I Tomescu. Conditional indexing lower bounds through self-reducibility. *arXiv preprint*, 2020. `arXiv:2002.00629`.

7 Massimo Equi, Veli Mäkinen, and Alexandru I Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless seth fails. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 608–622. Springer, 2021.

8 Tuvi Etzion. An algorithm for generating shift-register cycles. *Theoretical computer science*, 44:209–224, 1986.

9 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM (JACM)*, 57(1):4, 2009.

10 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

11 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

12 Daniel Gabric, Štěpán Holub, and Jeffrey Shallit. Generalized de bruijn words and the state complexity of conjugate sets. In *International Conference on Descriptional Complexity of Formal Systems*, pages 137–146. Springer, 2019.

13 Daniel Gabric, Štěpán Holub, and Jeffrey Shallit. Maximal state complexity and generalized de bruijn words. *Information and Computation*, page 104689, 2021.

14 Travis Gagie. r-indexing wheeler graphs. *arXiv preprint*, 2021. `arXiv:2101.12341`.

15 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for bwt-based data structures. *Theoretical computer science*, 698:67–78, 2017.

16 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477. SIAM, 2018.

17 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM (JACM)*, 67(1):1–54, 2020.

18 Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature biotechnology*, 2018.

19 Farhad Hemmati and Daniel J Costello. An algebraic construction for q-ary shift register sequences. *IEEE Transactions on Computers*, 100(12):1192–1195, 1978.

20 John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

21 Dominik Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1344–1357. SIAM, 2019.

22 Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics read alignment. *Journal of Computational Biology*, 27(4):500–513, 2020.

23 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357, 2012.

24 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

25 Abraham Lempel. m-ary closed sequences. *Journal of Combinatorial Theory, Series A*, 10(3):253–258, 1971.

26 Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint*, 2013. `arXiv:1303.3997`.

27 Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

28 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the burrows–wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007.

29 Gonzalo Navarro and Nicola Prezza. Universal compressed text indexing. *Theoretical Computer Science*, 762:41–50, 2019.

30 Adam M Novak, Erik Garrison, and Benedict Paten. A graph extension of the positional burrows-wheeler transform and its applications. In *International Workshop on Algorithms in Bioinformatics*, pages 246–256. Springer, 2016.

31 Nicola Prezza. On locating paths in compressed tries. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 744–760. SIAM, 2021.

32 Nicola Prezza. Subpath queries on compressed graphs: A survey. *Algorithms*, 14(1):14, 2021.

33 Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

34 Yan Shao, Yanbing Liu, and Jianlong Tan. Accelerating dfa construction by parallelizing subset construction. In *International Conference on Trustworthy Computing and Services*, pages 16–24. Springer, 2014.

35 Jouni Sirén. Indexing variation graphs. In *2017 Proceedings of the ninteenth workshop on algorithm engineering and experiments (ALENEX)*, pages 13–27. SIAM, 2017.

36 Jouni Sirén, Erik Garrison, Adam M Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020.

37 Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014.

R-enum: Enumeration of Characteristic Substrings in BWT-runs Bounded Space

Takaaki Nishimoto
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan

Yasuo Tabei
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan

Abstract

Enumerating characteristic substrings (e.g., maximal repeats, minimal unique substrings, and minimal absent words) in a given string has been an important research topic because there are a wide variety of applications in various areas such as string processing and computational biology. Although several enumeration algorithms for characteristic substrings have been proposed, they are not space-efficient in that their space-usage is proportional to the length of an input string. Recently, the *run-length encoded Burrows-Wheeler transform (RLBWT)* has attracted increased attention in string processing, and various algorithms for the RLBWT have been developed. Developing enumeration algorithms for characteristic substrings with the RLBWT, however, remains a challenge. In this paper, we present *r-enum (RLBWT-based enumeration)*, the first enumeration algorithm for characteristic substrings based on RLBWT. R-enum runs in $O(n \log\log(n/r))$ time and with $O(r \log n)$ bits of working space for string length n and number r of runs in RLBWT. Here, r is expected to be significantly smaller than n for highly repetitive strings (i.e., strings with many repetitions). Experiments using a benchmark dataset of highly repetitive strings show that the results of r-enum are more space-efficient than the previous results. In addition, we demonstrate the applicability of r-enum to a huge string by performing experiments on a 300-gigabyte string of 100 human genomes.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Enumeration algorithm, Burrows-Wheeler transform, Maximal repeats, Minimal unique substrings, Minimal absent words

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.21

Supplementary Material *Software (Source Code)*: `https://github.com/TNishimoto/renum` archived at `swh:1:dir:cde748f235b355d80783d13b86eaa048b7c965ea`

Acknowledgements We thank reviewers for their useful comments.

1 Introduction

Enumerating characteristic substrings (e.g., maximal repeats, minimal unique substrings and minimal absent words) in a given string has been an important research topic because there are a wide variety of applications in various areas such as string processing and computational biology. The usefulness of the enumeration of maximal repeats has been demonstrated in lossless data compression [20], bioinformatics [6, 23] and string classification with machine learning models [30, 28]. The enumeration of minimal unique substrings and minimal absent words has shown practical benefits in bioinformatics [24, 1, 14, 16] and data compression [18, 19]. There is therefore a strong need to develop scalable algorithms for enumerating characteristic substrings in a huge string.

The *Burrows-Wheeler transform (BWT)* [13] is for permutation-based lossless data compression of a string, and many enumeration algorithms for characteristic substrings leveraging BWT have been proposed. Okanohara and Tsujii [30] proposed an enumeration algorithm for maximal repeats that uses BWT and an enhanced suffix array [2]. Since their

algorithm takes linear time to the length n of a string and $O(n \log n)$ working space, applying it to a huge string is computationally demanding. Beller et al. [12] proposed an enumeration algorithm for maximal repeats that uses a *range distinct query* on the BWT of a string in $O(n \log \sigma)$ time and $n \log \sigma + o(n \log \sigma) + O(n)$ bits of working space for alphabet size σ of a string. Since the working space of their algorithm is linearly proportional to the length n of a string, a large amount of space is expected to be consumed for huge strings. Along the same line of research, Belazzougui et al. [9, 7] proposed an algorithm for enumerating characteristic substrings in $O(n \log \sigma)$ time and $n \log \sigma + o(n \log \sigma) + O(\sigma^2 \log^2 n)$ bits of working space, which resulted in the space usage being linearly proportional to the string length. Thus, developing a more space-efficient enumeration algorithm for the characteristic substrings of a string remains a challenging issue.

Run-length BWT (RLBWT) is a recent, popular lossless data compression, and it is defined as a run-length compressed BWT for strings. Thus, the compression performance of RLBWT has been shown to be high, especially for *highly repetitive strings* (i.e., strings with many repetitions) such as genomes, version-controlled documents, and source code repositories. Kempa and Kociumaka [26] showed an upper bound on the size of the RLBWT by using a measure of repetitiveness. Although several compressed data structures and string processing algorithms that use RLBWT have also been proposed (e.g., [8, 21, 29, 4, 25]), no previous algorithms for enumerating characteristic substrings based on RLBWT have been proposed. Such enumeration algorithms are expected to be much more space-efficient than existing algorithms for highly repetitive strings.

Contribution. We present the first enumeration algorithm for characteristic substrings based on RLBWT, which we call *r-enum (RLBWT-based enumeration)*. Following the idea of the previous works [12, 7], r-enum performs an enumeration by simulating traversals of a *Weiner-link tree* (e.g., [9]), which is a trie, each node of which represents a right-maximal repeat in a string T of length n. Each characteristic substring in T corresponds to a node in the Weiner-link tree of T. This is made possible in $O(n \log \log_w(n/r) + occ)$ time and $O(r \log n)$ bits of working space for number r of runs in RLBWT, machine word size $w = \Theta(\log n)$, and number occ of characteristic substrings. For a highly repetitive string such that $r = o(n \log \sigma / \log n)$ holds, r-enum is more space-efficient than the best previous algorithms taking $O(nd)$ time and $|RD| + O(n)$ bits of space, where $|RD|$ is the size of a data structure supporting range distinct queries and computing the LF function in $O(d)$ time; a pair $(|RD|, d)$ can be chosen as $(|RD|, d) = (n \log \sigma + o(n \log \sigma), O(\log \sigma))$ [15] or $(|RD|, d) = (O(n \log \sigma), O(1))$ [9, Lemmas 3.5 and 3.17]. Table 1 summarizes the running time and working space of state-of-the-art algorithms including those by Okanohara and Tsujii (OT method) [30], Beller et al. (BBO method) [12], and Belazzougui and Cunial (BC method) [7] in comparison with our r-enum.

Experiments using a benchmark dataset of highly repetitive strings show that r-enum is more space-efficient than the previous algorithms. In addition, we demonstrate the applicability of r-enum to a huge string by performing experiments on a 300-gigabyte string of 100 human genomes, which has not been shown in the previous work so far.

The outline of this paper is as follows. Section 2 introduces several basic notions, including the Weiner-link tree. In Section 3, we present a traversal algorithm for the Weiner-link tree of T in $O(r \log n)$ bits. Section 4 presents r-enum for finding the corresponding nodes to maximal repeats, minimal unique substrings, and minimal absent words. In Section 5, we slightly modify r-enum such that it outputs each characteristic substring and its occurrences in T instead of the corresponding node to the characteristic substring. Section 6 shows the performance of our method on benchmark datasets of highly repetitive strings.

Table 1 Summary of running time and working space of enumeration algorithms for (i) maximal repeats, (ii) minimal unique substrings (MUSs), and (iii) minimal absent words (MAWs) for each method. Last column represents main data structure used in each algorithm. Input of each algorithm is string T of length n or BWT of T, and each outputted characteristic substring is represented by pointer with $O(\log n)$ bits. We exclude inputs and outputs from working space. In addition, σ is alphabet size of T, $w = \Theta(\log n)$ is machine word size, r is the number of runs in RLBWT of T, and $occ = O(n\sigma)$ [17] is the number of minimal absent words for T. RD means data structure (i) supporting range distinct queries in $O(d)$ time per output element and (ii) computing LF function in $O(d)$ time; $|RD|$ is its size. We can choose $(|RD|, d) = (n \log \sigma + o(n \log \sigma), O(\log \sigma))$ [15] or $(|RD|, d) = (O(n \log \sigma), O(1))$ [9, Lemmas 3.5 and 3.17].

(i) Maximal repeats	Running time	Working space (bits)	Data structures
OT method [30]	$O(n)$	$O(n \log n)$	Enhanced suffix array
[9, Theorem 7.8]	$O(n)$	$O(n \log \sigma)$	BWT and RD
BBO method [12]	$O(nd)$	$\|RD\| + O(n)$	BWT and RD
BC method [7]	$O(nd)$	$\|RD\| + O(\sigma^2 \log^2 n)$	BWT and RD
r-enum (this study)	$O(n \log \log_w (n/r))$	$O(r \log n)$	RLBWT and RD

(ii) MUSs	Running time	Working space (bits)	Data structures
[9]	$O(n)$	$O(n \log \sigma)$	BWT and RD
BC method [7]	$O(nd)$	$\|RD\| + O(\sigma^2 \log^2 n)$	BWT and RD
r-enum (this study)	$O(n \log \log_w (n/r))$	$O(r \log n)$	RLBWT and RD

(iii) MAWs	Running time	Working space (bits)	Data structures
[5]	$O(n + occ)$	$O(n \log n)$	Suffix array
[9, Theorem 7.12]	$O(n + occ)$	$O(n \log \sigma)$	BWT and RD
BC method [7]	$O(nd + occ)$	$\|RD\| + O(\sigma^2 \log^2 n)$	BWT and RD
r-enum (this study)	$O(n \log \log_w (n/r) + occ)$	$O(r \log n)$	RLBWT and RD

2 Preliminaries

Let $\Sigma = \{1, 2, \ldots, \sigma\}$ be an ordered alphabet, T be a string of length n over Σ, and $|T|$ be the length of T. Let $T[i]$ be the i-th character of T (i.e., $T = T[1], T[2], \ldots, T[n]$), and $T[i..j]$ be the substring of T that begins at position i and ends at position j. For two strings T and P, $T \prec P$ means that T is lexicographically smaller than P. $Occ(T, P)$ denotes all the occurrence positions of P in T, i.e., $Occ(T, P) = \{i \mid i \in [1, n - |P| + 1] \text{ s.t. } P = T[i..(i + |P| - 1)]\}$. We assume that (i) the last character of T is a special character \$ not occurring in substring $T[1..n-1]$, (ii) $|T| \geq 2$, and (iii) every character in Σ occurs at least once in T. For two integers b and e ($b \leq e$), *interval* $[b, e]$ represents the set $\{b, b+1, \ldots, e\}$. Let $\mathsf{substr}(T)$ denote the set of all the distinct substrings of T (i.e., $\mathsf{substr}(T) = \{T[i..j] \mid 1 \leq i \leq j \leq n\}$).

In this paper, characteristic substrings of a string consist of *maximal repeats*, *minimal unique substrings*, and *minimal absent words*. A maximal repeat in T is defined as a substring P satisfying two conditions: (i) it occurs at least twice in the string (i.e., $|Occ(T, P)| \geq 2$), and (ii) either of the left or right extended substrings of it occurs fewer times than it (i.e., $|Occ(T, cP)|, |Occ(T, Pc)| < |Occ(T, P)|$ for any $c \in \Sigma$). A minimal unique substring is defined as substring P satisfying two conditions: (i) it occurs just once (i.e., $|Occ(T, P)| = 1$), and (ii) all the proper substrings of it occur at least twice in the string (i.e., $|Occ(T, P[2..|P|])|, |Occ(T, P[1..|P|-1])| \geq 2$). A minimal absent word is defined as string P satisfying two conditions: (i) it does not occur in a string (i.e., $|Occ(T, P)| = 0$), and (ii) all the

Sorted circular strings

i	SA	LCP	LF	F		L
1	11	0	2	$	a b a a b a b a b	a
2	10	0	8	a	$ a b a a b a b a	b
3	3	1	9	a	a b a b a b a $ a	b
4	8	1	10	a	b a $ a b a a b a	b
5	1	3	1	a	b a a b a b a b a	$
6	6	3	11	a	b a b a $ a b a a	b
7	4	5	3	a	b a b a b a $ a b	a
8	9	0	4	b	a $ a b a a b a b	a
9	2	2	5	b	a a b a b a b a $	a
10	7	2	6	b	a b a $ a b a a b	a
11	5	4	7	b	a b a b a $ a b a	a

Figure 1 Suffix array, LCP array, LF function, BWT, and circular strings for $T = abaabababa\$$.

proper substrings of it occur in the string (i.e., $|Occ(T, P[2..|P|])|, |Occ(T, P[1..|P|-1])| \geq 1$). For convenience, a minimal absent word is sometimes called a substring, although the string is not a substring of T.

Our computation model is a unit-cost word RAM with a machine word size of $w = \Theta(\log_2 n)$ bits. We evaluate the space complexity in terms of the number of machine words. A bitwise evaluation of the space complexity can be obtained with a multiplicative factor of $\log_2 n$. We assume the base-2 logarithm throughout this paper when the base is not indicated.

2.1 Rank and range distinct queries

Let $S \subseteq \{1, 2, \ldots, n\}$ be a set of d integers. A *rank* query $\mathsf{rank}(S, i)$ on S returns the number of elements no more than i in S, i.e., $\mathsf{rank}(S, i) = |\{j \mid j \in S \text{ s.t. } j \leq i\}|$. $R_{\mathsf{rank}}(S)$ is a *rank data structure* solving a rank query on S in $O(\log\log_w(n/d))$ time and with $O(dw)$ bits of space [10].

A range distinct query, $\mathsf{RD}(T, b, e)$ on a string T returns a set of 3-tuples (c, p_c, q_c) that consists of (i) a distinct character c in $T[b..e]$, (ii) the first occurrence p_c of the character c for a given interval $[b..e]$ in T, and (iii) the last occurrence q_c of the character c for $[b..e]$ in T. Formally, let $\Sigma(T[b..e])$ be the set of distinct characters in $T[b..e]$, i.e., $\Sigma(T[b..e]) = \{T[i] \mid i \in [b, e]\}$. Then, $\mathsf{RD}(T, b, e) = \{(c, p_c, q_c) \mid c \in \Sigma(T[b..e])\}$, where $p_c = \min(Occ(T, c) \cap [b, e])$, and $q_c = \max(Occ(T, c) \cap [b, e])$. $R_{\mathsf{RD}}(T)$ is a *range distinct data structure* solving a range distinct query on T in $O(|\mathsf{RD}(T, b, e)| + 1)$ time and with $O(n \log \sigma)$ bits of space [11].

2.2 Suffix and longest common prefix arrays

The *suffix array* [27] SA of string T is an integer array of size n such that $\mathsf{SA}[i]$ stores the starting position of the i-th suffix of T in lexicographical order. Formally, SA is a permutation of $\{1, 2, \ldots, n\}$ such that $T[\mathsf{SA}[1]..n] \prec \cdots \prec T[\mathsf{SA}[n]..n]$. The *longest common prefix array* (LCP array) LCP of T is an integer array of size n such that $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i]$ stores the length of the LCP of the two suffixes $T[\mathsf{SA}[i]..n]$ and $T[\mathsf{SA}[i-1]..n]$ for $i \in \{2, 3, \ldots, n\}$. We call the values in the suffix and LCP arrays *sa-values* and *lcp-values*, respectively. Moreover, let LF be a function such that (i) $\mathsf{SA}[\mathsf{LF}(i)] = \mathsf{SA}[i] - 1$ for any integer $i \in (\{1, 2, \ldots, n\} \setminus \{x\})$ and (ii) $\mathsf{SA}[\mathsf{LF}(x)] = n$, where x is an integer such that $\mathsf{SA}[x] = 1$. Figure 1 depicts the suffix array, LCP array, and LF function for a string.

2.3 BWT and RLBWT

The BWT [13] of string T is an array L built by permuting T as follows; (i) all the n rotations of T are sorted in lexicographical order, and (ii) $L[i]$ for any $i \in \{1, 2, \ldots, n\}$ is the last character at the i-th rotation in sorted order. Similarly, $F[i]$ for any $i \in \{1, 2, \ldots, n\}$ is the first character at the i-th rotation in sorted order. Formally, let $L[i] = T[\mathsf{SA}[\mathsf{LF}(i)]]$ and $F[i] = T[\mathsf{SA}[i]]$.

Since $L[i]$ and $L[j]$ represent two characters $T[\mathsf{SA}[i]-1]$ and $T[\mathsf{SA}[j]-1]$ for two distinct integers $i, j \in \{1, 2, \ldots, n\}$, the following relation holds between $\mathsf{LF}(i)$ and $\mathsf{LF}(j)$; $\mathsf{LF}(i) < \mathsf{LF}(j)$ if and only if either of two conditions holds: (i) $L[i] \prec L[j]$ or (ii) $L[i] = L[j]$ and $i < j$. Let C be an array of size σ such that $C[c]$ is the number of occurrences of characters lexicographically smaller than $c \in \Sigma$ in string T, i.e., $C[c] = |\{i \mid i \in [1, n] \text{ s.t. } T[i] \prec c\}|$. The following equation holds by the above relation for the LF function on BWT L: $\mathsf{LF}(i) = C[c] + Occ(L[1..i], L[i])$. We call the equation *LF formula*.

The RLBWT of T is the BWT encoded by run-length encoding, i.e., RLBWT is a partition of L into r substrings $L[\ell(1)..\ell(2)-1], L[\ell(2)..\ell(3)-1], \ldots, L[\ell(r)..\ell(r+1)-1]$ such that each substring $L[\ell(i)..\ell(i+1)-1]$ is a maximal repetition of the same character in L called a *run*. Formally, let $r = 1 + |\{i \mid i \in \{2, 3, \ldots, n\} \text{ s.t. } L[i] \neq L[i-1]\}|$, $\ell(1) = 1$, $\ell(r+1) = n+1$, and $\ell(j) = \min\{i \mid i \in \{\ell(j-1)+1, \ell(j-1)+2, \ldots, n\} \text{ s.t. } L[i] \neq L[i-1]\}$ for $j \in \{2, 3, \ldots, r\}$. Let $\mathcal{S}_{\mathsf{start}}$ denote the set of the starting position of each run in L, i.e., $\mathcal{S}_{\mathsf{start}} = \{\ell(1), \ell(2), \ldots, \ell(r)\}$. RLBWT is represented as r pairs $(L[\ell(1)], \ell(1))$, $(L[\ell(2)], \ell(2))$, $\ldots$, $(L[\ell(r)], \ell(r))$ using $2rw$ bits. $\sigma \leq r \leq n$ holds since we assume that every character in Σ occurs in T.

$\mathsf{LF}(i) = \mathsf{LF}(i-1) + 1$ holds for an integer $i \in \{2, 3, \ldots, n\}$ by LF formula if i is not the starting position of a run in L (i.e., $i \notin \mathcal{S}_{\mathsf{start}}$). Similarly, $\mathsf{LCP}[\mathsf{LF}(i)] = 1 + \mathsf{LCP}[i]$ holds if $i \notin \mathcal{S}_{\mathsf{start}}$.

Figure 1 depicts two arrays L and F for string $T = abaabababa\$$. Since the BWT of $T = abaabababa\$$ is $abbb\$baaaaa$, the RLBWT of the string T is $(a, 1), (b, 2), (\$, 5), (b, 6)$, and $(a, 7)$. The red and blue characters a are adjacent in a run on the BWT L. Hence, $\mathsf{LF}(8) = \mathsf{LF}(7) + 1$ holds by LF formula. Similarly, $\mathsf{LCP}[\mathsf{LF}(8)] = \mathsf{LCP}[8] + 1 = 1$ holds.

2.4 Suffix tree

The *suffix tree* [3] of T is a trie storing all the suffixes of T. Each node v represents the concatenated string on the path from the root of the tree to the node v. Let u_P denote the node representing a substring P of T. The *depth* of node u_P is defined as the length of its string P, i.e., the depth of u_P is $|P|$. Let $\mathsf{children}(P)$ denote the set of strings represented by the children of u_P. Formally, $\mathsf{children}(P) = \{Pc \mid c \in \Sigma \text{ s.t. } Pc \in \mathsf{substr}(T)\}$. We call u_P an *explicit node* if it has at least two children; otherwise, we call u_P an *implicit node*. The root u_ε of the suffix tree is explicit because T contains at least two distinct characters. Let $\mathcal{L}_d$ be the set of substrings represented by all the explicit nodes with depth d. Formally, $\mathcal{L}_d = \{P \mid P \in \mathsf{substr}(T) \text{ s.t. } |P| = d \text{ and } |\mathsf{children}(P)| \geq 2\}$.

The *suffix-tree interval* (a.k.a. *suffix-array interval*) for a substring P of T is an interval $[b, e]$ on the suffix array of T such that $\mathsf{SA}[b..e]$ represents all the occurrence positions of P in string T; that is, for any integer $p \in \{1, 2, \ldots, n\}$, $T[p..p+|P|-1] = P$ holds if and only if $p \in \{\mathsf{SA}[b], \mathsf{SA}[b+1], \ldots, \mathsf{SA}[e]\}$. The suffix-tree interval of the empty string ε is defined as $[1, n]$. Let $\mathsf{interval}(P) = [b, e]$ denote the suffix-tree interval for P.

The *rich representation* [9] $\mathsf{repr}(P)$ for P is a 3-tuple $(\mathsf{interval}(P), \{(c_1, b_1, e_1), (c_2, b_2, e_2), \ldots, (c_k, b_k, e_k)\}, |P|)$. Here, $Pc_1, Pc_2, \ldots, Pc_k$ are strings represented by the children of node u_P, and $[b_i, e_i] = \mathsf{interval}(Pc_i)$ for $i \in [1, k]$.

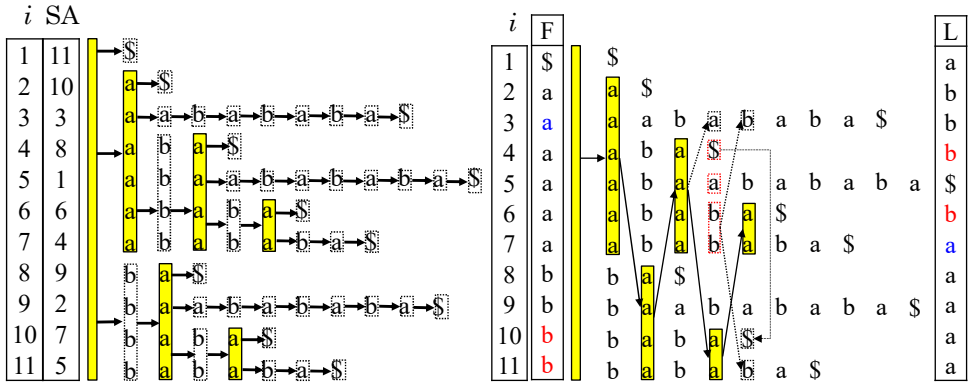


Figure 2 Left figure illustrates suffix tree of $T = abaababababa\$$ on sorted suffixes of T. We represent each node by its suffix-tree interval (rectangles) and omit characters on edges. Yellow and white rectangles are explicit and implicit nodes, respectively. Right figure illustrates Weiner-link tree of T and Weiner links on sorted suffixes of T. Tree consists of yellow rectangles and solid arrows. We omit characters on Weiner links. Solid and dotted arrows represent Weiner links pointing to explicit and implicit nodes, respectively. Red rectangles are children of node of aba in suffix tree of T.

Figure 2 illustrates the suffix tree of string $T = abaababababa\$$ (left figure). The three sets $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_2$ are $\{\varepsilon\}, \{a\}, \{ba\}$, respectively. The suffix-tree interval for substring $P = aba$ is $[4, 7]$, and $\mathsf{children}(P) = \{aba\$, abaa, abab\}$. The rich representation for P is $([4, 7], \{(\$, 4, 4), (a, 5, 5), (b, 6, 7)\}, 3)$.

2.5 Weiner links and Weiner-link tree

Weiner links are directed links on the suffix tree of T. Let cP be a substring of T, where c is a character, and P is a string. Then, node u_{cP} is the destination of a Weiner link with character c starting at node u_P. Hence, every node u_P must be the destination of exactly one Weiner link unless node u_P is the root of the suffix tree (i.e., $P = \varepsilon$). Node u_P is always explicit if u_{cP} is explicit. This is because the explicit node u_{cP} indicates that T has two substrings cPc_1 and cPc_2, where c_1 and c_2 are distinct characters. Let $\mathsf{WLink}(P)$ denote the set of strings such that the node for each string is the destination of a Weiner link starting at node u_P (i.e., $\mathsf{WLink}(P) = \{cP \mid c \in \Sigma \text{ s.t. } cP \in \mathsf{substr}(T)\}$).

The Weiner-link tree (a.k.a. *suffix-link tree*) for T is a graph such that (i) the nodes are all the explicit nodes in the suffix tree of T and (ii) the edges are all the Weiner links among the explicit nodes. Since any explicit node is the destination of a Weiner link starting at another explicit node, the graph results in a tree. Each child of a node u_P represents a string in $\mathsf{WLink}(P)$ in the Weiner-link tree.

Let p_c and q_c be the first and last occurrences of a character c on the suffix-tree interval for a substring P in the BWT L of T, respectively. Then, $[\mathsf{LF}(p_c), \mathsf{LF}(q_c)]$ is equal to the suffix-tree interval of cP. Hence, we can compute the suffix-tree intervals for the destinations of all the Weiner links starting at node u_P using a range distinct query and LF function. Formally, let $[b, e] = \mathsf{interval}(P)$ for a substring P of T. Then, the following equation holds:

$$\{\mathsf{interval}(cP) \mid cP \in \mathsf{WLink}(P)\} = \{[\mathsf{LF}(p_c), \mathsf{LF}(q_c)] \mid (c, p_c, q_c) \in \mathsf{RD}(L, b, e) \text{ s.t. } c \neq \$\} \quad (1)$$

Let Q_P denote an array of size σ for a substring P of T such that $Q_P[c]$ stores set $\{(c', b, e) \mid cPc' \in \mathsf{children}(cP)$ and let $[b, e] = \mathsf{interval}(cPc')\}$ for a character $c \in \Sigma$ if cP is a substring of T; otherwise, $Q_P[c] = \emptyset$. The array Q_P has three properties for any

character $c \in \Sigma$: (i) $cP \in \mathsf{WLink}(P)$ holds if and only if $|Q_P[c]| \geq 1$ holds, (ii) node u_{cP} is explicit if and only if $|Q_P[c]| \geq 2$ holds, and (iii) $Q_P[c] = \bigcup_{Pc' \in \mathsf{children}(P)} \{(c', b, e) \mid \hat{c}Pc' \in \mathsf{WLink}(Pc') \text{ s.t. } \hat{c} = c \text{ and let } [b, e] = \mathsf{interval}(cPc')\}$ holds. In other words, each child of node u_{cP} is the destination of a Weiner link starting at a child of node u_P in the suffix tree.

We can compute the children of a node u_P in the Weiner-link tree using Q_P without explicitly constructing the Weiner-link tree. Formally, the following lemma holds.

▶ **Lemma 1** (Lemma 4.1 in [9]). *Let* RD *be a data structure supporting a range distinct query on BWT L in $O((1+k)t_{\mathsf{RD}})$ time and computing LF function in $O(t_{\mathsf{LF}})$ time. Here, k is the number of elements output by the range distinct query. We can compute set $\mathcal{Y} = \{\mathsf{repr}(cP) \mid cP \in \mathsf{WLink}(P)\}$ using (i) $\mathsf{repr}(P)$, (ii) data structure* RD*, and (iii) an empty array X of size σ. After that, we can divide the set $\mathcal{Y}$ into two sets for explicit and implicit nodes. The computation time and working space are $O(h(t_{\mathsf{RD}} + t_{\mathsf{LF}}))$ and $O((\sigma + h')w)$ bits, respectively, where $h = \sum_{Pc' \in \mathsf{children}(P)} |\mathsf{WLink}(Pc')|$, and $h' = \sum_{cP \in \mathsf{WLink}(P) \cap \mathcal{L}_{|P|+1}} |\mathsf{children}(cP)|$.*

Proof. We compute the outputs with the following three steps. At the first step, we compute set $\{\mathsf{interval}(cP) \mid cP \in \mathsf{WLink}(P)\}$ and convert the empty array X into Q_P using Equation 1 and the third property of Q_P. At the second step, we output the rich representation $\mathsf{repr}(cP) = (\mathsf{interval}(cP), Q_P[c], |P| + 1)$ for each $cP \in \mathsf{WLink}(P)$ and divide the rich representations into two sets for explicit and implicit nodes using the second property of Q_P. At the last step, we remove all the elements from $Q_P[c]$ for each $cP \in \mathsf{WLink}(P)$ to recover X from Q_P. We perform the three steps using range distinct queries to the suffix-tree intervals for node u_P and its children, and the intervals stored in $\mathsf{repr}(P)$. Hence, the running time is $O(h(t_{\mathsf{RD}} + t_{\mathsf{LF}}))$ in total.

Next, we analyze the working space. The rich representation for a node $u_{P'}$ takes $O(w)$ bits if $u_{P'}$ is implicit because it has at most one child. Otherwise, $\mathsf{repr}(P')$ takes $O(|\mathsf{children}(P')|w)$ bits. Hence, the working space is $O((\sigma + h')w)$ bits in total. ◀

Figure 2 illustrates the Weiner-link tree of string $T = abaababababa\$$ (right figure). Since T contains two substrings $aaba$ and $baba$, the node of aba has two Weiner links pointing to the nodes of $aaba$ and $baba$, i.e., $\mathsf{WLink}(aba) = \{aaba, baba\}$. The Weiner-link tree contains the node of $baba$ but not that of $aaba$ because the former and latter nodes are explicit and implicit, respectively.

The three suffix-tree intervals for aba, $aaba$, and $baba$ are $[4, 7]$, $[3, 3]$, and $[10, 11]$, respectively. A range distinct query on $\mathsf{interval}(aba)$ in the BWT of T returns the set $\{(\$, 5, 5), (a, 7, 7), (b, 4, 6)\}$. Hence, $\mathsf{interval}(aaba) = [\mathsf{LF}(7), \mathsf{LF}(7)]$ and $\mathsf{interval}(baba) = [\mathsf{LF}(4), \mathsf{LF}(6)]$ hold by Equation 1 (See also red and blue characters on two arrays F and L in Figure 2). Figure 2 also illustrates the children of the node of aba in the suffix tree and Weiner links starting from the children. In this example, $Q_P[\$] = \emptyset$, $Q_P[a] = \{(b, 3, 3)\}$, and $Q_P[b] = \{(\$, 10, 10), (b, 11, 11)\}$, where $P = aba$. Hence, the node of $aaba$ has one child, and the node of $baba$ has two children in the suffix tree.

3 Traversing Weiner-link tree in $O(rw)$ bits of space

In this section, we present a breadth-first traversal algorithm for the Weiner-link tree of T in $O(rw)$ bits of working space. The traversal algorithm outputs n sets $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_0\}$, $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_1\}$, …, $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_{n-1}\}$ in left-to-right order. Here, each set $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_t\}$ represents the set of the rich representations for all the nodes with depth t in the Weiner-link tree. Hence each node u_P is represented as its rich representation $\mathsf{repr}(P)$.

3.1 Data structures

Let D_{LF} be an array of size r such that $D_{\mathsf{LF}}[i] = \mathsf{LF}(\ell(i))$ for $i \in \{1, 2, \ldots, r\}$ and $L' = L[\ell(1)], L[\ell(2)], \ldots, L[\ell(r)]$. Our traversal algorithm uses six data structures: (i) the RLBWT of T (i.e., r pairs $(L[\ell(1)], \ell(1))$, $(L[\ell(2)], \ell(2))$, $\ldots$, $(L[\ell(r)], \ell(r))$, which are introduced in Section 2.3), (ii) the rich representation for the root of the Weiner-link tree of T (i.e., $\mathsf{repr}(\varepsilon)$), (iii) D_{LF}, (iv) $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$, (v) $R_{\mathsf{RD}}(L')$, and (vi) an empty array X of size σ, where $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$ is the rank data structure introduced in Section 2.1, and it is built on the set $\mathcal{S}_{\mathsf{start}} = \{\ell(1), \ell(2), \ldots, \ell(r)\}$ introduced in Section 2.3. Similarly, $R_{\mathsf{RD}}(L')$ is the range distinct data structure introduced in Section 2.1, and it is built on the string L'. The six data structures require $O((r + \sigma)w)$ bits in total. We construct the six data structures in $O(n \log \log_w(n/r))$ time and $O(rw)$ bits of working space by processing the RLBWT of T (see Appendix A).

We use the two data structures D_{LF} and $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$ to compute LF function. Let x be the index of a run containing the i-th character of BWT L (i.e., $x = \mathsf{rank}(\mathcal{S}_{\mathsf{start}}, i)$) for an integer $i \in \{1, 2, \ldots, n\}$. $\mathsf{LF}(i) = \mathsf{LF}(\ell(x)) + |Occ(L[\ell(x)..i], L[i])| - 1$ holds by LF formula, and $|Occ(T[\ell(x)..i], L[i])| = i - \ell(x) + 1$ holds because $L[\ell(x)..i]$ consists of a repetition of the i-th character $L[i]$. Hence, $\mathsf{LF}(i) = D_{\mathsf{LF}}[x] + (i - \ell(x))$ holds, and we can compute LF function in $O(\log \log_w(n/r))$ time using D_{LF} and $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$.

We use the fifth data structure $R_{\mathsf{RD}}(L')$ to compute a range distinct query on the BWT L of T. Let b' and e' be the indexes of the two runs on L containing two characters $L[b]$ and $L[e]$, respectively (i.e., $b' = \mathsf{rank}(\mathcal{S}_{\mathsf{start}}, b)$ and $e' = \mathsf{rank}(\mathcal{S}_{\mathsf{start}}, e)$), for an interval $[b, e] \subseteq \{1, 2, \ldots, n\}$. Then the following relation holds between two sets $\mathsf{RD}(L, b, e)$ and $\mathsf{RD}(L', b', e')$.

▶ **Lemma 2.** $\mathsf{RD}(L, b, e) = \{(c, \max\{\ell(p), b\}, \min\{\ell(q + 1) - 1, e\}) \mid (c, p, q) \in \mathsf{RD}(L', b', e')\}$ *holds.*

Proof. $L[b..e]$ consists of $e' - b' + 1$ repetitions $L[b..\ell(b' + 1) - 1]$, $L[\ell(b' + 1)..\ell(b' + 2) - 1]$, $\ldots$, $L[\ell(e' - 1)..\ell(e') - 1]$, $L[\ell(e')..e]$. Hence, the following three statements hold: (i) The query $\mathsf{RD}(L', b', e')$ returns all the distinct characters in $L[b..e]$ (i.e., $\{L[i] \mid i \in [b, e]\} = \{L'[i] \mid i \in [b', e']\}$). (ii) Let p be the first occurrence of a character c on $[b', e']$ in L'. Then, the first occurrence of a character c on $[b, e]$ in L is equal to $\ell(p)$ if $\ell(p) \geq b$; otherwise, the first occurrence is equal to b. (iii) Similarly, let q be the last occurrence of the character c on $[b', e']$ in L'. Then, the last occurrence of c on $[b, e]$ in L is equal to $\ell(q + 1) - 1$ if $\ell(q + 1) - 1 \leq e$; otherwise, the last occurrence is equal to e. We obtain Lemma 2 with the three statements. ◀

Lemma 2 indicates that we can solve a range distinct query on L using two rank queries on $\mathcal{S}_{\mathsf{start}}$ and a range distinct query on L'. The range distinct query on L takes $O(\log \log_w(n/r) + k)$ time, where k is the number of output elements.

We use the empty array X to compute rich representations using Lemma 1. The algorithm of Lemma 1 takes $O((k + 1) \log \log_w(n/r))$ time using the six data structures since we can compute LF function and solve the range distinct query on L in $O(\log \log_w(n/r))$ and $O((k + 1) \log \log_w(n/r))$ time, respectively.

3.2 Algorithm

The basic idea behind our breadth-first traversal is to traverse the Weiner-link tree without explicitly building the tree in order to reduce the working space. The algorithm computes nodes sequentially using Lemma 1. Recall that the algorithm of Lemma 1 returns the rich

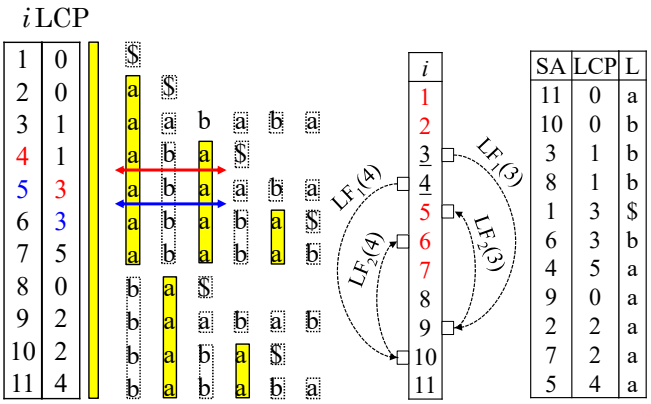


Figure 3 Left figure illustrates part of suffix tree of $T = abaabababa\$$ on sorted suffixes of T. Two colored integers in array i are positions in $\mathcal{K}'_2 = \{4, 5\}$. Similarly, two colored integers in LCP array correspond to positions in $\mathcal{P}_2 = \{5, 6\}$. Right figure illustrates array i, SA, LCP array, and BWT L for $T = abaabababa\$$. Red integers in array i are integers in set $\mathcal{S}_{\mathsf{start}} = \{1, 2, 5, 6, 7\}$. Similarly, underlined integers are integers in set $\mathcal{P}_1 = \{3, 4\}$.

representations for all the children of a given explicit node u_P in the Weiner-link tree (i.e., it returns set $\{\mathsf{repr}(cP) \mid cP \in \mathsf{WLink}(P) \text{ s.t. } cP \in \mathcal{L}_{|P|+1}\}$). This fact indicates that the set of the rich representations for all the nodes with a depth $t \geq 1$ is equal to the union of the sets of rich representations obtained by applying Lemma 1 to all the nodes with depth $t-1$ in the Weiner-link tree, i.e., $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_t\} = \bigcup_{P \in \mathcal{L}_{t-1}} \{\mathsf{repr}(cP) \mid cP \in \mathsf{WLink}(P)\}$.

Our algorithm consists of $(n-1)$ steps. At each t-th step, the algorithm (i) applies Lemma 1 to each representation in set $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_{t-1}\}$, (ii) outputs set $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_t\}$, and (iii) removes the previous set $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_{t-1}\}$ from working memory. The algorithm can traverse the whole Weiner-link tree in breadth-first order because we initially store the first set $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_0\}$ for the first step.

3.3 Analysis

The traversal algorithm requires $O(H_{t-1} \log \log_w(n/r))$ computation time at the t-th step. Here, H_t is the number of Weiner links starting from the children of the explicit nodes with depth t (i.e., $H_t = \sum_{P \in \mathcal{L}_t} \sum_{Pc \in \mathsf{children}(P)} |\mathsf{WLink}(Pc)|$). The running time is $O((\sum_{t=0}^{n} H_t) \log \log_w(n/r))$ in total. The term $\sum_{t=0}^{n} H_t$ represents the number of Weiner links starting from the children of all the explicit nodes in the suffix tree. Belazzougui et al. showed that $\sum_{t=0}^{n} H_t = O(n)$ holds [9, Observation 1]. Hence our traversal algorithm runs in $O(n \log \log_w(n/r))$ time.

Next, we analyze the working space of the traversal algorithm. Let $\mathcal{K}_t$ be the set of the children of all the explicit nodes with depth t in the suffix tree of T (i.e., $\mathcal{K}_t = \bigcup_{P \in \mathcal{L}_t} \mathsf{children}(P)$). Then, the algorithm requires $O((r + \sigma + |\mathcal{K}_{t-1}| + |\mathcal{K}_t|)w)$ bits of working space while executing the t-th step. Hence our traversal algorithm requires $O((r + \sigma + \max\{|\mathcal{K}_0|, |\mathcal{K}_1|, \ldots, |\mathcal{K}_n|\})w))$ bits of working space while the algorithm runs.

We introduce a set $\mathcal{K}'_t$ to analyze the term $|\mathcal{K}_t|$ for an integer t. The set $\mathcal{K}'_t$ consists of the children of all the explicit nodes with depth t except for the last child of each explicit node, i.e., $\mathcal{K}'_t = \bigcup_{P \in \mathcal{L}_t} \{Pc \mid Pc \in \mathsf{children}(P) \text{ s.t. } e' \neq e\}$, where $[b, e] = \mathsf{interval}(P)$, and $[b', e'] = \mathsf{interval}(Pc)$. $|\mathcal{K}_t| = |\mathcal{K}'_t| + |\mathcal{L}_t|$ holds because every explicit node has exactly one last child. $|\mathcal{L}_t| \leq |\mathcal{K}'_t|$ also holds because every explicit node has at least two children in the suffix tree of T. Hence, we obtain the inequality $|\mathcal{K}_t| \leq 2|\mathcal{K}'_t|$.

We also introduce a set $\mathcal{P}_t$ for an integer t. The set $\mathcal{P}_t$ consists of positions with lcp-value t on the LCP array of T except for the first lcp-value $LCP[1]$, i.e., $\mathcal{P}_t = \{i \mid i \in \{2, 3, \ldots, n\}$ s.t. $\mathsf{LCP}[i] = t\}$. Let P be the longest common prefix of $T[\mathsf{SA}[i-1]..n]$ and $T[\mathsf{SA}[i]..n]$ for position $i \in \mathcal{P}_t$, i.e., $P = T[\mathsf{SA}[i]..\mathsf{SA}[i] + \mathsf{LCP}[i] - 1]$. Then node u_P is explicit, and two nodes u_{Pc} and $u_{Pc'}$ are adjacent children of u_P, where $c = T[\mathsf{SA}[i-1] + \mathsf{LCP}[i]]$ and $c' = T[\mathsf{SA}[i] + \mathsf{LCP}[i]]$. The two adjacent children u_{Pc} and $u_{Pc'}$ $(Pc \prec Pc')$ of an explicit node u_P indicate that $\mathsf{LCP}[i] = |P|$ holds, where i is the left boundary of $\mathsf{interval}(Pc')$. Hence, there exists a one-to-one correspondence between $\mathcal{P}_t$ and $\mathcal{K}'_t$, and $|\mathcal{P}_t| = |\mathcal{K}'_t|$ holds. We obtain the inequality $|\mathcal{K}_t| \leq 2|\mathcal{P}_t|$ by $|\mathcal{P}_t| = |\mathcal{K}'_t|$ and $|\mathcal{K}_t| \leq 2|\mathcal{K}'_t|$.

In Figure 3, the left figure represents a part of a suffix tree. In this example, $\mathcal{P}_3 = \{5, 6\}$, and $\mathcal{K}'_3 = \{4, 5\}$. Obviously, the two positions 5 and 6 in $\mathcal{P}_3$ correspond to the two positions 4 and 5 in $\mathcal{K}'_3$ with the adjacent children of explicit node u_{aba}, respectively.

Next, we introduce two functions LF_x and $\mathsf{dist}(i)$ to analyze $|\mathcal{P}_t|$. The first function $\mathsf{LF}_x(i)$ returns the position obtained by recursively applying LF function to i x times, i.e., $\mathsf{LF}_0(i) = i$ and $\mathsf{LF}_x(i) = \mathsf{LF}_{x-1}(\mathsf{LF}(i))$ for $x \geq 1$. The second function $\mathsf{dist}(i) \geq 0$ returns the smallest integer such that $\mathsf{LF}_{\mathsf{dist}(i)}(i)$ is the starting position of a run on L (i.e., $\mathsf{LF}_{\mathsf{dist}(i)}(i) \in \mathcal{S}_{\mathsf{start}}$). Formally, $\mathsf{dist}(i) = \min\{x \mid x \geq 0$ s.t. $\mathsf{LF}_x(i) \in \mathcal{S}_{\mathsf{start}}\}$. The following lemma holds.

▶ **Lemma 3.** $\mathsf{LF}_{\mathsf{dist}(i)}(i) \neq \mathsf{LF}_{\mathsf{dist}(j)}(j)$ *holds for two distinct integers* $i, j \in \mathcal{P}_t$*, where* $t \geq 0$ *is an integer.*

Proof. We show that $\mathsf{LCP}[i] = \mathsf{LCP}[\mathsf{LF}_{\mathsf{dist}(i)}(i)] - \mathsf{dist}(i)$ holds for an integer $i \in \{1, 2, \ldots, n\}$. Let t be an integer in $[0, \mathsf{dist}(i)-1]$. Since $\mathsf{LF}_t(i) \notin \mathcal{S}_{\mathsf{start}}$, $\mathsf{LCP}[\mathsf{LF}_t(i)] = \mathsf{LCP}[\mathsf{LF}_{t+1}(i)]-1$ holds by LF formula. The LF formula produces $\mathsf{dist}(i)$ equations $\mathsf{LCP}[\mathsf{LF}_0(i)] = \mathsf{LCP}[\mathsf{LF}_1(i)] - 1$, $\mathsf{LCP}[\mathsf{LF}_1(i)] = \mathsf{LCP}[\mathsf{LF}_2(i)]-1$, …, $\mathsf{LCP}[\mathsf{LF}_{\mathsf{dist}(i)-1}(i)] = \mathsf{LCP}[\mathsf{LF}_{\mathsf{dist}(i)}(i)]-1$. Hence, $\mathsf{LCP}[i] = \mathsf{LCP}[\mathsf{LF}_{\mathsf{dist}(i)}(i)] - \mathsf{dist}(i)$ holds by the $\mathsf{dist}(i)$ equations.

Next, we prove Lemma 3. The two integers i and j must be the same if $\mathsf{LF}_{\mathsf{dist}(i)}(i) = \mathsf{LF}_{\mathsf{dist}(j)}(j)$ because $\mathsf{dist}(i) = \mathsf{dist}(j)$ holds by three equations $\mathsf{LCP}[i] = \mathsf{LCP}[j]$, $\mathsf{LCP}[i] = \mathsf{LCP}[\mathsf{LF}_{\mathsf{dist}(i)}(i)] - \mathsf{dist}(i)$, and $\mathsf{LCP}[j] = \mathsf{LCP}[\mathsf{LF}_{\mathsf{dist}(j)}(j)] - \mathsf{dist}(j)$. The equation $i = j$ contradicts the fact that $i \neq j$. Hence, $\mathsf{LF}_{\mathsf{dist}(i)}(i) \neq \mathsf{LF}_{\mathsf{dist}(j)}(j)$ holds. ◀

The function $\mathsf{LF}_{\mathsf{dist}(i)}(i)$ maps the integers in $\mathcal{P}_t$ into distinct integers in set $\mathcal{S}_{\mathsf{start}}$ by Lemma 3. The mapping indicates that $|\mathcal{P}_t| \leq |\mathcal{S}_{\mathsf{start}}| = r$ holds for any integer t. In Figure 3, the right figure represents the mapping between $\mathcal{P}_1 = \{3, 4\}$ and $\mathcal{S}_{\mathsf{start}} = \{1, 2, 5, 6, 7\}$ on a BWT. In this example, $\mathsf{LF}_1(3) = 9 \notin \mathcal{S}_{\mathsf{start}}$, $\mathsf{LF}_2(3) = 5 \in \mathcal{S}_{\mathsf{start}}$, $\mathsf{LF}_1(4) = 10 \notin \mathcal{S}_{\mathsf{start}}$, and $\mathsf{LF}_2(4) = 6 \in \mathcal{S}_{\mathsf{start}}$. Hence, $\mathsf{LF}_{\mathsf{dist}(i)}(i)$ maps the two positions 3 and 4 in $\mathcal{P}_1$ into the two positions 5 and 6 in $\mathcal{S}_{\mathsf{start}}$, respectively, which indicates that $|\mathcal{P}_1| \leq |\mathcal{S}_{\mathsf{start}}|$ holds.

Finally, we obtain $\max\{|\mathcal{K}_0|, |\mathcal{K}_1|, \ldots, |\mathcal{K}_n|\} \leq 2r$ by $|\mathcal{K}_t| \leq 2|\mathcal{P}_t|$ and $|\mathcal{P}_t| \leq r$. Hence, the working space of our traversal algorithm is $O((r + \sigma)w)$ bits. We obtain the following theorem using $\sigma \leq r$.

▶ **Theorem 4.** *We can output* n *sets* $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_0\}$, $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_1\}$, …, $\{\mathsf{repr}(P) \mid P \in \mathcal{L}_{n-1}\}$ *in left-to-right order in* $O(n \log \log_w(n/r))$ *time and* $O(rw)$ *bits of working space by processing the RLBWT of* T*.*

4 Enumeration of characteristic substrings in $O(rw)$ bits of space

In this section, we present r-enum, which enumerates maximal repeats, minimal unique substrings, and minimal absent words using RLBWT. While the enumeration algorithm proposed by Belazzougui and Cunial [7] finds nodes corresponding to characteristic substrings by traversing the Weiner-link tree of T in a depth-first manner, r-enum adopts the breadth-first search presented in Section 3. The next theorem holds by assuming $\sigma \leq r$.

▶ **Theorem 5.** *R-enum can enumerate (i) maximal repeats, (ii) minimal unique substrings, and (iii) minimal absent words for T in $O(n \log \log_w(n/r))$, $O(n \log \log_w(n/r))$, and $O(n \log \log_w(n/r) + occ)$ time, respectively, by processing the RLBWT of T with $O(rw)$ bits of working space, where occ is the number of minimal absent words for T, and $occ = O(\sigma n)$ holds [17]. Here, r-enum outputs rich representation $\mathsf{repr}(P)$, pair $(\mathsf{interval}(P'), |P'|)$, and 3-tuple $(\mathsf{interval}(P''), |P''|, c)$ for a maximal repeat P, minimal unique substring P', and minimal absent word $P''c$, respectively, where P, P', P'' are substrings of T, and c is a character.*

We prove Theorem 5(i) for maximal repeats. R-enum leverages the following relation between the maximal repeats and nodes in the Weiner-link tree.

▶ **Lemma 6.** *A substring P of T is a maximal repeat if and only if P satisfies two conditions: (i) node u_P is explicit (i.e., $P \in \mathcal{L}_{|P|}$), and (ii) $\mathsf{rank}(\mathcal{S}_{\mathsf{start}}, b) \neq \mathsf{rank}(\mathcal{S}_{\mathsf{start}}, e)$, where $\mathsf{interval}(P) = [b, e]$.*

Proof. $|Occ(T, P)| \geq 2$ and $|Occ(T, P)| > |Occ(T, Pc)|$ hold for any character $c \in \Sigma$ if and only if node u_P is explicit. From the definition of BWT, $|Occ(T, P)| > |Occ(T, cP)|$ holds for any character $c \in \Sigma$ if and only if $L[b..e]$ contains at least two distinct characters, i.e., $\mathsf{rank}(\mathcal{S}_{\mathsf{start}}, b) \neq \mathsf{rank}(\mathcal{S}_{\mathsf{start}}, e)$ holds. Hence, Lemma 6 holds. ◀

R-enum traverses the Weiner-link tree of T, and it verifies whether each explicit node u_P represents a maximal repeat using Lemma 6, i.e., it verifies $\mathsf{rank}(\mathcal{S}_{\mathsf{start}}, b) \neq \mathsf{rank}(\mathcal{S}_{\mathsf{start}}, e)$ or $\mathsf{rank}(\mathcal{S}_{\mathsf{start}}, b) = \mathsf{rank}(\mathcal{S}_{\mathsf{start}}, e)$ using the two rank queries on $\mathcal{S}_{\mathsf{start}}$. We output its rich representation $\mathsf{repr}(P)$ if P is a maximal repeat. The rich representation $\mathsf{repr}(P)$ for u_P stores $[b, e]$, and our breadth-first traversal algorithm stores the data structure $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$ for rank queries on $\mathcal{S}_{\mathsf{start}}$. Hence, we obtain Theorem 5(i).

Similarly, r-enum can also find the nodes corresponding to minimal unique substrings and minimal absent words using their properties while traversing the Weiner-link tree. See Appendixes B and C for the proofs of Theorem 5(ii) and (iii), respectively.

5 Modified enumeration algorithm for original characteristic substrings and their occurrences

Let $\mathsf{output}(P)$ be the element representing a characteristic substring P outputted by r-enum. In this section, we slightly modify r-enum and provide three additional data structures, R_{str}, R_{occ}, and R_{eRD}, to recover the original string P and its occurrences in T from the element $\mathsf{output}(P)$. The three data structures R_{str}, R_{occ} and R_{eRD} require $O(rw)$ bits of space and support *extract*, *extract-sa*, and *extended range distinct queries*, respectively. An extract query returns string P for a given pair $(\mathsf{interval}(P), |P|)$. An extract-sa query returns all the occurrences of P in T (i.e., $\mathsf{SA}[b..e]$) for a given pair $(\mathsf{interval}(P), \mathsf{SA}[b])$, where $\mathsf{interval}(P) = [b, e]$. An extended range distinct query $\mathsf{eRD}(L, b, e, \mathsf{SA}[b])$ returns 4-tuple $(c, p_c, q_c, \mathsf{SA}[p_c])$ for each output $(c, p_c, q_c) \in \mathsf{RD}(L, b, e)$ (i.e., $\mathsf{eRD}(L, b, e, \mathsf{SA}[b]) = \{(c, p_c, q_c, \mathsf{SA}[p_c]) \mid (c, p_c, q_c) \in \mathsf{RD}(L, b, e)\}$). We omit the detailed description of the three data structures because each data structure supports the queries using the well-known properties of RLBWT. Formally, the following lemma holds.

▶ **Lemma 7.** *Let $k = |\mathsf{eRD}(L, b, e, \mathsf{SA}[b])|$. The three data structures R_{str}, R_{occ}, and R_{eRD} of $O(rw)$ bits of space can support extract, extract-sa, and extended range distinct queries in $O(|P| \log \log_w(n/r))$, $O((e - b + 1) \log \log_w(n/r))$, and $O((k + 1) \log \log_w(n/r))$ time, respectively. We can construct the three data structures in $O(n \log \log_w(n/r))$ time and $O(rw)$ bits of working space by processing the RLBWT of T.*

Table 2 Details of dataset. Table details size in megabytes (MB), string length (n), alphabet size (σ), number of runs in BWT (r), and number of maximal repeats (m) for each data.

Data name	Size [MB]	n	σ	r	m
einstein.de.txt	93	92,758,441	118	101,370	79,469
einstein.en.txt	468	467,626,544	140	290,239	352,590
world leaders	47	46,968,181	90	573,487	521,255
influenza	155	154,808,555	16	3,022,822	7,734,058
kernel	258	257,961,616	161	2,791,368	1,786,102
cere	461	461,286,644	6	11,574,641	10,006,383
coreutils	205	205,281,778	237	4,684,460	2,963,022
Escherichia Coli	113	112,689,515	16	15,044,487	12,011,071
para	429	429,265,758	6	15,636,740	13,067,128
100genome	307,705	307,705,110,945	6	36,274,924,494	52,172,752,566

Proof. See Appendix D. ◀

We modify r-enum as follows. The modified r-enum outputs pair $(\mathsf{repr}(P), \mathsf{SA}[b])$, 3-tuple $(\mathsf{interval}(P'), |P'|, \mathsf{SA}[b'])$, and 4-tuple $(\mathsf{interval}(P''), |P''|, c, \mathsf{SA}[b''])$ instead of rich representation $\mathsf{repr}(P)$, pair $(\mathsf{interval}(P'), |P'|)$, and 3-tuple $(\mathsf{interval}(P''), |P''|, c)$, respectively. Here, (i) $P, P', P''c$ are a maximal repeat, minimal unique substring, and minimal absent word, respectively, and (ii) b, b', b'' are the left boundaries of $\mathsf{interval}(P)$, $\mathsf{interval}(P')$, and $\mathsf{interval}(P''c)$, respectively. We replace each range distinct query $\mathsf{RD}(L, b, e)$ used by r-enum with the corresponding extended range distinct query $\mathsf{eRD}(L, b, e, \mathsf{SA}[b])$ to compute the sa-values $\mathsf{SA}[b]$, $\mathsf{SA}[b']$, and $\mathsf{SA}[b'']$. See Appendix E for a detailed description of the modified r-enum. Formally, the following theorem holds.

▶ **Theorem 8.** *R-enum can also output the sa-values* $\mathsf{SA}[b]$, $\mathsf{SA}[b']$, *and* $\mathsf{SA}[b'']$ *for each maximal repeat* P, *minimal unique substring* P', *and minimal absent word* $P''c$, *respectively, using an additional data structure of* $O(rw)$ *bits of space. Here,* b, b', b'' *are the left boundaries of* $\mathsf{interval}(P)$, $\mathsf{interval}(P')$, *and* $\mathsf{interval}(P''c)$, *respectively, and the modification does not increase the running time.*

Proof. See Appendix E. ◀

We compute each characteristic substring and all the occurrences of the substring in T by applying extract and extract-sa queries to the corresponding element outputted by the modified r-enum. For example, the outputted pair $(\mathsf{repr}(P), \mathsf{SA}[b])$ for a maximal repeat P contains $\mathsf{interval}(P), |P|$, and $\mathsf{SA}[b]$. Hence, we can obtain P by applying an extract query to pair $(\mathsf{interval}(P), |P|)$. Similarly, we can obtain all the occurrences of P in T by applying an extract-sa query to pair $(\mathsf{interval}(P), \mathsf{SA}[b])$. Note that we do not need to compute the occurrences of minimal absent words in T since the words do not occur in T.

6 Experiments

We demonstrate the effectiveness of our r-enum for enumerating maximal repeats on a benchmark dataset of highly repetitive strings in a comparison with the state-of-the-art enumeration algorithms of the OT [30], BBO [12] and BC [7] methods, which are reviewed in Section 1. In this experiment, all the methods output suffix-tree intervals for maximal repeats.

Table 3 Peak memory consumption of each method in mega bytes (MB).

		Memory (MB)			
Data name	Size [MB]	r-enum	BBO	BC	OT
einstein.de.txt	93	**2**	100	100	1,642
einstein.en.txt	468	**4**	488	488	8,278
world leaders	47	**5**	37	37	832
influenza	155	**18**	77	73	2,741
kernel	258	**21**	297	292	4,567
cere	461	**86**	265	224	8,166
coreutils	205	**32**	268	237	3,634
Escherichia Coli	113	109	116	**55**	1,995
para	429	**116**	262	204	7,599

The OT method enumerates maximal repeats using the BWT and enhanced suffix array of a given string, where the enhanced suffix array consists of suffix and LCP arrays. The OT method runs in $O(n)$ time and with $O(n \log n)$ bits of working space for T.

The BBO method traverses the Weiner-link tree of a given string T using Lemma 1 and a breadth-first search, and it outputs maximal repeats by processing all the nodes in the tree. We used the SDSL library [22] for an implementation of the Huffman-based wavelet tree to support range distinct queries, and we did not implement the technique of Beller et al. for storing a queue for suffix-tree intervals in $n + o(n)$ bits. Hence, our implementation of the BBO method takes the BWT of T and runs in $O(n \log \sigma)$ time and with $|WT_{\mathbf{huff}}| + O(\max\{|\mathcal{K}_0|, |\mathcal{K}_1|, \ldots, |\mathcal{K}_n|\})w)$ bits of working space, where (i) $|WT_{\mathbf{huff}}| = O(n \log \sigma)$ is the size of the Huffman-based wavelet tree, and (ii) $\mathcal{K}_0, \mathcal{K}_1, \ldots, \mathcal{K}_n$ are introduced in Section 3.3. The latter term can be bounded by $O(rw)$ bits by applying the analysis described in Section 3.3 to their enumeration algorithm.

The BC method also traverses the Weiner-link tree by Lemma 1 and a depth-first search. The method stores a data structure for range distinct queries and a stack data structure of size $O(\sigma^2 \log^2 n)$ bits for the depth-first search. We also used the SDSL library [22] for the Huffman-based wavelet tree to support range distinct queries. Hence, our implementation of the BC method runs in $O(n \log \sigma)$ time and with $|WT_{\mathbf{huff}}| + O(\sigma^2 \log^2 n)$ bits of working space.

We used a benchmark dataset of nine highly repetitive strings in the Pizza & Chili corpus downloadable from `http://pizzachili.dcc.uchile.cl`. In addition, we demonstrated the scalability of r-enum by enumerating maximal repeats on the concatenation of 100 human genomes (307 gigabytes) built from 1,000 human genomes [33]. Table 2 details our dataset.

We used memory consumption and execution time as evaluation measures for each method. Since each method takes the BWT of a string as an input and outputs suffix-tree intervals for maximal repeats, the execution time consists of two parts: (i) the preprocessing time for constructing data structures built from an input BWT, and (ii) the enumeration time after the data structures are constructed. We performed all the experiments on 48-core Intel Xeon Gold 6126 (2.60 GHz) CPU with 2 TB of memory. The source codes used in the experiments are available at `https://github.com/TNishimoto/renum`.

Table 4 Execution time of each method in seconds (s).

	Execution time (s)			
Data name	r-enum	BBO	BC	OT
einstein.de.txt	172	84	70	**13**
einstein.en.txt	856	487	387	**80**
world leaders	97	24	16	**10**
influenza	267	69	49	**30**
kernel	559	281	178	**57**
cere	985	277	186	**110**
coreutils	445	285	179	**42**
Escherichia Coli	253	68	40	**29**
para	961	262	173	**102**

Table 5 Execution time in hours and peak memory consumption in mega-bytes (MB) of r-enum on 100genome.

Size [MB]	n/r	Execution time (hours)	Memory (MB)
307,705	8.5	25	319,949

6.1 Experimental results on benchmark dataset

In the experiments on the nine highly repetitive benchmark strings, we ran each method and with a single thread. Table 3 shows the peak memory consumption of each method. The BBO and BC methods consumed approximately 1.0-2.0 and 0.9-1.5 bytes per byte of input, respectively. The memory usage of the BC method was no more than that of the BBO method on each of the nine strings. The OT method consumed approximately 18 bytes per character, which was larger than the memory usage of the BBO method. Our r-enum consumed approximately 7-23 bytes per run in BWT. The memory usage of r-enum was the smallest on most of the nine strings except for the file Escherichia Coli. In the best case, the memory usage of r-enum was approximately 122 times less than that of the BC method on einstein.en.txt because the ratio $n/r \approx 1611$ and alphabet size $\sigma = 140$ were large.

Table 4 shows the execution time for each method. The OT method was the fastest among all the methods, and it took approximately 13-110 seconds. The execution times of the BBO and BC methods were competitive, and each execution of them finished within 487 seconds on the nine strings. R-enum was finished in 985 seconds even for the string data (cere) with the longest enumeration time. These results show that r-enum can space-efficiently enumerate maximal substrings in a practical amount of time, although r-enum was approximately 10 times slower than the fastest method (i.e., OT method).

6.2 Experimental results on 100 human genomes

We tested r-enum on 100genome, which is a 307-gigabyte string of 100 human genomes. For this experiment, we implemented computations of Weiner-links from nodes with the same depth in parallel. It is easy to achieve the parallelization, because r-rnum uses Lemma 1 to compute Weiner-links and we can apply the Lemma to each node independently. We ran the parallelized r-enum with 48 threads.

Table 5 shows the total execution time and peak memory consumption of r-enum on 100genome. R-enum consumed approximately 25 hours and 319 gigabytes of memory for enumeration. The result demonstrates the scalability and practicality of r-enum for

enumerating maximal repeats on a huge string existing in the real-world. The execution time was 0.6 seconds per 10^6 characters. The memory consumption was almost the same as the size of the input file. It was relatively large compared with the memory consumption of r-enum on the nine benchmark strings because the ratio of $n/r \approx 8.4$ for 100genome was small.

7 Conclusion

We presented r-enum, which can enumerate maximal repeats, minimal unique substrings, and minimal absent words working in $O(rw)$ bits of working space. Experiments using a benchmark dataset of highly repetitive strings showed that r-enum is more space-efficient than the previous methods if $n/r > 27.5$. In addition, we demonstrated the applicability of r-enum to a huge string by performing experiments on a 300-gigabyte string of 100 human genomes. Our method was not so effective for the 300-gigabyte string because the string is weakly compressible. On the other hand, it is oblivious that r-enum is more space-efficient than the previous methods for huge strings such that n/r is sufficiently large. Our future work is to reduce the running time and working space of r-enum.

We showed that breadth-first traversal of a Weiner-link tree can be performed in $O(rw)$ bits of working space. The previous breadth-first traversal algorithm by the BBO method requires $|RD| + O(n)$ bits of working space, where $|RD|$ is the size of a data structure supporting range distinct queries on the BWT of T. In addition to enumerations of characteristic substrings, traversal algorithms of a Weiner-link tree can be used for constructing three data structures: (i) LCP array, (ii) suffix tree topology, and (iii) the merged BWT of two strings [32]. Constructing these data structures in $O(rw)$ bits of working space by modifying r-enum would be an interesting future work.

We also showed that the data structure RD' for our traversal algorithm supported two operations: (i) a range distinct query in $O(\log \log_w(n/r))$ time per output element and (ii) computations of LF function in $O(\log \log_w(n/r))$ time. The result can replace the pair $(|RD|, d)$ presented in Table 1 with pair $(O(rw), O(\log \log_w(n/r)))$, which improves the BBO and BC methods so that they work in $O(rw + n)$ and $O(rw + \sigma^2 \log^2 n)$ bits of working space, respectively. We think that the working space of the BC method with RD' is practically smaller than that of r-enum, because $\sigma^2 \log^2 n$ is practically smaller than rw in many cases. This insight indicates that we can enumerate characteristic substrings with a lower memory consumption of $O(rw)$ bits by combining r-enum with the BC method even for a large alphabet size. Thus, the following method could improve the space efficiency for enumerations: executing the BC method with RD' for a small alphabet (i.e., $\sigma < \sqrt{r/\log n}$) and executing r-enum for a large alphabet.

References

1 Paniz Abedin, M. Oguzhan Külekci, and Shama V. Thankachan. A survey on shortest unique substring queries. *Algorithms*, 13:224, 2020.

2 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53–86, 2004.

3 Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, pages 85–96, 1985.

4 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.

5 Carl Barton, Alice Héliou, Laurent Mouchard, and Solon P. Pissis. Linear-time computation of minimal absent words using suffix array. *BMC Bioinformatics*, 15:388, 2014.

6 Verónica Becher, Alejandro Deymonnaz, and Pablo Ariel Heiber. Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome. *Bioinformatics*, 25:1746–1753, 2009.

7 Djamal Belazzougui and Fabio Cunial. Space-efficient detection of unusual words. In *Proceedings of SPIRE*, pages 222–233, 2015.

8 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Proceedings of CPM*, pages 26–39, 2015.

9 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Transactions on Algorithms*, 16:17:1–17:54, 2020.

10 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11:31:1–31:21, 2015.

11 Djamal Belazzougui, Gonzalo Navarro, and Daniel Valenzuela. Improved compressed indexes for full-text document retrieval. *Journal of Discrete Algorithms*, 18:3–13, 2013.

12 Timo Beller, Katharina Berger, and Enno Ohlebusch. Space-efficient computation of maximal and supermaximal repeats in genome sequences. In *Proceedings of SPIRE*, pages 99–110, 2012.

13 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *Technical report*, 1994.

14 Panagiotis Charalampopoulos, Maxime Crochemore, Gabriele Fici, Robert Mercas, and Solon P. Pissis. Alignment-free sequence comparison using absent words. *Information and Computation*, 262:57–68, 2018.

15 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

16 Maxime Crochemore, Gabriele Fici, Robert Mercas, and Solon P. Pissis. Linear-time sequence comparison using minimal absent words & applications. In *Proceedings of LATIN*, pages 334–346, 2016.

17 Maxime Crochemore, Filippo Mignosi, and Antonio Restivo. Automata and forbidden words. *Information Processing Letters*, 67:111–117, 1998.

18 Maxime Crochemore, Filippo Mignosi, Antonio Restivo, and Sergio Salemi. Data compression using antidictionaries. *Proceedings of the IEEE*, 88:1756–1768, 2000.

19 Maxime Crochemore and Gonzalo Navarro. Improved antidictionary based compression. In *Proceedings of SCCC*, pages 7–13, 2002.

20 Isamu Furuya, Takuya Takagi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Takuya Kida. MR-RePair: Grammar compression based on maximal repeats. In *Proceedings of DCC*, pages 508–517, 2019.

21 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM*, 67:2:1–2:54, 2020.

22 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Proceedings of SEA*, pages 326–337, 2014.

23 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

24 Bernhard Haubold, Nora Pierstorff, Friedrich Möller, and Thomas Wiehe. Genome comparison without alignment using shortest unique substrings. *BMC Bioinformatics*, 6:123, 2005.

25 Dominik Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proceedings of SODA*, pages 1344–1357, 2019.

26 Dominik Kempa and Tomasz Kociumaka. Resolution of the burrows-wheeler transform conjecture. In *Proceedings of FOCS*, pages 1002–1013, 2020.

27 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22:935–948, 1993.

28 Tomonari Masada, Atsuhiro Takasu, Yuichiro Shibata, and Kiyoshi Oguri. Clustering documents with maximal substrings. In *Proceedings of ICEIS*, pages 19–34, 2011.

29 Tatsuya Ohno, Kensuke Sakai, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online RLBWT and its application to LZ77 parsing. *Journal of Discrete Algorithms*, 52-53:18–28, 2018.

30 Daisuke Okanohara and Jun'ichi Tsujii. Text categorization with all substring features. In *Proceedings of SDM*, pages 838–846, 2009.

31 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80:1986–2011, 2018.

32 Nicola Prezza and Giovanna Rosone. Space-efficient construction of compressed suffix trees. *Theoretical Computer Science*, 852:138–156, 2021.

33 The 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491:56–65, 2012.

A Algorithm for constructing data structures in Section 3.1

Recall that our traversal algorithm uses six data structures: (i) the RLBWT of T, (ii) $\mathsf{repr}(\varepsilon)$, (iii) D_{LF}, (iv) $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$, (v) $R_{\mathsf{RD}}(L')$, and (vi) an empty array X of size σ. Let δ be the permutation of $[1, r]$ satisfying either of two conditions for two distinct integers $i, j \in \{1, 2, \ldots, r\}$: (i) $L[\ell(\delta[i])] \prec L[\ell(\delta[j])]$ or (ii) $L[\ell(\delta[i])] = L[\ell(\delta[j])]$ and $i < j$. Then $D_{\mathsf{LF}}[1] = 1$ and $D_{\mathsf{LF}}[i] = D_{\mathsf{LF}}[i-1] + |L[\ell(\delta[i-1])..\ell(\delta[i-1]+1)-1]|$ hold by LF formula. We construct the permutation δ in $O(n)$ time and $O(rw)$ bits of working space using LSD radix sort. After that we construct the array D_{LF} in $O(r)$ time using δ.

Next, we construct $\mathcal{S}_{\mathsf{start}}$ and L' in $O(r)$ time by processing the RLBWT of T. After that, we construct $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$ in $O(|\mathcal{S}_{\mathsf{start}}| \log\log_w(n/r))$ time and $O(|\mathcal{S}_{\mathsf{start}}|w)$ bits of working space by processing $\mathcal{S}_{\mathsf{start}}$ [21, Appendix A.1]. Similarly, we construct $R_{\mathsf{RD}}(L')$ in $O(r)$ time and $O(r \log \sigma)$ bits of working space [9, Lemma 3.17].

Finally, $\mathsf{repr}(\varepsilon)$ consists of 3-tuple $([1, n], \mathsf{RD}(L, 1, n), 0)$. We compute $\mathsf{RD}(L, 1, n)$ by Lemma 2. Hence the construction time is $O(n \log\log_w(n/r))$ in total, and the working space is $O(rw)$ bits.

B Proof of Theorem 5 (ii)

For simplicity, we focus on minimal unique substrings with a length of at least 2. Every minimal unique substring with a length of at least 2 is a substring cPc' of T, where c, c' are characters, and P is a string with a length of at least 0. R-enum uses an array Right_P of size σ for detecting a minimal unique substring cPc'. $\mathsf{Right}_P[c]$ stores $|Occ(T, Pc)|$ for each $c \in \Sigma$. A substring cPc' of T is a minimal unique substring if and only if cPc' satisfies three conditions: (i) $|Occ(T, cPc')| = 1$, (ii) $|Occ(T, cP)| \geq 2$, and (iii) $\mathsf{Right}_P[c] \geq 2$ hold. We can verify the three conditions using $\mathsf{repr}(P)$ and Lemma 1, and hence, the following lemma holds.

▶ **Lemma 9.** *Let $\mathcal{M}(P)$ be the set of minimal unique substrings such that the form of each minimal unique substring is cPc', where c, c' are characters, and P is a given string. We can compute the output by r-enum for the set $\mathcal{M}(P)$ (i.e., $\{(\mathsf{interval}(cPc'), |cPc'|) \mid cPc' \in \mathcal{M}(P)\}$) using (i) $\mathsf{repr}(R)$, (ii) the data structures presented in Section 3.1, and (iii) an empty array X' of size σ. The running time and working space are $O(h \log\log_w(n/r))$ and $O((\sigma + h')w)$ bits, respectively. Here, $h = \sum_{Pc' \in \mathsf{children}(P)} |\mathsf{WLink}(Pc')|$, and $h' = \sum_{cP \in \mathsf{WLink}(P) \cap \mathcal{L}_{|P|+1}} |\mathsf{children}(cP)|$.*

Proof. $\mathsf{Right}_P[c]$ is stored in $\mathsf{repr}(P)$ for $c \in \Sigma$, and the pair $(\mathsf{interval}(cPc'), |cPc'|)$ is stored in $\mathsf{repr}(cP)$. We compute the output by four steps. (i) We convert X' into Right_P by processing $\mathsf{repr}(P)$. (ii) We compute the rich representations for all the strings in $\mathsf{WLink}(P)$ by Lemma 1.

(iii) We process the rich representation for each string $cP \in \mathsf{WLink}(P)$ and output pair $(\mathsf{interval}(cPc'), |cPc'|)$ for each child cPc' if $|\mathsf{interval}(cPc')| = 1$ and $\mathsf{Right}_P[c'] \geq 2$. (iv) We recover X' from Right_P. Hence, Lemma 9 holds. ◀

Node u_P is always explicit (i.e., u_P is a node of the Weiner-link tree) if a substring cPc' is a minimal unique substring of T because $|Occ(T, P)| > |Occ(T, Pc')|$ holds from the definition of the minimal unique substring. Hence, we can compute $(\mathsf{interval}(cPc'), |cPc'|)$ for each minimal unique substring cPc' in T by applying Lemma 9 to all the nodes in the Weiner-link tree.

R-enum prepares an empty array X' of size σ and computes the output for set $\mathcal{M}(P)$ by applying Lemma 9 to each node u_P. The running time and working space are $O((\sum_{t=0}^{n} H_t) \log\log_w(n/r)) = O(n \log\log_w(n/r))$ and $O((r + \sigma + \max\{|\mathcal{K}_0|, |\mathcal{K}_1|, \ldots, |\mathcal{K}_n|\})w)) = O(rw)$ bits, respectively. Here, H_t and $\mathcal{K}_t$ are the terms introduced in Section 3.3.

C Proof of Theorem 5 (iii)

For simplicity, we focus on minimal absent words with a length of at least 2. Minimal absent words have similar properties to the properties of the minimal unique substrings explained in the proof of Theorem 5(ii), i.e., the characteristic substrings have two properties: (i) a string cPc' is a minimal absent word for T if and only if cPc' satisfies three conditions: (1) $|Occ(T, cPc')| = 0$, (2) $|Occ(T, cP)| \geq 1$, and (3) $\mathsf{Right}_P[c'] \geq 1$ hold, and (ii) node u_P is always explicit if a substring cPc' is a minimal absent word for T because $|Occ(T, P)| > |Occ(T, Pc')|$ holds from the definition of the minimal absent word. We obtain the following lemma by modifying the proof of Lemma 9.

▶ **Lemma 10.** *Let $\mathcal{W}(P)$ be the set of minimal absent words such that the form of each minimal unique substring is cPc', where c, c' are characters, and P is a given string. We can compute the output by r-enum for the set $\mathcal{W}(P)$ (i.e., $\{(\mathsf{interval}(cP), |cP|, c') \mid cPc' \in \mathcal{W}(P)\}$) using (i) $\mathsf{repr}(R)$, (ii) the data structures presented in Section 3.1, and (iii) an empty array X' of size σ. The running time and working space are $O(h \log\log_w(n/r) + |\mathcal{W}(P)|)$ and $O((\sigma + h')w)$ bits, respectively.*

We can compute $(\mathsf{interval}(cP), |cP|, c')$ for each minimal absent word cPc' for T by applying Lemma 10 to all the nodes in the Weiner-link tree. R-enum prepares an empty array X' of size σ and computes the output for set $\mathcal{W}(P)$ by applying Lemma 10 to each node u_P. The running time and working space are $O(n \log\log_w(n/r) + occ)$ and $O(rw)$ bits, respectively.

D Proof of Lemma 7

Proof for data structure R_{str}. Data structure R_{str} consists of (i) the RLBWT of T, (ii) π, (iii) D_{LF}, (iv) $R_{\mathsf{rank}}(\mathcal{S}'_{\mathsf{start}})$, and (v) $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$. Here, (i) π is the permutation on $\{1, 2, \ldots, r\}$ satisfying $\mathsf{LF}(\ell(\pi[1])) < \mathsf{LF}(\ell(\pi[2])) < \cdots < \mathsf{LF}(\ell(\pi[r]))$, (ii) D_{LF} is the array introduced in Section 3, and (iii) $\mathcal{S}'_{\mathsf{start}} = \{\mathsf{LF}(\ell(\pi[1])), \mathsf{LF}(\ell(\pi[2])), \ldots, \mathsf{LF}(\ell(\pi[r]))\}$. $\pi[i] = \delta[i]$ holds for any integer $i \in \{1, 2, \ldots, r\}$. $\mathcal{S}'_{\mathsf{start}}$ and D_{LF} can be constructed in $O(r)$ time after the permutation π is constructed. We already showed that δ could be constructed in $O(n)$ time by processing the RLBWT of T, which was explained in Appendix A. Hence, we can construct R_{str} in $O(n \log\log_w(n/r))$ time and $O(rw)$ bits of working space.

We introduce the inverse function LF^{-1} of LF function (i.e., $\mathsf{LF}^{-1}(\mathsf{LF}(i)) = i$ holds for $i \in \{1, 2, \ldots, n\}$) to solve the extract query. Recall that $\mathsf{LF}(i) = D_{\mathsf{LF}}[x] + (i - \ell(x))$ holds, which is shown in Section 3.1, where $x = \mathsf{rank}(\mathcal{S}_{\mathsf{start}}, i)$. Similarly, $\mathsf{LF}^{-1}(i) = \ell(\pi[y]) + (i - D_{\mathsf{LF}}[\pi[y]])$ holds by the LF formula for any integer $i \in \{1, 2, \ldots, n\}$, where $y = \mathsf{rank}(\mathcal{S}'_{\mathsf{start}}, i)$. Hence, we can compute $\mathsf{LF}^{-1}(i)$ in $O(\log\log_w(n/r))$ time using the data structure R_{str}.

Let LF_x^{-1} be the function that returns the position obtained by recursively applying the inverse LF function to i x times (i.e., $\mathsf{LF}_0^{-1}(i) = i$, and $\mathsf{LF}_x^{-1}(i) = \mathsf{LF}^{-1}(\mathsf{LF}_{x-1}^{-1}(i))$). Then, $T[\mathsf{SA}[i]..\mathsf{SA}[i]+d-1] = L[\mathsf{LF}_1^{-1}(i)], L[\mathsf{LF}_2^{-1}(i)], \ldots, L[\mathsf{LF}_d^{-1}(i)]$ holds for any integer $d \geq 1$ because $\mathsf{SA}[\mathsf{LF}^{-1}(i)] = \mathsf{SA}[i]+1$ holds unless $\mathsf{SA}[i] = n$. R_{str} can support random access to the BWT L in $O(\log\log_w(n/r))$ time using a rank query on set $\mathcal{S}_{\mathsf{start}}$. Hence, we can compute $T[\mathsf{SA}[i]..\mathsf{SA}[i]+d-1]$ in $O(d\log\log_w(n/r))$ time using R_{str} for two given integers i and d.

We explain an algorithm solving an extract query for a given rich representation $\mathsf{repr}(P)$. Let $\mathsf{interval}(P) = [b, e]$. Then, $\mathsf{SA}[b]$ stores the index of a suffix of T having P as a prefix, i.e., $T[\mathsf{SA}[b]..\mathsf{SA}[b]+|P|-1] = P$ holds. We recover the prefix P from $\mathsf{SA}[b]$ using R_{str}. Hence R_{str} supports the extract query in $O(|P|\log\log_w(n/r))$ time. ◀

Proof for data structure R_{occ}. Next, we leverage a function ϕ to solve the extract-sa query. The function $\phi(\mathsf{SA}[i])$ returns $\mathsf{SA}[i+1]$ for $i \in \{1, 2, \ldots, n-1\}$. R_ϕ is a data structure of $O(rw)$ bits proposed by Gagie et al. [21], and we can compute ϕ function in $O(\log\log_w(n/r))$ time by R_ϕ. The data structure can be constructed in $O(n\log\log_w(n/r))$ time and $O(rw)$ bits of working space by processing the RLBWT of T [21]. The second data structure R_{occ} consists of R_ϕ, and we solve the extract-sa query by recursively applying the function ϕ to $\mathsf{SA}[b]$ $(e-b)$ times. Hence R_{occ} can support the extract-sa query in $O((e-b+1)\log\log_w(n/r))$ time. ◀

Proof for data structure R_{eRD}. Let D_{SA} be an array of size r such that $D_{\mathsf{SA}}[i]$ stores the sa-value at the starting position of the i-th run in BWT L for $i \in \{1, 2, \ldots, r\}$, i.e., $D_{\mathsf{SA}}[i] = \mathsf{SA}[\ell(i)]$. Let $(c, p_c, q_c, \mathsf{SA}[p_c])$ be a 4-tuple outputted by query $\mathsf{eRD}(L, b, e, \mathsf{SA}[b])$ and x be the index of the run containing character $L[p_c]$ (i.e., $x = \mathsf{rank}(\mathcal{S}_{\mathsf{start}}, p_c)$). Then $\mathsf{SA}[p_c] = D_{\mathsf{SA}}[x]$ if $p_c = \ell(x)$; otherwise $\mathsf{SA}[p_c] = \mathsf{SA}[b]$ holds because p_c is equal to $\ell(x)$ or b under Lemma 2. The relationship among $\mathsf{SA}[p_c]$, $D_{\mathsf{SA}}[x]$, and $\mathsf{SA}[b]$ is called the *toehold lemma* (e.g., [31, 21]). The toehold lemma indicates that we can compute $\mathsf{SA}[p_c]$ in $O(\log\log_w(n/r))$ time for each output $(c, p_c, q_c) \in \mathsf{RD}(L, b, e)$ using (i) the array D_{SA}, (ii) data structure $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$, and (iii) the RLBWT of T if we know the first sa-value $\mathsf{SA}[b]$ in $[b, e]$.

Next, we explain R_{eRD}. R_{eRD} consists of $R_{\mathsf{RD}}(L')$, D_{SA}, $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$, and the RLBWT of T. We already showed that we could construct $R_{\mathsf{RD}}(L')$ and $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$ in $O(n\log\log_w(n/r))$ time by processing the RLBWT of T. We construct the array D_{SA} by computing all the sa-values in $\mathsf{SA}[1..n]$ in left-to-right order using data structure R_ϕ. R_ϕ can be constructed in $O(n\log\log_w(n/r))$ time by processing the RLBWT of T. Hence, the construction time for R_{eRD} is $O(n\log\log_w(n/r))$ time in total, and the working space is $O(rw)$ bits.

We solve extended range distinct query $\mathsf{eRD}(L, b, e, \mathsf{SA}[b])$ using the toehold lemma after solving the corresponding range distinct query $\mathsf{RD}(L, b, e)$ using $R_{\mathsf{RD}}(L')$, $R_{\mathsf{rank}}(\mathcal{S}_{\mathsf{start}})$, and the RLBWT of T. The running time is $O((k+1)\log\log_w(n/r))$, where $k = |\mathsf{eRD}(L, b, e, \mathsf{SA}[b])|$. ◀

E Proof of Theorem 8

We extend Lemma 1. Let $\mathsf{eRepr}(P)$ for P be a 4-tuple $(\mathsf{interval}(P), \{(c_1, b_1, e_1, \mathsf{SA}[b_1]), (c_2, b_2, e_2, \mathsf{SA}[b_2]), \ldots, (c_k, b_k, e_k, \mathsf{SA}[b_k])\}, |P|, \mathsf{SA}[b])$. Here, (i) b is the left boundary of $\mathsf{interval}(P)$, (ii) $Pc_1, Pc_2, \ldots, Pc_k$ are strings represented by the children of node u_P, and (iii) $[b_i, e_i] = \mathsf{interval}(Pc_i)$ for $i \in [1, k]$. We call $\mathsf{eRepr}(P)$ an *extended rich representation*.

Let $\mathsf{eRepr}(cP) = (\mathsf{interval}(cP), \{(c'_1, b'_1, e'_1, \mathsf{SA}[b'_1]), (c'_2, b'_2, e'_2, \mathsf{SA}[b'_2]), \ldots, (c'_{k'}, b'_{k'}, e'_{k'}, \mathsf{SA}[b'_{k'}])\}, |cP|, \mathsf{SA}[b'])$ for a character c. Let x_i be an integer such that $\mathsf{LF}(x_i) = b'_i$ for a 4-tuple $(c'_i, b'_i, e'_i, \mathsf{SA}[b'_i])$ in $\mathsf{eRepr}(P)$, and let $y(i)$ be an integer such that $x_i \in [b_{y(i)}, e_{y(i)}]$

holds. Then, there exists a tuple $(\hat{c}, p_{\hat{c}}, q_{\hat{c}}, \mathsf{SA}[p_{\hat{c}}]) \in \mathsf{eRD}(L, b_{y(i)}, e_{y(i)}, \mathsf{SA}[b_{y(i)}])$ such that $p_{\hat{c}} = x_i$ holds. $\mathsf{SA}[b'_i] = \mathsf{SA}[p_{\hat{c}}] - 1$ holds by LF function. Next, let j be an integer such that b'_j is the smallest in set $\{b'_1, b'_2, \ldots, b'_{k'}\}$. Then $\mathsf{SA}[b'] = \mathsf{SA}[b'_j]$ holds because b'_j is equal to the left boundary of $\mathsf{interval}(cP)$. Hence, Lemma 1 can output extended rich representations instead of rich representations by replacing the range distinct queries used by the algorithm of Lemma 1 with the corresponding extended range distinct queries. Formally, the following lemma holds.

▶ **Lemma 11.** *We can compute set* $\{\mathsf{eRepr}(cP) \mid cP \in \mathsf{WLink}(P)\}$ *for a given rich representation* $\mathsf{eRepr}(P)$ *in* $O(h \log \log_w(n/r))$ *time using* R_{eRD} *and the six data structures introduced in Section 3.1, where* $h = \sum_{Pc' \in \mathsf{children}(P)} |\mathsf{WLink}(Pc')|$.

Next, we modify our traversal algorithm for the Weiner-link tree of T. The modified traversal algorithm uses Lemma 11 instead of Lemma 1. Hence, we obtain the following lemma.

▶ **Lemma 12.** *We can output* n *sets* $\{\mathsf{eRepr}(P) \mid P \in \mathcal{L}_0\}$, $\{\mathsf{eRepr}(P) \mid P \in \mathcal{L}_1\}$, …, $\{\mathsf{eRepr}(P) \mid P \in \mathcal{L}_{n-1}\}$ *in left-to-right order in* $O(n \log \log_w(n/r))$ *time and* $O(rw)$ *bits of working space by processing the RLBWT of* T.

Finally, we prove Theorem 8 using the modified traversal algorithm, i.e., Lemma 12.

Proof for maximal repeats. The node u_P representing a maximal repeat P is explicit, and hence, $\mathsf{eRepr}(P)$ is outputted by the modified traversal algorithm. R-enum uses the modified traversal algorithm instead of our original traversal algorithm. Hence, r-enum can output the extended rich representations for all the maximal repeats in T without increasing the running time. ◀

Proof for minimal unique substrings. Let cPc' be a minimal unique substring of T such that its occurrence position is $\mathsf{SA}[b']$. Recall that r-enum computes $\mathsf{repr}(cP)$ by applying Lemma 1 to $\mathsf{repr}(P)$ and outputs $(\mathsf{interval}(cPc'), |cPc'|)$ by processing $\mathsf{repr}(cP)$. The extended rich representation $\mathsf{eRepr}(cP)$, which corresponds to $\mathsf{repr}(cP)$, contains $\mathsf{interval}(cPc'), |cPc'|$, and $\mathsf{SA}[b']$. The modified r-enum (i) traverses the Weiner-link tree by the modified traversal algorithm, (ii) computes $\mathsf{eRepr}(cP)$ by applying Lemma 11 to $\mathsf{eRepr}(P)$, and (iii) outputs $(\mathsf{interval}(cPc'), |cPc'|, \mathsf{SA}[b'])$ by processing $\mathsf{eRepr}(cP)$. The running time is $O(n \log \log_w(n/r))$ time in total. ◀

Proof for minimal absent words. Let cPc' be a minimal absent word for T, and let b'' be the left boundary of $\mathsf{interval}(cP)$. $\mathsf{eRepr}(cP)$ contains the sa-value $\mathsf{SA}[b'']$, and hence, we can compute $(\mathsf{interval}(cP), |cP|, c', \mathsf{SA}[b'])$ for the minimal absent word cPc' by modifying the algorithm used by the modified r-enum for minimal unique substrings.

The modified r-enum (i) traverses the Weiner-link tree by the modified traversal algorithm, (ii) computes $\mathsf{eRepr}(cP)$ by applying Lemma 11 to $\mathsf{eRepr}(P)$, and (iii) outputs $(\mathsf{interval}(cP), |cP|, c', \mathsf{SA}[b'])$ by processing $\mathsf{eRepr}(cP)$. The running time is $O(n \log \log_w(n/r) + occ)$ time in total. ◀

F Omitted table

Table 6 Execution time of each method. Execution time is separately presented as enumeration and preprocessing times in seconds (s).

	Preprocessing time [s]				Enumeration time [s]			
Data name	r-enum	BBO	BC	OT	r-enum	BBO	BC	OT
einstein.de.txt	1	1	1	7	171	83	69	**6**
einstein.en.txt	3	3	3	50	853	484	384	**30**
world leaders	1	1	1	7	96	23	15	**3**
influenza	2	1	1	17	265	68	48	**13**
kernel	2	2	2	41	557	279	176	**16**
cere	5	3	3	78	980	274	183	**32**
coreutils	2	2	2	29	443	283	177	**13**
Escherichia Coli	5	1	1	18	248	67	39	**11**
para	6	3	3	71	955	259	170	**31**

A Linear Time Algorithm for Constructing Hierarchical Overlap Graphs

Sangsoo Park
Samsung Electronics, Seoul, Korea

Sung Gwan Park
Samsung Electronics, Seoul, Korea

Bastien Cazaux
LIRMM, Université Montpellier, CNRS, Montpellier, France

Kunsoo Park
Seoul National University, Seoul, Korea

Eric Rivals
LIRMM, Université Montpellier, CNRS, Montpellier, France

Abstract

The hierarchical overlap graph (HOG) is a graph that encodes overlaps from a given set P of n strings, as the overlap graph does. A best known algorithm constructs HOG in $O(||P|| \log n)$ time and $O(||P||)$ space, where $||P||$ is the sum of lengths of the strings in P. In this paper we present a new algorithm to construct HOG in $O(||P||)$ time and space. Hence, the construction time and space of HOG are better than those of the overlap graph, which are $O(||P|| + n^2)$.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases overlap graph, hierarchical overlap graph, shortest superstring problem, border array

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.22

Funding S. Park, S.G. Park and K. Park were supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2018-0-00551, Framework of Practical Algorithms for NP-hard Graph Problems). B. Cazaux and E. Rivals acknowledge the funding from Labex NumeV (GEM flagship project, ANR 2011-LABX-076), and from the Marie Skłodowska-Curie Innovative Training Networks ALPACA (grant 956229).

1 Introduction

For a set of strings, a *superstring* of the set is a string that has all strings in the set as substrings. The *shortest superstring* problem is to find a shortest superstring of a set of strings. This problem is known to play an important role in *DNA assembly*, which is the problem to restore the entire genome from short sequencing reads. Despite its importance, the shortest superstring problem is known to be NP-hard [6]. As a result, extensive research has been done to find good approximation algorithms for the shortest superstring problem [2, 18, 11, 13, 19, 20].

The shortest superstring problem is reduced to finding a shortest hamiltonian path in a graph that encodes overlaps between the strings [2, 12, 16], which is the *distance graph* or equivalent *overlap graph*. The overlap graph [15] of a set of strings is a graph in which each string constitutes a node and an edge connecting two nodes shows the longest overlap between them. Many approaches for approximating the shortest superstring problem focus on the overlap graph, and try to find good approximations of its hamiltonian path [11, 13].

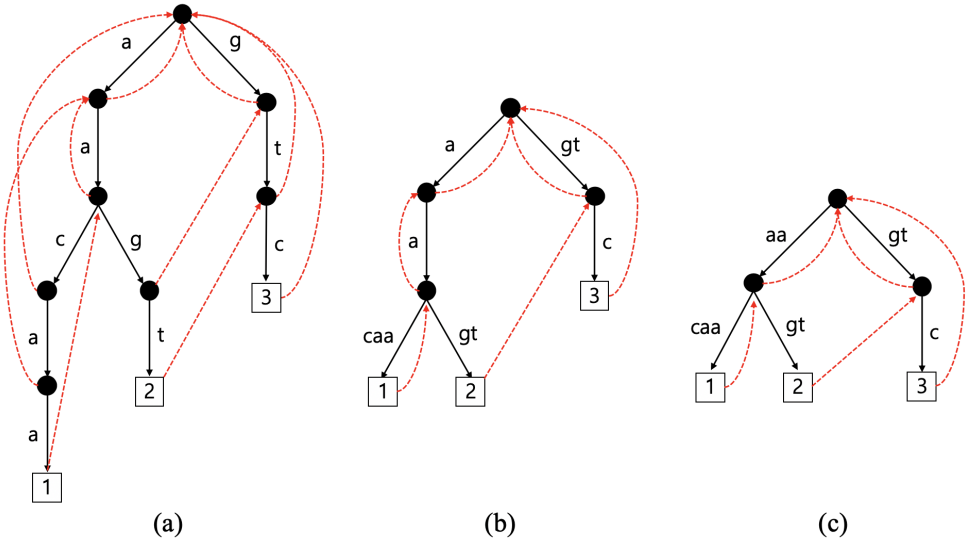


Figure 1 Data structures built with $P = \{aacaa, aagt, gtc\}$. (a) Aho–Corasick trie. (b) Extended Hierarchical Overlap Graph. (c) Hierarchical Overlap Graph.

Given a set of strings $P = \{s_1, s_2, ..., s_n\}$, computing the overlap graph of P is equivalent to solving the *all-pair suffix-prefix problem*, which is to find the longest overlap for every pair of strings in P. The best theoretical bound for this problem is $O(||P|| + n^2)$ [8], where $||P||$ is the sum of lengths of the strings in P. Since the input size of the problem is $O(||P||)$ and the output size is $O(n^2)$, this bound is optimal. There has also been extensive research on the all-pair suffix-prefix problem in the practical point of view [7, 10, 17] because it is the first step in DNA assembly.

Recently, Cazaux and Rivals [4, 5] proposed a new graph which stores the overlap information, called the *hierarchical overlap graph* (HOG). HOG is a graph with two types of edges (which will be defined in Section 2) in which a node represents either a string or the longest overlap between a pair of strings. The *extended hierarchical overlap graph* (EHOG) is also a graph with two types of edges in which a node represents either a string or an overlap between a pair of strings (which may be not the longest one). For example, Figure 1 shows EHOG and HOG built with $P = \{aacaa, aagt, gtc\}$. Even though HOG and EHOG may be the same for some input instances, there is a series of instances where the ratio of EHOG size over the HOG size tends to infinity with respect to the number of nodes. Therefore, HOG has an advantage over EHOG in both practical and theoretical points of view.

HOG also has a couple of advantages compared to the overlap graph [5]. First, HOG uses only $O(||P||)$ space, while the overlap graph needs $O(||P|| + n^2)$ space in total. For input instances with many short strings, HOG uses a considerably smaller amount of space than the overlap graph. Second, HOG contains the relationship between the overlaps themselves, since the overlaps appear as nodes in HOG. In contrast, the overlap graph stores only the lengths of the longest overlaps, and thus we cannot find the relationship between two overlaps easily. Therefore, HOG stores more information than the overlap graph, while using less space.

There have been many works to compute HOG and EHOG efficiently. Computing the EHOG from P costs $O(||P||)$ time, which is optimal [3]. For computing the HOG, Cazaux and Rivals proposed an $O(||P|| + n^2)$ time algorithm using $O(||P|| + n \times \min(n, \max\{|s| : s \in P\}))$ space [5]. Recently, Park et al. [14] gave an $O(||P|| \log n)$ time algorithm using $O(||P||)$ space by using the segment tree data structure.

In this paper we present a new algorithm to compute HOG, which uses $O(||P||)$ time and space, which are both optimal. The algorithm is based on the Aho–Corasick trie [1] and the border array [9]. Therefore, the construction time and space of HOG are better than those of the overlap graph, which are $O(||P|| + n^2)$, and this fact may lead to many applications of HOG. For example, consider the problem of finding *optimal cycle cover* in the overlap graph built with a set $P = \{s_1, s_2, ..., s_n\}$ of strings. Typically this problem needs to be solved in finding good approximations of shortest superstrings. A greedy algorithm to solve the optimal cycle cover problem on the overlap graph was given in [2], which takes $O(||P|| + n^2)$ time. Recently, Cazaux and Rivals proposed an $O(||P||)$ time algorithm to solve the optimal cycle cover problem given the HOG or EHOG of P [4]. By using our result in this paper, the optimal cycle cover problem can be solved in $O(||P||)$ time and space by using HOG instead of the overlap graph.

The rest of the paper is organized as follows. In Section 2 we give preliminary information for HOG and formalize the problem. In Section 3 we present an $O(||P||)$ time and space algorithm for computing HOG. In Section 4 we conclude and discuss a future work.

2 Preliminaries

2.1 Basic notation

We consider strings over a constant-size alphabet Σ. The length of a string s is denoted by $|s|$. Given two integers $1 \leq l \leq r \leq |s|$, the substring of s which starts from l and ends at r is denoted by $s[l..r]$. Note that $s[l..r]$ is a prefix of s when $l = 1$, and a suffix of s when $r = |s|$. If a prefix (suffix) of s is different from s, we call it a proper prefix (suffix) of s. Given two strings s and t, an *overlap* from s to t is a string which is both a proper suffix of s and a proper prefix of t. Given a set $P = \{s_1, s_2, ..., s_n\}$ of strings, the sum of $|s_i|$'s is denoted by $||P||$.

2.2 Hierarchical Overlap Graph

We define *hierarchical overlap graph* and *extended hierarchical overlap graph* as in [5].

▶ **Definition 1** (Hierarchical Overlap Graph). Given a set $P = \{s_1, s_2, \ldots, s_n\}$, we define $Ov(P)$ as the set of the *longest* overlap from s_i to s_j for $1 \leq i, j \leq n$. The *hierarchical overlap graph* of P, denoted by HOG(P), is a directed graph with a vertex set $V = P \cup Ov(P) \cup \{\epsilon\}$ and an edge set $E = E_1 \cup E_2$, where $E_1 = \{(x, y) \in V \times V \mid x$ is the longest proper prefix of $y\}$ and $E_2 = \{(x, y) \in V \times V \mid y$ is the longest proper suffix of $x\}$.

▶ **Definition 2** (Extended Hierarchical Overlap Graph). Given a set $P = \{s_1, s_2, \ldots, s_n\}$, we define $Ov^+(P)$ as the set of all overlaps from s_i to s_j for $1 \leq i, j \leq n$. The *extended hierarchical overlap graph* of P, denoted by EHOG(P), is a directed graph with a vertex set $V^+ = P \cup Ov^+(P) \cup \{\epsilon\}$ and an edge set $E^+ = E_1^+ \cup E_2^+$, where $E_1^+ = \{(x, y) \in V^+ \times V^+ \mid x$ is the longest proper prefix of $y\}$ and $E_2^+ = \{(x, y) \in V^+ \times V^+ \mid y$ is the longest proper suffix of $x\}$.

Figure 1 shows the Aho–Corasick trie [1], EHOG, and HOG built with $P = \{aacaa, aagt, gtc\}$. It is shown in [5] that EHOG is a contracted form of the Aho–Corasick trie and HOG is a contracted form of EHOG.

As in the Aho–Corasick trie, each node u in HOG or EHOG corresponds to a string (denoted by $S(u)$), which is a concatenation of all labels on the path from the root (node representing ϵ) to u.

There are two types of edges in EHOG and HOG as in the Aho–Corasick trie: a tree edge and a failure link. An edge (u, v) is a tree edge (an edge in E_1^+ or E_1, solid line in Figure 1) in an EHOG (HOG), if $S(u)$ is the longest proper prefix of $S(v)$ in the EHOG (HOG). It is a failure link (an edge in E_2^+ or E_2, dotted line in Figure 1) in an EHOG (HOG), if $S(v)$ is the longest proper suffix of $S(u)$ in the EHOG (HOG).

Given a set $P = \{s_1, s_2, \ldots, s_n\}$ of strings, we can build an EHOG of P in $O(||P||)$ time and space [5]. Furthermore, given EHOG(P) and $Ov(P)$, we can compute HOG(P) in $O(||P||)$ time and space [5]. Therefore, the bottleneck of computing HOG(P) is computing $Ov(P)$ efficiently.

3 Computing HOG in linear time

In this section we introduce an algorithm to build the HOG of $P = \{s_1, s_2, \ldots, s_n\}$ in $O(||P||)$ time. We assume that there are no two different strings $s_i, s_j \in P$ such that s_i is a substring of s_j for simplicity of presentation. Our algorithm directly computes HOG(P) (and $Ov(P)$) from the Aho–Corasick trie of P in $O(||P||)$ time.

Let us assume we have the Aho–Corasick trie of P including the failure links. We define $R(u)$ for each node u of the trie, as follows:

$$R(u) = \{i \in \{1, \ldots, n\} \mid S(u) \text{ is a proper prefix of } s_i\}. \tag{1}$$

That is, $R(u)$ is a set of string indices in the subtree rooted at u if u is an internal node, or an empty set if u is a leaf node.

For each input string s_i, we will do the following operation separately, which is to find the longest overlap from s_i to any string in P. Consider a path $(v_0, v_1, \ldots, v_l)$ which starts from the leaf representing s_i and follows the failure links until it reaches the root, i.e., $S(v_0) = s_i$ and v_l is the root of the tree. By definition of the failure link, the strings corresponding to nodes appearing in the path are suffixes of s_i. If there are an index j and a node v_k on the path such that $j \in R(v_k)$, $S(v_k)$ is both a suffix of s_i and a proper prefix of s_j, so $S(v_k)$ is an overlap from s_i to s_j.

$S(v_k)$ for $0 < k \leq l$ is the longest overlap from s_i to s_j if and only if $j \in R(v_k)$ and there is no m such that $0 \leq m < k$ and $j \in R(v_m)$. If there exists such m, then $S(v_m)$ is a longer overlap from s_i to s_j than $S(v_k)$, so $S(v_k)$ is not the longest overlap. Therefore, we get the following lemma.

▶ **Lemma 3.** $S(v_k)$ is the longest overlap from s_i to s_j if and only if $j \in R(v_k) - R(v_{k-1}) - \ldots - R(v_0)$.

Therefore, if $|R(v_k) - R(v_{k-1}) - \ldots - R(v_0)| > 0$, $S(v_k)$ is the longest overlap from s_i to s_j for $j \in R(v_k) - R(v_{k-1}) - \ldots - R(v_0)$, and thus $v_k \in Ov(P)$. Therefore, we aim to compute $|R(v_k) - R(v_{k-1}) - \ldots - R(v_0)|$ for every $0 < k \leq l$.

Given an index k, we define $k + 1$ auxiliary sets of indices $I_k(k), I_k(k-1), \ldots, I_k(0)$ in a recursive manner as follows.

- $I_k(k) = R(v_k)$
- $I_k(m) = I_k(m+1) - R(v_m)$ for $m = k-1, k-2, \ldots, 0$

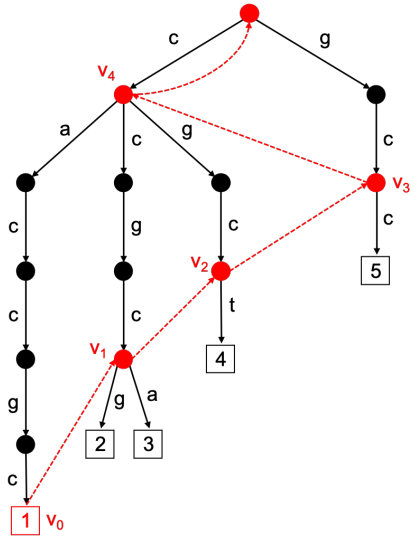


Figure 2 Aho–Corasick trie with $P = \{caccgc, ccgcg, ccgca, cgct, gcc\}$.

By definition, $I_k(0)$ is $R(v_k) - R(v_{k-1}) - \ldots - R(v_0)$ in Lemma 3 and we want to compute $|I_k(0)|$. For every $0 \leq m < k$, $I_k(m) = I_k(m+1) - R(v_m) \subseteq I_k(m+1)$ and thus $|I_k(m)| = |I_k(m+1)| - |I_k(m+1) - I_k(m)|$ holds. By summing up all these equations for $0 \leq m < k$, we get $|I_k(0)| = |I_k(k)| - \sum_{m=0}^{k-1} |I_k(m+1) - I_k(m)|$. Since $I_k(k) = R(v_k)$ and $I_k(m+1) - I_k(m) = I_k(m+1) - (I_k(m+1) - R(v_m)) = I_k(m+1) \cap R(v_m)$, we have

$$|I_k(0)| = |R(v_k)| - \sum_{m=0}^{k-1} |I_k(m+1) \cap R(v_m)|. \tag{2}$$

We also define a new function $up(u)$ for a node u as follows.

▶ **Definition 4.** Given a node u in the Aho–Corasick trie, $up(u)$ is defined as the first ancestor of u (except u itself) in the path that starts at u and follows the failure links until it reaches the root node. We define an ancestor on the tree which consists of tree edges in the Aho–Corasick trie.

Note that $up(u)$ is well defined when u is not the root node, since the root node is always an ancestor of u. When u is the root node, $up(u)$ is empty.

Now we analyze the value of $|I_k(m+1) \cap R(v_m)|$ in Equation (2) for each $0 \leq m < k$ as follows. We use a path $(v_0, v_1, ..., v_5)$ in Figure 2 as a running example, i.e., $l = 5$ and $0 < k \leq 5$.

▶ **Lemma 5.** $|I_k(m+1) \cap R(v_m)|$ is $|R(v_m)|$ if $up(v_m) = v_k$; it is 0 otherwise.

Proof. We divide the relationship between v_m and v_k into cases.

1. v_m is outside the subtree rooted at v_k
   Let us assume that $I_k(m+1) \cap R(v_m)$ is not empty and there exists $j \in I_k(m+1) \cap R(v_m)$. Then $j \in R(v_k) \cap R(v_m)$ should hold since $I_k(m+1) \subseteq I_k(k) = R(v_k)$. Therefore, both v_m and v_k should be ancestors of the leaf corresponding to s_j. Because $|S(v_m)| > |S(v_k)|$, v_k should be an ancestor of v_m. Since v_m is outside the subtree rooted at v_k, v_k cannot be an ancestor of v_m, which is a contradiction. Therefore such j does not exist, which shows that $I_k(m+1) \cap R(v_m) = \emptyset$ and $|I_k(m+1) \cap R(v_m)| = 0$.

For example, consider the case with $k = 4$ and $m = 3$ in Figure 2. Since $I_4(4) = R(v_4) = \{1, 2, 3, 4\}$ and $R(v_3) = \{5\}$, we can see that $I_4(4) \cap R(v_3) = \emptyset$.

2. v_m is inside the subtree rooted at v_k
In this case, v_k is an ancestor of v_m and we further divide it into cases.

a. There exists q such that $m < q < k$ and v_q is an ancestor of v_m.
We get $R(v_m) \subseteq R(v_q)$ because v_q is an ancestor of v_m. Since $I_k(m+1) = R(v_k) - R(v_{k-1}) - ... - R(v_{m+1})$ and $m < q < k$, we have $I_k(m+1) \subseteq R(v_k) - R(v_q)$. Therefore, $I_k(m+1) \cap R(v_m) \subseteq (R(v_k) - R(v_q)) \cap R(v_q) = \emptyset$. That is, $I_k(m+1) \cap R(v_m) = \emptyset$ and $|I_k(m+1) \cap R(v_m)| = 0$.

b. For any q such that $m < q < k$, v_q is not an ancestor of v_m.
Here we show that $R(v_m) \subseteq I_k(m+1)$. Let us consider an index $x \in R(v_m)$. Since v_k is an ancestor of v_m, we have $x \in R(v_k)$. Moreover, for any q such that $m < q < k$, neither v_q is an ancestor of v_m nor v_m is an ancestor of v_q. That is, $R(v_q) \cap R(v_m) = \emptyset$ and thus $x \notin R(v_q)$. Therefore, we have $x \in I_k(m+1) = R(v_k) - R(v_{k-1}) - \ldots - R(v_{m+1})$. In conclusion, $R(v_m) \subseteq I_k(m+1)$ and thus $|I_k(m+1) \cap R(v_m)| = |R(v_m)|$.
For example, consider the case with $k = 4$ and $m = 1$ in Figure 2. Since $I_4(2) = R(v_4) - R(v_3) - R(v_2) = \{1, 2, 3\}$ and $R(v_1) = \{2, 3\}$, we can see that $R(v_1) \subseteq I_4(2)$ and $I_4(2) \cap R(v_1) = R(v_1)$.

In summary, $|I_k(m+1) \cap R(v_m)| = |R(v_m)|$ in case 2(b), and 0 otherwise. In case 2(b), v_k is an ancestor of v_m and there is no q such that $m < q < k$ and v_q is an ancestor of v_m. In other words, v_k is the first ancestor of v_m in the path starting from v_m and following the failure links repeatedly, which means that $up(v_m) = v_k$. ◀

▶ **Theorem 6.** For every $0 < k \leq l$, $|I_k(0)| = |R(v_k)| - \sum_{v_m} |R(v_m)|$, where $0 \leq m < k$ and $up(v_m) = v_k$.

Proof. From Equation (2), we have $|I_k(0)| = |R(v_k)| - \sum_{m=0}^{k-1} |I_k(m+1) \cap R(v_m)|$. By Lemma 5, we have $\sum_{m=0}^{k-1} |I_k(m+1) \cap R(v_m)| = \sum_{v_m : up(v_m) = v_k} |R(v_m)|$. By merging the two equations, we have the theorem. ◀

Now let us consider the relationship between u and $up(u)$. $S(up(u))$ is a proper suffix of $S(u)$ because $up(u)$ can be reached from u through failure links. Furthermore, $S(up(u))$ is a proper prefix of $S(u)$ because $up(u)$ is an ancestor of u. That is, $S(up(u))$ is a *border* [9] of $S(u)$. Moreover, we visit every suffix of $S(u)$ in the trie in the decreasing order of lengths and $S(up(u))$ is the first border we visit, so $S(up(u))$ is the *longest border* of $S(x)$. Since each node in the Aho–Corasick trie corresponds to a prefix of some s_i, we can compute $up(u)$ for all nodes u by computing the border array of every s_i as follows. Let $pnode_i(l)$ be the node which corresponds to $s_i[1..l]$, and $border_i(l)$ be the length of the longest border of $s_i[1..l]$. Then we have the following equation for every s_i and $1 \leq l \leq |s_i|$:

$$up(pnode_i(l)) = pnode_i(border_i(l)). \tag{3}$$

If we store $pnode_i$ and $border_i$ using arrays, we can compute $pnode_i, border_i$, and $up(u)$ in $O(||P||)$ time and space, because $border_i$ can be computed in $O(||P||)$ time using an algorithm in [9].

▶ **Example 7.** Let us consider Figure 2, which is an Aho–Corasick trie built with a set $P = \{s_1 = caccgc, s_2 = ccgcg, s_3 = ccgca, s_4 = cgct, s_5 = gcc\}$ of strings. For each string, we compute its corresponding border array, and get $border_1 = (0, 0, 1, 1, 0, 1)$, $border_2 = (0, 1, 0, 1, 0)$, $border_3 = (0, 1, 0, 1, 0)$, $border_4 = (0, 0, 1, 0)$, and $border_5 = (0, 0, 0)$. We also

Algorithm 1 Build HOG in linear time.

```
1:  procedure BUILD-HOG(P)
2:      Build the Aho–Corasick trie with P
3:      Compute border arrays border_i for 1 ≤ i ≤ n
4:      Compute up(u) for each node u
5:      Compute |R(u)| for each node u
6:      Mark root as included in HOG(P)
7:      For each node u, initialize Child(u) with an empty set
8:      for i ← 1 to n do
9:          u ← leaf corresponding to s_i in Aho–Corasick trie
10:         Mark u as included in HOG(P)
11:         while u ≠ root do
12:             I(u) ← |R(u)|
13:             for all u' ∈ Child(u) do
14:                 I(u) ← I(u) − |R(u')|
15:             if I(u) > 0 then
16:                 Mark u as included in HOG(P)
17:             Child(u) ← an empty set
18:             Add u to Child(up(u))
19:             u ← failure link of u
20:     Build HOG(P) with marked nodes
```

store $pnode_i$'s by traversing the Aho–Corasick trie. Now we can compute up by using $pnode_i$ and $border_i$. For example, let us consider $v_1 = pnode_2(4)$, which represents $ccgc$. Since the longest border of $ccgc$ is c, which has length 1, we have $border_2(4) = 1$. As a result, we have $up(v_1) = up(pnode_2(4)) = pnode_2(border_2(4)) = pnode_2(1) = v_4$ by Equation (3). Note that v_4 represents c, which is the longest border of $ccgc$.

We are ready to describe an algorithm to compute HOG of P in $O(||P||)$ time and space. First, we build the Aho–Corasick trie with P and a border array for each s_i. By using the border arrays, we compute $up(u)$ for every node u except the root. Next, we compute $|R(u)|$ for each node u by the post-order traversal of the Aho–Corasick trie. For each string s_i, we start from the leaf node corresponding to s_i and follow the failure links until we reach the root. For each node v_k that we visit, we compute its corresponding $|I_k(0)| = |R(v_k)| - \sum_{v_m : up(v_m) = v_k} |R(v_m)|$. If $|I_k(0)| > 0$, we mark v_k to be included in HOG. Algorithm 1 shows an algorithm to compute HOG. Lines 2–5 compute the preliminaries for the algorithm, while lines 6–19 compute the nodes to be included in HOG. Note that the loop of lines 8–19 works separately for each input string s_i. We consider v_k in the order of increasing k, and thus if $up(v_m) = v_k$, then $m < k$. Hence, Child(v_k) in line 13 stores every v_m such that $up(v_m) = v_k$ by line 18 of previous iterations. For each node $u = v_k$ in lines 11–19, $I(u)$ correctly computes $|I_k(0)|$ since we get $|R(v_k)|$ in line 12 and subtract every $|R(v_m)|$ where $v_k = up(v_m)$ in lines 13–14. According to Theorem 6, lines 12–14 correctly computes $|I_k(0)|$. We build HOG(P) in line 20 by removing the unmarked nodes and contracting the edges while traversing the Aho–Corasick trie once, as in [5].

▶ **Example 8.** Consider again the Aho–Corasick trie built with $P = \{s_1 = caccgc, s_2 = ccgcg, s_3 = ccgca, s_4 = cgct, s_5 = gcc\}$ in Figure 2. Let us consider a path starting from a node representing s_1 and following the failure links until the root node. The path

$(v_0, v_1, v_2, v_3, v_4, v_5)$ is marked with dotted lines in Figure 2. By definition of up, we get $up(v_0) = up(v_1) = up(v_2) = v_4$ and $up(v_3) = up(v_4) = v_5$. Therefore, we can compute $|I_k(0)|$'s as follows.

- $|I_0(0)| = |R(v_0)| = 0$
- $|I_1(0)| = |R(v_1)| = 2$
- $|I_2(0)| = |R(v_2)| = 1$
- $|I_3(0)| = |R(v_3)| = 1$
- $|I_4(0)| = |R(v_4)| - |R(v_0)| - |R(v_1)| - |R(v_2)| = 4 - 0 - 2 - 1 = 1$

Note that $|R(v_0)| = 0$ by definition of $R(u)$. Since v_1, v_2, v_3, and v_4 have positive $|I_k(0)|$'s, we mark them to be included in HOG. We do this process for every s_i.

Now we show that Algorithm 1 runs in $O(||P||)$ time and space. Computing an Aho–Corasick trie, a border array for each string, and $up(u)$ and $|R(u)|$ for each node u costs $O(||P||)$ time and space. Furthermore, for a given index i, lines 13–14 are executed at most $|s_i|$ times since line 18 is executed at most $|s_i|$ times, and thus the sum of $|\mathrm{Child}(u)|$ is at most $|s_i|$. Therefore, lines 9–19 run in $O(|s_i|)$ time for given i, and thus lines 8–19 run in $O(||P||)$ time in total. Also they use $O(|s_i|)$ additional space to store the Child list. Lastly, we can build HOG(P) with marked nodes in $O(||P||)$ space and time [5]. Therefore, Algorithm 1 runs in $O(||P||)$ time and space. We remark that Algorithm 1 can be modified so that it builds the HOG from an EHOG instead of an Aho–Corasick trie, while it still costs $O(||P||)$ time and space.

▶ **Theorem 9.** Given a set P of strings, HOG(P) can be built in $O(||P||)$ time and space.

4 Conclusion

We have presented an $O(||P||)$ time and space algorithm to build the HOG, which improves upon an earlier $O(||P|| \log n)$ time solution. Since the input size of the problem is $O(||P||)$, the algorithm is optimal.

There are some interesting topics about HOG and EHOG which deserve the future work. As mentioned in the introduction, the *shortest superstring problem* gained a lot of interest [2, 18, 11, 13]. Since many algorithms dealing with the shortest superstring problem are based on the overlap graph, HOG may give better approximation algorithms for the shortest superstring problem by using the additional information that HOG has when compared to the overlap graph.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
2. A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *Journal of the ACM*, 41(4):630–647, 1994. doi:10.1145/179812.179818.
3. B. Cazaux, R. Cánovas, and E. Rivals. Shortest DNA cyclic cover in compressed space. In *DCC*, pages 536–545, 2016. doi:10.1109/DCC.2016.79.
4. B. Cazaux and E. Rivals. A linear time algorithm for shortest cyclic cover of strings. *Journal of Discrete Algorithms*, 37:56–67, 2016. doi:10.1016/j.jda.2016.05.001.
5. B. Cazaux and E. Rivals. Hierarchical overlap graph. *Information Processing Letters*, 155:105862, 2020. doi:10.1016/j.ipl.2019.105862.
6. J. Gallant, D. Maier, and J. Astorer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980. doi:10.1016/0022-0000(80)90004-5.

7 G. Gonnella and S. Kurtz. Readjoiner: A fast and memory efficient string graph-based sequence assembler. *BMC Bioinformatics*, 13(1):82, 2012. doi:10.1186/1471-2105-13-82.

8 D. Gusfield, G. M. Landau, and B. Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Information Processing Letters*, 41(4):181–185, 1992. doi:10.1016/0020-0190(92)90176-V.

9 D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.

10 J. Lim and K. Park. A fast algorithm for the all-pairs suffix–prefix problem. *Theoretical Computer Science*, 698:14–24, 2017. doi:10.1016/j.tcs.2017.07.013.

11 M. Mucha. Lyndon words and short superstrings. In *SODA*, pages 958–972. SIAM, 2013. doi:10.1137/1.9781611973105.69.

12 E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21 Suppl 2:ii79–ii85, 2005. doi:10.1093/bioinformatics/bti1114.

13 K. Paluch. Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring, 2014. arXiv:1401.3670.

14 S. G. Park, B. Cazaux, K. Park, and E. Rivals. Efficient construction of hierarchical overlap graphs. In *SPIRE*, pages 277–290, 2020. doi:10.1007/978-3-030-59212-7_20.

15 H. Peltola. Algorithms for some string matching problems arising in molecular genetics. In *IFIP Congress*, pages 53–64, 1983.

16 P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001. doi:10.1073/pnas.171285098.

17 M. H. Rachid and Q. Malluhi. A practical and scalable tool to find overlaps between sequences. *BioMed Research International*, 2015, 2015. doi:10.1155/2015/905261.

18 Z. Sweedyk. A $2\frac{1}{2}$-approximation algorithm for shortest superstring. *SIAM Journal on Computing*, 29(3):954–986, 2000. doi:10.1137/S0097539796324661.

19 J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57(1):131–145, 1988. doi:10.1016/0304-3975(88)90167-3.

20 E. Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(1):313–323, 1990. doi:10.1007/BF01840391.

Efficient Algorithms for Counting Gapped Palindromes

Andrei Popa
Department of Computer Science, University of Bucharest, Romania

Alexandru Popa
Department of Computer Science, University of Bucharest, Romania

Abstract

A gapped palindrome is a string uvu^R, where u^R represents the reverse of string u. In this paper we show three efficient algorithms for counting the occurrences of gapped palindromes in a given string S of length N. First, we present a solution in $O(N)$ time for counting all gapped palindromes without additional constraints. Then, in the case where the length of v is constrained to be in an interval $[g, G]$, we show an algorithm with running time $O(N \log N)$. Finally, we show an algorithm in $O(N \log^2 N)$ time for a more general case where we count gapped palindromes uvu^R, where u^R starts at position i with $g(i) \leq v \leq G(i)$, for all positions i.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases pattern matching, gapped palindromes, suffix tree

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.23

1 Introduction

A gapped palindrome is a string uvu^R, where u^R is the reverse of the string u. Gapped palindromes are a generalization of palindromes, which are strings of the form uu^R. In this paper we study the counting of occurrences of gapped palindromes in given string S.

Palindromes have many applications in bioinformatics, more precisely in the study of DNA and RNA sequences, where problems involving pairs of equal substrings and the repetition of certain substrings appear naturally and were intensively studied (see, for example, [20, 15, 11, 10, 14, 18, 12]). Manacher [13] presents an algorithm that finds for every position i of the string the longest palindrome centered in i in $O(N)$ time. Using Manacher's algorithm we can count all occurrences of palindromes in linear time. We refer the reader to the book of Gusfield [7] and the references therein for more details regarding palindromes.

Gapped palindromes were first studied by Kolpakov and Kucherov in [9], where they design algorithms to find all maximal gapped palindromes in a string that obey two types of constraints termed long-armed and length-constrained palindromes. Dumitran, Gawrychowski and Manea present in [4] an algorithm which finds for each position i the longest string u such that uvu^R is a substring which has u^R starting at position i and $g \leq |v| < G$, where g and G are two given natural numbers. Another algorithm shown in the same paper solves a similar problem where $|v|$ is only constrained by a lower bound function $g(i)$. Both algorithms have running time $O(N)$.

Brodal et al. [2] show an algorithm which finds all tuples (a, b, c, d) such that $S[a..b] = S[c..d]$, $c - b$ is in a given interval $[g, G]$ and the pair of ranges $[a..b]$ and $[c..d]$ is maximal (it cannot be extended to the left or to the right such that the previous two properties still hold). The running time of Brodal et al.'s algorithm is $O(N \log N + z)$, where z is the number of the pairs found.

Counting palindromes was previously studied in [6] where it is shown that counting all distinct palindromes in a string can be done in linear time. Rubinchik and Shur describe in [16] an algorithm which counts distinct palindromes in a substring of a given string S of length N in time $O(\log N)$, with $O(N \log N)$ preprocessing time.

In this paper we study counting occurrences of gapped palindromes without traversing each of them, the running time of our algorithms not depending on this number. We consider that the counting problem is a natural variation of the already studied related problems and we offer solutions for counting occurrences of gapped palindromes under various constraints. The first problem we solve is counting all strings of the form uvu^R in a given string S without any other constraints, for which we give a linear time complexity algorithm. For the second problem, which has the additional constraint that for g and G given natural numbers, $g \leq |v| \leq G$, we describe a solution in $O(N \log N)$ time. Finally, we present the solution for the problem where for each position i of the given string S we find all substrings uvu^R where u^R begins at position i and $g(i) \leq |v| \leq G(i)$, for the given functions g and G, the algorithm having running time $O(N \log^2 N)$.

2 Preliminaries

A substring of a string S which starts at position i and ends at position j (inclusive) is denoted by $S[i..j]$. The length of the string S is $|S|$. We denote $|S|$ by N. The i-th character of the string is denoted by S_i, the first character is S_1 and the last is $S_{|S|}$. A string S which has the property that $S = S^R$ is called a *palindrome*. We denote by *gapped palindrome* a string $P = uvu^R$, where u and v are arbitrary strings and $|u| > 0$. For $|v| \leq 1$, P is also a palindrome. Note that a gapped palindrome P might be split into uvu^R in multiple ways. In this paper we consider occurrences of uvu^R and $u'v'u'^R$ different if either $u \neq u'$, $v \neq v'$ or the positions at which the two substrings start are different.

A data structure used in all our algorithms is the *suffix tree* introduced by Weiner in [19], also described in Gusfield [7]. A suffix tree of a string S is a tree T where each edge (u, v) (u is the parent of v) is labeled by a nonempty substring $S[i..j]$ of S. In this tree, the concatenation of the labels of all edges on a path from the root to a certain node n forms a string which we denote by $H(n)$. A fundamental property of the suffix tree structure is that any two edges (u, v_1), (u, v_2) are labeled by strings which start with different characters. Thus, two different nodes n_1 and n_2 have the property that $H(n_1) \neq H(n_2)$. Moreover, it holds that $lcp(H(n_1), H(n_2)) = H(lca(n_1, n_2))$, where $lca(n_1, n_2)$ represents the lowest common ancestor of the nodes n_1 and n_2 and $lcp(H(n_1), H(n_2))$ is the longest common prefix of the strings $H(n_1)$ and $H(n_2)$. For each suffix $S[i..N]$ of S there is a node n such that $S[i..N] = H(n)$ and by the previous property, the node n is unique, thus we denote it by $T(i)$. We term $T(i)$ *the associated node* of the suffix $S[i..N]$. If S is of the form $S'\$$ where $\$$ is a character that does not appear in S, then the associated node of a suffix is a leaf. We denote the root of the tree by n_r, $M(n)$ is the substring which labels the edge from the parent of the node n to n (in particular $M(n_r) = \epsilon$ where ϵ is the empty string).

By $Anc(n)$ we denote the set of ancestors of the node n (including n) and by $Tree(n)$ we denote the set of all nodes in the subtree of n. By the property of the lcp function we have that $lcp(S[i..N], S[j..N]) = H(lca(T(i), T(j)))$.

Assuming the alphabet of a string is of a constant size, the suffix tree can be constructed in $O(N)$ time, as shown by Weiner in [19]. There is also an online algorithm to construct the suffix tree in linear time found by Ukkonen [17]. Given a suffix tree of a string, Gusfield [7] shows how to compute $lcp(S[i..N], S[j..N])$ in $O(1)$ time.

For a given string S, Manacher shows in [13] an algorithm which computes Pal_i representing the length of the longest palindrome centered in i, for all $1 \leq i \leq N$, in time complexity $O(N)$. By summing all these values we obtain an algorithm to count all occurrences of palindromes in S in linear time.

Harel and Tarjan [8] describe a method to partition a rooted tree T such that the nodes of each partition form a path and any two nodes in the same partition are one the ancestor of the other. An important property of the partitioning, is that a path from any node to the root intersects $O(\log N)$ partitions. We refer to this kind of partitioning of a tree by *heavy path decomposition*.

An AVL Tree is a data structure introduced in [1]. An AVL tree is a balanced binary tree which holds a set of ordered elements. It can perform operations such as adding a new element to the set, removing an element, asking for the smallest element greater than a given value in $O(\log N)$, where N is the number of elements it holds. A property that we use in this paper is that two AVL trees of sizes N and M can be merged in $O\left(\log \binom{N+M}{N}\right)$ time, as shown by Brown And Tarjan in [3].

In our algorithms we use *binary indexed trees* for computing and updating efficiently partial sums on an array where we have operations of the form "add x to the element on the position p". The binary indexed tree is described by Fenwick in [5]. For a binary indexed tree F associated to an array A, we denote the operation $A[p] = A[p] + x$ by $F[p] = F[p] + x$ and the query $\sum_{i=1}^{p} A[i]$ by $F(p)$. Both operations are executed in running time $O(\log N)$ where N is the length of the array.

We name *reversed binary indexed tree* a similar structure F' where $F'(i) = \sum_{i=p}^{N} A[i]$. The reversed binary indexed tree F' can be built using a standard binary indexed tree F where on an operation $A[p] = A[p] + x$ we execute $F[|A| - p + 1] = F[|A| - p + 1] + x$ and $F'(i) = F(N - p + 1)$.

3 Counting all gapped palindromes

In this section we study the following problem:

▶ **Problem 1** (Counting all occurrences of gapped palindromes). *Let S be a given string of length N. Count how many tuples (a, b, c, d) exist such that $1 \leq a \leq b < c \leq d \leq N$ and $S[a..b]^R = S[c..d]$.*

We denote the set of these tuples by Q.

We begin with a fundamental observation.

▶ Remark 2. For fixed values c and b, the number of tuples (a, b, c, d) such that $S[a..b]^R = S[c..d]$ is $|lcp(S[1..b]^R, S[c..N])|$.

Proof. Let $L = |lcp(S[1..b]^R, S[c..N])|$. For any $l \in \{1..L\}$ we have that $S[b - l + 1..b]^R = S[c..c + l - 1]$, thus assigning $a := b - l + 1$ and $d := c + l - 1$ the property $S[a..b]^R = S[c..d]$ holds. Also we have that $S[b - l + 1..b]^R \neq S[c..c + l - 1]$ for any $l > L$, so for any assignment $a := b - l + 1$ and $d := c + l - 1$ the property $S[a..b]^R = S[c..d]$ does not hold. For other values of a and d, the strings $S[a..b]^R$ and $S[c..d]$ have different length. Thus, the number of tuples (a, b, c, d) with the desired property is $|\{1..L\}|$, which, in turn, is equal to $|lcp(S[1..b]^R, S[c..N])|$. ◀

Let T be the suffix tree built on the string $S' = S\#S^R\$$ of length N', where the characters $\#$ and $\$$ do not appear in the string S. We define j_r such that for $j \in \{1..N\}$ we have $S'[j..N] = S'[N+2..j_r]^R$ and for $j \in \{N+2..2\cdot N+1\}$ we have $S'[N+2..j]^R = S'[j_r..N]$. Thus j_r can be expressed as $2 \cdot N + 2 - j$.

▶ Remark 3. For some b and c such that $1 \leq b, c \leq N$ we have that:

$$lcp(S[1..b]^R, S[c..N]) = lcp(S'[b_r..N'], S'[c..N'])$$

Proof. By the structure of S' we have that $S'[b_r..N'-1] = S[1..b]^R$ and $S'[c..N] = S[c..N]$. Let L be $|lcp(S[1..b]^R, S[c..N])|$. If $L < min(|S[1..b]^R|, |S[c..N]|)$ then $S_{b-L} \neq S_{c+L}$, so $S'_{b_r+L} \neq S'_{c+L}$, thus $lcp(S'[b_r..N'], S'[c..N']) = L$. If $L = |S[1..b]^R|$ or $L = |S[c..N]|$, then either $S'_{b_r+L} = \$$ or $S'_{c+L} = \#$. Because the characters $\$$ and $\#$ are unique in S' and $b_r + L \neq c + L$, we have that $S'_{b_r+L} \neq S'_{c+L}$, thus $lcp(S'[b_r..N'], S'[c..N']) = L$. ◀

The following remark indicates how to find pairs of a reversed prefix and a suffix which have a certain longest common prefix:

▶ Remark 4. For a prefix $S[1..j]$ and a suffix $S[i..N]$, we have that $lca(T(i), T(j_r)) = n$ is equivalent to the existence of two nodes n_1 and n_2 children of the node n such that $T(i) \in Tree(n_1)$ and $T(j_r) \in Tree(n_2)$.

Using Remarks 3 and 4 we design Algorithm 1. Informally, Algorithm 1 performs as follows. For each node n of the suffix tree T built on the string S' we count the pairs of a reversed prefix and a suffix which have the *lcp* equal to $H(n)$, let this number be *cnt*. This number represents the number of pairs (b, c) such that $lcp(S[1..b]^R, S[c..N]) = H(n)$ holds. We add to the final answer the value $|H(n)| \times cnt$, where $|H(n)|$ comes from the number of pairs (a, d) such that $S[a..b]^R = S[c..d]$ for each of the (b, c) pairs. To count the number of pairs (b, c), we iterate over the children of n and we count the number of pairs formed by reversed prefixes and suffixes in the subtree of the current child with suffixes and reversed prefixes in the subtrees of the previous children.

▶ **Lemma 5.** *The Algorithm 1 gives the value* $ans_1 = \sum_{b=1}^{N} \sum_{c=1}^{N} lcp(S[1..b]^R, S[c..N])$ *in* $O(N)$ *time.*

Proof. Remarks 3 and 4 prove that the algorithm correctly counts for each node n the number of pairs (b, c) such that $lcp(S[1..b]^R, S[c..N]) = H(n)$. Every pair (b, c) is processed when $n = lca(T(b_r), T(c))$, adding the value $H(n)$ to ans_1, thus the value ans_1 is $\sum_{b=1}^{N} \sum_{c=1}^{N} lcp(S[1..b]^R, S[c..N])$. The time complexity of the algorithm is linear since its core is a simple *depth-first* traversal (with some constant time additional computations). ◀

Algorithm 1 does not count the elements of the answer set Q because the condition $b < c$ does not hold. Nevertheless, we can assert the following:

▶ Remark 6. Let

$$Q' = \{(a, b, c, d) \mid 1 \leq a \leq b \leq N, 1 \leq c \leq d \leq N, S[a..b]^R = S[c..d]\}$$

Then $ans_1 = |Q'|$.

Proof. It follows immediately from Lemma 5 and Remark 2. ◀

We observe that $Q \subset Q'$. We aim to find $|Q|$ by computing $|Q'|$ and $|Q' - Q|$.

Algorithm 1 Counting excluding the condition $b < c$.

```
1  Procedure iterate(n: the current node of the suffix tree)
       Data: Sf[u] - the number of the suffixes of S in the subtree of the node u
       Pr[u] - the number of the prefixes of S in the subtree of the node u
2      pr ← 0;
3      sf ← 0;
4      cnt ← 0;
5      if n is leaf then
6          i ← T^{-1}(n) ;          // n is the associated node of a suffix of S'
7          if i ≤ N then                    // i corresponds to a suffix of S
8              sf ← 1;
9          else if i > N + 1 and i ≤ 2 · N + 1 then  // i_r corresponds to a prefix
               of S
10             pr ← 1;
11         end
12     else
13         for ch ← child of n do
14             iterate(ch) ;
15             cnt ← cnt + Sf[ch] · pr;
16             cnt ← cnt + Pr[ch] · sf;
17             pr ← pr + Pr[ch];
18             sf ← sf + Sf[ch];
19         end
20     end
21     ans_1 ← cnt × |H(n)|;
22     Sf[u] ← sf;
23     Pr[u] ← pr;
24 iterate(n_r) // start the iteration from the root of the suffix tree
```

▶ Remark 7. Let (a, b, c, d) be a tuple in Q. Then $(a, b, c, d) \in Q'$ but also $(c, d, a, b) \in Q'$ and $(a, b, c, d) \neq (c, d, a, b)$. Moreover, if $(a, b, c, d) \in Q'$ and $[a, b] \cap [c, d] = \emptyset$, then $(a, b, c, d) \in Q$ or $(c, d, a, b) \in Q$.

In other words, a tuple $(a, b, c, d) \in Q$ "appears twice" in Q' and all the elements in Q' which have this property are those for which the substrings $S[a..b]$ and $S[c..d]$ do not intersect (either $b < c$ or $d < a$). We use Remark 7 to compute the final answer using ans_1 and the number of tuples (a, b, c, d) such that $S[a..b]$ and $S[c..d]$ do not intersect.

▶ **Lemma 8.** *Let $I = \{(a, b, c, d) \in Q' \mid [a, b] \cap [c, d] \neq \emptyset\}$ and $ans_2 = |I|$. Then the answer to the Problem 1 is $ans = (ans_1 - ans_2)/2$.*

Proof. From Remark 7 we have that $|Q| = |Q' \setminus I|/2$, and as $I \subset Q'$, we have $|Q' \setminus I| = |Q'| - |I|$. ◀

We say that a palindrome $S[a..b]$ is *centered* at $(a + b)/2$ and the value $(a + b)/2$ is the center of the palindrome.

▶ **Lemma 9.** *Let $(a, b, c, d) \in I$. Then $S[a..b]$ and $S[c..d]$ are contained in a palindrome centered at $(b + c)/2$. More precisely, there are l and r such that $l \leq a, b, c, d \leq r$ and $S[l..r]$ is a palindrome and $(l + r)/2 = (b + c)/2$.*

Proof. Assume that $b \leq d$. The case $b > d$ is solved similarly.

As $S[a..b]$ and $S[c..d]$ intersect, we have that $b \geq c$. Thus S has the following structure:

$$S = XS[a..c-1]S[c..b]S[b+1..d]Y$$

Due to the fact that $S[a..b]^R = S[c..d]$ and $b \leq d$ we have that $S[a..c{-}1\ c..b]^R = S[c..b\ b{+}1..d]$, thus $S[c..b]^R = S[c..b]$ and $S[a..c{-}1]^R = S[b{+}1..d]$. Therefore S has the structure $XUAU^RY$, where A is a palindrome centered at $(b+c)/2$. It follows that $S[a..d] = UAU^R$ also has the center $(b+c)/2$. ◀

To count $(a,b,c,d) \in I$, we compute for each $c_e \in [1,N]$, $2 \cdot c_e \in \mathbb{N}$ the value V_{c_e} representing the number of tuples which are included in a palindrome centered at c_e and $c_e = (b+c)/2$. Then $ans_2 = \sum_{c_e} V_{c_e}$.

▶ **Lemma 10.** *Let $S[l..r]$ be the longest palindrome centered at c_e. Then $V_{c_e} = \lceil (r-l+1)/2 \rceil^2$.*

Proof. Observe that a can take any integer value in the interval $[l, \lfloor (l+r)/2 \rfloor]$, b can take any integer value in $[\lceil (l+r)/2 \rceil, r]$. By Lemma 9, the values c and d are the symmetric of b and a with respect to c_e. More precisely $c = (l+r) - b$ and $d = (l+r) - a$. Thus the number of tuples (a,b,c,d) is $(\lfloor (l+r)/2 \rfloor - l + 1) \cdot (r - (\lceil (l+r)/2 \rceil) + 1)$ which is equal to $\lceil (r-l+1)/2 \rceil^2$. ◀

We use all the previous results in designing Algorithm 2. We can thus state the theorem:

▶ **Theorem 11.** *Problem 1 can be solved in $O(N)$ time.*

Proof. Algorithm 2 computes the desired value $ans = |Q|$. The correctness of Algorithm 1 is given by Lemma 5. The step 4 is correct by Lemma 10. Lemma 8 proves the correctness of the step 6.

The suffix tree of the string S' in step 1 is computed in $O(N)$ time using Ukkonen's algorithm [17]. Algorithm 1 runs in linear time by Lemma 5. The lengths of the longest palindromes centered in each position are computed using Manacher's algorithm [13] in $O(N)$. Step 5 is done in linear time and Step 6 takes constant time. Thus the entire algorithm runs in $O(N)$ time. ◀

Algorithm 2 The algorithm for solving Problem 1.

1 build *suffix tree* on the string $S' = S\#S^R\$$;
2 compute ans_1 using Algorithm 1 ;
3 for each center c compute the longest palindrome in S centered at c ;
4 compute the values $V_c = \lceil (r-l+1)/2 \rceil^2$ where $S[l..r]$ is the longest palindrome centered at c ;
5 compute ans_2 as $\sum_{c=2}^{2 \cdot N} V_{c/2}$;
6 compute $ans = (ans_1 - ans_2)/2$;

4 Palindromes with constraints on the length of the gap

In this section we consider a version of Problem 1 with the additional constraint that Q contains only tuples with the property that $g \leq c - b - 1 \leq G$, for given g and G positive integers.

▶ **Problem 12** (Counting palindromes with gap length constraint). *Let S be a string of length N and g, G positive integers. Let*

$$Q = \{(a, b, c, d) \mid 1 \leq a \leq b < c \leq d \leq N, g \leq c - b - 1 \leq G, S[a..b]^R = S[c..d]\}$$

. Find $|Q|$.

We use an adaptation of Algorithm 1. Instead of counting for each node n all the pairs (i, j) such that $lcp(S[1..j]^R, S[i..N]) = H(n)$, we count only those that also hold the constraint on the gap, namely $g \leq i - j - 1 \leq G$. This approach obtains the correct answer directly without the need of a strategy to subtract wrongly counted tuples as in the Problem 1. However, a data structure on an ordered set is needed, thus increasing the running time of the algorithm.

Brodal et al. [2] describe an algorithm that solves a similar problem. For a node n of the suffix tree, Brodal et al.'s algorithm processes pairs of suffixes belonging to the subtrees of two different children. For a suffix $S[i..N]$ in one subtree the algorithm finds the smallest index j of a suffix $S[j..N]$ in the other subtree, such that $j \geq i + v$. The value v is independent of i and j, but depends on the node n. The algorithm also iterates on some interval of suffixes starting from j, thus its time complexity depends on the number of suffixes individually found. Because in our problem we need only to find the number of the elements in an interval, we change the algorithm to have a running time depending only on N.

The modification of the strategy presented in Brodal et al. [2] needs the following lemma:

▶ **Lemma 13.** *Let E be a list of sorted elements and T an AVL tree of sizes N and M respectively, such that $N \leq M$. For each element i of E we can find the biggest element j in T such that $j \leq i$ and its position in T (in other words, how many elements of T are smaller than j) in time $O\left(\log \binom{N+M}{N}\right)$.*

Proof. From Lemma 3 of Brodal et al. [2] we know that for each i in E we can find the smallest j in T such that $j \geq i$ in time $O\left(\log \binom{N+M}{N}\right)$. In our algorithm we use a similar operation.

Now we describe the method of finding the position of the element j. We denote by $L(n)$ the left child of the node n and by $R(n)$ the right child of the node n. For each node u from the AVL we keep a table $Size(u)$ representing the size of the subtree rooted in the node u. When inserting a new element in the AVL tree, we update the value $Size(u)$ to be $Size(L(u)) + Size(R(u)) + 1$ at the step the node u is visited. For finding the position of an element, during the traversal of the tree, we keep a variable p which counts the number of smaller elements than the element in the current node which are not situated in the current subtree. When visiting the left child, we leave p unchanged, when visiting the right child we add to p the value $Size(L(u)) + 1$, and when returning to the parent we undo the change made when we visited the node u.

The running time of the algorithm does not change, thus the operation can be performed in $O\left(\log \binom{N+M}{N}\right)$. ◀

▶ **Theorem 14.** *Problem 12 can be solved in running time $O(N \log N)$.*

Proof. We apply the strategy described by Brodal et al. in [2]. We change the suffix tree to have each node with at most two children in order to use "the smaller half trick". For each node n we maintain an AVL tree that stores all the suffixes i in the subtree of n and another AVL tree that stores all the prefixes j in the subtree of n. The AVL tree for a node n is computed as the union of the AVL trees of its children, which can be done in time $O\left(\log \binom{s_1+s_2}{s_1}\right)$ where s_1 and s_2 are the number of nodes in the two subtrees of node n. All union operations are processed in $O(N \log N)$ time.

For a node n of the suffix tree, let n_1 be the child with the smaller subtree and n_2 the child with the bigger subtree. For each suffix i of S in the subtree of n_1 we find how many prefixes j of S in the subtree of n_2 exist such that $g \leq i - j - 1 \leq G$. We also find for each prefix j in the subtree of n_1 the number of suffixes i in the subtree of n_2 such that $g \leq i - j - 1 \leq G$. We add both numbers to the answer.

We describe the first case, the other one being similar. For each suffix i in the subtree of n_1 count the number of prefixes j_1 in the subtree of n_2 that hold the condition $g \leq i - j_1 - 1$ and the number of prefixes j_2 which hold the condition $G + 1 \leq i - j_2 - 1$, the result being the difference of the two numbers. We rewrite $g \leq i - j_1 - 1$ as $j \leq i - g - 1$. We merge the sorted list of the suffixes of n_1 with the AVL tree which holds the prefixes of the subtree rooted at n_2. The elements i from the sorted list are treated as they would be $i - g - 1$ and we return the largest j in the AVL which is smaller than or equal to $i - g - 1$ and its position. If the position of j is p_j, then the number of prefixes counted for the suffix i is p_j.

The running time of our algorithm is $O(N \log N)$, being just a modification of the algorithm described by Brodal et al. in [2]. ◀

5 Gapped palindromes with positional constraints on the length of the gap

▶ **Problem 15** (Counting gapped palindromes for every position)**.** *Let S be a string of size N. We say that for an index i a pair (j, r) is valid if $j < i$ and $S[i..i + r - 1]^R = S[j - r + 1..j]$. Given two functions g and G, count for each index i the number P_i representing the number of valid pairs (j, r) for which the property $g(i) \leq i - j - 1 \leq G(i)$ holds.*

We begin by reducing the problem of counting pairs (j, r) where j is bounded from both sides to one where j is only upper bounded.

▶ Remark 16. Let P_i^g be the number of valid pairs (j, r) of i such that $i - j - 1 \geq g(i)$ and P_i^G the number of valid pairs (j, r) of i such that $i - j - 1 \geq G(i) + 1$ (equivalently $j \leq i - g(i) - 1$ and $j \leq i - G(i) - 2$ respectively). Then $P_i = P_i^g - P_i^G$.

The following remark shows how to compute the values P_i^g and P_i^G based on Remark 2:

▶ Remark 17. The values P_i^g and P_i^G can be computed in the following way:

$$P_i^g = \sum_{j=1}^{i-g(i)-1} |lcp(S[1..j]^R, S[i..N])|$$

$$P_i^G = \sum_{j=1}^{i-G(i)-2} |lcp(S[1..j]^R, S[i..N])|$$

We provide a data structure to compute the value $\sum_{j=1}^{K} |lcp(S[1..j]^R, S[i..N])|$ where initially $K = 0$ and we make updates of the form $K = K + 1$. We denote by $Q(i) = \sum_{j=1}^{K} lcp(S[1..j]^R, S[i..N])$ at a certain state given by K.

Let W be an array indexed by the set of the nodes of T.

Initially we have that $W[n] = 0, \forall n \in T$. On an update $K = K + 1$ we execute $W[u] = W[u] + |M(u)|$ for each u ancestor of $T((K+1)_r)$ (recall that $(K+1)_r = 2 \cdot N + 2 - (K+1)$).

▶ Remark 18. Let A be the set of prefixes processed at a certain moment, more precisely $A = \{T(j_r) \mid j \leq K\}$, K is the index of the last prefix added to the structure. Then $W[n] = |M(n)| \cdot |A \cap Tree(n)|$.

We can find the value $Q(i)$ using the following lemma.

▶ **Lemma 19.**

$$Q(i) = \sum_{u \in Anc(T(i))} W[u]$$

Proof. By Remark 18 we have that

$$\begin{aligned}
\sum_{u \in Anc(T(i))} W[u] &= \sum_{u \in Anc(T(i))} |M(u)| \cdot |A \cap Tree(u)| \\
&= \sum_{u \in Anc(T(i))} \sum_{v \in A \cap Tree(u)} |M(u)| \\
&= \sum_{\substack{(u,v) \in Anc(T(i)) \times A \\ v \in Tree(u)}} |M(u)| \\
&= \sum_{\substack{(u,v) \in Anc(T(i)) \times A \\ u \in Anc(v)}} |M(u)| \\
&= \sum_{v \in A} \sum_{u \in Anc(T(i)) \cap Anc(v)} |M(u)| \\
&= \sum_{v \in A} \sum_{u \in Anc(lca(T(i),v))} |M(u)|
\end{aligned}$$

but $A = \{T(j_r) \mid j \leq K\}$, thus

$$\begin{aligned}
&= \sum_{j \leq K} \sum_{u \in Anc(lca(T(i),T(j_r)))} |M(u)| \\
&= \sum_{j \leq K} |H(lca(T(i), T(j_r)))| \\
&= Q(i)
\end{aligned}$$

◀

We describe a method to update and compute the sum over the array W. Build the *heavy path decomposition* of the suffix tree T. Let $Ch(n)$ be the partition corresponding to the node n. By looking at the partitions of the decomposition as sequences, we define $Pos(n)$ being the position of n in the sequence $Ch(n)$ and $L_l(p)$ the node at position p in the partition l. We denote by $Par(n)$ the parent of the node n and $Par(n_r) = nil$ where nil denotes the absence of a value. The partition has the following property:

▶ **Property 20.** *$Par(L_l(i)) = L_l(i-1)$ for each $i > 1$. Moreover, if u is an ancestor of v, then $L_{Ch(u)}(i)$ is also an ancestor of v, for all $i \leq Pos(u)$.*

Harel and Tarjan show in [8] a property of the heavy path decomposition which we reformulate here:

▶ **Property 21.** *The number of distinct partitions encountered on a path from a node n to the root is $O(\log N)$. In other words, consider an algorithm which has a variable n initially equal to some node of the tree. The number of steps in a loop of the form "while $n \neq nil$ do $l = Ch(n)$, $n = Par(L_l(1))$" is of the order $O(\log N)$.*

By Properties 20 and 21 we design Algorithms 3 and 4, which update the structure when executing $K = K + 1$ and query the structure respectively.

Algorithm 3 The update procedure.

```
1  Procedure update()
       Data: St_l[j] - for each l an array with the values W[u] corresponding to the
             chain l, meaning St_l[j] = W[L_l(j)]
2      n ← T((K+1)_r) ;          // the leaf corresponding to the added prefix
3      K := K + 1;
4      while n ≠ nil do
5          p ← Pos(n);
6          l ← Ch(n);
7          for j ← 1..p do      // operation executed by an efficient structure
8              St_l[j] = St_l[j] + |M(L_l(j))| ;          // add |M(u)| on the chain
9          end
10         n ← Par(L_l(1));
11     end
```

Algorithm 4 The query function.

```
1  Function query(i: the index of the suffix)
       Data: St_l[j] - for each l the array with the values W[u] corresponding l to the
             chain l, meaning St_l[j] = W[L_l(j)]
2      n ← T(i) ;                // the leaf corresponding to the added suffix
3      sum ← 0;
4      while n ≠ nil do
5          p ← Pos(n);
6          l ← Ch(n);
7          for j ← 1..p do      // operation executed by an efficient structure
8              sum = sum + St_l(j)
9          end
10         n ← Par(L_l(1));
11     end
12     return sum
```

▶ **Lemma 22.** *Algorithms 3 and 4 simulate correctly the operations on W. The time complexity of Algorithm 3 is $O(\log N)$ operations of the form "assign $B[i] = B[i] + A[i]$ for all $i \leq p$", where p is a given integer. Algorithm 4 runs in $O(\log N)$ time operations of the form "compute $\sum_{i=1}^{p} B[i]$" for a given p.*

Proof. We prove only the statements about Algorithm 3, those concerning Algorithm 4 can be proven similarly.

Consider line 7 of Algorithm 3. If we reverse the direction of the loop, then, by Property 20, the nodes $L_l(j)$ on the next line provide the iteration of the ancestors of the node in order, from n to the chain above. Line 10 takes the next node on the path from n to the root which is processed at the next iteration of the exterior loop (when j on line 7 is equal to p). Thus, every ancestor of n is eventually visited.

The time complexity of the algorithm is a direct consequence of Property 21, because in each iteration of the exterior loop (line 4 of Algorithm 3) an operation of the form $St_l[j] = St_l[j] + |M(L_l(j))|$ is executed. ◀

Finally we provide a data structure to perform the operations on the arrays St_l in Algorithms 3 and 4. More precisely, we solve the following problem:

▶ **Problem 23.** *Let A_i be a sequence and B be an array, both of size N, initially $B[i] = 0, \forall i$. Execute a series of operations of the form:*

- *update: given p, assign $B[i] = B[i] + A_i$ for all $i \leq p$*
- *query: given p, find $\sum_{i=1}^{p} B[i]$*

▶ **Lemma 24.** *Problem 23 can be solved by executing each operation in running time $O(\log N)$.*

Proof. Let $S_i = \sum_{j=1}^{i} A_j$, be the partial sum sequence of A which can be computed in linear time and P_i the sequence of values p which were used to update the values of B up to the current step. Let L be the length of the sequence P. For a given position p we can compute the partial sum query in the following way:

$$\sum_{i=1}^{p} B[i] = \sum_{j=1}^{L} \sum_{i=1}^{p} A_i^j$$

where $A_i^j = A_i$ for $i \leq P_j$ and $A_i^j = 0$ otherwise. Then, let $P^{\leq p}$ be the sequence with values of P such that $P_i \leq p$ and $P^{>p}$ the sequence with values $P_i > p$, with lengths $L^{\leq p}$ and $L^{>p}$ respectively. We have:

$$\sum_{i=1}^{p} B[i] = \sum_{j=1}^{L^{\leq p}} S[P_j^{\leq p}] + \sum_{j=1}^{L^{>p}} S[p]$$

$$\sum_{i=1}^{p} B[i] = \left(\sum_{j=1}^{L^{\leq p}} S[P_j^{\leq p}] \right) + S[p] \cdot L^{>p}$$

or, for a better view:

$$\sum_{i=1}^{p} B[i] = \sum_{j=1}^{L^{\leq p}} S[P_j^{\leq p}] + S[p] \cdot \sum_{j=1}^{L^{>p}} 1$$

Thus, we separate the computation of $\sum_{i=1}^{p} B[i]$ in two, easier to compute, sums. The first sum, $\sum_{j=1}^{L^{\leq p}} S[P_j^{\leq p}]$ can be computed using a binary indexed tree F_1, which for a given p we update by executing $F_1[p] = F_1[p] + S_p$. We compute the second sum, $S[p] \cdot \sum_{j=1}^{L^{>p}} 1$ by using a reversed binary indexed tree F_2, which for a given p we update by executing $F_2[p] = F_2[p] + 1$. The answer for a query of the form $\sum_{i=1}^{p} B[i]$ is $F_1(p) + S_p \cdot F_2(p+1)$. ◀

All the above lemmas and remarks are joint together in Algorithm 5.

Algorithm 5 The algorithm which solves Problem 15.

```
1  build the suffix tree of the string S' = S#S^R$;
2  compute the pairs (i, i − g(i) − 1) and (i, i − G(i) − 2) and sort them by the second
     field;
3  for K ← 0 to N − 1 do
4  |  for (i, _) ← remaining pairs which have the second field equal to K do
5  |  |  P_i^g or P_i^G := query(i);
6  |  end
7  |  update();
8  end
9  for i ← 1 to N do
10 |  P_i = P_i^g − P_i^G;
11 end
```

We conclude with the following theorem:

▶ **Theorem 25.** *Problem 15 can be solved in time complexity* $O(N \log^2 N)$.

Proof. By Lemma 22 we have that Algorithms 3 and 4 correctly simulate the operations on the array W and the Lemma 19 shows that the operations on the array W correctly computes the value $Q(i)$. Remark 17 shows that Algorithm 5 correctly computes $P^g(i)$ and $P^G(i)$, then computes P_i using Remark 16. Thus, the correctness of the algorithm is proven.

Using Lemma 24 we obtain that Algorithms 3 and 4 have the total time complexity $O(\log^2 N)$. Algorithm 5 does N calls of Algorithm 3. The query function is called two times for each position i (once for $P^g(i)$ and once for $P^G(i)$), thus there are $2 \cdot N$ calls of Algorithm 4. Therefore the total running time of the algorithm is $O(N \log^2 N)$. ◀

6 Conclusions and future work

In this paper we show a linear time algorithm that counts the number of gapped palindromes in a string. Then, we show an algorithm with time complexity $O(N \log N)$ for a variation of the problem in which we have a lower and an upper bound on the length of the gap of the palindromes counted. Finally, we show an algorithm with $O(N \log^2 N)$ time complexity for a more general case where we count gapped palindromes uvu^R, where u^R starts at position i with $g(i) \leq v \leq G(i)$, for all positions i.

As an open problem, we believe that it is possible to solve Problem 15 using an algorithm that has $O(N \log N)$ running time. One possible research direction is to adapt the algorithm in Section 4. However, we mention that a straightforward adaptation is not possible for the following reason. In our suffix tree traversal, we count for every prefix/suffix from the smallest subtree the corresponding pair from the other subtree, while the goal is to count for every suffix the corresponding prefixes irrespective of the subtree they belong to.

Another approach to obtain a $O(N \log N)$ algorithm for Problem 15 is to use a suffix array instead of a suffix tree. Then, we tried to use the fact that the length of the longest common prefix of two suffixes from the suffix array is the minimum length of the longest common prefix of all the pairs of consecutive suffixes between the two given suffixes. The difficulty consists in building a data structure which handles the change of the minimum longest common prefix of two consecutive suffixes in an interval and queries sums on partial minimum longest common prefix of two consecutive suffixes both in time complexity $O(\log N)$.

References

1. Georgy Adelson-Velsky and Evgenii Landis. An algorithm for organization of information. *Doklady Akademii Nauk SSSR*, 146(2):263–266, 1962.
2. Gerth Stølting Brodal, Rune B Lyngsø, Christian NS Pedersen, and Jens Stoye. Finding maximal pairs with bounded gap. In *Annual Symposium on Combinatorial Pattern Matching*, pages 134–149. Springer, 1999.
3. Mark R Brown and Robert E Tarjan. A fast merging algorithm. *Journal of the ACM (JACM)*, 26(2):211–226, 1979.
4. Marius Dumitran, Paweł Gawrychowski, and Florin Manea. Longest Gapped Repeats and Palindromes. *Discrete Mathematics & Theoretical Computer Science*, Vol. 19 no. 4, FCT '15, 2017. doi:10.23638/DMTCS-19-4-4.
5. Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and experience*, 24(3):327–336, 1994.
6. Richard Groult, Élise Prieur, and Gwénaël Richomme. Counting distinct palindromes in a word in linear time. *Information Processing Letters*, 110(20):908–912, 2010.
7. Dan Gusfield. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News*, 28(4):41–60, 1997.
8. Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.
9. Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theoretical Computer Science*, 410(51):5365–5373, 2009.
10. Tomoko Kuroda-Kawaguchi, Helen Skaletsky, Laura G Brown, Patrick J Minx, Holland S Cordum, Robert H Waterston, Richard K Wilson, Sherman Silber, Robert Oates, Steve Rozen, et al. The azfc region of the y chromosome features massive palindromes and uniform recurrent deletions in infertile men. *Nature genetics*, 29(3):279–286, 2001.
11. David RF Leach. Long dna palindromes, cruciform structures, genetic instability and secondary structure repair. *Bioessays*, 16(12):893–900, 1994.
12. Le Lu, Hui Jia, Peter Dröge, and Jinming Li. The human genome-wide distribution of dna palindromes. *Functional & integrative genomics*, 7(3):221–227, 2007.
13. Glenn Manacher. A new linear-time"on-line"algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM (JACM)*, 22(3):346–351, 1975.
14. Sjoerd Repping, Helen Skaletsky, Julian Lange, Sherman Silber, Fulco van der Veen, Robert D Oates, David C Page, and Steve Rozen. Recombination between palindromes p5 and p1 on the human y chromosome causes massive deletions and spermatogenic failure. *The American Journal of Human Genetics*, 71(4):906–922, 2002.
15. Steve Rozen, Helen Skaletsky, Janet D Marszalek, Patrick J Minx, Holland S Cordum, Robert H Waterston, Richard K Wilson, and David C Page. Abundant gene conversion between arms of palindromes in human and ape y chromosomes. *Nature*, 423(6942):873–876, 2003.
16. Mikhail Rubinchik and Arseny M Shur. Counting palindromes in substrings. In *International Symposium on String Processing and Information Retrieval*, pages 290–303. Springer, 2017.
17. Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
18. Peter E Warburton, Joti Giordano, Fanny Cheung, Yefgeniy Gelfand, and Gary Benson. Inverted repeat structure of the human genome: the x-chromosome contains a preponderance of large, highly homologous inverted repeats that contain testes genes. *Genome research*, 14(10a):1861–1869, 2004.
19. Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.
20. DA Wilson and CA Thomas Jr. Palindromes in chromosomes. *Journal of molecular biology*, 84(1):115–138, 1974.

AWLCO: All-Window Length Co-Occurrence

Joshua Sobel
Department of Computer Science, University of Rochester, NY, USA

Noah Bertram
Department of Mathematics, University of Rochester, NY, USA

Chen Ding
Department of Computer Science, University of Rochester, NY, USA

Fatemeh Nargesian
Department of Computer Science, University of Rochester, NY, USA

Daniel Gildea
Department of Computer Science, University of Rochester, NY, USA

Abstract

Analyzing patterns in a sequence of events has applications in text analysis, computer programming, and genomics research. In this paper, we consider the *all-window-length* analysis model which analyzes a sequence of events with respect to windows of all lengths. We study the exact co-occurrence counting problem for the all-window-length analysis model. Our first algorithm is an offline algorithm that counts all-window-length co-occurrences by performing multiple passes over a sequence and computing single-window-length co-occurrences. This algorithm has the time complexity $O(n)$ for each window length and thus a total complexity of $O(n^2)$ and the space complexity $O(|I|)$ for a sequence of size n and an itemset of size $|I|$. We propose `AWLCO`, an online algorithm that computes all-window-length co-occurrences in a single pass with the time complexity of $O(n)$ and space complexity of $O(\sqrt{n|I|})$, assuming perfect hashing. Following this, we generalize our use case to patterns in which we propose an algorithm that computes all-window-length co-occurrence with time complexity $O(n|I|)$, assuming perfect hashing, with an additional pre-processing step and space complexity $O(\sqrt{n|I|} + |I|)$, plus the overhead of the Aho-Corasick algorithm [3].

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases Itemsets, Data Sequences, Co-occurrence

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.24

Related Version *Full Version*: `https://arxiv.org/abs/2011.14460`

Funding Partially funded by NSF awards CCF-1717877, IIS-1813823, CCF-1934986 and CNS-1909099.

Acknowledgements We would like to give special thanks to Lu Zhang and Katherine Seeman for their efforts for the implementation and experimental evaluation of our algorithms. We thank one of our reviewers for suggesting the Aho-Corasick algorithm.

1 Introduction

Analyzing regularities in streams and event sequences has applications in data analytics as well as programming languages, natural language processing, and genomics. Examples of an event sequence include a sequence of system logs, memory requests by a program, tweets by a user, a series of symptoms, a sequence of words in a document, or an RNA sequence. One metric of regularity is *co-occurrence* [13, 21] – the number of times that an entire set of items or more broadly of patterns is contained within a sliding window of an arbitrary size. For example, consider the sequence "*abccba*" and window size three. This sequence

of events contains four such windows: "*abc*", "*bcc*", "*ccb*", and "*cba*". We see that both "*a*" and "*b*" appear together in two windows. Thus, itemset $\{a, b\}$ co-occurs twice for window size of three. In the sequence "*cat dog cat*" with window size seven, we see that the words, referred to as patterns, "*cat*" and "*dog*" both appear as substrings in two windows, and thus the pattern set $\{cat, dog\}$ has co-occurrence of two with window size seven.

Most applications assume that the window is given by a user or defined in an ad hoc manner. Existing counting algorithms for streams often assume the *sliding-window* model of computation, that is, answering queries or mining is done over the last w most recent data elements [6, 7]. Successful pattern-searching tools, such as *ShapeSearch*, enable the search for desired patterns within a fixed window size in trendlines [20]. However, in certain applications of co-occurrence analysis, the query is about identifying the time windows that satisfy certain conditions on the co-occurrence. For instance, in text analysis, what is the time window in which a set of events are very likely to appear? Or, at which time window does the co-occurrence of a set of words in a document become random? Or, how often do two or multiple gene expression patterns co-occur in an RNA sequence? These applications require the analysis of all possible window lengths, possibly as large as the size of the sequence.

The All-Window-Length Analysis Model. In this paper, we consider a new analysis model of computation for streams and sequences, the *all-window-length analysis model*, where the analysis of a sequence of data elements is done in *one pass* for all window lengths, starting from the size of a pattern up to the size of a sequence. Unlike single-window-length analysis, in this model, window length becomes a variable. We consider the co-occurrence counting of items and patterns in this analysis model. A pattern is a string with characters drawn from alphabet $\mathbb{A}$. Given a sequence T of size n, and an itemset I consisting of patterns, find the number of windows in which every pattern in I occurs for all window lengths $x \in \{1, \ldots, n\}$ in T. This model enables us to perform analysis without apriori knowledge of window-size, i.e., a window size can be chosen and analyzed on demand at query time. For a sequence T of size n and an itemset I consisting of $|I|$ unique tokens, the co-occurrence analysis considers $\sum_{x=1}^{n}(n - x + 1)$ windows. We propose efficient exact algorithms and theoretical analysis for the co-occurrence counting of sets of items and patterns under this analysis model. Note that this analysis model is different from the setting of counting frequent itemset in a stream, in which data elements arrive in baskets of arbitrary lengths, and the goal is to find the itemset that appears in s fraction of the baskets, where s is a support threshold [1, 2, 14, 17].

Applications. We expect the all-window-length analysis model to open research opportunities that lead to solving problems in natural language processing, the optimization of the memory layout of programs, and accelerating the search for RNA sequences in genomes. In natural language processing, the co-occurrence of words within a sliding window is the basis for training word embeddings, which are vector representations of a word's meaning and usage [15, 16]. Different window sizes are useful for different purposes; embeddings derived from smaller windows tend to represent syntactic information while larger windows represent semantic information [18]. Identifying an effective window length for training word embeddings requires the efficient exploration of the relationship between window size and co-occurrence frequency of words [12].

The application of all-window-length co-occurrence analysis in programming languages is in the optimization of the memory layout of programs. Modern processor performance is dependent on cache performance and cache block utilization. A set of data elements belong to the same *affinity group* if they are always accessed close to each other. This closeness is defined by k-linkedness. A reference affinity forms a unique partition of data for every

k, and the relation between different ks is hierarchical, meaning the affinity groups at link length k are a finer partition of the groups at $k+1$. Reference affinity has been used to optimize the memory layout in data structure splitting [24], whole-program code layout [10], and both [22]. Finding affinity groups requires the analysis of the access co-occurrence of data elements in memory access traces for all ks.

Research has shown that analyzing nucleotide co-occurrence over the entire human genome provides a powerful insight into the evolution of viruses [8, 19]. Co-occurrence is a method for tracking cooperative genomic interactions as a major force underlying virus evolution. Existing co-occurrence network construction tools such as `cooccurNet` [25] consider pairs of nucleotides or amino acids for analysis and apply filters on the significance of the co-occurrence of genes. The distance in a co-occurrence network counts for the relatedness of genes. An all-window-length analysis of the co-occurrence gene sequences provides further insight into pattern analysis in genomics.

Results. In this paper, we propose an efficient algorithm that computes all-window-length exact co-occurrence of patterns in a single pass. For co-occurrence of itemsets of size one or two, our past work proposed a linear time algorithm (in sequence length) to compute co-occurrence for all window lengths [13]. To analyze co-occurrence, first, we introduce an algorithm to calculate co-occurrence that runs in $O(n)$ time, is easily understood, and uses $O(|I|)$ space for single-window-length co-occurrence, where n is the length of the sequence, and I is the set of co-occurring items. However, to find the co-occurrence across all window lengths, the algorithm would require to compute the co-occurrence for each window length separately and use $O(n^2)$ time, which is impractical for large datasets.

We propose `AWLCO`, a time- and space-efficient algorithm that computes the exact co-occurrence of itemsets for all window lengths, in a single pass. The algorithm computes co-occurrence by finding *gaps* in the sequence, or substrings of the sequence that do not contain subsets of the queried pattern. This is a novel approach to compute co-occurrence and provides an improved algorithm, since the stored gaps are not bound to any window lengths. Thus, the collection of gaps allows the co-occurrence to be determined for all window lengths in a single pass through the gaps. Furthermore, we propose a simple approach for computing all of the gaps for an itemset in a single pass through the sequence. The relevant gaps can be found by iterating through the sequence and keeping track of the items and the orders they last appeared. We theoretically prove that gaps are only relevant and counted if the current item encountered in the sequence is the item that was seen furthest in the past, thus, drastically reducing the amount of space and updates needed. `AWLCO` enables all-window-length queries in $O(n)$ time by using $O(\sqrt{n|I|})$ additional space, assuming a perfect hashing function.

Finally, we generalize our problem to finding the co-occurrence of a set of patterns. We argue that finding an algorithm that handles multiple elements at the same index of a sequence would solve all window length pattern co-occurrence. We present an algorithm for pattern co-occurrence counting with time complexity $O(n|I|)$, assuming perfect hashing, and space complexity $O(\sqrt{n|I|} + |I|)$, with additional space overhead from the Aho-Corasick algorithm [3].

2 Problem Definition

We begin by fixing a vocabulary $\mathbb{A}$ that we will be working in. Let T be a sequence with elements in $\mathbb{A}$. Sequence T can be considered as a stream. Let n be the length of the sequence T and for any natural number l, let $[l] = \{1, \ldots, l\}$. A sequence will have its indices zero

indexed, i.e., $T[0]$ is the first element that appeared in the sequence and $T[i]$ is the element that appeared at position i. We use $T[i \dots j]$ to denote a sub-string of T. For example, $T[0 \dots j]$ indicates the first $j+1$ elements of sequence T. An itemset I is a finite non-empty subset of $\mathbb{A}$. For a sequence T, a window is a sub-string of T, or a contiguous selection of elements of T. For sequence T we define the window at index i of length x where $x \leq i+1$, $\omega(T, i, x)$, to be the window containing the i-th element of T and the $x-1$ previous elements of T. When it is clear what sequence is being referenced, we will refer simply to $\omega(i, x)$. For example, for the sequence $T =$"$abcdef$", $\omega(3,3)$ is "bcd". We define the co-occurrence count as the number of windows of length x in sequence T that contain all elements of the itemset I.

▶ **Definition 1.** Single-window length co-occurrence problem*: Given a sequence T and an itemset I, find the co-occurrence count of itemset I in windows of length x in sequence T.*

$$\textit{co-occurrence}\,(T, I, x) = |\{\omega(i, x) \,:\, i \in \{x-1, \dots, n-1\}, \forall e \in I, e \in \omega(i, x)\}| \quad (1)$$

▶ **Example 2.** Consider the sequence T="abcabe". The co-occurrence count of itemset $\{a, b\}$ in all windows with size four, $\textit{co-occurrence}\,(abcabe, \{a, b\}, 4)$, is three.

In this paper, we consider the new problem of finding co-occurrence counts of I in T for all window lengths.

▶ **Definition 3.** All-window length co-occurrence problem*: Given a sequence T of size n, and an itemset I, find the co-occurrence counts of itemset I in all windows of lengths $x \in \{|I|, \dots, n\}$ in sequence T.*

In Section 3, we define a baseline algorithm for finding all window length co-occurrence counts based on finding the single window length co-occurrence count. In Section 4, we describe our algorithm for simultaneously finding co-occurrence counts of all window lengths in linear time in the length of the sequence assuming perfect hashing and with space complexity of $O(\sqrt{n|I|})$.

A pattern is a string with characters drawn from alphabet $\mathbb{A}$. A pattern e's ith component is denoted $e[j]$ and the length of the pattern is $|e|$. A pattern occurs in a sequence T if there exists $j \in \{0, ..., n\}$ such that for all $i \in \{0, \dots, |e|-1\}$, $T[j+i] = e[i]$.

▶ **Definition 4.** All-window length pattern co-occurrence problem*: Given a sequence T of length n, and an itemset I consisting of patterns, find the number of windows in which every pattern in I occurs for all window lengths $x \in \{1, \dots, n\}$ in sequence T.*

3 Single-Window-Length Co-occurrence

Consider an item $e \in \mathbb{A}$ and a sequence T. The time elapsed since last access of e at index i, $\text{tesla}(T, e, i)$, is the difference between i and the greatest index where e occurs in T up to and possibly including i, and, in the case that there is no occurrence of e in the interval up to i, we define it to be ∞. When the choice of T is clear we use the shorthand $\text{tesla}(e, i)$ instead. There is a direct connection between the tesla values for items in the itemset and the number of times the items of the itemset co-occur.

▶ **Lemma 5.** *Itemset I co-occurs in a window $\omega(i, x)$ if and only if* $\max\{\text{tesla}(e, i) | e \in I\} < x$.

Proof. The statement implies that for each $e \in I$, $\text{tesla}(e, i) < x$, which implies that $e \in \omega(i, x)$. Conversely, if each $e \in \omega(i, x)$, then we have $\text{tesla}(e, i) < x$; therefore, we have $\max\{tesla(e, i) | e \in I\} < x$. ◀

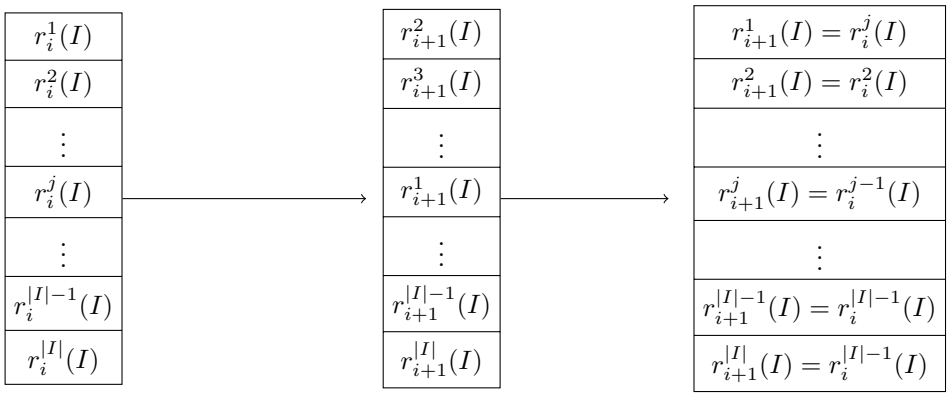


Figure 1 The book-stack, when $T[i+1] = r_i^j(I)$. This change is shown in the first two book-stacks. The third reflects the book-stack at index $i+1$ after it has been updated.

▶ **Example 6.** Consider the sequence $T =$"*abcabe*" and itemset $\{a, b\}$. Suppose we have processed $T[0 \dots 3]$ and we know $\text{tesla}(a, 3) = 0$ and $\text{tesla}(b, 3) = 2$. Since the max tesla value is two, the itemset does not co-occur in the size two window $\omega(3, 2)$.

By the lemma, the co-occurrence defined in Equation 1 can be computed by iterating through each index of the sequence and counting the number of times $\max\{\text{tesla}(e, i) | e \in I\} < x$.

$$\textit{co-occurrence}\,(T, I, x) = |\{i \in \{x-1, \dots, n-1\} : \max\{\text{tesla}(e, i) | e \in I\} < x\}| \quad (2)$$

Book Stack. We now wish to have a systematic way of ordering items according to their corresponding time elapsed since last access. Let Q denote the set of non-empty subsets of $\mathbb{A}$. Let A be some element of Q and suppose that $A = \{e^1, e^2, \dots, e^{|A|}\}$, and they are labeled in such a way that at index i in our sequence,

$$\text{tesla}(e^1, i-1) \leq \text{tesla}(e^2, i-1) \leq \cdots \leq \text{tesla}(e^{|A|}, i-1).$$

Now let $r_i^j : Q \to \mathbb{A}$ be given by $r_i^j(A) = e^j$, for $j \in \{1, ..., |A|\}$. That is to say that, r arranges the members of A in a finite sequence according to $\text{tesla}(\cdot, i-1)$. This notation is robust as it allows for weak ordering and will be used to consider a generalized case later on. We call the realization of r_i^j a *book-stack*, i.e., $\mathcal{S}_i = [(r_i^1(I), \text{tesla}(r_i^1(I), i)), \dots, (r_i^{|I|}(I), \text{tesla}(r_i^{|I|}(I), i))]$ based on the above ordering, given a set A. We define $\mathcal{S}_i.\texttt{retrieve}(j) = \text{tesla}(r_i^j, i)$. We define $\mathcal{S}_i.\texttt{find} : A \to \{1, \dots, |A|\}$, such that $\texttt{find}(a) = j$, where $r_i^j(A) = a$. We define $\mathcal{S}_i.\texttt{update} : \{1, \dots, |A|\} \to \times_{l=1}^{|A|} A$, in which $\mathcal{S}_i.\texttt{update}(j) = (r_{i+1}^1(A), \dots, r_{i+1}^{|A|}(A))$, where we have

$$r_{i+1}^l(A) = \begin{cases} r_i^j(A), & l = 1 \\ r_i^{l+1}(A), & 1 \leq l < j \\ r_i^j(A), & j < l \leq |A|. \end{cases}$$

We therefore define $\mathcal{S}_{i+1} = \mathcal{S}_i.\texttt{update}(\mathcal{S}_i.\texttt{find}(T[i]))$. It is straightforward to see that the `update` guarantees the correct ordering for r_{i+1}^j based on $\text{tesla}(\cdot, i)$. Figure 1 illustrates `update` to a book-stack data structure step by step. By an abuse of notation, in our algorithms we refer to $\mathcal{S}_i$ with $\mathcal{S}$.

Algorithm 1, SINGLECOUNTING demonstrates co-occurrence count for a specific window length. The co-occurrences of an itemset can be calculated for multiple window lengths by repeating Algorithm 1 and varying the argument x.

Algorithm 1 SINGLECOUNTING.

Input: Sequence T of length n, Itemset I, Window Length x
Result: *co-occurrence*(T, I, x)

```
count←0
S ←empty book-stack
for each item e ∈ I do
    S += (e, −∞)
end
for i = 0 to n − 1 do
    if T[i] ∈ I then
        j ← S.find(T[i])
        S.update(j)
    end
    if i ≥ x − 1 and i - S.retrieve(|I|)< x then
        count←count+1
    end
end
return count
```

▶ **Example 7.** Consider the sequence $T = abcabe$ and the itemset $I = \{a, b\}$. The algorithm initializes the $\mathcal{S}$ by adding $(e, -\infty)$ for each item e in I, representing that element e has never been seen. Table 1 shows the state of $\mathcal{S}$ and the resultant max tesla value every time an element of T is processed. At any step the max tesla value can be found by taking the current index in the sequence and subtracting the last access time of the item in the bottom of the book-stack.

3.1 Complexity Analysis

The book-stack can be implemented as a doubly linked list of items. Finding elements on the bottom of the book-stack can then be done in constant time. We can maintain a hash table from each element to the corresponding node in the book-stack. Each node can be accessed in constant time. The book-stack will only take $|I|$ space and no additional space is needed, thus the total space is $O(|I|)$. In addition, each element of the sequence is accessed once, and only constant time operations are performed, giving a time complexity of $O(n)$. For co-occurrence of a single window length, this algorithm performs optimally with respect to time complexity. This is because there is an intrinsic linear cost in computing co-occurrence, as each element in the sequence must be examined in the worst case. In the next section, we present a solution that in linear time can calculate the co-occurrence for all window lengths.

4 All Window-Length Co-occurrence

4.1 Counting Co-occurring Windows

To find the co-occurrence of an itemset $I = \{e_1, e_2, \ldots, e_{|I|}\}$ in sequence T with window length x we must count how many x-length windows in T contain I. We will make use of the fact that counting the windows containing I is equivalent to counting the windows that

Table 1 Book Stack changes for single-window co-occurrence counting.

initial	a (i=0)	b (i=1)	c (i=2)	a (i=3)	b (i=4)	e (i=5)
max tesla	$0-(-\infty)=\infty$	$1-0=1$	$2-0=2$	$3-1=2$	$4-3=1$	$5-3=2$
$a(-\infty)$	$a(0)$	$b(1)$	$b(1)$	$a(3)$	$b(4)$	$b(4)$
$b(-\infty)$	$b(-\infty)$	$a(0)$	$a(0)$	$b(1)$	$a(3)$	$a(3)$

Figure 2 Gaps for certain elements in a sequence. The uppermost pattern illustrates the three gaps "a"-gaps, the middle pattern shows the "b"-gaps, and the bottom pattern shows the three gaps that contain neither "a" nor "b".

do not contain I, since we know the total number of x-length windows is $n-x+1$. For a sequence T of length n and an itemset $\{e_1\}$ we denote the x-length windows that do not contain e_1 as $\overline{\{e_1\}}_x$. For larger itemsets we extend the notation analogously where $\overline{\{e_1,e_2\}}_x$ are the x-length windows that do not contain e_1 and do not contain e_2. A window is a *non co-occurrent* window as long as there is at least one element in I that is not contained in the window. Therefore, the co-occurrence of I is the total number of x-length windows minus the number of x-length windows that do not contain at least one item of I.

$$\textit{co-occurrence}\,(T,I,x) = (n-x+1) - |\overline{\{e_1\}}_x \cup \ldots \cup \overline{\{e_{|I|}\}}_x| \tag{3}$$

Using the inclusion-exclusion principle we can rewrite the co-occurrence as follows.

$$\textit{co-occurrence}\,(T,I,x) = (n-x+1) - \sum_{\substack{A\subseteq I:\\ A\neq\emptyset}} (-1)^{|A|+1}\,|\overline{A}_x| \tag{4}$$

▶ **Example 8.** Consider again figure 2 with $I=\{a,b\}$ and $x=2$. We have that $\overline{\{a\}}_2 \cup \overline{\{b\}}_2 = \{w(i,2) : 1\le i\le 3, i=5, 7\le i\le 10\}$, hence $\left|\overline{\{a\}}_2 \cup \overline{\{b\}}_2\right| = 8$. The nonempty subsets of I are $\{a,b\}$, $\{a\}$, and $\{b\}$. We have that $\overline{\{a,b\}}_2 = \{w(8,2)\}$, $\overline{\{a\}}_2 = \{w(3,2), w(7,2), w(8,2)\}$, and $\overline{\{b\}}_2 = \{w(i,2) : 1\le i\le 2, i=5, 8\le i\le 10\}$. This means that $\left|\overline{\{a\}}_2 + \overline{\{b\}}_2 - \overline{\{a,b\}}_2\right| = 8$, agreeing with $\left|\overline{\{a\}}_2 \cup \overline{\{b\}}_2\right|$. The idea here is that when using $\overline{\{a\}}_2$ and $\overline{\{b\}}_2$ to count $\overline{\{a\}}_2 \cup \overline{\{b\}}_2$ we double count $w(8,2)$, so we need to subtract by 1 to get the correct result.

We know that an A-gap of size k contains $k-x+1$ windows of length x in which none of A occurs. Thus, $|\overline{A}_x| = \sum_{k=x}^{n}(k-x+1)N_A(k)$, where $N_A(k)$ is the number of A-gaps of length k. Working with the right term of equation (4),

$$\sum_{\substack{A\subseteq I:\\ A\neq\emptyset}} (-1)^{|A|+1}\,|\overline{A}_x| = \sum_{\substack{A\subseteq I:\\ A\neq\emptyset}} (-1)^{|A|+1} \sum_{k=x}^{n}(k-x+1)N_A(k) \tag{5}$$

$$= \sum_{k=x}^{n}(k-x+1) \sum_{\substack{A\subseteq I:\\ A\neq\emptyset}} (-1)^{|A|+1} N_A(k). \tag{6}$$

Now, let us define:

$$H[k] = \sum_{\substack{A \subseteq I: \\ A \neq \emptyset}} (-1)^{|A|+1} N_A(k) \tag{7}$$

We call the collection of $H[k]$'s for all values of k a *gap histogram*, H. The co-occurrence of I in x-length windows of sequence T is then calculated as follows.

$$\textit{co-occurrence}\,(T, I, x) = (n - x + 1) - \sum_{k=x}^{n} (k - x + 1) H[k] \tag{8}$$

An elegant property of this equation is that by storing the cumulative counts in a gap histogram we can simultaneously calculate the co-occurrence for all window lengths.

▶ **Example 9.** Return to the sequence shown in Figure 2 for $I = \{a, b\}$ and $x = 2$. Because there are two length one $\{a, b\}$-gaps and one length one $\{b\}$-gap, we obtain $H[1] = -2+1 = -1$. Additionally we have one length two $\{a, b\}$-gap, one length two $\{a\}$-gap, and one length two $\{b\}$-gap which nets $H[2] = 1 + 1 - 1 = 1$. There is one length three $\{a\}$-gap and one length three $\{b\}$-gap, so $H[3] = 1 + 1 = 2$. Finally, there is a single length four $\{b\}$-gap so $H[4] = 1$. To summarize, $H[i] = 1$ for $i = 2, 4$, $H[3] = 2$, $H[1] = -1$, and $H[i] = 0$ otherwise.

Using gap histograms to store cumulative counts has a space complexity of $\sqrt{n|I|}$. In Theorem 12, we will formally discuss the space complexity in more detail. Calculating the co-occurrence from the gap histogram instead of directly counting co-occurrent windows is beneficial since calculating gaps does not require a window length as input and yet the gap information is still sufficient to easily calculate the co-occurrence for all window lengths. Thus, all that is needed to calculate all window length co-occurrence is an algorithm to generate the gap histogram.

The simplest way to generate the gap histogram is to iterate through the sequence, keeping track of where gaps begin and end. Whenever an item in the given itemset is found at some index i it marks the end of a gap for any subset of I containing that item and also marks the beginning of a new gap spanning from $T[i + 1...k - 1]$, where k is either the index of the next occurrence of an element in the subset of I in the sequence or n if another element does not occur before the end of the sequence. Note that if an element in I occurs in two adjacent indices in the sequence (i and $i + 1$), we obtain the gap $[i + 1, i]$ which we treat as a length 0 gap and discard. The length l of each newly ended gap can be updated in the histogram by either incrementing or decrementing $H[l]$ depending on whether the subset size was odd or even respectively. This approach has run time $O(n2^{|I|})$ and performs poorly for large itemsets. It is inefficient since whenever an item from I is encountered in the sequence, we need to consider $2^{|I|-1}$ subsets of I and update the histogram (subtract or add counts) accordingly. A better approach is presented next.

4.2 Efficient Gap Counting

Since updates to the histogram have negating effects on each other (Equation 7), many of the histogram entries do not change when an item of the itemset is observed in the sequence. It turns out when an item of the itemset is observed in a sequence, we only need to update the histogram for the gaps related to the *first and second least recently seen items* of I. To keep track of the tesla's, as we iterate through the sequence we can maintain a book-stack data structure that contains each item in I along with the time that it last appeared in the sequence, so that the most recently seen item appears at the top of book-stack.

Observe that, when an item e from I is seen in the sequence at index i, a maximal gap representing each subset of I containing e is added to the histogram. Furthermore, for any one of those sets G, the length of the added gap is the minimum tesla value attained by an item in G at index $i-1$. Note that in this context we take $\text{tesla}(e,i) = i$ if the element has not yet been encountered in the sequence. These gaps account for all of the gaps in the sequence except for gaps that include the final element of the sequence, these gaps are handled specially.

For the following theorem, we first provide some notation. Let

$$H_i = (H_i[1], H_i[2], \ldots, H_i[n])$$

be the histogram up to index i in the sequence. This is only for notational convenience. It is important to note that in the actual program, there is only one histogram, rather than n.

▶ **Theorem 10.** *For any $0 < i < n$, suppose $T[i] = r_i^{|I|}(I)$ then*

$$H_i[k] = \begin{cases} H_{i-1}[k] + 1 & \text{if } k = \text{tesla}(r_i^{|I|}(I), i-1) \\ H_{i-1}[k] - 1 & \text{if } k = \text{tesla}(r_i^{|I|-1}(I), i-1) \\ H_{i-1}[k] & \text{otherwise.} \end{cases}$$

If $T[i] \neq r_i^{|I|}(I)$ then $H_i[k] = H_{i-1}[k]$ for all k. In other words, the histogram is only updated when the next element in the sequence is the item that was just at the bottom of the book-stack.

Proof Sketch. We have a maximal gap for every subset A of I containing $T[i]$. The length of this A-gap is $\min_{e\in A} \text{tesla}(e, i-1)$, hence the addition to the histogram from A is $(-1)^{|A|+1}$ to the kth spot where $k = \min_{e\in A} \text{tesla}(e, i-1)$. Suppose $T[i] \neq \arg\max_{e\in I} \text{tesla}(e, i-1)$ i.e., $T[i]$ is not the item seen furthest in the past most recently. There are the same number of even and odd subsets of I in which $T[i] = \arg\min_{e\in A} \text{tesla}(e, i-1)$ hence these subsets contribute no net updates to H. For the remaining subsets, the same argument follows, hence there are no net updates.

Now suppose that $T[i] = \arg\max_{e\in I} \text{tesla}(e, i-1)$. Similar to the above, for each item in I not equal to $r_i^{|I|-1}(I)$ and $T[i]$, there are the same number of even and odd subsets of I in which $T[i] = \arg\min_{e\in A} \text{tesla}(e, i-1)$. But for $r_i^{|I|-1}(I)$ there is but one subset in which this is satisfied, namely, $\{r_i^{|I|-1}(I), T[i]\}$, and there is also one subset in which $T[i]$ satisfies this, $\{T[i]\}$. Therefore we have

$$H_i[\text{tesla}(T[i], i-1)] = H_{i-1}[\text{tesla}(T[i], i-1)] + 1,$$
$$H_i[\text{tesla}(r_i^{|I|-1}(I), i-1)] = H_{i-1}[\text{tesla}(r_i^{|I|}(I), i-1)] - 1. \quad \blacktriangleleft$$

The theorem does not handle the case for H_n, which we now address. The argument is similar to the proof above for H_i with $i < n$, except that $T[i]$ is undefined. All gaps necessarily close at the end of the sequence. This means that $|C_k| = \sum_{\ell=0}^{|I|} \binom{|I|-j}{\ell}$, for all but $j = |I|$. For $j = |I|$ there is but one set for which $r_i^{|I|}(I) = \arg\min_{e\in A} \text{tesla}(e, n)$, namely, $\{r_i^{|I|}(I)\}$. Thus $H_n[k] = H_{n-1}[k]$ for all k except when $k = \text{tesla}(n, i-1)$ in which $H_n[k] = H_{n-1}[k] - 1$.

The incremental updates that we have derived above result in algorithm `AWLCO`, shown in Algorithm 2.

4.3 Complexity Analysis

The next two theorems assume that the histogram can be implemented as a hashtable with perfect hashing. Without perfect hashing the histogram must contain space for all entries from $1 - n$ and thus will be linear in space to maintain a constant run time or constant histogram updates must be sacrificed to obtain a worst case n^2 runtime.

▶ **Theorem 11** (Time Complexity). *The time complexity of all-window length co-occurrence algorithm is linear in the length of the sequence.*

Proof. The algorithm iterates over the sequence once and possibly updates the book-stack and the histogram for each element in the sequence. Since updating the book-stack and updating the histogram are both done in constant time, the generation of the histogram is done in linear time in the length of the sequence. Once a histogram is computed, the co-occurrence for every window length is computed in a linear time by summing the histogram as shown in Equation 8. Thus, the algorithm provides an $O(n)$ method to calculate all window length co-occurrence. ◀

▶ **Theorem 12** (Space Complexity). *The space complexity of the algorithm is $O(\sqrt{n|I|})$ where n is the length of the sequence and $|I|$ is the size of the itemset.*

Proof. Space is used to maintain the book-stack and the histogram. The book-stack will use $O(|I|)$ space. Note that for any item e in the itemset the total length of gaps for $\{e\}$ is at most the length of the sequence. Thus, we have that the sum of all of the lengths of single-item gaps is bounded above by $n|I|$. Furthermore, whenever an item of the itemset is on the bottom of the book-stack a maximum of two new gaps are added to the histogram. The length of the gap associated to the bottom item in the book-stack is equal to the length of a single-item gap. The length of the other gap is bounded above by the length of the first gap. Therefore, the sum of the length of all gaps added to the histogram is bounded above by $2n|I|$. Note the size of the histogram is the number of distinct gap lengths added to it. In the worst case, gaps are greedily added to the histogram such that there is a length $1, 2, \ldots, k$ size gap added. In this case, if the total number of gaps added is k, the total length of the gaps is $\frac{k(k+1)}{2}$. We know that the sum of the gaps length in a histogram is bounded by $2n|I|$. Thus, we have that $\frac{k(k+1)}{2} \leq 2n|I|$. Solving for k, we have that $k^2 + k \leq 4n|I|$ and $k \leq 2\sqrt{n|I|}$. Thus, the total space used is bounded above by $|I| + 2\sqrt{n|I|}$ which gives a space complexity of $O(\sqrt{n|I|})$. Note that this last line is true because we make the assumption $|I| \leq n$. If not the co-occurrence is simply 0 for every window length. ◀

5 Pattern Co-occurrence

We now wish to generalize our algorithm in two ways. The first is to patterns and the second is to a stream in which multiple events can occur at the same index, for which the latter turns out to be a subproblem of the first. Pattern co-occurrence is explained first. We define a pattern as a string with characters drawn from our alphabet $\mathbb{A}$. A pattern e's ith component is denoted $e[i]$ and the length of the pattern is $|e|$. A pattern occurs in a sequence T if there exists $j \in [|T|]$ such that $T[j \ldots j + |e| - 1] = e$, also let all such j be denoted in the set $b(e)$. Thus, pattern co-occurrence for an itemset I is defined as the number of windows in which every pattern in I occurs. We wish to find an algorithm that can compute the co-occurrence for all window lengths in one pass for patterns. It is clear that tesla is no longer well-defined. Let e be a pattern. So define $\text{btesla}(e, i) = i - \max(|b(e) \cap \{0, \ldots, i\}|)$, which is the distance between i and the most recent start of the pattern. If $b(e) \cap \{0, \ldots, i\}$ is empty, then let it be i.

We can use our previous definition of an A-gap for $A \subseteq I$, but the size of an A gap is now found differently. Previously, the size of an A-gap closed at time i would be $\min_{e \in A} \text{tesla}(e, i - 1)$, but now it is $\min_{e \in A} \text{btesla}(e, i - 1)$, since an A-gap still occurs if all but the tail ends of members of A are within said gap. Supposing that no two patterns in consideration end at

Algorithm 2 `AWLCO`.

Input: Sequence T, ItemSet I
Result: Co-occurrence of all window lengths
$H \leftarrow$ empty histogram
$cooc \leftarrow []$
$\mathcal{S} \leftarrow$ empty book-stack
for $e \in I$ **do**
  $\mathcal{S}$ += (e, $-\infty$)
end
// Read through entire sequence
for $i = 0$ *to* $n - 1$ **do**
  current$\leftarrow T[i]$
  // When element is seen, update bottom two gaps
  if $current \in I$ **then**
    if $\mathcal{S}$.find($current$) $= |I|$ **then**
      $f \leftarrow i - \mathcal{S}$.retrieve($|I|$)
      $s \leftarrow i - \mathcal{S}$.retrieve($|I| - 1$)
      $H[f] \leftarrow H[f] + 1$
      $H[s] \leftarrow H[s] - 1$
    end
    j $\leftarrow \mathcal{S}$.find(current)
    $\mathcal{S}$.update(j)
  end
end
// Final gap from bottom of book-stack
$f \leftarrow i - \mathcal{S}$.retrieve($|I|$)
$H[f] \leftarrow H[f] + 1$
for $x = |I|$ *to* $|T| - |I| + 1$ **do**
  $S_x \leftarrow 0$
  for $k = x$ *to* $|T|$ **do**
    $S_x \leftarrow S_x + (k - x + 1)H[k]$
  end
  $cooc[x] \leftarrow (|T| - x + 1) - S_x$
end
return $cooc$

the same time, it is easy to see that Theorem 10 still holds in this case, using btesla in place of tesla. Thus finding an algorithm that handles multiple events at the same index would solve all window length pattern co-occurrence as well. We now proceed to solve the problem in the case that multiple events can occur at the same index.

5.1 Multiple Item Co-occurrence

It is now natural to define co-occurrence for sets of items. We let $T[i] \subseteq \mathbb{A}$, rather than just one element of $\mathbb{A}$, for all i. A co-occurring window for some itemset $I \subseteq \mathbb{A}$ is a window in which for all $e \in I$, there exists a set $A \in w(i, x)$ such that $e \in A$. Thus the co-occurrence is the sum of these co-occurring windows. This is the natural extension. We will now present the following theorem relating to the updates of H. Let X_i denote the set of items that occur at $T[i]$.

$T[i-k_1]$		$T[i-k_2]$			$T[i]$
x_1	x_2	x_3	y_3	x_4	x_1
		y_1			x_2
		y_2			x_3
					x_4
					x_5

Figure 3 Illustration of Theorem 13. The set of patterns I consists of $x_1, x_2, \ldots$, which were seen at $T[i]$, and all other patterns $y_1, y_2, \ldots$. Here A is the set $\{x_1, x_2\}$ of patterns seen at $T[i]$ that were last seen further in the past than any of the other patterns $y_1, y_2, \ldots$. We add one to $H[k_1]$, where k_1 is the time elapsed since x_1 was last seen. We subtract one from $H[k_2]$, where k_2 is the time elapsed since y_1 was last seen.

▶ **Theorem 13.** *Suppose that for all* $a \in I \cap X_i$, $\text{tesla}(a, i-1) < \text{tesla}(e, i-1)$ *for any* $e \in I \setminus X_i$. *Then for any* $1 < i < n$,

$$H_i[k] = \begin{cases} H_{i-1}[k] + 1 \text{ for } k = \text{tesla}(r_i^{|I \setminus X_i|}(I \setminus X_i), i-1) \\ H_{i-1}[k] - 1 \text{ for } k = \text{tesla}(r_i^{|I|}(I), i-1) \\ H_{i-1}[k] \text{ otherwise.} \end{cases}$$

Otherwise, $H_i[k] = H_{i-1}[k]$ *for all* k.

▶ Remark 14. This is a generalization of Theorem 10. Observe that in the case when $|A| = 1$, this reduces to that result. Moreover, when $i = n$ the same update given in Theorem 10 follows.

Proof Sketch. The incremental updates to H correspond to all the subsets of I that contain at least one member of X_i. Weakly order X_i according to tesla. Now let A_j where $j \in [X_i]$, be the set of subsets of I that contains x_j. Let $\langle A \rangle_i$ be the updates corresponding to some set A. Therefore

$$H_i - H_{i-1} = \sum_{J \subseteq [|K|]} (-1)^{|J|+1} \left\langle \bigcap_{j \in J} A_j \right\rangle.$$

Consider each $\mathcal{K}_J = \bigcap_{i \in J} A_i$. For each one, the update is the same if one removes all members of X_i besides the one corresponding to the smallest number in J, call this set $\mathcal{K}'_J$. Using Theorem 10, the update is $+1$ for $k = \text{tesla}(a, i-1)$, a being the item described before, and also is -1 for $k = \text{tesla}(a, i-1)$ for a being the furthest item seen in the past not in $\mathcal{K}_J \cap I$. But this implies that J that are not of the form $J_m = \{|X_i| - m, \ldots, |X_i|\}$ do not contribute to the update. For any $0 \leq m < |X_i|$, the positive update corresponding to $\mathcal{K}_{J_m}$ cancels with the negative update corresponding to $\mathcal{K}_{J_{m+1}}$. This process telescopes leaving only the positive update corresponding to $\mathcal{K}_{J_0}$ and the negative update corresponding to $\mathcal{K}_{J_{|X_i|}}$. This gives the desired result. ◀

A full proof is given in the appendix. Figure 3 provides an illustration of Theorem 13. With this result we can now construct a similar algorithm to those before, with a few modifications. Maintain a book-stack as before, but notice that it is no longer a strict ordering. For example, if $X_i = \{e_1, e_2\} \subseteq I$, then one of e_1 and e_2 will occupy the top of the book-stack and the other will occupy the second to top spot. To check whether $\max_{e \in X_i \cap I} \text{tesla}(e, i) < \min_{e \in I \setminus X_i} \text{tesla}(e, i)$, we partition the book-stack using $p \in \{0, ..., |I|\}$,

where p is defined as follows: for all $j \leq p$, $\text{tesla}(r_i^j(I), i-1) = \text{tesla}(r_i^{|I|}(I), i-1)$, and for all $j > p$, $\text{tesla}(r_i^j(I), i-1) > \text{tesla}(r_i^{|I|}(I), i-1)$. Thus checking if the non-trivial conditions given in Theorem 13 hold is easy as we just check that $r_i^j(I) \leq p$ for every j corresponding to a member in X_i. It is also easy to update the histogram if these conditions hold, as we just update according to $r_j^p(I)$ and $r_j^{p+1}(I)$. The pseudocode is given in Algorithm 3 in the appendix.

5.2 Complexity Analysis

We again consider pattern co-occurrence rather than the simplified multiple item co-occurrence, to find the complexity of the algorithm in general. We complete a pass over the stream as we do in algorithm 2, but with a few additions. At each index we must maintain the partition which in the worst case takes linear scan of I at each index. Maintaining the book-stack then also requires $O(|I|)$ operations. We can use the state machine in the Aho-Corasick algorithm [3] to recognize when a pattern is completed in this same pass. For this addition, the algorithm requires an initial time linear to the sum of the lengths of all of the patterns to construct the necessary finite state machine for Aho-Corasick. Thus, the time complexity is $O(n|I|)$ with an additional pre-processing complexity of $O(\sum_{e \in I} |e|)$ due to Aho-Corasick. Space complexity for the histogram and book-stack remains $O(\sqrt{n|I|} + |I|)$[1], however additional space is now needed for user's desired implementation of the Aho-Corasick algorithm.

6 Related Work

Counting in Streams. In the count-distinct problem, the goal is to know the number of unique elements in a stream [9, 14]. In the bit-counting problem, the goal is to maintain the frequency count of 1's in the last k bits of a bit stream of size N. Datar et al. propose an approximate algorithm with for the bit-counting problem with $O(\log^2 k)$ space complexity [6]. Existing counting algorithms for streams assume the *sliding-window* model of computation, that is answering queries or mining is done over the last w elements seen so far [7]. However, `AWLCO` introduces a new analysis model – all-window-length analysis model – which is compelled to analyze and query all windows of all lengths starting from the beginning of a stream or anytime in the past. To that end, `AWLCO` presents an efficient and exact itemset counting algorithm for the all-window-length analysis model.

The frequent itemset mining in stream is a well-studied problem that adheres to the counting problem [5]. The seminal work by Manku and Motwani presents an algorithm for estimating the frequency count of itemsets in a stream and identifying those itemsets that occur in at least a fraction θ of the stream seen so far with some error parameter ϵ [14]. For example, when the input is a stream of transactions where each transaction is a set of items, the goal is to find the most frequent itemsets within transactions. The challenge is to consider variable-length itemsets and avoid the combinatorial enumeration of all possible itemsets. Many existing frequent itemset mining algorithms (with exception of [4, 11]) obtain approximate results with error bounds. A variation of frequent itemset mining is the problem of mining frequent co-occurrence patterns across multiple data streams [21]. The definition of co-occurrence patterns is slightly different than co-occurrence itemsets considered by

[1] When considering patterns rather than single elements, $|I| \leq n$ is not necessarily true, so we include an additional factor of $|I|$ for completeness.

AWLCO. A co-occurrence pattern is a group of items that appear consecutively showing tight correlations between these items. A frequent co-occurrence pattern is the pattern that appears in at least θ streams within a time period of length τ and the appearance of the pattern in each stream happens within a time window of δ or smaller. In this paper, AWLCO presents an all-window length frequency counting for a query itemset. A natural extension of the itemset frequency counting of presented by AWLCO is mining frequent itemsets in all window-lengths.

Affinity Analysis. Zhong et al. defined *reference affinity* for data elements on an access trace. A set of data elements belong to the same affinity group if they are always accessed close to each other [24]. The closeness is defined by $k-$linked-ness. They proved that reference affinity forms a unique partition of data for every k, and the relation between different ks is hierarchical, i.e. the affinity groups at link length k are a finer partition of the groups at $k+1$. This definition requires *strict* co-occurrence in that every occurrence of a group element must be accompanied by all other elements of the group. *Weak reference affinity* [23] introduces a second parameter, affinity threshold. It adheres to the unique and hierarchical partition properties with respect to both parameters. Zhang et al. showed that neither strict reference affinity nor weak reference affinity can efficiently be computed [22]. Thus they gave a heuristic solution and adapted it to use sampling. The average time complexity of their algorithm is $O(N\delta\omega^2 + N\delta\pi)$, where N is the length of the trace, δ is the sampling rate, ω is the size of the affinity group, and π is the average time length of windows containing accesses to all members of the group ω. Lavaee et al. gave an $O(L\delta\omega^2)$ algorithm to compute the affinity for all sub-groups of sizes up to ω [10]. Reference affinity has been used to optimize the memory layout in data structure splitting [24], whole-program code layout [10], and both [22].

7 Discussion and Future Work

Applications. The all-window-length co-occurrence has applications in text analysis, the optimization of the memory layout of programs, and accelerating the search for RNA sequences in genomes. In terms of practical applications, our plan is to develop interactive tools that enable the exploration of sequences of events and genomics data. Projects such as cooccurNet [25] provide a basis that can be extended with all-window-length co-occurrence analysis functionalities.

Mining Problems. In this paper, we expounded co-occurrence counting of itemsets and patterns in the all-window-length analysis model. Going forward, we study mining algorithms in this analysis model, including mining frequent closed itemsets, i.e., given a sequence T find the top-k itemsets that have highest co-occurrences in an arbitrary window size and for a frequent itemset X, there exists no super-pattern $X \subset Y$, with the same co-occurrence as X. The algorithm requires to mine frequent itemsets for all window lengths in one pass.

Extending to Timestamped Sequences. The proposed algorithms operate on a sequence of data points taken at equally spaced points in time. Thus, our sequences are discrete-time data. We plan to study co-occurrence counting and frequent itemset mining in a series of data points indexed in continuous time order. In the continuous setting, we define $T = \{(e, \omega) : e \in I, \omega \in [0, \tau]\}$, where $\tau \geq 0$, to be a set of timestamps in consideration. We

can now consider the co-occurrence of items and patterns over an interval of time, which can now be considered as events not items. We wish to compute the probability of a groups of events happening in time scale r:

$$\Pr(I \in [a,b], T, r) = \Pr_{y \sim U([a,b])}(\forall e \in I, \exists (e,\omega) \in T : |y - \omega| < r).$$

This naturally leads to an analytic definition and suggests a continuous analog of co-occurrence.

References

1. Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 207–216, 1993. doi:10.1145/170036.170072.
2. Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994. doi:10.5555/645920.672836.
3. 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.
4. Joong Hyuk Chang and Won Suk Lee. Finding recent frequent itemsets adaptively over online data streams. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 487–492, 2003. doi:10.1145/956750.956807.
5. Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1530–1541, 2008. doi:10.14778/1454159.1454225.
6. Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002. doi:10.1137/S0097539701398363.
7. Mayur Datar and Rajeev Motwani. The sliding-window computation model and results. In *Data Stream Management - Processing High-Speed Data Streams*, pages 149–165. Springer, 2016. doi:10.1007/978-0-387-47534-9_8.
8. Xiangjun Du, Zhuo Wang, Aiping Wu, Lin Song, Yang Cao, Haiying Hang, and Taijiao Jiang. Networks of genomic co-occurrence capture characteristics of human influenza A (H3N2) evolution. *Genome Research*, 18(1):178–187, January 2008. doi:10.1101/gr.6969007.
9. Philippe Flajolet and G. Nigel Martin. Probabilistic counting. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 76–82, 1983. doi:10.1109/SFCS.1983.46.
10. Rahman Lavaee, John Criswell, and Chen Ding. Codestitcher: inter-procedural basic block layout optimization. In *Proceedings of the International Conference on Compiler Construction*, pages 65–75, 2019. doi:10.1145/3302516.3307358.
11. Carson Kai-Sang Leung and Quamrul I. Khan. DSTree: A tree structure for the mining of frequent sets from data streams. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 928–932, 2006. doi:10.1109/ICDM.2006.62.
12. Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proceedings of the 52nd Anuual Meeting of the Association for Computational Linguistics (ACL)*, pages 302–308, 2014. doi:10.3115/v1/P14-2050.
13. Yumeng (Lucinda) Liu, Daniel Busaba, Chen Ding, and Daniel Gildea. All timescale window co-occurrence: Efficient analysis and a possible use. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, pages 289–292, Riverton, NJ, USA, 2018. IBM Corp. doi:10.5555/3291291.3291322.
14. Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 346–357, 2002. doi:10.1016/B978-155860869-6/50038-X.

15 Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, pages 3111–3119, 2013. doi:10.5555/2999792.2999959.

16 Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, 2014.

17 Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman. *Mining Massive Datasets*. Cambridge University Press, 2014. doi:10.5555/2124405.

18 Hinrich Schütze. *Ambiguity Resolution in Language Learning – Computational and Cognitive Models*. Number 10 in CSLI Lecture Notes Series. Center for the Study of Language and Information, Stanford, California, 1997.

19 Jason W. Shapiro and Catherine Putonti. Gene co-occurrence networks reflect bacteriophage ecology and evolution. *mBio*, 9(2), 2018. doi:10.1128/mBio.01870-17.

20 Tarique Siddiqui, Paul Luh, Zesheng Wang, Karrie Karahalios, and Aditya G. Parameswaran. Shapesearch: A flexible and efficient system for shape-based exploration of trendlines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 51–65, 2020. doi:10.1145/3318464.3389722.

21 Ziqiang Yu, Xiaohui Yu, Yang Liu, Wenzhu Li, and Jian Pei. Mining frequent co-occurrence patterns across multiple data streams. In *International Conference on Extending Database Technology (EDBT)*, pages 73–84, 2015. doi:10.5441/002/edbt.2015.08.

22 Chengliang Zhang, Chen Ding, Mitsunori Ogihara, Yutao Zhong, and Youfeng Wu. A hierarchical model of data locality. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–29, 2006. doi:10.1145/1111037.1111040.

23 Chengliang Zhang, Yutao Zhong, Chen Ding, and Mitsunori Ogihara. Finding reference affinity groups in trace using sampling method. Technical report, Department of Computer Science, University of Rochester, 2004.

24 Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–266, 2004. doi:10.1145/996841.996872.

25 Yuanqiang Zou, Zhiqiang Wu, Lizong Deng, Aiping Wu, Fan Wu, Kenli Li, Taijiao Jiang, and Yousong Peng. cooccurNet: an R package for co-occurrence network construction and analysis. *Bioinformatics*, 33(12):1881–1882, 2017. doi:10.1093/bioinformatics/btx062.

A Appendix

Proof of Theorem 10

Proof. We have a maximal gap for every subset of I containing $T[i]$. This collection of subsets can be written as

$$C = \{A \subseteq I | A = \{T[i]\} \cup B, B \subseteq I \setminus \{T[i]\}\}.$$

For each $A \in C$, the update to H is $(-1)^{|A|+1}$ to $H\left[\text{tesla}(r_i^1(A), i-1)\right]$ as we have found an A-gap of size $\text{tesla}(r_i^1(A), i-1)$ at index i. Let $C_k = \{A \in C | \text{tesla}(r_i^1(A), i-1) = k\}$. The incremental updates can be expressed by

$$H_i[k] = H_{i-1}[k] + \sum_{A \in C_k} (-1)^{|A|+1}, \tag{9}$$

for each k. Suppose $T[i] = r_i^{j_0}(I)$ where $j_0 < |I|$, i.e., $T[i]$ is not the item seen furthest in the past most recently. Then there are $\binom{|I|-j}{\ell}$ sets $A \in C$ of length $\ell + 1$ in which $T[i] = r_i^1(A)$. Thus for $k = \text{tesla}(T[i], i-1)$, we have

$$H_i[k] - H_{i-1}[k] = \sum_{A \in C_k} (-1)^{|A|+1} = \sum_{\ell=0}^{|I|-j} \binom{|I|-j}{\ell} (-1)^{\ell} = (1-1)^{|I|-j} = 0. \tag{10}$$

Now for every $j < j_0$ (which means that j never equals $|I|-1$ in this case), we have that there are $\binom{|I|-j-1}{\ell}$ members $A \in C$ of length $\ell+2$ in which, $r_i^j(I) = r_i^1(A)$. Therefore for $k = \text{tesla}(r_i^j(I), i-1)$,

$$H_i[k] - H_{i-1}[k] = \sum_{A \in C_k} (-1)^{|A|+1} = \sum_{\ell=0}^{|I|-j-1} \binom{|I|-j-1}{\ell} (-1)^{(\ell+1)} = -(1-1)^{|I|-j-1} = 0.$$

But for $|I| \geq j > j_0$, there are no such sets $A \in C$ in which $r_i^j(I) = r_i^1(A)$, as $T[i] = r_i^{j_0}(I)$ is contained in all $A \in C$.

But if $j_0 = |I|$, i.e., $T[i] = r_i^{|I|}(I)$, then for each $j < |I|-1$, there are again $\binom{|I|-j-1}{\ell}$ members $A \in C$ of length $\ell+2$ in which, $r_i^j(I) = r_i^1(A)$, so again equation (10) holds for $k = \text{tesla}(r_i^j(I), i-1)$, giving no net updates for such k. But there is exactly one $A \in C$ in which $r_i^{|I|-1}(I) = r_i^1(A)$, namely, $\{r_i^{|I|-1}(I), T[i]\}$, and there is also one $A \in C$ in which $T[i] = r_i^{|I|}(I) = r_i^1(A)$, which is $\{T[i]\}$. Therefore we have

$$H_i[\text{tesla}(r_i^{|I|}(I), i-1)] = H_{i-1}[\text{tesla}(r_i^{|I|}(I), i-1)] + 1,$$
$$H_i[\text{tesla}(r_i^{|I|-1}(I), i-1)] = H_{i-1}[\text{tesla}(r_i^{|I|}(I), i-1)] - 1. \quad \blacktriangleleft$$

Proof of Theorem 13

Proof. Suppose without loss of generality that $X_i \subseteq I$. We wish to find $H_i - H_{i-1}$. Denote

$$X_i = \{r_i^1(X_i), r_i^2(X_i), \ldots, r_i^{|X_i|}(X_i)\} = \{x_1, x_2, \ldots, x_{|X_i|}\},$$

as r defined before. Now let

$$U_i = \{A \subseteq I : A = B \cup \{x_j\}, B \subseteq I \setminus \{x_j\}, j \in [|X_i|]\},$$

which in words, is all subsets of I that contain at least one member of X_i. Observe that

$$U_i = \bigcup_{j=1}^{|X_i|} \{A \subseteq I : A = B \cup \{x_j\}, B \subseteq I \setminus \{x_j\}\}.$$

Now let $K_j = \{A \subseteq I : A = B \cup \{x_j\}, B \subseteq I \setminus \{x_j\}\}$ for all j. Therefore $U_i = \bigcup_{j=1}^{|X_i|} K_j$.

The update rule is known for each K_j based on our previous result. The remains the of the proof is as follows. We can leverage the update rule currently known to compute the total update. But the intersection of K_j's is non-empty, meaning if we update according to each K_j, we would be overcounting some members of U. Once this is determined, we will find the update rule according for each arbitrary intersection of these K_j's, which completes the proof.

Define $\langle \cdot \rangle_i$ to be a mapping from subsets of I to an integer valued n dimensional vector. $\langle A \rangle_i^k$ is the sum of the number of maximal gaps of length k ending at index i given by even subsets of A, minus the sum of the number of maximal gaps of length k ending at index i given by the odd subsets of A. Using this new definition, $\langle U_i \rangle_i^k = H_i[k] - H_{i-1}[k]$. We can now appeal to the inclusion exclusion principle to write that

$$H_i - H_{i-1} = \langle U_i \rangle = \left\langle \bigcup_{j=1}^{|X_i|} K_j \right\rangle = \sum_{J \subseteq [|X_i|]} (-1)^{|J|+1} \left\langle \bigcap_{j \in J} K_j \right\rangle. \tag{11}$$

The right hand side of the above equality will now be used.

Denote for any $J \subseteq [|X_i|]$,

$$\mathcal{K}_J = \bigcap_{j \in J} K_j.$$

Let $\mathcal{X}_J$ be the set of members of X_i that lie in every member of $\mathcal{K}_J$. Observe that

$$\mathcal{X}_J = \bigcap_{G \in \mathcal{K}_J} G.$$

It also follows that $\mathcal{X}_J = \bigcup_{j \in J} \{x_j\}$. Moreover, we can write

$$\mathcal{K}_J = \{A \subseteq I : A = \mathcal{X}_J \cup B, B \subseteq I \setminus \mathcal{X}_J\}.$$

For each set $A \in \mathcal{K}_J$, there is a corresponding set A' in $\mathcal{K}'_J = \{A \subseteq I : A = \{r_i^1(\mathcal{X}_J)\} \cup B, B \subseteq I \setminus \mathcal{X}_J\}$, in which $\langle A \rangle = (-1)^{|J|+1} \langle A' \rangle$. This correspondence is easy to find. Let $A \in \mathcal{K}_J$. Thus $A = \mathcal{X}_J \cup B$, for some $B \in I \setminus \mathcal{X}_J$. Then the corresponding set $A' \in \mathcal{K}'_J$ is $\{r_i^1(\mathcal{X}_J)\} \cup B$. This is clear, because items that lie in every $A \in \mathcal{K}_J$ that never satisfy $\arg\min_{e \in A} \text{tesla}(e, i-1)$ for all A never contribute towards any updates and hence can be ignored, except they may change the parity of the set and hence change the sign of the update. From here, we can apply the first theorem taking I in that theorem to be $I \setminus \mathcal{X}_J$, which gives

$$\langle \mathcal{K}_J \rangle_i^k = (-1)^{|J|+1} \langle \mathcal{K}'_J \rangle_i^k = (-1)^{|J|+1} \begin{cases} 1 \text{ for } k = \text{tesla}(r_i^{|I \setminus \mathcal{X}_J|}((I \setminus \mathcal{X}_J) \cup \{x^*\}), i-1) \\ -1 \text{ for } k = \text{tesla}(r_i^1(\mathcal{X}_J), i-1) \\ 0 \text{ otherwise,} \end{cases} \tag{12}$$

when $r_i^1(\mathcal{X}_J) = r_i^{|I \setminus \mathcal{X}_J|+1}((I \setminus \mathcal{X}_J) \cup \{r_i^1(\mathcal{X}_J)\})$. Every update is 0 otherwise.

Now assume for all $x \in X_i$ and $e \in I \setminus X_i$, $\text{tesla}(x, i-1) \geq \text{tesla}(e, i-1)$. For if this does not hold for some $x' \in X_i$, then by the above, no updates occur due to x', so analysis is the same.

We now wish to compute the right hand side of equation (11). We can employ equation (12) for each $\mathcal{K}_J$. If $r_i^1(\mathcal{X}_J) \neq r_i^{|I \setminus \mathcal{X}_J|+1}((I \setminus \mathcal{X}_J) \cup \{r_i^1(\mathcal{X}_J)\})$, that is, the first ranked item of $\mathcal{X}_J$ is not ranked below all of $I \setminus \mathcal{X}_J$, then $\langle \mathcal{K}_J \rangle = \mathbf{0}$. We claim that the J in which $\langle \mathcal{K}_J \rangle \neq \mathbf{0}$ are of the following form:

$$J_m = \{|X_i| - b : b \in [m]\}, \tag{13}$$

for $0 \leq m < |X_i|$. We first show that if $J \neq J_m$ for some m, then $\langle \mathcal{K}_J \rangle = \mathbf{0}$. If $J \neq J_m$ for some m, then there exists b_0 such that $r_i^{|X_i|-b_0}(X_i) \notin \mathcal{X}_J$, and there is some b_1 such that $b_1 > b_0$ and $r_i^{|X_i|-b_1}(X_i) \in \mathcal{X}_J$. Since $b_1 \leq |X_i| - 1$, $b_0 < |X_i| - 1$ which gives that $|X_i| - b_0 > 1$. Let c_0 and c_1 be such that $r_i^{c_0}((I \setminus \mathcal{X}_J) \cup \{r_i^1(\mathcal{X}_J)\}) = r_i^{|X_i|-b_0}(X_i)$ and $r_i^{c_1}((I \setminus \mathcal{X}_J) \cup \{r_i^1(\mathcal{X}_J)\}) = r_i^1(\mathcal{X}_J)$. We have that $c_0 > c_1$. Now since $c_0 \leq |I \setminus \mathcal{X}_J| + 1$, $c_1 \neq |I \setminus X_J| + 1$. Therefore $r_i^1(\mathcal{X}_J) \neq r_i^{|I \setminus X_J|+1}((I \setminus \mathcal{X}_J) \cup \{r_i^1(\mathcal{X}_J)\})$, hence $\langle \mathcal{K}_J \rangle = \mathbf{0}$.

Now suppose that $J = J_m$ for some m. Let c_1 be such that $r_i^{c_1}(I) = r_i^1(\mathcal{X}_J)$. We then have that for any $c < c_1$, $c \in J$, moreover, $r_i^c(I) \notin (I \setminus \mathcal{X}_J) \cup \{r_i^1(\mathcal{X}_J)\}$. Thus $r_i^1(\mathcal{X}_J) = r_i^{|I \setminus \mathcal{X}_J|+1}((I \setminus \mathcal{X}_J) \cup \{r_i^1(\mathcal{X}_J)\})$, for if not, then there would be $c_0 > c_1$ in which $r_i^{c_0}(I) \in (I \setminus \mathcal{X}_J) \cup \{r_i^1(\mathcal{X}_J)\}$, a contradiction.

From this, the right hand side of equation (11) becomes

$$\sum_{J\subseteq[|X_i|]} (-1)^{|J|+1} \langle \mathcal{K}_J \rangle = \sum_{m=0}^{|X_i|-1} (-1)^m \langle \mathcal{K}_{J_m} \rangle . \tag{14}$$

We now have for $J = J_m$, $r_i^1(\mathcal{X}_J) = r_i^{|X_i|-m}(X_i)$. Also when $m < |X_i| - 1$, we have that

$$r_i^{|I\setminus\mathcal{X}_J|}(I \setminus \mathcal{X}_J \cup \{r_i^1(\mathcal{X}_J)\}) = r_i^{|I\setminus\mathcal{X}_J|}(I \setminus \mathcal{X}_J) = r_i^{|X_i|-(m+1)}(X_i). \tag{15}$$

But when $m = |X_i| - 1$, $J = [|X_i|]$, therefore

$$r_i^{|I\setminus\mathcal{X}_J|}(I \setminus \mathcal{X}_J \cup \{r_i^1(\mathcal{X}_J)\}) = r_i^{|I\setminus\mathcal{X}_J|}(I \setminus \mathcal{X}_J) = r_i^{|I\setminus X_i|}(I \setminus X_i). \tag{16}$$

Now for $m < |X_i| - 1$, we can rewrite equation (11) to get

$$\langle \mathcal{K}_{J_m} \rangle_i^k = (-1)^m \begin{cases} 1 \text{ for } k = \text{tesla}(r_i^{|X_i|-(m+1)}(X_i), i-1) \\ -1 \text{ for } k = \text{tesla}(r_i^{|X_i|-m}(X_i), i-1) \\ 0 \text{ otherwise.} \end{cases} \tag{17}$$

Now define

$$u(m)_i^k = \begin{cases} 1 \text{ for } k = \text{tesla}(r_i^{|X_i|-(m+1)}(X_i), i-1) \\ -1 \text{ for } k = \text{tesla}(r_i^{|X_i|-m}(X_i), i-1) \\ 0 \text{ otherwise,} \end{cases} \tag{18}$$

for $m < |X_i| - 1$ and

$$u(|X_i|-1)_i^k = \begin{cases} 1 \text{ for } k = \text{tesla}(r_i^{|I\setminus X_i|}(I \setminus X_i), i-1) \\ -1 \text{ for } k = \text{tesla}(r_i^1(X_i), i-1) \\ 0 \text{ otherwise,} \end{cases} \tag{19}$$

Taking $u(m)_i = (u_i^1(m), u_i^2(m), \ldots, u_i^n(m))$, we can write

$$\sum_{m=0}^{|X_i|-1} (-1)^m \langle \mathcal{K}_{J_m} \rangle = \sum_{m=0}^{|X_i|-1} (-1)^m (-1)^m u(m)_i^k = \sum_{m=0}^{|X_i|-1} u(m)_i^k. \tag{20}$$

Observe that

$$u(0)_i^k + u(1)_i^k = \begin{cases} 1 \text{ for } k = \text{tesla}(r_i^{|X_i|-2}(X_i), i-1) \\ -1 \text{ for } k = \text{tesla}(r_i^{|X_i|}(X_i), i-1) \\ 0 \text{ otherwise.} \end{cases} \tag{21}$$

Applying this for all $m < |X_i| - 1$ gives

$$\sum_{m=0}^{|X_i|-2} u(m)_i^k = \begin{cases} 1 \text{ for } k = \text{tesla}(r_i^1(X_i), i-1) \\ -1 \text{ for } k = \text{tesla}(r_i^{|X_i|}(X_i), i-1) \\ 0 \text{ otherwise.} \end{cases} \tag{22}$$

So combining this with $u(|X_i| - 1)_i^k$, we get

$$\sum_{m=0}^{|X_i|-1} u(m)_i^k = \begin{cases} 1 \text{ for } k = \text{tesla}(r_i^{|I\setminus X_i|}(I \setminus X_i), i-1) \\ -1 \text{ for } k = \text{tesla}(r_i^{|I|}(I), i-1) \\ 0 \text{ otherwise,} \end{cases} \tag{23}$$

since $r_i^{|X_i|}(X_i) = r_i^{|I|}(I)$. Combining equation (23) with equations (20), (14), and (11) (and considering the components of each of those equations), we finally get,

$$H_i[k] - H_{i-1}[k] = \begin{cases} 1 \text{ for } k = \text{tesla}(r_i^{|I \setminus X_i|}(I \setminus X_i), i-1) \\ -1 \text{ for } k = \text{tesla}(r_i^{|I|}(I), i-1) \\ 0 \text{ otherwise,} \end{cases} \tag{24}$$

proving the result ($X_i = A$ in the statement of the theorem). ◀

Algorithms

On the following page, we give the pseudocode for the algorithm PAWLCO.

Algorithm 3 `PAWLCO`.

Input: Trace T, ItemSet I
Result: Co-occurrence of all window lengths

$H \leftarrow$ empty histogram, $cooc \leftarrow []$, $\mathcal{S} \leftarrow$ empty book-stack
; **for** $e \in I$ **do**
  $\mathcal{S}$ += (e, $-\infty$)
end
$p \leftarrow |I|$, $m \leftarrow 1$
while $\mathcal{S}$.retrieve($|I|$) = $\mathcal{S}$.retrieve($|I| - m$) **do**
  $m \leftarrow m + 1$
end
$p \leftarrow |I| - m$
for $i = 0$ *to* $n - 1$ **do**
  $C \leftarrow \{e \in I \,|\, e[0] = T[i - |e| + 1]...e[|e| - 1] = T[i]\}$
  min $\leftarrow i$ **for** $c \in C$ **do**
    if $\mathcal{S}$.find(c) $< min$ **then**
      min $\leftarrow \mathcal{S}$.find(c)
    end
  end
  if $min > \mathcal{S}$.retrieve($p + 1$) **then**
    $f \leftarrow i - \mathcal{S}$.retrieve($|I|$), $s \leftarrow i - \mathcal{S}$.retrieve($p + 1$)
    $H[f] \leftarrow H[f] + 1$, $H[s] \leftarrow H[s] - 1$
  end
  for *current* $\in C$ **do**
    j $\leftarrow \mathcal{S}$.find(current)
    $\mathcal{S}$.update(j)
    `// Maintain partition`
    if $j = p = |I|$ **then**
      $m \leftarrow 2$
      while $\mathcal{S}$.retrieve*(/I/-1)* = $\mathcal{S}$.retrieve*(/I/-m)* **do**
        $m \leftarrow m + 1$
      end
      $p \leftarrow |I| - m$
    end
    if $p \leq j < |I|$ **then**
      $p \leftarrow p + 1$
    end
  end
end
`// Final gap from bottom of book-stack`
$f \leftarrow i - \mathcal{S}$.retrieve(|I|)
$H[f] \leftarrow H[f] + 1$
for $x = 0$ *to* $|T| - |I| + 1$ **do**
  $S_x \leftarrow 0$
  for $k = x$ *to* $|T|$ **do**
    $S_x \leftarrow S_x + (k - x + 1)H[k]$
  end
  $cooc[x] \leftarrow (|T| - x + 1) - S_x$
end
return *cooc*

Optimal Completion and Comparison of Incomplete Phylogenetic Trees Under Robinson-Foulds Distance

Keegan Yao
Department of Computer Science and Engineering, University of Connecticut, Storrs, CT, USA

Mukul S. Bansal
Department of Computer Science and Engineering, University of Connecticut, Storrs, CT, USA

Abstract

The comparison of phylogenetic trees is a fundamental task in phylogenetics and evolutionary biology. In many cases, these comparisons involve trees inferred on the same set of leaves, and many distance measures exist to facilitate such comparisons. However, several applications in phylogenetics require the comparison of trees that have non-identical leaf sets. The traditional approach for handling such comparisons is to first restrict the two trees being compared to just their common leaf set. An alternative, conceptually superior approach that has shown promise is to first *complete* the trees by adding missing leaves so that the completed trees have identical leaf sets. This alternative approach requires the computation of optimal completions of the two trees that minimize the distance between them. However, no polynomial-time algorithms currently exist for this optimal completion problem under any standard phylogenetic distance measure.

In this work, we provide the first polynomial-time algorithms for the above problem under the widely used Robinson-Foulds (RF) distance measure. This hitherto unsolved problem is referred to as the *RF(+) problem*. We (i) show that a recently proposed linear-time algorithm for a restricted version of the RF(+) problem is a 2-approximation for the RF(+) problem, and (ii) provide an exact $O(nk^2)$-time algorithm for the RF(+) problem, where n is the total number of distinct leaf labels in the two trees being compared and k, bounded above by n, depends on the topologies and leaf set overlap of the two trees. Our results hold for both rooted and unrooted binary trees.

We implemented our exact algorithm and applied it to several biological datasets. Our results show that completion-based RF distance can lead to very different inferences regarding phylogenetic similarity compared to traditional RF distance. An open-source implementation of our algorithms is freely available from `https://compbio.engr.uconn.edu/software/RF_plus`.

2012 ACM Subject Classification Applied computing → Molecular evolution; Mathematics of computing → Combinatorial optimization; Theory of computation → Dynamic programming; Mathematics of computing → Trees

Keywords and phrases Phylogenetic tree comparison, Robinson-Foulds Distance, Optimal tree completion, Algorithms, Dynamic programming

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.25

Supplementary Material *Software (Source Code)*:
`https://compbio.engr.uconn.edu/software/RF_plus`

Funding *Keegan Yao*: University of Connecticut Summer Undergraduate Research Fund
Mukul S. Bansal: US National Science Foundation award IIS 1553421

1 Introduction

Phylogenetic trees, or simply *phylogenies*, are leaf-labeled trees that depict the evolutionary relationships between different species, genes, or other biological entities such as cells in an organism or individuals from a population. In phylogenetic trees, leaf nodes represent extant entities while internal nodes represent hypothetical ancestors. Many different methodologies,

algorithms, and data types exist for estimating phylogenies, and there is often considerable uncertainty and error in their inference, with different methods or data types suggesting different evolutionary relationships between the same extant entities. Many distance (or similarity) measures have therefore been developed for systematically comparing different phylogenetic trees, including the widely used Robinson-Foulds distance [29], triplet and quartet distances [14, 17], nearest neighbor interchange (NNI) and subtree prune and regraft (SPR) distances [31, 18, 34], maximum agreement subtrees [19, 2, 15], nodal distance [9], geodesic distance [23] and others. However, these distance measures implicitly assume that the two trees being compared have identical leaf sets, an assumption that is often violated in practice. Indeed, several applications, such as supertree construction [24, 6, 10, 32, 1], phylogenetic database search [28, 30, 11, 25], and clustering of phylogenies [20, 35], require the computation of distances between trees with partially overlapping leaf sets.

The traditional approach to comparing two trees with only partially overlapping leaf sets is to first restrict (i.e., prune down) both trees to their shared leaf set. This restriction based approach, though simple to conceptualize and compute, can result in the loss of valuable topological information through scrapping of leaves that are not common to both trees. An alternative approach to comparing trees with non-identical leaf sets is to *complete* or *fill in* each of the input trees to the union of their leaf sets in a way which minimizes the distance between them, and then compute their distance. This approach, though conceptually more complex, successfully incorporates all topological information in both the trees being compared. In addition to its more complete use of topological information, the completion based approach also has the benefit of a larger range of attainable values due to comparisons over larger extended trees rather than smaller induced trees. Despite these advantages, no polynomial-time algorithms currently exist for completion based comparison under any standard phylogenetic distance measure. In this work, we provide the first polynomial-time algorithms for optimal completion and comparison of incomplete phylogenetic trees under the widely used Robinson-Foulds (RF) distance measure. Following existing literature [4], we refer to completion based RF distance as *RF(+)*, the traditional restriction based RF distance as *RF(-)*, and the problem of computing the RF(+) distance between two trees as the *RF(+) problem*. Figure 1 illustrates the difference between RF(-) and RF(+) distances.

Previous work. The idea of completion based Robinson-Foulds distance arose at least a decade ago when Cotton and Wilkinson introduced majority-rule supertrees [13] and defined two variants, majority-rule(-) and majority-rule(+) supertrees, based on RF(-) and RF(+), respectively. Completion based majority-rule(+) supertrees and some variants were subsequently shown to have many desirable properties [16]. Later, Kupczok [22] characterized the RF(+) distance for the restricted special case where the leaf set of one tree is a subset of the leaf set of the other in terms of incompatible splits between the two trees. For this restricted special case, referred to as the *One Tree RF(+) (OT-RF(+))* problem [4], an $O(n^2)$-time algorithm was proposed by Christensen et. al. in 2017 [12], where n is the total number of distinct leaf labels in the two trees being compared. More recently, Bansal proposed an optimal $O(n)$-time algorithm for this OT-RF(+) problem [3, 4]. Bansal also proposed a restricted formulation of the RF(+) problem, called the *Extraneous-Clade-Free RF(+) (EF-RF(+)) problem*, which is based on computing optimal completions that avoid the creation of *any* subtrees formed by joining together two subtrees unique to each one of the two input trees. Essentially, the EF-RF(+) problem disallows certain types of completions; specifically, it ignores how subtrees exclusive to one input tree impact the overall optimal position where subtrees from the other input tree should be added. Bansal showed that the EF-RF(+) problem can be solved in $O(n)$ time [4]. These linear-time algorithms for the OT-RF(+) and EF-RF(+) problems are applicable to both rooted and unrooted trees.

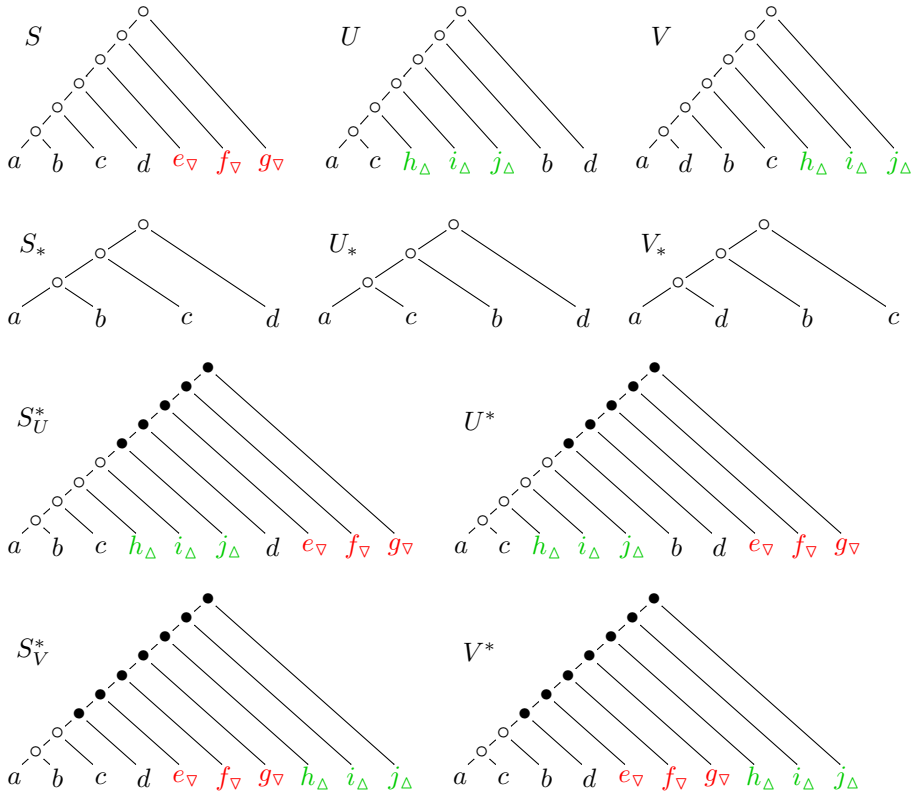


Figure 1 RF(-) and RF(+) distances. The figure shows a "base" tree S and two other trees U and V, with $Le(U) = Le(V)$, being compared to S. S_*, U_* and V_* represent the trees S, U and V, respectively, when restricted to the common leaf set. U^* and V^* are the optimal RF(+) completions of U and V with respect to S. S^*_U and S^*_V are the optimal RF(+) completions of S with respect to U and V, respectively. Filled in nodes represent matched nodes (Definition 2.2). Here, $RF(S_*, U_*) = 2$ and $RF(S_*, V_*) = 4$ while $RF(S^*_U, U^*) = 8$ and $RF(S^*_V, V^*) = 4$. Thus, in this example, U is closer to S than V under RF(-) but V is closer to S than U under RF(+).

Our Contributions. In this work, we provide the first polynomial-time algorithms for the RF(+) problem for both rooted and unrooted trees. Specifically, we make the following contributions: First, we show that the EF-RF(+) distance between two trees is a 2-approximation for the RF(+) distance between those trees. Since the EF-RF(+) problem can be solved in $O(n)$ time, this yields a linear time 2-approximation algorithm for the RF(+) problem. Second, we provide an $O(nk^2)$-time exact algorithm for the RF(+) problem, where k, bounded above by n, is the number of maximal subtrees exclusive to one input tree. And third, we perform an extensive experimental study which demonstrates that the use of RF(+) distance can lead to very different inferences regarding phylogenetic similarity compared to RF(-) distance. We also find that, in practice, EF-RF(+) distances are often very close to RF(+) distances, suggesting that the linear-time algorithm for computing EF-RF(+) distances could be an excellent heuristic for estimating RF(+) distances between large trees.

The rest of this manuscript is organized as follows: Preliminaries and problem definitions appear in the next section. We describe the linear time 2-approximation algorithm in Section 3, and the exact algorithm in Section 4. Section 5 shows how our algorithms can be

extended to unrooted trees, and Section 6 describes the results of our experimental study. Concluding remarks appear in Section 7. Proofs of all lemmas and theorems from Sections 3 and 4 appear in the Appendix.

2 Definitions and Preliminaries

We follow basic definitions and problem formulations from [4]. All trees will be unordered. Given a tree T, we denote its node set, edge set, and leaf set by $V(T)$, $E(T)$, and $Le(T)$, respectively. The set of all non-leaf (i.e., internal) nodes of T is denoted by $I(T)$. If T is rooted, the root node of T is denoted by $rt(T)$, the parent of a node $v \in V(T)$ by $pa_T(v)$, its set of children by $Ch_T(v)$, and the (maximal) subtree of T rooted at v by $T(v)$. If two nodes in T have the same parent, they are called *siblings* of each other. If $pa_T(v)$ has exactly two children, then we will denote the sibling of v as $sib_T(v)$. The *least common ancestor*, denoted $lca_T(L)$, of a set $L \subseteq Le(T)$ in T is defined to be the node $v \in V(T)$ such that $L \subseteq Le(T(v))$ and $L \not\subseteq Le(T(u))$ for any child u of v. For convenience, given a collection of vertices $a_1, \ldots, a_m$ in T, we will define $lca_T(a_1, \ldots, a_m) = lca_T(Le(T(a_1)) \cup \cdots \cup Le(T(a_m)))$. Given a rooted tree T and $a, b \in V(T)$, we say that $a \leq b$ if $a \in V(T(b))$, and $a < b$ if $a \in V(T(b))$ and $a \neq b$. A rooted tree is *binary* if all of its internal nodes have exactly two children, while an unrooted tree is *binary* if all its nodes have degree either 1 or 3. Throughout this work, the term *tree* refers to binary trees with uniquely labeled leaves.

Let T be a rooted or unrooted tree. Given a set $L \subseteq Le(T)$, let T_L be the minimal subtree of T with leaf set L. We define the *leaf induced subtree* $T[L]$ of T on leaf set L to be the tree obtained from T_L by successively removing each non-root node of degree two and adjoining its two neighbors.

▶ **Definition 2.1** (Completion of a tree). *Given a tree T and a set L' such that $Le(T) \subseteq L'$, a* completion *of T on L' is a tree T' such that $Le(T') = L'$ and $T'[Le(T)] = T$.*

If T is a rooted tree, for each node $v \in V(T)$, the *clade* $C_T(v)$ is defined to be the set of all leaf nodes in $T(v)$; i.e. $C_T(v) = Le(T(v))$. We denote the set of all clades of a rooted tree T by $Clade(T)$. This concept can be extended to unrooted trees as follows. If T is an unrooted tree, each edge $(u, v) \in E(T)$ defines a partition of the leaf set of T into two disjoint subsets $Le(T_u)$ and $Le(T_v)$, where T_u is the subtree containing node u and T_v is the subtree containing node v, obtained when edge (u, v) is removed from T. The partition induced by any edge $(u, v) \in E(T)$ is called a *split* and is represented by the set $\{Le(T_u), Le(T_v)\}$. The set of all splits in an unrooted tree T is denoted by $Split(T)$.

▶ **Definition 2.2** (Matched and mismatched nodes). *Given rooted trees S and T, and a node $v \in V(S)$, we call v a* matched node *with respect to T if $C_S(v) \in Clade(T)$, and a* mismatched node *otherwise. Analogously, $C_S(v)$ is called a* matched clade *if $C_S(v) \in Clade(T)$, and a* mismatched clade *otherwise.*

The *symmetric difference* of two sets A and B, denoted by $A \Delta B$, is the set $(A \setminus B) \cup (B \setminus A)$. We now define the Robinson-Foulds distance and the two problems that we solve in this paper.

▶ **Definition 2.3** (Robinson-Foulds distance). *The* Robinson-Foulds (RF) distance, *$RF(S, T)$, between two trees S and T is defined to be $|Clade(S) \Delta Clade(T)|$ if S and T are rooted trees, and $|Split(S) \Delta Split(T)|$ if S and T are unrooted trees.*

▶ **Problem 1** (Rooted $RF(+)$ (R-RF(+))). *Given two rooted binary trees S and T, compute a binary completion S^* of S on $Le(S) \cup Le(T)$ and a binary completion T^* of T on $Le(S) \cup Le(T)$ such that $RF(S^*, T^*)$ is minimized.*

▶ **Problem 2** (Unrooted $RF(+)$ (U-RF(+))). *Given two unrooted binary trees S and T, compute a binary completion S^* of S on $Le(S) \cup Le(T)$ and a binary completion T^* of T on $Le(S) \cup Le(T)$ such that $RF(S^*, T^*)$ is minimized.*

These problems can equivalently be viewed as maximizing the number of matched clades or minimizing the number of mismatched clades between completions of the input trees. Our algorithms for the problems above rely on first computing exact solutions for restricted variants of those problems. These restricted variants of R-RF(+) and U-RF(+) were first proposed and defined in [4] and are referred to as the *Extraneous-Clade-Free R-RF(+) (EF-R-RF(+))* and *Extraneous-Split-Free U-RF(+) (EF-U-RF(+))* problems. These restricted variants are based on computing optimal completions that do not contain *any* subtrees formed by joining together two subtrees unique to each one of the two input trees. Next, we first define *extraneous clades* and *extraneous splits*, and then state the EF-R-RF(+) and EF-U-RF(+) problems.

▶ **Definition 2.4** (Extraneous clade [4]). *Suppose S and T are rooted trees. Given completions S' and T' of S and T, respectively, on $Le(S) \cup Le(T)$, we define a clade of S' or T' to be an* extraneous clade *if it contains leaves from both S and T but no leaves that are common to S and T.*

An extraneous split is simply the analogous notion for unrooted trees and we refer the reader to [4] for a formal definition. The corresponding problem variants can now be defined as follows:

▶ **Problem 3** (Extraneous-Clade-Free R-RF(+) (EF-R-RF(+)) [4]). *Given two rooted trees S and T, compute a completion S' of S on $Le(S) \cup Le(T)$ and a completion T' of T on $Le(S) \cup Le(T)$ such that S' and T' do not contain any extraneous clades and $RF(S', T')$ is minimized.*

▶ **Problem 4** (Extraneous-Split-Free U-RF(+) (EF-U-RF(+)) [4]). *Given two unrooted trees S and T such that $|Le(S) \cap Le(T)| \geq 2$, compute a completion S' of S on $Le(S) \cup Le(T)$ and a completion T' of T on $Le(S) \cup Le(T)$ such that S' and T' do not contain any extraneous splits and $RF(S', T')$ is minimized.*

Figure 2 provides examples of completions with and without extraneous clades. Both the EF-R-RF(+) and EF-U-RF(+) problems can be solved optimally in linear time [4].

Note. In the remainder of this section, as well as in Sections 3 and 4 we focus on only the rooted version of RF(+), i.e., on the R-RF(+) problem, and implicitly assume that the two trees being compared, S and T, are rooted.

Node coloring scheme for rooted trees. For ease of presentation, we assign a color to some of the nodes of the two rooted input trees as follows. These node colorings can also be used to define red and green subtrees.

▶ **Definition 2.5** (Red and Green Nodes). *Let S and T be two arbitrary rooted trees. A node $v \in V(S)$ is called a* red node *(with respect to T) if $Le(S(v)) \subseteq Le(S) \setminus Le(T)$. Analogously, a node $v \in V(T)$ is called a* green node *(with respect to S) if $Le(T(v)) \subseteq Le(T) \setminus Le(S)$.*

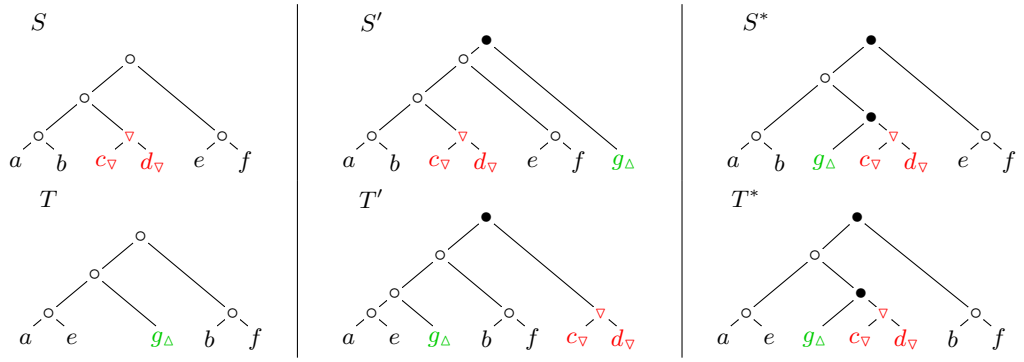


Figure 2 EF-RF(+) and RF(+) completions. S', T' are optimal EF-R-RF(+) completions (without extraneous clades) of S and T, respectively, and completions S^*, T^* are optimal RF(+) completions. Nodes labeled with downward and upward pointing triangles are red and green nodes, respectively, as defined in Definition 2.5. Filled in nodes correspond to matched clades.

▶ **Definition 2.6** (Red and Green Subtrees). *A subtree $S(u)$, where $u \in V(S)$, is called a* red subtree *of S if u is a red node. A subtree $T(u)$, where $u \in V(T)$, is called a* green subtree *of T if u is a green node. A subtree $S(u)$, where $u \in V(S)$, is called a* maximal red subtree *of S if $S(u)$ is a red subtree and either $u = rt(S)$ or $pa_S(u)$ is not red. A subtree $T(u)$, where $u \in V(T)$, is called a* maximal green subtree *of T if $T(u)$ is a green subtree and either $u = rt(T)$ or $pa_T(u)$ is not green. Note that all nodes in a red (green) subtree must be red (green).*

Under this node coloring, completing a tree S with respect to tree T entails adding all the green leaves of T into S and completing a tree T with respect to tree S entails adding, or *grafting*, all the red leaves of S into T. Importantly, as we show later in Theorem 3.1, under R-RF(+) problem, there exist optimal completions of S and T in which all grafted subtrees are maximal red or green subtrees. In other words, to optimally complete S we must only add the maximal green subtrees of T to S, and vice versa.

Notational conventions. S and T will denote the two given (input) trees to be completed/-compared. Going forward, we will generally use S' and T' to represent completions (optimal or non-optimal) with *no* extraneous clades, and S^* and T^* to represent completions that *may* include extraneous clades.

3 EF-R-RF(+) is a 2-Approximation for R-RF(+)

Observe that any optimal pair of R-RF(+) completions can be modified into a pair of (not necessarily optimal) EF-R-RF(+) completions by breaking apart any existing extraneous clades and reinserting the red/green leaves in a manner that avoids forming extraneous clades. In this section, we will show how to perform such a modification of optimal R-RF(+) completions so that the resulting increase in RF distance is appropriately bounded. This will establish that EF-R-RF(+) distance is a 2-approximation for R-RF(+) distance and will yield a linear-time 2-approximation algorithm for the R-RF(+) problem. We will first establish the presence of *canonical* optimal R-RF(+) completions that satisfy some desirable structural properties.

Notation and terminology. Given completions S^* and T^* of S and T, if there exists an extraneous clade $C_{T^*}(v)$ for some vertex $v \in T^*$, then we will call the subtree $T^*(v)$ an *extraneous subtree*. If the children s and t of v satisfy $C_{T^*}(s) \in \mathit{Clade}(S)$ and $C_{T^*}(t) \in \mathit{Clade}(T)$, then we will denote the extraneous subtree by $\{s,t\}$. To simplify notation, we will write $pa_{T^*}\{s,t\}$ to express the parent $pa_{T^*}(lca_{T^*}(s,t))$ of the root node of the extraneous subtree $\{s,t\}$ in completion T^*. Likewise, we will write $sib_{T^*}\{s,t\}$ to express $sib_{T^*}(lca_{T^*}(s,t))$, i.e., the sibling of the root node of extraneous subtree $\{s,t\}$ in T^*.

Next, we show that there always exists an optimal pair of R-RF(+) completions in which all extraneous clades are of the form $\{s,t\}$, and any such extraneous clade appears in *both* completions. We refer to such optimal R-RF completions S^* and T^* of S and T as *canonical optimal R-RF(+) completions.*

▶ **Theorem 3.1.** *Let S and T be rooted binary trees. Then, there exist optimal completions S^* and T^* under the R-RF(+) problem with the following properties:*

1. *Every subtree inserted into S^* is a maximal green subtree of T, and every subtree inserted into T^* is a maximal red subtree of S,*
2. *Every extraneous subtree in S^* and T^* is of the form $\{s,t\}$, where s is the root of a maximal red subtree in S and t is the root of a maximal green subtree in T,*
3. *Every extraneous subtree $\{s,t\}$ which is a subtree of S^* is also a subtree of T^* and vice versa.*

Decomposition of canonical optimal R-RF(+) completions. Given an extraneous subtree $\{s,t\}$ in canonical optimal R-RF(+) completions S^*, T^* of S and T, where $s \in V(S)$ and $t \in V(T)$, we define a *decomposition* of the extraneous subtree $\{s,t\}$ as a modification of the completions S^* and T^*, yielding new completions S' and T' with strictly fewer extraneous subtrees, as follows:

1. If either none or both of the nodes $pa_{S^*}\{s,t\}$ and $pa_{T^*}\{s,t\}$ are matches (in S^* and T^*), then the decomposition occurs as described below.
   - In tree T^*, prune out the grafted subtree $S(s)$ and regraft it at the parent edge of node $sib_{T^*}\{s,t\}$.
   - In tree S^*, prune out the grafted subtree $T(t)$ and regraft it at the parent edge of node $pa_{S^*}\{s,t\}$. If $pa_{S^*}\{s,t\} = rt(S^*)$, then create a new root node with children t and $pa_{S^*}\{s,t\}$.
2. Otherwise, if exactly one of the nodes $pa_{S^*}\{s,t\}$ and $pa_{T^*}\{s,t\}$ is a matched node (in S^* and T^*), then the decomposition occurs as described below. Without loss of generality, assume that $pa_{S^*}\{s,t\}$ is a match and $pa_{T^*}\{s,t\}$ a mismatch.
   - In tree S^*, prune out the grafted subtree $T(t)$ and regraft it at the parent edge of node $sib_{S^*}\{s,t\}$.
   - In tree T^*, prune out the grafted subtree $S(s)$ and regraft it at the parent edge of that unique node $u \in V(T^*)$ for which $C_{T^*}(u) = C_{S^*}(pa_{S^*}\{s,t\})$. If $u = rt(S^*)$, then create a new root node with children s and $pa_{S^*}\{s,t\}$. Note that u must exist since $pa_{S^*}\{s,t\}$ is a matched node.

This decomposition is illustrated in Figure 3. The following lemma characterises how the RF distance between S^* and T^* is impacted as their extraneous subtrees are decomposed.

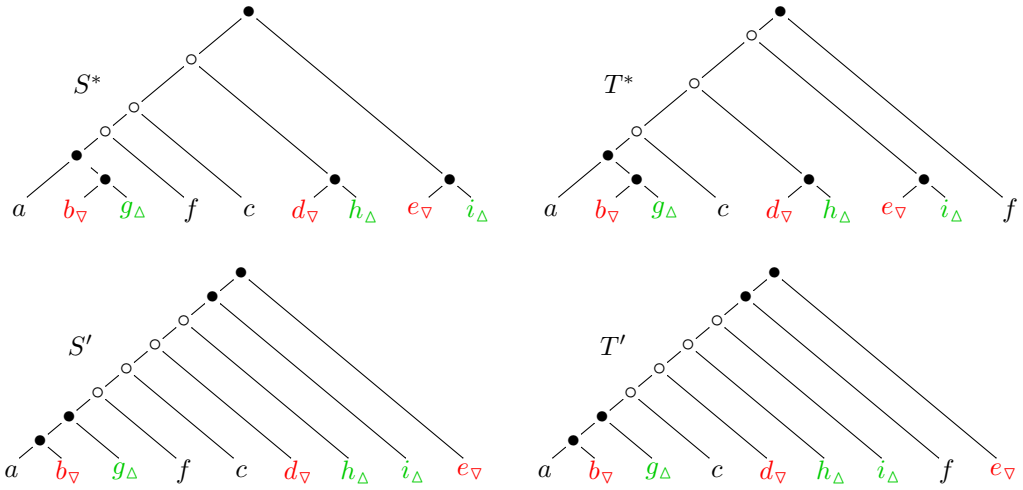


Figure 3 Decomposition of extraneous clades. Shown here is a decomposition of completions S^* and T^* into completions S' and T'. Nodes labeled with downward and upward pointing triangles are red and green nodes, respectively. Extraneous subtree $\{b, g\}$ is of type 1 where both parents match, extraneous subtree $\{d, h\}$ is of type 1 where neither parent is a match, and extraneous subtree $\{e, i\}$ is of type 2. Matches between corresponding completions are denoted by filled in nodes.

▶ **Lemma 3.2.** *Let S' and T' denote the trees obtained by decomposing extraneous subtree $\{s, t\}$ in completions S^* and T^*, respectively.*

1. *If $pa_{S^*}\{s, t\}$ and $pa_{T^*}\{s, t\}$ are both matched nodes then $RF(S', T') = RF(S^*, T^*)$.*
2. *If exactly one of $pa_{S^*}\{s, t\}$ and $pa_{T^*}\{s, t\}$ is a matched node then $RF(S', T') = RF(S^*, T^*)$.*
3. *If neither $pa_{S^*}\{s, t\}$ nor $pa_{T^*}\{s, t\}$ is a matched node then $RF(S', T') = RF(S^*, T^*) + 2$.*

The 2-approximation now follows by appropriately bounding the number of extraneous subtrees $\{s, t\}$ that fall in category 3 of the above lemma.

▶ **Theorem 3.3.** *Let S^* and T^* represent optimal completions of S and T, respectively, under the R-RF(+) problem. Let S' and T' represent optimal completions of S and T respectively under the EF-R-RF(+) problem. Then, $RF(S', T') \leq 2 \cdot RF(S^*, T^*)$.*

4 An Efficient Exact Algorithm for R-RF(+) Distance

As shown above, optimal EF-R-RF(+) completions 2-approximate RF(+) distance. We now show how to construct optimal R-RF(+) completions by modifying optimal EF-R-RF(+) completions.

Notation and terminology. We refer to EF-R-RF(+) completions resulting from the *TwoTreeCompletion* Algorithm of [4] as *canonical EF-R-RF(+) completions*. This is due to the way that maximal red and green subtrees are topologically well placed in such completions. We will refer to the placement of a maximal colored subtree under the *TwoTreeCompletion* Algorithm as a *canonical EF-R-RF(+) position*. The placement of each maximal red subtree R of S, rooted at r, in canonical EF-R-RF(+) completion T' of T has the useful property that all leaves $a \in Le(S) \cap Le(T)$ where $lca_S(a, r) = pa_S(r)$ also satisfy $lca_{T'}(a, r) = pa_{T'}(r)$, and all leaves $b \in Le(S) \cap Le(T)$ where $lca_{T'}(b, r) > pa_{T'}(r)$ also satisfy $lca_S(b, r) > pa_S(r)$.

By Theorem 3.1, we know that there exists an optimal pair of R-RF(+) completions where the only extraneous subtrees are of the form $\{s,t\}$. We will first show that a canonical pair of R-RF(+) completions can be constructed by taking a canonical pair of EF-R-RF(+) completions and pairing up extraneous subtrees of the form $\{s,t\}$ in an optimal manner. We will then design a recurrence relation which computes the best possible change to the RF distance caused by pairing up extraneous subtrees of the form $\{s,t\}$, and show that this change to the RF distance can be computed in near linear time depending on the leaf-set overlap between the input trees.

▶ **Lemma 4.1.** *There exist canonical R-RF(+) completions S^* and T^* of rooted binary trees S and T such that every subtree grafted into S^* and T^* is either in an extraneous subtree or in its canonical EF-R-RF(+) position.*

In the remainder of this section, let S', T' and S^*, T^* represent canonical EF-R-RF(+) and R-RF(+) completions of S and T, respectively. We will soon define the subproblems that are the basis of our dynamic programming algorithm. Before doing so, we motivate the dynamic programming recurrence relation with the following lemma, which describes a new useful tree T'' that is easier to construct from T' and preserves the important topological structure of T^*. Our dynamic programming algorithm actually constructs T'', and we can then easily use T'' to generate S^* and T^*.

▶ **Lemma 4.2.** *Let T'' be the tree obtained by taking T^* and regrafting* every *extraneous subtree $\{s,t\}$ along the parent edge of $lca_{T^*}(lca_{T^*}(Le(sib_S(s))),t)$. Then $RF(S',T'') = RF(S^*,T^*) + 2m$, where m is the number of extraneous subtrees $\{s,t\}$ contained in T^*.*

Note that T'' itself may not be a completion of T. In particular, in the construction of T'', pruning and regrafting the maximal green subtree $T(t)$ is necessary if the extraneous subtree $\{s,t\}$ is formed and $lca_{T'}(s,t) \neq pa_{T'}(t)$. Moving any subtree of T in T' changes T' to no longer be a completion of T. Figure 4 shows a concrete example.

▶ **Definition 4.3.** *Let the colors red and green be associated with the binary values 0 and 1, respectively. For $v \in V(T')$ and $c \in \{0,1\}$, let $cMax(c,v)$ be the total number of maximal subtrees of color c in $T'(v)$. Moreover, let m be an integer such that $0 \leq m \leq cMax(c,v)$. We define $Cost(v,m,c)$ to be $\min_{\widehat{T}}(RF(S',\widehat{T}) - 2p - RF(S',T'))$, where $\widehat{T}$ is obtained from T' by regrafting maximal red and green subtrees in $T'(v)$ under the constraint that each extraneous subtree $\{s,t\}$ is grafted along the parent edge of $lca_{T'(v)}(s,t)$ and exactly m maximal c-colored subtrees in $T'(v)$ have been regrafted along the parent edge of v, excluding extraneous subtrees (see Figure 5 for an example), and p denotes the number of extraneous subtrees of the form $\{s,t\}$ in $\widehat{T}$.*

In the trivial case when v is the root of a maximal c-colored subtree, we will say that it is possible to push one red subtree up to the parent edge of v or down from the parent edge of v.

Note that the *Cost*() subproblem builds the optimal RF(+) distance. However, the cost is defined based on Lemma 4.2 by constructing T'' and subtracting out the extraneous subtrees as they are produced. Moreover, we subtract the constant term $RF(S',T')$ to express the cost as the *change* in RF distance.

We point out that the choice of $\widehat{T}$ implying $Cost(rt(T'),0,0)$ is exactly T'' by Lemmas 4.1 and 4.2. Furthermore, for any internal node v in T', and for the choice of m,c which imply the optimal cost value of $Cost(rt(T'),0,0)$ via the upcoming recurrence relation, the tree $\widehat{T}(v)$ which admits $Cost(v,m,c)$ is exactly equal to $T''(v)$. In this sense, each $\widehat{T}$ captures an entire subtree of T''. Note that on a local scale, in any specific $\widehat{T}$ there may be a red or green

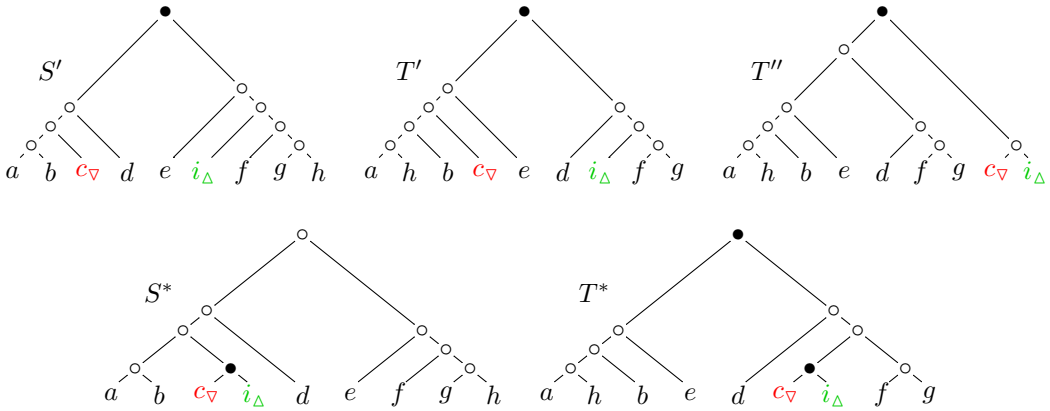


Figure 4 The tree T''. This figure shows the relationship between T', T'', and T^*. In this example, observe that there is exactly one extraneous subtree $\{s, t\}$ in the optimal completions S^* and T^*, and that $RF(S', T'') = RF(S^*, T^*) + 2$. Moreover, T'' in this example cannot be a completion of T since the green leaf i has been regrafted. But constructing T'' is simply an intermediary step for constructing completions S^* and T^*. Matches are denoted by filled in nodes.

subtree regrafted outside of an extraneous subtree and outside of its canonical EF-R-RF(+) position. However, it can be concluded that either eventually these red and green subtrees will be paired in extraneous subtrees for some later $\widehat{T}$, or the particular cost value does not imply the optimal $Cost(rt(T'), 0, 0)$.

The next lemma provides a recurrence relation that can compute each $Cost(v, m, c)$ efficiently. In this recurrence relation, a subscript of L or R denotes the *left* or *right* child, respectively. For example, if a vertex v is an internal node in T then v_L is the left child of v, and if c is a color associated with vertex v then c_L is a color associated with vertex v_L. Note that the trees are unordered, so we use "left" and "right" here only to distinguish between the two children of an internal node.

▶ **Lemma 4.4.** *Let $f(m_i, v_i, c_i)$ equal 2 when $m_i > 0$ and v_i is a match with color other than c_i, and 0 otherwise. Let $g_c(m_L, m_R, c_L, c_R)$ equal $2 \cdot \min\{m_L, m_R\}$ when $c_L \neq c_R$, and 0 when $c_L = c_R = c$. Then,*

$$Cost(v, m, c) = \min_{m_L, m_R, c_L, c_R} \left\{ \begin{array}{l} Cost(v_L, m_L, c_L) + Cost(v_R, m_R, c_R) \\ \quad + f(m_L, v_L, c_L) + f(m_R, v_R, c_R) - g_c(m_L, m_R, c_L, c_R) \end{array} \right\}$$

if v is an internal node of T', and $Cost(v, m, c) = 0$ if v is a leaf of T', where:

(a) *$c, c_L, c_R \in \{0, 1\}$, and either $c_L \neq c_R$ or $c_L = c_R = c$,*

(b) *$0 \leq m \leq cMax(c, v)$,*

(c) *If $c_L \neq c_R$, then $m_i - m_j = m$ for $i, j \in \{L, R\}, i \neq j$ satisfying $c_i = c$,*

(d) *If $c_L = c_R = c$, then $m_L + m_R = m$*

The functions f and g_c from Lemma 4.4 both track *local* changes in matched and mismatched nodes. In particular, f tracks a local change between $RF(S', T')$ and $RF(S', T'')$ while g_c tracks a local change between $RF(S', T'')$ and $RF(S^*, T^*)$. We now provide our dynamic programming algorithm for computing the R-RF(+) distance between S and T.

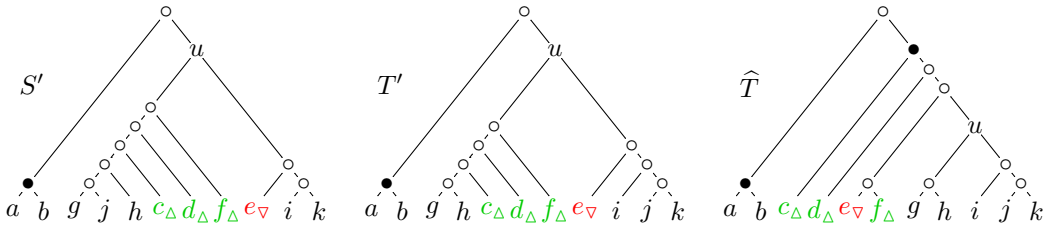


Figure 5 Illustration of tree $\widehat{T}$. The figure shows an example of what the tree $\widehat{T}$ might look like after computing $Cost(u, 2, 1)$, where c and d have both been regrafted *iteratively* along the parent edge of u *and* not regrafted into an extraneous subtree. Note that the extraneous subtree $\{e, f\}$ has also been regrafted along the parent edge of u, though it does not contribute to the value of $m = 2$. In particular, $u = lca_{T'}(e, f)$, so the extraneous subtree $\{e, f\}$ will appear at the same position in $\widehat{T}$ and T''. Moreover, f is not included as one of the two maximal green subtrees grafted onto the parent edge of u since it is a part of an extraneous subtree. For each choice of vertex v, integer m and color c implying to the minimum $Cost(rt(T'), 0, 0)$ value, the corresponding optimal $\widehat{T}$ provides the topolgical structure of T'' when restricted to the subtree rooted at v.

Algorithm 1 *Compute-R-RF+(S,T).*

```
1: Compute the EF-R-RF(+) completions S' and T' of S and T.
2: for v in T' in postorder do
3:    if v is a leaf then
4:       Set Cost(v,0,0) = Cost(v,0,1) = 0.
5:       if v is the root of a maximal red (0) or green (1) subtree then
6:          Set Cost(v,1,c_v) = 0, where c_v is the color of v.
7:    else
8:       for each color c and value 0 ≤ m ≤ cMax(c,v) do
9:          Compute Cost(v,m,c) using the recurrence relation from Lemma 4.4
10: return RF(S',T') + Cost(rt(T'),0,0)
```

The algorithm above can be easily augmented to compute optimal completions by backtracking and determining the optimal values of m and c at each vertex of T' implying $Cost(rt(T'), 0, 0)$. Using these optimal m and c values, we can determine when opposite colored subtrees converge and construct T''. From T'', we simply move each extraneous subtree $\{s, t\}$ into the canonical EF-R-RF(+) position for $T(t)$ to build T^* and form the same extraneous subtrees in S' to build S^*.

▶ **Theorem 4.5.** *The RF(+) distance between two rooted binary trees S and T can be computed in $O(nk^2)$ time, where $n = |Le(S) \cup Le(T)|$ and k is the number of maximal red* and *green subtrees in S and T.*

5 Extension to Unrooted Trees

Our algorithm for the R-RF(+) problem can be easily adapted for the U-RF(+) problem. Specifically, the following algorithm computes the unrooted RF(+) distance between two unrooted input trees S and T with at least one leaf in common.

Algorithm 2 *Compute-U-RF+(S, T).*

1: Let l be any leaf from $Le(S) \cap Le(T)$. Produce two rooted trees $\widehat{S}$ and $\widehat{T}$ by rooting S and T, respectively, on the edge which connects l to the rest of each tree.
2: Compute the RF(+) distance d between $\widehat{S}$ and $\widehat{T}$ using Algorithm *Compute-R-RF+(S,T)*.
3: Return d

The correctness of this algorithm is easy to establish based on the well-understood association between rooted and unrooted RF distances [10, 4], and further technical details and proofs are therefore omitted. This yields the following two theorems.

▶ **Theorem 5.1.** *The U-RF(+) problem can be solved in $O(nk^2)$ time, where $n = |\, Le(S) \cup Le(T)|$ and k is the number of maximal red and green subtrees in the corresponding EF-U-RF(+) completion of S or T.*

▶ **Theorem 5.2.** *Let S^* and T^* represent optimal completions of unrooted trees S and T, respectively, under the U-RF(+) problem. Let S' and T' represent optimal completions of S and T, respectively, under the EF-U-RF(+) problem. Then, $RF(S', T') \leq 2 \cdot RF(S^*, T^*)$.*

6 Experimental Evaluation

We implemented our exact algorithm and performed experiments to assess the impact of using RF(+) distance instead of RF(-) distance on inferences related to tree similarity. We also conducted experiments to assess how well the linear-time algorithm for computing EF-RF(+) distances approximates RF(+) distances in practice. All our experiments were performed using real biological phylogenetic tree datasets on marsupials [8] (158 trees), legumes [33] (22 trees), and placental mammals [7] (726 trees).

Experiment 1: Conflicts between RF(+) and RF(-). Given two trees S and T, let $RF^+(S,T)$ and $RF^-(S,T)$, respectively, denote the RF(+) and RF(-) distances between them. We used the above datasets to measure the number of times that for any "base" tree S, there is a tree T_1 which is closer to S than T_2 under one of RF(+) or RF(-) but not closer under the other distance measure. This motivates the following definitions to describe each possible case of a change in order.

Type-1 Triples: Triple (S, T_1, T_2) is Type-1 if $RF^-(S,T_1) < RF^-(S,T_2)$ but $RF^+(S,T_1) > RF^+(S,T_2)$, or $RF^-(S,T_2) < RF^-(S,T_1)$ but $RF^+(S,T_2) > RF^+(S,T_1)$. A Type-1 triple indicates when the ordering of T_1 and T_2 by distance from S strictly changes as the distance function changes between RF(-) and RF(+).

Type-2 Triples: Triple (S, T_1, T_2) is Type-2 if $RF^-(S,T_1) = RF^-(S,T_2)$ but $RF^+(S,T_1) \neq RF^+(S,T_2)$. A Type-2 triple indicates when T_1 and T_2 have equal distance to S under RF(-) but not under RF(+).

Type-3 Triples: Triple (S, T_1, T_2) is Type-3 if $RF^-(S,T_1) \neq RF^-(S,T_2)$ but $RF^+(S,T_1) = RF^+(S,T_2)$. A Type-3 triple indicates when T_1 and T_2 have equal distance to S under RF(+) but not under RF(-).

Observe that the magnitude of difference between RF(+) and RF(-) distances depends on the level of overlap between the trees being compared. To account for this effect, we define the *leaf-overlap ratio* of a pair of trees (S,T) to be the following ratio: $|\, Le(S) \cap Le(T)|$ divided by $\min\{|\, Le(S)|, |\, Le(T)|\}$, and the leaf-overlap ratio of a triple of trees S, T_1, and T_2 to be the minimum pairwise leaf-overlap ratio between (S,T_1) and (S,T_2).

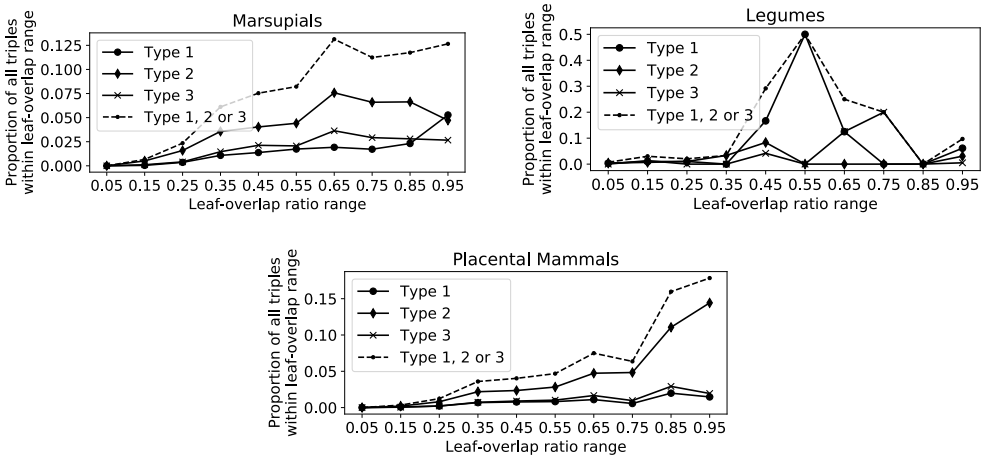


Figure 6 Fraction of conflicting triples for different leaf-overlap ratios. The figure contains three plots, one for each dataset, which each show the fraction of triples of type-1, type-2, and type-3 for different ranges of leaf-overlap ratio, among all triples of trees within the same leaf-overlap ratio range in that dataset. The dotted line represents the total number of conflicting triples (i.e., all triples of type 1, 2 or 3). x-axis labels denote the center of each interval of size 0.1. Each leaf-overlap ratio range is a closed interval and *includes* the boundary, e.g., x-axis label 0.15 represents the range $[0.1 - 0.2]$.

We performed this experiment for each subset of three trees from each dataset, and Figure 6 shows its results. As the figure shows, the proportion of conflicting triples (type-1, 2, or 3) tends to increase as the triple leaf-overlap ratio increases. In particular, at least 10% of all triples show a conflict (either of type-1, 2, or 3) when the leaf-overlap ratio is 0.7 or greater. Even for leaf-overlap ratio as small as 0.4, we find that 5% of all triples show a conflict. This demonstrates that RF(+) and RF(-) frequently differ starkly in their assessments of relative similarities between trees. Observe that the results on the Legumes dataset are vastly different from the results on the other two datasets. This is mainly because the Legumes dataset consists of only 22 trees, which is significantly smaller than the 158 tree and 726 tree datasets. For instance, the number of triples within each leaf overlap ratio range (interval size 0.1) is between 8,214,518 and 50,815,687 for the placental mammals dataset, between 3,287 and 1,652,701 for the Marsupials dataset, but only 6, 16, 5, and 0, respectively, for the Legumes dataset for leaf overlap ratio ranges $[0.5 - 0.6]$, $[0.6 - 0.7]$, $[0.7 - 0.8]$, and $[0.8 - 0.9]$.

Experiment 2: Impact on phylogenetic database search and clustering. Next, we assessed the potential impact of using RF(+) distance on applications related to phylogenetic database search and clustering. Specifically, we assessed how, for each "query" tree in each dataset, the sets of the "closest" trees to it differed under RF(+) and RF(-). Specifically, we measured how the sets of (i) the most similar trees and (ii) the most similar 10% of trees (i.e., top 10% closest matches) differed when using RF(+) and RF(-) distances. To avoid any ambiguity in defining these sets, we include all trees with equal distance, even if that results in sets of different sizes under RF(+) and RF(-).

For our comparison of the most similar trees, we found that the sets of closest trees under RF(+) and RF(-) all had a distance of 0 to the query tree and were identical, for all choices of the query tree in all datasets. To perform a more meaningful comparison, we therefore required a minimum leaf-overlap ratio of 0.7, i.e., only those trees with a minimum leaf-overlap ratio of 0.7 with the query tree could be compared with the query tree. Likewise,

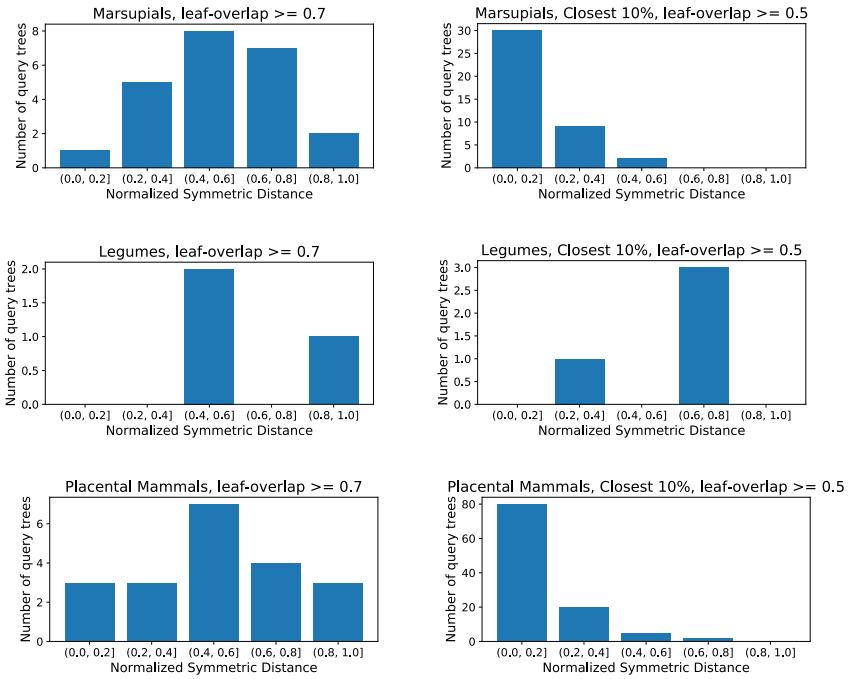


Figure 7 Difference between sets of closest trees under RF(+) and RF(-). Plots in the left column show the number of query trees where the set of closest trees with a minimum leaf-overlap ratio of 0.7 differ under RF(+) and RF(-) distances for each of the three biological data sets. Plots in the right column show the number of query trees where the set of closest 10% of trees with a minimum leaf-overlap ratio of 0.5 differ under RF(+) and RF(-) distances. Results are presented for varying levels of difference between the sets (labels on the x-axes). The sizes of the datasets, in order from top to bottom, are 158 trees, 22 trees and 726 trees. Each tree in each of these datasets was used as a query tree for this analysis.

for our comparison of the most similar 10% of trees, we found that the sets of closest 10% of trees were generally identical under RF(+) and RF(-) if no minimum leaf-overlap ratio was imposed. We therefore imposed a minimum leaf-overlap ratio of 0.5 for the analysis, which was the smallest ratio for which a non-negligible fraction of query trees returned differing sets under RF(+) and RF(-). Figure 7 shows the results of both these analyses. We find that there are several query trees in each dataset for which there is a large difference (normalised symmetric difference greater than, say, 0.4) between their sets of closest trees under RF(+) and RF(-). For the sets of closest 10% of trees, we find that over 25% of trees in the marsupials dataset, 18% of trees in the legumes dataset, and almost 15% of trees in the placental mammals dataset return different sets of closest 10% of trees under RF(+) and RF(-) distances. These results demonstrate how using RF(+) distance can substantially impact phylogenetic database search and phylogenetic tree clustering, especially when the trees under consideration have a sufficiently large overlap in their leaf sets.

Experiment 3: Comparison of EF-RF(+) and RF(+). Finally, we used simulated and real datasets to compare the distances inferred under EF-RF(+) and RF(+), and to study the runtime and scalability of our implementation. For our analysis with simulated data, we generated two datasets of random trees using the birth-death model implemented in SaGePhy [21] (specific parameter values: height of tree = 1.0, birth rate = 5.0 and death

rate = 0.05). The first simulated dataset consisted of 100 randomly generated trees, each with between 200 and 300 leaves. The second simulated dataset also consisted of 100 randomly generated trees, but each with between 900 and 1000 leaves. The average leaf-set sizes for these two datasets were 244.95 and 941.14, respectively, and the average pairwise leaf-overlap ratio for both datasets was approximately 0.5. For each pair of trees in each dataset, we measured how close the EF-RF(+) distance is to the RF(+) distance for that pair. Figure 8 plots the distribution of the ratio of RF(+) distance to EF-RF(+) distance for the two datasets. As that figure shows, the ratio of RF(+) distance to EF-RF(+) distance is approximately 0.92, on average, and roughly follows a Gaussian distribution.

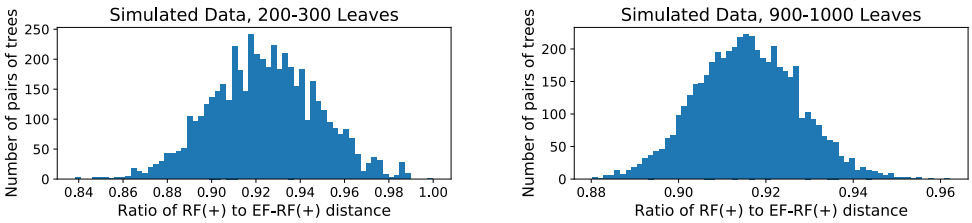


Figure 8 Comparison of EF-RF(+) and RF(+) distances on simulated trees. The two plots show the distribution of the ratio of RF(+) distance to EF-RF(+) distance for the two simulated datasets consisting of randomly generated birth-death trees. Each dataset contains 100 trees and results are shown for all $\binom{100}{2}$ pairs of trees in each dataset.

For the three biological datasets, we found that the ratio of RF(+) distance to EF-RF(+) distance was equal to one for an overwhelmingly large proportion of pairs of trees within all three datasets. Specifically, for the marsupials, legumes, and placental mammals datasets, the average ratios of RF(+) distance to EF-RF(+) distance were 0.998, 0.993, and 0.995, respectively. In fact, 99.07%, 93.81%, and 96.82% of the pairs in these datasets, respectively, had identical EF-RF(+) and RF(+) distances. Even when the trees being compared were restricted to have at least 0.4 leaf-overlap ratio, 95.97%, 78.79%, and 95.59% of the pairs in marsupials, legumes, and placental mammals datasets, respectively, had identical EF-RF(+) and RF(+) distances. This discrepancy between results for simulated data and real data is not surprising since we expect any pair of randomly generated trees to have smaller maximal red and green subtrees and greater RF(-) distance, presenting more opportunities to improve the distance by creating extraneous clades. Together, these results on simulated and real datasets show that EF-RF(+) distance, which is linear-time computable, is generally very close to RF(+) distance in practice.

Runtime comparison. We also measured the runtimes of the two algorithms and found that, on average, computing EF-RF(+) distances took 0.06 seconds for the first simulated dataset and 0.25 seconds for the second simulated dataset. Corresponding average runtimes for computing RF(+) distances were 0.17 seconds and 1.04 seconds, respectively. All timed experiments were run on a single core of a 2.1 GHz Intel Xeon processor.

7 Conclusion

Completion based comparison of incomplete phylogenetic trees is an emerging, promising approach for tree comparison. In this work, we developed the first polynomial-time exact algorithm for the RF(+) problem. We also established a linear-time 2-approximation algorithm for the problem. These algorithms allow for more principled comparison of

incomplete phylogenetic trees than was hitherto possible, and our experimental analysis shows that RF(+) distance can lead to very different inferences regarding phylogenetic similarity compared to traditional RF distance. Moreover, our results suggest that the linear-time 2-approximation algorithm for the RF(+) problem almost always computes optimal or near-optimal RF(+) distances in practice.

In addition to their utility for improved tree comparison and clustering, our solutions for the RF(+) problem also have implications for phylogenomics. Many modern phylogenomics methods for reconstructing evolutionary histories and understanding genome-scale patterns of evolution are designed to work with complete phylogenies from genomic loci across the genomes of the considered species [5, 26, 27, 20, 12], and loci that yield incomplete phylogenies are often discarded, resulting in only a fraction of the available genomic sequence information being used for the phylogenomic analysis. Thus, problems related to optimal completion of incomplete phylogenies (i.e., imputation of complete phylogenies) arise naturally in phylogenomics. Our algorithms for the RF(+) problem may provide a principled way to impute such complete phylogenies.

The current work is restricted to comparison of binary trees under the Robinson-Foulds metric, and it can be extended in many useful ways. A possible next step could include consideration of non-binary trees in computing distances between incomplete trees. Future work could also entail development of similar completion based methods under other distance/similarity measures such as triplet/quartet distances [14, 17], nearest neighbor interchange (NNI) and subtree prune and regraft (SPR) distances [31, 18, 34], and nodal distance [9]. Furthermore, the idea of computing optimal completions could be extended to multi-labeled trees, which arise frequently in practice due to evolutionary events such as gene duplication.

References

1 Wasiu A. Akanni, Mark Wilkinson, Christopher J. Creevey, Peter G. Foster, and Davide Pisani. Implementing and testing bayesian and maximum-likelihood supertree methods in phylogenetics. *Royal Society Open Science*, 2(8), 2015. doi:10.1098/rsos.140436.

2 Amihood Amir and Dmitry Keselman. Maximum agreement subtree in a set of evolutionary trees: Metrics and efficient algorithms. *SIAM Journal on Computing*, 26(6):1656–1669, 1997. doi:10.1137/S0097539794269461.

3 Mukul S. Bansal. Linear-time algorithms for some phylogenetic tree completion problems under robinson-foulds distance. In *Comparative Genomics - 16th International Conference, RECOMB-CG 2018, Magog-Orford, QC, Canada, October 9-12, 2018, Proceedings*, pages 209–226, 2018. doi:10.1007/978-3-030-00834-5_12.

4 Mukul S. Bansal. Linear-time algorithms for phylogenetic tree completion under robinson–foulds distance. *Algorithms for Molecular Biology*, 15:6, 2020.

5 Mukul S. Bansal, Guy Banay, Timothy J. Harlow, J. Peter Gogarten, and Ron Shamir. Systematic inference of highways of horizontal gene transfer in prokaryotes. *Bioinformatics*, 29(5):571–579, 2013.

6 Mukul S. Bansal, J. Gordon Burleigh, Oliver Eulenstein, and David Fernández-Baca. Robinson-foulds supertrees. *Algorithms for Molecular Biology*, 5(1):18, February 2010.

7 Robin Beck, Olaf Bininda-Emonds, Marcel Cardillo, Fu-Guo Liu, and Andy Purvis. A higher-level MRP supertree of placental mammals. *BMC Evol. Biol.*, 6(1):93, 2006. doi:10.1186/1471-2148-6-93.

8 Marcel Cardillo, Olaf R. P. Bininda-Emonds, Elizabeth Boakes, and Andy Purvis. A species-level phylogenetic supertree of marsupials. *Journal of Zoology*, 264:11–31, 2004.

9 Gabriel Cardona, Mercè Llabrés, Francesc Rosselló, and Gabriel Valiente. Nodal distances for rooted phylogenetic trees. *Journal of Mathematical Biology*, 61(2):253–276, August 2010. doi:10.1007/s00285-009-0295-2.

10 Ruchi Chaudhary, J Gordon Burleigh, and David Fernandez-Baca. Fast local search for unrooted robinson-foulds supertrees. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 9(4):1004–1013, 2012.

11 Duhong Chen, J Gordon Burleigh, Mukul S Bansal, and David Fernández-Baca. Phylofinder: an intelligent search engine for phylogenetic tree databases. *BMC Evolutionary Biology*, 8(1):90, 2008.

12 Sarah Christensen, Erin K. Molloy, Pranjal Vachaspati, and Tandy Warnow. Optimal Completion of Incomplete Gene Trees in Polynomial Time Using OCTAL. In Russell Schwartz and Knut Reinert, editors, *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*, volume 88 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

13 James A. Cotton, Mark Wilkinson, and Mike Steel. Majority-rule supertrees. *Systematic Biology*, 56(3):445–452, 2007. doi:10.1080/10635150701416682.

14 Douglas E. Critchlow, Dennis K. Pearl, Chunlin Qian, and Daniel Faith. The triples distance for rooted bifurcating phylogenetic trees. *Systematic Biology*, 45(3):323–334, 1996. doi:10.1093/sysbio/45.3.323.

15 Damien M. de Vienne, Tatiana Giraud, and Olivier C. Martin. A congruence index for testing topological similarity between trees. *Bioinformatics*, 23(23):3119–3124, 2007. doi:10.1093/bioinformatics/btm500.

16 Jianrong Dong and David Fernandez-Baca. Properties of majority-rule supertrees. *Systematic Biology*, 58(3):360–367, 2009. doi:10.1093/sysbio/syp032.

17 George F. Estabrook, F. R. McMorris, and Christopher A. Meacham. Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. *Systematic Zoology*, 34(2):193–200, 1985. URL: http://www.jstor.org/stable/2413326.

18 J. Felsenstein. *Inferring Phylogenies*. Sinauer Assoc., Sunderland, Mass, 2003.

19 C. R. Finden and A. D. Gordon. Obtaining common pruned trees. *Journal of Classification*, 2(1):255–276, December 1985. doi:10.1007/BF01908078.

20 Kevin Gori, Tomasz Suchan, Nadir Alvarez, Nick Goldman, and Christophe Dessimoz. Clustering genes of common evolutionary history. *Molecular Biology and Evolution*, 33(6):1590–1605, 2016. doi:10.1093/molbev/msw038.

21 Soumya Kundu and Mukul S Bansal. Sagephy: An improved phylogenetic simulation framework for gene and subgene evolution. *Bioinformatics*, 35(18):3496–3498, 2019.

22 Anne Kupczok. Split-based computation of majority-rule supertrees. *BMC Evolutionary Biology*, 11(1):205, July 2011. doi:10.1186/1471-2148-11-205.

23 Anne Kupczok, Arndt Von Haeseler, and Steffen Klaere. An exact algorithm for the geodesic distance between phylogenetic trees. *Journal of Computational Biology*, 15(6):577–591, 2008.

24 Harris T. Lin, J. Gordon Burleigh, and Oliver Eulenstein. Triplet supertree heuristics for the tree of life. *BMC Bioinformatics*, 10(1):S8, January 2009. doi:10.1186/1471-2105-10-S1-S8.

25 Michelle M. McMahon, Akshay Deepak, David FernÃ¡ndez-Baca, Darren Boss, and Michael J. Sanderson. Stbase: One million species trees for comparative biology. *PLOS ONE*, 10(2):1–17, February 2015. doi:10.1371/journal.pone.0117987.

26 S. Mirarab, R. Reaz, Md. S. Bayzid, T. Zimmermann, M. S. Swenson, and T. Warnow. ASTRAL: genome-scale coalescent-based species tree estimation. *Bioinformatics*, 30(17):i541–i548, 2014. doi:10.1093/bioinformatics/btu462.

27 Siavash Mirarab, Md. Shamsuzzoha Bayzid, Bastien Boussau, and Tandy Warnow. Statistical binning enables an accurate coalescent-based estimation of the avian tree. *Science*, 346(6215), 2014. doi:10.1126/science.1250463.

28 William H Piel, MJ Donoghue, MJ Sanderson, and LUT Netherlands. Treebase: a database of phylogenetic information. In *Proceedings of the 2nd International Workshop of Species 2000*, 2000.

29 D.F. Robinson and L.R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1):131–147, 1981. doi:10.1016/0025-5564(81)90043-2.

30 Jason TL Wang, Huiyuan Shan, Dennis Shasha, and William H Piel. Fast structural search in phylogenetic databases. *Evolutionary Bioinformatics*, 2005(1):0–0, 2007.

31 M.S. Waterman and T.F. Smith. On the similarity of dendrograms. *Journal of Theoretical Biology*, 73(4):789–800, 1978. doi:10.1016/0022-5193(78)90137-6.

32 Christopher Whidden, Norbert Zeh, and Robert G. Beiko. Supertrees based on the subtree prune-and-regraft distance. *Systematic Biology*, 63(4):566–581, 2014. doi:10.1093/sysbio/syu023.

33 M.F. Wojciechowski, M.J. Sanderson, K.P. Steele, and A. Liston. Molecular phylogeny of the "Temperate Herbaceous Tribes" of Papilionoid legumes: a supertree approach. In P.S. Herendeen and A. Bruneau, editors, *Advances in Legume Systematics*, volume 9, pages 277–298. Royal Botanic Gardens, Kew, 2000.

34 Yufeng Wu. A practical method for exact computation of subtree prune and regraft distance. *Bioinformatics*, 25(2):190–196, 2009. doi:10.1093/bioinformatics/btn606.

35 Ruriko Yoshida, Kenji Fukumizu, and Chrysafis Vogiatzis. Multilocus phylogenetic analysis with gene tree clustering. *Annals of Operations Research*, March 2017. doi:10.1007/s10479-017-2456-9.

A Appendix

Proof of Theorem 3.1. Let S^* and T^* be arbitrarily chosen optimal completions of S and T under R-RF(+). We will modify S^* and T^* to be of the desired form. To do so, we first show that any maximal red subtree in S and any maximal green subtree of T can be made subtrees of S^* and T^* without increasing the RF distance between them (condition 1). Suppose there exist two maximal matched red subtrees R_1 and R_2 of S^* and T^* which neighbor each other in the original tree S. Let r_1 and r_2 be the roots of R_1 and R_2.

1. Suppose both $C_{T^*}(pa_{T^*}(r_1)) \setminus C_{T^*}(r_1)$ and $C_{T^*}(pa_{T^*}(r_2)) \setminus C_{T^*}(r_2)$ contain some non-green leaves. Observe that every matched clade in T^* containing $C_{T^*}(r_1) \cup C_{T^*}(r_2)$ must also contain $C_{T^*}(lca_{T^*}(r_1, r_2))$ because R_1 and R_2 neighbor each other in S by assumption. Therefore, we can regraft R_2 to neighbor R_1 in T^* without increasing the RF distance between S^* and T^*. Moreover, if there are any green subtrees inserted along the path from R_1 to R_2 in S^*, then they can be regrafted along the parent edge of $lca_{S^*}(r_1, r_2)$ without increasing the Robinson-Foulds distance.
2. Suppose, without loss of generality, that $C_{T^*}(pa_{T^*}(r_2)) \setminus C_{T^*}(r_2)$ contains only green leaves. That is, suppose R_2 is contained in an extraneous subtree, whose root could be a match without ancestoring R_1. First, regraft R_2 in T^* to neighbor R_1. Then, regraft all green subtrees from the path in S^* connecting R_2 and R_1 to the parent edge of $lca_{S^*}(r_1, r_2)$, *preserving the topological structure of the green leaves.* This does not increase the RF distance between S^* and T^*. Notice that any originally matched clades containing $Le(R_2)$ are mismatched. However, preserving the topological structure of the green leaves from any matched clades containing $Le(R_2)$ also retains the same number of matches except for one representing the smallest match containing R_2. This is because the only subtree removed (*in both* S^* *and* T^*) from these matched extraneous subtrees is R_2. Furthermore, the matched clade $Le(R_1) \cup Le(R_2)$ is formed in both S^* and T^*, which counteracts this lost match.

If this is done iteratively for all such R_1 and R_2, then we conclude that there exist optimal completions S^* and T^* where every maximal red subtree in S is also a subtree of S^* *and* T^*. The same argument applies for maximal green subtrees.

Now we will show that S^* and T^* can be modified to only contain extraneous subtrees of the form $\{s,t\}$ without increasing the RF distance (condition 2). We will simultaneously show that an extraneous subtree $\{s,t\}$ is a subtree of S^* if and only if it is a subtree of T^* by construction (condition 3). Observe that if $Le(U) \cap Le(V) \cap Le(S) \neq \varnothing$ for two maximal extraneous subtrees U and V of S^* and T^* respectively, then $Le(U) \cap Le(V) \cap Le(S) \subseteq Le(R)$ for a *single* maximal red subtree R of S. Likewise if $Le(U) \cap Le(V) \cap Le(T) \neq \varnothing$, then $Le(U) \cap Le(V) \cap Le(T) \subseteq Le(Y)$ for a single maximal green subtree Y of T. Therefore, every maximal extraneous subtree in S^* or T^* satisfies one of the following two cases.

1. Without loss of generality, let U be a maximal extraneous subtree of S^* rooted at u such that for *every* maximal extraneous subtree V of T^*, $Le(U) \cap Le(V) \cap Le(S) = \varnothing$ or $Le(U) \cap Le(V) \cap Le(T) = \varnothing$. Then, every *extraneous* clade contained in $Le(U)$ must be a mismatch. Hence, every maximal green subtree of U can be regrafted along the parent edge of $pa_{S^*}(u)$ without increasing the Robinson-Foulds distance from T^*. This results in destroying *all* extraneous subtrees contained in U because $pa_{S^*}(u)$ is an ancestor of a maximal extraneous subtree and therefore possesses uncolored descendants.
2. Let U and V be maximal extraneous subtree of S^* and T^*, rooted at u and v respectively, satisfying $Le(U) \cap Le(V) \cap Le(S) \neq \varnothing$ and $Le(U) \cap Le(V) \cap Le(T) \neq \varnothing$. Then every matched *extraneous* clade contained in $Le(U)$ and $Le(V)$ must contain elements of $Le(U) \cap Le(V) \cap Le(S)$ and $Le(U) \cap Le(V) \cap Le(T)$. Every maximal green subtree of U with no leaves in $Le(U) \cap Le(V) \cap Le(T)$ can be regrafted along the parent edge of u without increasing the RF distance. Likewise, every maximal red subtree of V with no leaves in $Le(U) \cap Le(V) \cap Le(S)$ can be regrafted along the parent edge of v without increasing the RF distance. Moreover, as described before, $Le(U) \cap Le(V) \cap Le(S) \subseteq Le(R)$ and $Le(U) \cap Le(V) \cap Le(T) \subseteq Le(Y)$ for a single maximal red subtree R of S and a single maximal green subtree Y of T. Hence, we are only left with the extraneous subtree $\{rt_{S^*}(R), rt_{S^*}(Y)\}$ in S^* and $\{rt_{T^*}(R), rt_{T^*}(Y)\}$ in T^*.

Once every maximal extraneous subtree in S^* and T^* is handled according to the appropriate case above, we are left with two optimal completions S^* and T^* of the desired form. ◀

Proof of Lemma 3.2. *Case 1*: In this case, both $pa_{S^*}\{s,t\}$ and $pa_{T^*}\{s,t\}$ are matched nodes. Here, we must have $Le(S^*(pa_{S^*}\{s,t\})) = Le(T^*(pa_{T^*}\{s,t\}))$. This holds because $C_{S^*}(pa_{S^*}\{s,t\})$ and $C_{T^*}(pa_{T^*}\{s,t\})$ are *both* matches, and the smallest proper supersets of $C_{T^*}(s) \cup C_{T^*}(t)$ in S^* and T^* respectively. By definition, the decomposition replaces the matched clades $C_{S^*}(s) \cup C_{S^*}(t)$ and $C_{T^*}(s) \cup C_{T^*}(t)$ with $C_{S^*}(pa_{S^*}\{s,t\}) \setminus C_{S^*}(t)$ and $C_{T^*}(pa_{T^*}\{s,t\}) \setminus C_{T^*}(t)$ in S^* and T^*, respectively. Since $Le(S^*(pa_{S^*}\{s,t\})) = Le(T^*(pa_{T^*}\{s,t\}))$, we conclude that $C_{S^*}(pa_{S^*}\{s,t\}) \setminus C_{S^*}(t)$ and $C_{T^*}(pa_{T^*}\{s,t\}) \setminus C_{T^*}(t)$ are then matches in the resulting trees S' and T'.

Case 2: We now consider the case when exactly one of the nodes $pa_{S^*}\{s,t\}$ and $pa_{T^*}\{s,t\}$ is a matched node. Without loss of generality, suppose $pa_{S^*}\{s,t\}$ is a match and $pa_{T^*}\{s,t\}$ is not a match. For convenience, let x denote $pa_{S^*}\{s,t\}$, y denote $pa_{T^*}\{s,t\}$, and let u be the element of $V(T^*)$ such that $C_{S^*}(x) = C_{T^*}(u)$. Then, observe that $C_{S^*}(x) \supset C_{T^*}(y)$, i.e., $y < u$ in T^*. Moreover, every node v along the path from y to u in T^* must be a mismatch since $C_{T^*}(t) \subset C_{T^*}(v)$ and $C_{S^*}(t) \cap C_{S^*}(sib_{S^*}\{s,t\}) = \varnothing$ but $C_{T^*}(v) \cap C_{S^*}(sib_{S^*}\{s,t\}) \neq \varnothing$ for arbitrary choice of v. Now, applying the decomposition of extraneous subtree $\{s,t\}$ to S^* and T^* yields the modified trees S' and T'. Observe that this modification changes exactly the $\{s,t\}$ clade, and all clades along the path from y to u in T^*. In S', the new clade

formed at the subtree rooted at $pa_{S'}(t)$ must be a matched node since $C_{S'}(pa_{S'}(t)) = C_{T'}(u)$. Moreover, in T', all clades $C_{T'}(v)$ along the path from y to u remain mismatches except for $C_{T'}(u)$ because it still holds that $C_{T'}(t) \subset C_{T'}(v)$ and $C_{S'}(t) \cap C_{S'}(sib_{S'}\{s,t\}) = \varnothing$ but $C_{T'}(v) \cap C_{S'}(sib_{S'}\{s,t\}) \neq \varnothing$ for arbitrary choice of v along the path. Thus, after the decomposition, the number of matched clades in S' (w.r.t. T') remains the same as the number of matched clades in S^* (w.r.t. T^*).

Case 3: If neither $pa_{S^*}\{s,t\}$ nor $pa_{T^*}\{s,t\}$ is a matched node, then, following the same argument as in Case 1, S' will have one less matched node (w.r.t. T') than S^* (w.r.t. T^*). Namely, the clades $C_{S^*}(pa_{S^*}\{s,t\}) \setminus C_{S^*}(t)$ and $C_{T^*}(pa_{T^*}\{s,t\}) \setminus C_{T^*}(t)$ are mismatched clades in S' and T' respectively. Consequently, T' will have one less matched node as well. Thus, $RF(S',T') = RF(S^*,T^*) + 2$. ◀

Proof of Theorem 3.3. Let $d = \frac{1}{2}RF(S^*,T^*)$ and let e be the number of extraneous clades in S^*. Then, we have that d is also the number of mismatches in S^*, or equivalently in T^*. Observe that at most d of the e extraneous clades have mismatched parent nodes in both trees. Thus, by Lemma 3.2, decomposing all e extraneous clades will *increase* the RF distance by at most $2d = RF(S^*,T^*)$. Therefore, the decomposed extraneous clade free completion will have an RF distance of at most $2 \cdot RF(S^*,T^*)$. ◀

Proof of Lemma 4.1. Consider arbitrary canonical R-RF(+) completions S^* and T^*. We will show that any grafted subtree in S^* and T^* that is not in its canonical EF-R-RF(+) position or in an extraneous subtree can be regrafted into its canonical EF-R-RF(+) position without increasing the RF distance. Without loss of generality, suppose there exists a maximal red subtree R, with r denoting $rt(R)$, in T^* such that R is neither in its canonical EF-R-RF(+) position nor in an extraneous subtree. Let u represent the canonical EF-R-RF(+) position of subtree R in completion T^*. Thus, $u \neq pa_{T^*}(r)$. Then, we have two possible cases: either $pa_{T^*}(r)$ is an ancestor of u or not ($pa_{T^*}(r) > u$ or $pa_{T^*}(r) \not> u$).

1. Suppose $pa_{T^*}(r) > u$. We will prove that $pa_{T^*}(r)$ can be regrafted in position u without increasing the RF distance. Since $pa_{T^*}(r) > u$, for any arbitrary node c on the path from $pa_{T^*}(r)$ to u, there exists a subtree C of $T^*(c)$ rooted at one of the children of c (the subtree not containing u) satisfying $pa_{T^*}(r) > c = lca_{T^*}(u, Le(C)) > u$ and $pa_{S^*}(r) < lca_{S^*}(r, Le(C))$. Since $pa_{T^*}(r) > lca_{T^*}(u, Le(C)) > u$, we have that $pa_{T^*}(r) > lca_{T^*}(Le(C), a) > a$ for *all* leaves $a \in Le(S) \cap Le(T)$ such that $a < pa_{S^*}(r)$. Since for each such a, we have that $a < pa_{S^*}(r) < lca_{S^*}(a, Le(C))$ and $a < lca_{T^*}(a, Le(C)) = c < pa_{T^*}(r)$, every match containing $Le(C)$ must also contain $Le(R)$. In particular, c is not a match. This is true for *every* node c along the path from $pa_{T^*}(r)$ to u. We can therefore regraft R at position u without increasing the RF distance because every node along the path from $pa_{T^*}(r)$ to u is already mismatched.
2. Now suppose $pa_{T^*}(r) \not> u$. We will prove that R can be regrafted along the parent edge of $lca_{T^*}(pa_{T^*}(r), u)$ (equivalent position to u if u is an ancestor of $pa_{T^*}(r)$) without increasing the RF distance. This will then reduce the case where $pa_{T^*}(r)$ is not an ancestor of u to the previous case where $pa_{T^*}(r)$ is an ancestor of u. If $pa_{T^*}(r)$ is not an ancestor of u, then there exist some $a_1, \ldots, a_k \in Le(S) \cap Le(T)$ such that $pa_{S^*}(r) > a_i$ and $lca_{T^*}(pa_{T^*}(r), a_i) > pa_{T^*}(r)$ for all values of i. Therefore, $pa_{T^*}(r)$ is not a match, as well as every node on the same path up to the node $lca_{T^*}(pa_{T^*}(r), a_1, \ldots, a_k)$ which contains every a_i in its clade $C_{T^*}(lca_{T^*}(pa_{T^*}(r), a_1, \ldots, a_k))$. Then, regrafting R at the parent edge of $lca_{T^*}(a_1, \ldots, a_k, pa_{T^*}(r)) = lca_{T^*}(pa_{T^*}(r), u)$ will not increase the RF distance since there are no matches to become mismatched. ◀

Proof of Lemma 4.2. For binary trees U and V, let $\mathcal{M}_U^V$ denote the LCA map from U to V. That is, on input $u \in V(U)$, $\mathcal{M}_U^V(u)$ returns $lca_V(C_U(u))$. We will show that $RF(S', T'') - RF(S', T') = RF(S^*, T^*) - RF(S', T') + 2m$. Observe that the only changes from S' and T' to S^*, T^* and T'' are the formations of the extraneous subtrees $\{s, t\}$. Then, it suffices to confirm that for every extraneous subtree $\{s, t\}$, the number of mismatched clades in $T''(pa_{T''}\{s, t\})$ equals the number of mismatched clades in $T^*(\mathcal{M}_{S^*}^{T^*}(pa_{S^*}\{s, t\}))$ plus the number of extraneous subtrees. For an arbitrary extraneous subtree $\{s, t\}$ in T^*, we first count the mismatched clades in $T''(pa_{T''}\{s, t\})$. Then, we count the mismatched clades in $T^*(\mathcal{M}_{S^*}^{T^*}(pa_{S^*}\{s, t\}))$ and compare.

1. Suppose v lies along the path from $pa_{T''}\{s, t\}$ to the parent of the canonical EF-R-RF(+) position for $T(t)$ in T''. Moreover, suppose u lies along the path from $pa_{T''}\{s, t\}$ to the parent of the canonical EF-R-RF(+) position for $S(s)$ in T''. Then $C_{S'}(\mathcal{M}_{T''}^{S'}(v)) \supseteq C_{T''}(v) \cup C_{S'}(t)$ since v is an *ancestor* of the canonical EF-R-RF(+) position of $T(t)$ in T'' and hence $\mathcal{M}_{T''}^{S'}(v)$ is an ancestor of the canonical EF-R-RF(+) position of $T(t)$ in S'. Moreover, $C_{T''}(v) \cap C_{S'}(t) = \varnothing$ if $v \neq pa_{T''}\{s, t\}$ by construction of T''. Hence if $v \neq pa_{T''}\{s, t\}$, then v is mismatched with respect to S'. Likewise, $C_{S'}(\mathcal{M}_{T''}^{S'}(u)) \supseteq C_{T''}(u) \cup C_{S'}(s)$ and $C_{T''}(u) \cap C_{S'}(s) = \varnothing$ if $u \neq pa_{T''}\{s, t\}$. Hence if $u \neq pa_{T''}\{s, t\}$, then u is mismatched with respect to S'. Note that by construction, $C_{T''}(pa_{T''}\{s, t\}) = C_{T'}(lca_{T'}(s, t))$. Hence $pa_{T''}\{s, t\}$ is matched with respect to S' if and only if $lca_{T'}(s, t)$ is, and every other node along either path is mismatched.
   Note that the only remaining node impacted in the formation of $\{s, t\}$ is the root of the extraneous subtree in T''. This node must be mismatched with respect to S' since S' is an extraneous free completion.
2. Now suppose v lies along the path from $pa_{T^*}\{s, t\}$ (the canonical EF-R-RF(+) position for $T(t)$ in T^*) to $\mathcal{M}_{S^*}^{T^*}(pa_{S^*}\{s, t\})$ (the least common ancestor of the EF-R-RF(+) positions in T^*). Moreover, suppose u lies along the path from $\mathcal{M}_{S^*}^{T^*}(pa_{S^*}\{s, t\})$ to the parent of the canonical EF-R-RF(+) position for $S(s)$ in T^*. Observe that $\mathcal{M}_{T^*}^{S^*}(v)$ is an ancestor of the extraneous subtree $\{s, t\}$ in S^*, and therefore $\mathcal{M}_{T^*}^{S^*}(v)$ is an ancestor of the canonical EF-R-RF(+) position for $S(s)$ in S^*. Then $C_{S^*}(\mathcal{M}_{T^*}^{S^*}(v)) \supseteq C_{T^*}(v) \cup C_{S^*}(sib_{S^*}\{s, t\})$, where $C_{T^*}(v) \cap C_{S^*}(sib_{S^*}\{s, t\}) = \varnothing$ if $v \neq \mathcal{M}_{S^*}^{T^*}(pa_{S^*}\{s, t\})$. Additionally, notice that $\mathcal{M}_{T^*}^{S^*}(u)$ is an ancestor of the canonical EF-R-RF(+) position for $S(s)$ in S^*, and therefore $\mathcal{M}_{T^*}^{S^*}(u)$ is an ancestor of the extraneous subtree $\{s, t\}$. Then $C_{S^*}(\mathcal{M}_{T^*}^{S^*}(u)) \supseteq C_{T^*}(u) \cup C_{S^*}(s)$, where $C_{T^*}(u) \cap C_{S^*}(s) = \varnothing$ if $u \neq \mathcal{M}_{S^*}^{T^*}(pa_{S^*}\{s, t\})$. It follows that if $u \neq \mathcal{M}_{S^*}^{T^*}(pa_{S^*}\{s, t\})$, then u is a mismatched node. Likewise, if $v \neq \mathcal{M}_{S^*}^{T^*}(pa_{S^*}\{s, t\})$, then v is a mismatched node. Furthermore, $C_{T^*}(\mathcal{M}_{S^*}^{T^*}(pa_{S^*}\{s, t\}))$ is a matched clade with respect to S^* if and only if $C_{T'}(lca_{T'}(s, t))$ is a matched clade with respect to S'.
   Note, again, that the only remaining node impacted in the formation of $\{s, t\}$ is the root of the extraneous subtree $\{s, t\}$. Since S^* and T^* are canonical R-RF(+) completions, we know that this node must be matched in S^* and T^*.

Now, observe that the union of paths connecting the canonical EF-R-RF(+) positions for $S(s)$ and $T(t)$ to $pa_{T^*}\{s, t\}$ in T^* is the same size as the union of paths connecting the canonical EF-R-RF(+) positions for $S(s)$ and $T(t)$ to $pa_{T''}\{s, t\}$ in T''. Moreover, *every* node in each union of paths (except the common ancestor) is mismatched. Finally, the root of $\{s, t\}$ is mismatched in T'' but matched in T^*. Since the choice of $\{s, t\}$ was arbitrary, we conclude with $RF(S', T'') - RF(S', T') = RF(S^*, T^*) - RF(S', T') + 2m$, where m is the number of extraneous subtrees in T^*. Equivalently, $RF(S', T'') = RF(S^*, T^*) + 2m$. ◀

Proof of Lemma 4.4. Let S, T be two input binary rooted trees, and let S', T' be their canonical EF-R-RF(+) completions. By the proof of Lemma 4.1, we observe two important points: First, it can only be beneficial to move a maximal red or green subtree if the maximal subtree is eventually paired in an extraneous subtree. And second, a maximal red or green subtree will increase the RF distance by a lower amount if it is paired in an extraneous subtree closer to the canonical EF-R-RF(+) position. The recurrence relation follows by induction.

Base Case: No extraneous clades can be formed at a leaf node and there are no matches to become mismatched. Hence, the cost at each leaf is indeed zero.

Inductive Step: Assume we have computed all $Cost(x, \cdot, \cdot)$ for all descendants x of an internal node v. Let $c \in \{0, 1\}$ and $0 \leq m \leq cMax(c, v)$ be arbitrarily given. We first show that twice the *maximal number of new extraneous subtrees* $\{s, t\}$ *that can be formed at* v given c_L, c_R, m_L and m_R is equal to $g_c(m_L, m_R, c_L, c_R)$. There are two cases to consider: 1. $c_L = c_R = c$ and 2. $c_L \neq c_R$ (at least one of c_L and c_R must equal c).

1. Suppose $c_L = c_R = c$ and let m_L, m_R be arbitrary nonnegative values such that $m_L + m_R = m$. Then by the first observation above, the condition $m_L + m_R = m$ is optimal to regraft m subtrees of color $c_L = c_R = c$ along the parent edge of v. By the second observation above, if there are any extraneous subtrees that can be paired at v then it is optimal to do so at v. We cannot pair any maximal red and green subtrees at v because $c_L = c_R = c$, which means that all m subtrees regrafted along the parent edge of v are the same color. Hence, twice the number of *new* extraneous subtrees that can be formed at v is equal to $g_c(m_L, m_R, c_L, c_R) = 0$ when $c_L = c_R = c$.
2. Now suppose without loss of generality that $c_L \neq c_R$ and let m_L, m_R be arbitrary nonnegative values such that $|m_L - m_R| = m$. Then by the two observations above, the condition $|m_L - m_R| = m$ is optimal to regraft the $m_L + m_R$ subtrees on the parent edge of v. By the second observation above, if there are any extraneous subtrees that can be paired at v then it is optimal to do so at v. Note that since $c_L \neq c_R$, we can pair exactly $\min\{m_L, m_R\}$ extraneous subtrees at v. Hence, twice the number of *new* extraneous subtrees that can be formed at v is equal to $g_c(m_L, m_R, c_L, c_R) = 2\min\{m_L, m_R\}$.

We now show that, regardless of the choice of colors c_L and c_R, the new increase in RF distance between S' and T' *only by regrafting* m_L *and* m_R *subtrees from* $T'(v_L)$ *and* $T'(v_R)$ *at the parent edge of* v*, respectively*, is equal to $f(m_L, v_L, c_L) + f(m_R, v_R, c_R)$. Once a subtree is regrafted at the parent edge of v_L, the only clade that can become mismatched by regrafting the subtree on the parent edge of v is $C_{T'}(v_L)$. This clade only becomes mismatched if it is a matched clade and it is not contained in a maximal c_L-colored subtree. Once the clade is mismatched, regrafting all remaining m_L maximal subtrees on the parent edge of v cannot make v mismatched again. Therefore, the act of pruning and regrafting m_L maximal c_L-colored subtrees from the parent edge of v_L to the parent edge of v increases the RF distance between S' and T' by $f(m_L, v_L, c_L)$, one for each of S' and T' if a match becomes mismatched. By symmetry, the new increase in RF distance between S' and T' from pruning and regrafting m_R maximal c_R-colored subtrees from v_R to v is equal to $f(m_R, v_R, c_R)$.

We have determined that the maximal number of new extraneous subtrees which can be formed is equal to $g_c(m_L, m_R, c_L, c_R)$, and the new increase in RF distance is $f(m_L, v_L, c_L) + f(m_R, v_R, c_R)$. Then the change in cost from v_L and v_R to v is equal to $f(m_L, v_L, c_L) + f(m_R, v_R, c_R) - g_c(m_L, m_R, c_L, c_R)$. Note if a maximal c_L-colored subtree of $T'(v_L)$ is regrafted along the parent edge of v, it must first already be regrafted along parent edge

of v_L by construction. Then, the cost of regrafting m_L subtrees at the parent edge of v_L must be $Cost(v_L, m_L, c_L)$. By symmetry, the right subtree adds a cost of $Cost(v_R, m_R, c_R)$. Moreover, the cost values also subtract the number of extraneous subtrees formed in $T'(v_L)$ and $T'(v_R)$.

Hence, the value of $RF(S', \widehat{T}) - 2p - RF(S', T')$ given *fixed* c_L, c_R, m_L and m_R is $Cost(v_L, m_L, c_L) + Cost(v_R, m_R, c_R) + f(m_L, v_L, c_L) + f(m_R, v_R, c_R) - g_c(m_L, m_R, c_L, c_R)$. By definition, the cost $Cost(v, m, c)$ is equal to the minimum over all methods of moving maximal colored subtrees in $T'(v)$ while leaving m maximal c-colored subtrees regrafted along the parent edge of v and unpaired in an extraneous subtree. Then, taking the minimum over all possible c_L, c_R, m_L and m_R values provides the optimal cost value. ◀

Proof of Theorem 4.5. We note that a pair of canonical extraneous free completions can be computed in $O(n)$ time. To compute the optimal cost values at each vertex of an EF-R-RF(+) completion, Algorithm *Compute-R-RF+(S,T)* has a total of three nested for loops, over (1) the postorder traversal, (2) the values of c and m, and (3) the values of c_L, c_R, m_L and m_R when the recurrence relation is invoked. The total time complexity is then the product of the sizes of each nested loop. Note there are a constant number of colors.

1. The postorder traversal has $O(n)$ nodes to parse.
2. Notice m must be bounded above by $\max\{cMax(0, v), cMax(1, v)\} \leq cMax(0, rt(T')) + cMax(1, rt(T')) = k$ for any vertex v. Hence, we have another multiplicative $O(k)$ factor.
3. For each $Cost(v, m, c)$ value, we observe that the number of possible values of m_L and m_R considered is again bounded above by k, adding another multiplicative $O(k)$ factor.

Thus, the total runtime to compute all cost values is $O(nk^2)$. Once all cost values are computed, the RF(+) distance can be computed in $O(1)$ time. ◀